A Group Based Approach for Coordinating Active Objects

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Juan Carlos Cruz
von Kolumbien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Prof. Dr. Stéphane Ducasse

Institut für Informatik und angewandte Mathematik

A Group Based Approach for Coordinating Active Objects

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Juan Carlos Cruz
von Kolumbien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Prof. Dr. Stéphane Ducasse

Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan:
Bern, 19 June 2006 Prof. Dr. P. Messerli

Abstract

Concurrent technology is an interesting technology to build today systems, it allows multiprocessing, it increases the throughput and responsiveness of the applications and it provides a more appropriate structure to applications that are naturally concurrent. Unfortunately concurrent technology is perceived in general by software developers as complex and difficult to use. It is evident that the concurrent model is harder to reason than the sequential model, but in our opinion this does not explain completely why this technology is perceived as complex and difficult to use. We believe that the main reason why this technology is not widely used today is that we still lack methodologies, models, patterns, and languages, that facilitate the modeling, the specification, the construction and the understanding of concurrent systems.

Recent research in the area of software engineering have suggested to manage the complexity of building complex systems by managing separately the different aspects that compose those systems. One approach in this direction in the domain of concurrent and distributed systems is that of coordination models and languages introduced by Gerlernter and Carriero in 1992. The coordination models and languages approach promotes the separation of the computation and coordination aspects in the building and the specification of concurrent and distributed systems. According to the coordination models and languages approach a complete programming model can be built out of two separate pieces: the computation model and the coordination model. The computation model concerns the specification of the elements that compose those systems and the coordination model the specification of the glue that binds all the elements together.

We believe - and this is the main claim of this thesis - that the separation of the coordination and computation aspects in the specification and construction of concurrent object-oriented systems as promoted by coordination models and languages reduces the complexity of building such kind of systems and makes concurrent technology easier to use and to understand.

We propose in this thesis the use of active objects and coordination models and languages for the specification and construction of concurrent object-oriented systems. Active objects are objects integrating concurrency and coordination models and languages are models and languages that specify the way the active objects composing the systems are glued together. Our approach is based on the definition of a coordination model and language called *CoLaS* for the specification of the coordination aspect in concurrent object-oriented systems based on active objects. The CoLaS coordination model and language introduces a high level coordination abstraction called *Coordination Group* that allows programmers to design, to specify, to implement and to validate the coordination of groups of collaborating active objects in concurrent object-oriented systems.

What is new in our approach is the use of active objects and a coordination model and language based on the notion of coordination groups to manage the specification and the construction of concurrent object-oriented systems. Until now few works have been done in this direction, none of them combining the idea of groups and message interception to perform the coordination of the active objects.

Acknowledgments

It is very difficult for me to decide who to thank the first in this acknowledgments, so many persons have participated in a way or another to this adventure all along these years. Although, I believe that definitely my mother must be first person to thank. Unfortunately she is not any more with us to read what I write. She was the one who always motivated me all along my life to learn more and to try to get the best of me. She always wanted to give to her children what she did not have and what she considered was the most important legacy she could leave us, knowledge. She could have been a scientist, or a great politician, she could have changed the world, but she could not because she did not have the opportunity to get educated! So she wanted her children did it at her place. How many times I had to explain her that a Ph.D. work was not an easy work, that a Ph.D implied a lot of effort, work and time. How many times I heard her complaining about why I have not finished my Ph.D. (I have to say she was not the only one!). It took me a lot of time to finish it but I did it, and I did it mainly for you mom. I miss you a lot. I would have given everything I have to have you here with me today.

I am also sure my father would have been extremely proud of me today. He always was! He was a very simple person but very wise. He was confronted with a difficult life since he was very young. I learned from him so many values that I consider as fundamental in life: honesty, passion for work, respect and punctuality. And yes! I did not learn to be punctual here in Switzerland, that is something I learned from my father. He died when I was very young, I did not enjoy his love and wisdom for too long time. I had to make my life without him, without his support. I would have loved to have him close to me all along of my life. But that is life I suppose! I have not forgotten you dad, not a single second of my life, I miss our long sessions talking about life.

I am very lucky in life I found my great love, I met him when I was living in France and since that first day in 94 that we met when have been always together, Martin you are my life! You supported me all along these years in this work, you always motivated me when I lost my self-confidence. This success is also yours. I love you so much:-). I would also like to thank your parents: Heinz and Marie who have also become mine. They give me all the love that I do not have anymore from mine.

In the Software Composition Group, I must start with Oscar who opened one day the doors of his group to me and indirectly changed my life. He has a lot of qualities, I admire his intelligence and the quality of his work. Things were not easy at the beginning, the group was too young at that time I believe. A lot of things changed when Stéphane Ducasse and Serge Demeyer came to our group. They were two excellent post-docs and they did an extremely good job. Stéphane Ducasse is the person I have to thank the most because of this work. We started it together, he tought me how to write papers for conferences, he tought me how to do research. And, above all, he always kept faith in me, even when things were difficult. He always defended my work, he always believed in me. Thank you, and thank you again Stéphane.

I consider myself the “living” memory of the SCG group, I know all the persons that came to our group all along these years. I want to thank first Tamar Richner, my office colleague. We had very good discussions about all different subjects, we also suffered together and went through a lot of moments of doubt. I never considered you different because you were a woman, never! I am an admirer of your work, for me you have a rare quality which is to get the point immediately. She got her Ph.D and I was so happy for her. Now, we meet from time to time to talk about life, like in the past. We do not share the same office anymore but

we keep close contact. My other office colleague was Franz Achermann, extremely nice person, very Swiss, respectful, good worker, intelligent, always ready to help you, we had very good exchanges. I still remember that day in which he discovered he was also doing coordination. It was very funny!

Sander Tichelaar my master student, now Ph.D. and friend. Excellent colleague, good worker, and socially intelligent. He gave me the frame maker templates I used to write this thesis. Roel was also an excellent colleague, he is also somebody who believed in me, and tried to motivate me to finish. Finally the two Argentinian girls: Gabriela Arevalo and Laura Ponisio. You can not imagine how many “caserolazos” Gabriela did to me, to push me to finish my thesis. A “caserolazo” is the Argentinian NOISY way to push people to make things, or at least to try!. She was also an excellent colleague and is a dear friend. It is pity that she did not cross Mr. Right in Switzerland, she would stay with us today. Well, there were so many persons in the group with which I shared so many things. My other Ph.D. colleagues: Willem, Karl, Bob, Jean-Guy, the two Markus, Luca, Matthias, Michele, Alex, Doru and Orla; the master students I directed: Thomas and Daniel and our office managers: Isabelle Huber and Therese Schmid. Many thanks to all of you for your support and friendship!

Preface

Few months ago when I was reading the march issue of the Dr.Dobb magazine, I was surprised when I found an article entitled “A Fundamental Turn Toward Concurrency in Software“[Sutt05a] presenting concurrency as a fundamental turn in software today. When I started this thesis almost 10 years ago, I read a similar article about the importance of concurrency! I was definitely surprised to find that 10 years later concurrency is still considered as the next revolution about how to write software. When you do work for your thesis for so long time, you start to lose confidence in what you do and doubt about the value of your work. After reading this article I felt more confident than ever that my work in the specification of a coordination model and language for concurrent object -oriented systems was still the future.

Table of Contents

CHAPTER 1

Introduction

We interact today more and more with concurrent applications even without knowing it; the automatic banking systems that we use to perform our banking operations, the control systems roaming and tracking our mobile phones, the retail point-of-sales systems where we buy books, the reservation systems we use to book the hotels and flights for our holidays and business travels, etc. are some examples of these kinds of systems. And, new application domains appear every day! We need to be prepared to accept the challenge of building those new systems.

Although concurrent technology is an interesting technology to build today systems, this technology is still perceived by software engineers as complex and difficult to use. They are partially right; the programming model programmers have to reason in their heads is much harder that the one for sequential control flow. Nevertheless, we believe that the main reason why this technology is not widely used today is that we still lack methodologies, models, patterns and languages that facilitate the modeling, the specification, the construction and the understanding of concurrent systems. We propose in this thesis the use of active objects and coordination models and languages for the specification and construction of concurrent object-oriented systems. Active objects are objects integrating concurrency and coordination models and languages are models and languages that specify the way the different elements composing the systems are glued together. In our case, the coordination model specifies the way the active objects are glued together in concurrent object-oriented systems. We believe - and this is the main claim of this thesis - that the separation of the coordination and computation aspects in the specification and construction of concurrent object-oriented systems as promoted by the coordination models and languages approach reduces the complexity of building such kinds of systems and makes concurrent technology easier to use and to understand.

The major advantages of the usefulness of concurrent systems are [Mage99a]: performance gain from multiprocessing software, increased application throughput, increased application responsiveness and more appropriate structure (some programs are just naturally concurrent). Although it is evident that it sounds advantageous to move from sequential to concurrent systems, it is clear that concurrent systems are more complex than sequential systems. Concurrent systems require the explicit specification of synchronization to avoid data corruption and starvation of processes [Nier00a] and run time overhead is introduced by the creation and manipulation of the threads in which the processes run. The benefits introduced by the concurrency must be weighted against its costs in resource consumption, efficiency and program complexity before deciding to build a concurrent system [Lea99a].

It is in general well accepted today that the object-oriented paradigm provides good foundations for the new challenges of concurrent computing [Brio98a]. The concurrent object-oriented paradigm integrates two simple concepts: objects and concurrency. The two concepts are strong enough to structure complex computational systems. There has been a large number of proposals about how to combine object-oriented and concurrency features, some of them are summarized in [Papa95a][Brio98a][McHa93a]. Not all the proposals have been equally successful in showing the benefits of the integration of the two concepts. The main

reason is that object-oriented features and concurrency features are not orthogonal, and consequently they cannot be combined arbitrarily [Mats94a]. There are basically three different approaches to structure a concurrent object-based system [Papa95a]: the orthogonal, the homogenous and the heterogeneous approach. In the orthogonal approach concurrency execution is independent of objects. In the homogenous approach all objects are considered as "active" entities that have control over concurrent invocations. And in the heterogeneous approach both active and passive objects are provided. From our point of view the most interesting approach is the active objects approach. In the active objects approach the objects themselves rather than the threads that invoke their operations have the responsibility to schedule concurrent requests, the active objects remain independent self-contained computational entities.

Although the active objects approach have showed the benefits of the object-oriented paradigm for concurrent computing, building and maintaining concurrent object-oriented systems using active objects is still very difficult. From our point of view one of the most important problems found in building and maintaining those systems is that the functionality of the active objects that compose the systems and they way they cooperate and synchronize are mixed within the active objects code. The mixing of cooperation and synchronization concerns makes the concurrent systems built difficult to understand, modify and customize. We need concurrent object-oriented programming languages with abstractions that enforce the separation of the two concerns.

People doing research in software engineering have suggested to manage the complexity of building complex systems by managing separately the different aspects that compose those systems [Kicz97a]. One approach in this direction in the domain of concurrent and distributed systems is that of *coordination models and languages* [Gele92a]. The coordination models and languages approach promotes the separation of the computation and coordination aspects in the building and specification of concurrent and distributed systems. According to the coordination model and languages approach a complete programming model can be built out of two separate pieces: the computation model and the coordination model. The computation model concerns the specification of the elements that compose those systems and the coordination model the specification of the glue that binds all the elements together.

Although coordination is a fundamental aspect of object-oriented programming languages for concurrent systems, existing concurrent object-oriented programming languages provide only limited support for its specification and abstraction [Frol93a][Aksi92a]. It is fundamental that concurrent object-oriented programming languages help programmers to deal with the complexity of constructing concurrent object-oriented systems if we want the "concurrency revolution" to finally happen. From our point of view concurrent object-oriented programming languages must provide high level coordination abstractions supporting the separation and specification of the coordination aspect. We propose in this thesis to follow the coordination model and languages approach to define a coordination model and language called *CoLaS* for the specification of the coordination aspect in concurrent object-oriented systems based on active objects. The CoLaS coordination model and language introduces a high level coordination abstraction called *Coordination Group* that allows programmers to design, to specify and to implement the coordination of groups of collaborating active objects in concurrent object-oriented systems.

What is new in our approach is the use of active objects and the use of a coordination model and language based on the notion of coordination groups to manage the specification and the construction of concurrent object-oriented systems. Until now few works have been done in this direction [Frol93a][Aksi92a][Papa94a], but none of them combining the idea of groups and message interception to perform the coordination of the active objects.

1.1 The Problem

We already mentioned some of the problems we believe existing object-oriented programming languages have in supporting the specification, development and maintenance of the coordination aspect in concurrent object-oriented applications. We will try to summarize them here and to explain their implications.

- **Lack of high level coordination abstractions**. Existing concurrent object-oriented languages provide only low level coordination abstractions. In Java for example, the coordination is modelled at a very low level: threads model asynchronous activities; the *synchronized* keyword, the *wait*, *notify* and *notifyAll* methods are used to coordinate the activities across threads. The Java 2 (Platform SE 5.0) [Sun04a] includes a new package of concurrency utilities: thread pools, asynchronous execution of tasks, synchronization utilities such as counting semaphores; atomic locks; and condition variables. While the set of provided constructs introduced recently in Java can be used to solve non trivial coordination problems, in practice only expert programmers are able to handle them appropriately. Java programmers tend to rely on design patterns [Lea99a] to solve common coordination problems.
- **Lack of coordination abstractions for complex interactions.** Existing concurrent object-oriented languages do not support the expression and abstraction of complex object interactions and large scale synchronizations involving more that just a pair of objects [Aksi92a][Frol93a]. The message send model used in concurrent object-oriented languages can only specify communications that involve two partner objects at a time and its semantics cannot be easily extended.
- **Lack of separation of computation and coordination concerns**. In most concurrent object-oriented systems the coordination is hardcoded inside the objects behavior. Those systems are difficult to understand, to customize and to evolve. The concurrent object-oriented languages used to build those systems do not provide abstractions that enforce the separation of computation and coordination concerns. The lack of separation of computation and coordination concerns has as a consequence a design with poor potential for reuse. The concurrent objects cannot be reused independently of the way they are coordinated and the coordination patterns cannot be reused independently of the concurrent objects they coordinate.
- **Lack of support for the evolution of the coordination code.** Three main changes in the coordination can impact the coordination in a system: 1) the addition and the removal of coordinated objects to and from the coordination, 2) the definition of new coordination patterns and 3) the modification of the coordination policies specifying the coordination. The changes range broadly from local redefinition and recompilation of the coordination and/or the active objects code to the overall redefinition and recompilation of the system.
- **Lack of support for the validation of the coordination code**. Existing concurrent object-oriented languages do not support the verification of the concurrent code. It is impossible to verify safety and liveness properties [Andr96a] of the code to guarantee the “normal” execution of the concurrent programs.

1.2 The Approach

We propose in this thesis to define a coordination model and language called CoLaS for the specification of the coordination aspect in concurrent object-oriented systems based on active objects. CoLaS is a coordination model and language based on the notion of *Coordination Group*. A coordination group is an entity that specifies, controls and enforces the coordination of groups of collaborating active objects. We consider that the primary tasks of the coordination in concurrent object-oriented systems should be: 1) to support the creation of active objects, 2) to enforce cooperation actions between active objects, 3) to synchronize the occurrence of those actions and 4) to enforce proactive behavior [Andr96a] on the systems based on the state of the coordination. The coordination groups in CoLaS supports the four primary tasks.

The CoLaS coordination model is built out of two kinds of entities: the participants and the coordination groups. The participants are the entities coordinated and the coordination groups are the entities that control and enforce the coordination of the participants. The participants in the CoLaS coordination model are active objects: objects that have control over concurrent method invocations.

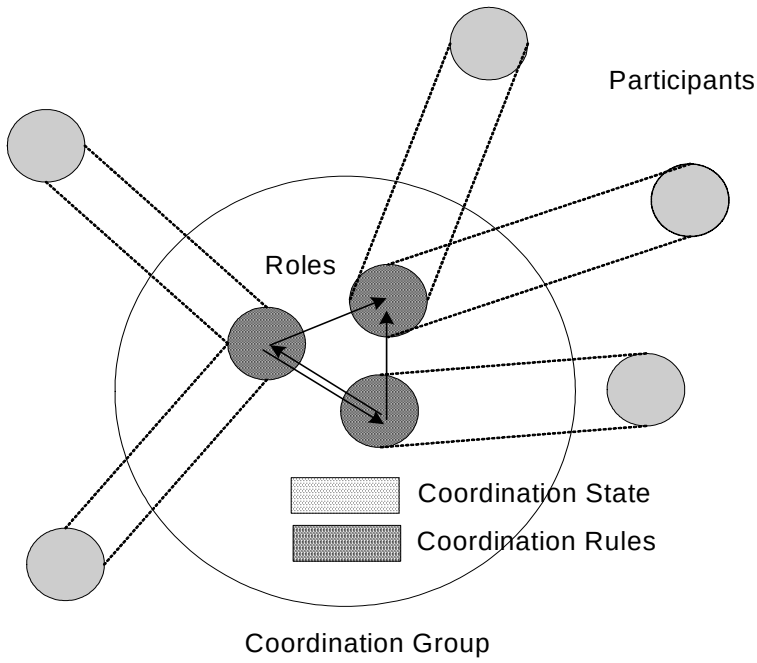


Figure 1.1 : A coordination group

A coordination group is composed of three elements (**Figure 1.1**): the roles specification, the coordination state and the coordination rules. The roles specification defines the different roles that participants may play in the group. Each role specifies the minimum requirements it imposes to an active object to play the role. The coordination state defines general information needed to perform the coordination, information like: whether some action has occurred or not in the system, the number of times some action has occurred

in the system. In general the coordination state specifies information related to the state of the coordination group and to the participants of the coordination group. Finally, the coordination rules define the different rules governing the coordination of the group. The coordination rules specify: cooperation actions between participants, synchronizations on the execution of participants actions and proactions or actions initiated by the participants independently of the messages exchanged.

One of the most important characteristics of the CoLaS model is its capacity to dynamically adapt the coordination specified in the coordination groups. The CoLaS model supports three types of dynamic coordination changes: (1) new participants can join and leave the groups at any time, (2) new groups can be created and destroyed dynamically and (3) new coordination rules can be added and existing removed from the groups

We believe the CoLaS coordination model and language tackles the most important problems that existing object-oriented programming languages have in supporting the development and maintenance of the coordination aspect in concurrent object-oriented applications:

- **Lack of high level coordination abstractions**. The coordination groups are high level coordination abstractions hiding the low level details of how the coordination is done. The programmers focus exclusively on expressing the coordination between the different roles using the coordination rules. The enforcement of the coordination is done transparently by the coordination groups.
- **Lack of coordination abstractions for complex interactions**. The coordination groups specify the coordination independently of the number of roles and the number of participants playing the roles. Also, the coordination specified in the coordination rules allows programmers to specify complex coordination protocols including message exchanges and synchronization constraints among multiple participants.
- **Lack of separation of computation and coordination concerns**. The coordination groups are specified independently of the internal representation of their participants: the coordination groups do not know which participants will play their roles, neither the participants know in which coordination groups are they playing a role. This allows a clear separation of computation and coordination concerns in a concurrent object-oriented system. The separation of coordination and computation concerns promotes design of concurrent object-oriented applications with greater potential of reuse: the coordinated entities can be reused independently of how they are coordinated and the coordination can be reused independently of the coordinated entities.
- **Lack of support for the evolution of the coordination code.** The coordination groups support the dynamic evolution of coordination, new coordination groups can be created and destroyed, new participants can join and leave the coordination groups and the coordination rules governing the coordination can be adapted to satisfy new coordination requirements.
- **Lack of support for the validation of the coordination code.** We provide in this thesis an approach to validate the most important properties of the coordination code specified in the coordination groups. The approach consists of transforming the coordination groups into Petri Nets in which reachability analysis techniques are used to validate formal properties. The main problem with our approach is that the validation process is not directly done in the coordination groups but in the Petri Nets. It is difficult to interpret the validation results obtained in the Petri Nets in the coordination groups.

1.3 Contributions of this Thesis

We consider that there are four main contributions in this thesis:

1. **Introduction of a group based approach for coordination of concurrent activities in object systems [Cruz99a].** We introduce CoLaS, a coordination model based on the notion of coordination groups. A coordination group is a high-level coordination abstraction that supports the specification, the control and the enforcement of groups of collaborating active objects in concurrent and distributed systems. The coordination groups enforce the separation of coordination and computation concerns in concurrent object-oriented systems, they allow the specification of complex interactions and support the evolution of the coordination requirements.
2. **Introduction of a coordination service for CORBA [Cruz99b][Cruz01a].** We define CORODS, a coordination service for the CORBA (Common Object Request Broker) standard [OMG95a] based on CoLaSD, an extension of the CoLaS model to support distribution. The CORODS coordination service supports the creation, the reference, the modification and the destruction of heterogeneous groups of distributed collaborating active objects. CORBA is a middleware proposed by the Object Management Group (OMG) to provide a standard for interoperability between independently developed components across networks of computers. The CORODS coordination service supports the coordination of heterogeneous distributed objects.
3. **Introduction of a platform for experimenting with the specification of rule-based coordination models [Cruz02a].** 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 definition of rules governing their coordination. We define OpenCoLaS a framework for experimenting with the specification of rule-based coordination models and languages. The OpenCoLaS framework allows programmers to specify new coordination rules in rule-based coordination models and languages.
4. **A survey of coordination abstractions.** We present a survey of coordination abstractions in existing coordination model and languages. The survey includes the most important existing concurrent object-oriented languages and coordination models and languages. We consider this survey as a first step towards the specification of a taxonomy of coordination abstractions in existing object-oriented and coordination languages.
5. **A methodology for the validation of formal properties of CoLaS coordination code**. We present a new methodology for the modeling and verification of formal properties of the coordination groups. The methodology consists of transforming the coordination groups in Predicate-Action Petri Nets. Reachability analysis is then used in the Petri Nets to validate formal properties.

1.4 Thesis Outline

The goal of this thesis is to specify a coordination model and language for concurrent object-oriented systems based on active objects. We claim that by separating the specification of the coordination aspect from the computation aspect in those systems we simplify their specification, understanding, construction, evolution and validation of properties. We have already identified in this introduction the most important problems that existing concurrent object-oriented languages have in supporting the specification of the coordination aspect: they are: 1) the lack of high-level coordination abstractions, 2) the lack of coordination abstractions for complex interactions, 3) the lack of separation of computation and coordination concerns,

4) the lack of support for the evolution of the coordination code and 5) the lack of support for the validation of the coordination code. For us these are the five big challenges to overcome in the specification of a coordination model and language for concurrent object-oriented systems. We propose in this thesis a coordination model and language called CoLaS based on the notion of coordination groups. A coordination group is an entity that encapsulates and enforces the coordination of groups of collaborating active objects. The primary tasks of the coordination group are: 1) to support the creation of active objects, 2) to enforce cooperation actions between active objects, 3) to synchronize the occurrence of those actions and 4) to enforce proactive behavior in the group of active objects.

We have organized the presentation of this thesis in the following way:

Chapter 2 defines the requirements for an ideal coordination model and language for concurrent object-oriented systems. In the first part of the chapter we provide an introduction to the coordination domain. We provide answers to some fundamental questions related to the understanding of the coordination like: Why we need to coordinate? What should be coordinated? Which are possible ways to coordinate? We additionally identify a list of coordination problems in concurrent systems and we propose a simple approach to identify them in general. In the second part of the chapter we present our analysis of the advantages and disadvantages that coordination abstractions in existing coordination models and language have in the specification of an ideal coordination model and language for concurrent object-oriented systems. The coordination abstractions analyzed correspond to those included in the survey of coordination abstractions in existing coordination models and languages included in Appendix A of this thesis. We conclude the chapter with the specification of the list of requirements we consider to be fundamental for the specification of an ideal coordination language for concurrent object-oriented systems. These requirements are used in in this thesis in the evaluation of our approach.

Chapter 3 introduces CoLaS, our coordination model and language. We present the two elements that compose the CoLaS coordination model: the participants and the coordination groups. The coordination groups are the entities that encapsulate and enforce the coordination of the participants. The specification of a coordination group contains: 1) the specification of the roles that participants may play in the group, 2) the specification of the coordination state of the group and 3) the specification of the coordination rules ruling the coordination behavior of the group. We use the Electronic Vote example introduced in [Mins97a] to illustrate the different elements that compose the CoLaS model. We conclude the chapter with an evaluation of the CoLaS model with respect to the list of requirements identified in Chapter 2 as ideal for the specification of coordination model and language for concurrent object-oriented systems.

Chapter 4 introduces CORODS, a coordination service for CORBA [OMG95a]. In this chapter we analyse the limitations of CORBA to support the construction and evolution of Open Distributed Systems. We propose the use of coordination models and languages, in particular the CoLaS coordination model to solve some of them. As we already pointed out we believe that the separation of computation and coordination concerns in systems, in particular in Open Distributed Systems facilitates their abstraction, understanding and evolution. The CoLaS coordination model is extended in this chapter to satisfy the new requirements imposed by the distribution, in particular the possibility of failures in the participants. The new coordination model called CoLaSD is introduced in CORBA as a coordination service named CORODS. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network. By using the CORODS service it is possible to perform coordination in distributed object systems guaranteeing at the same time their interoperabililty.

Chapter 5 introduces OpenCoLaS, a framework for experimenting with the specification of rule-based coordination models and languages. This chapter is divided into two parts: the first part illustrates the structure of the framework, the second part illustrates some results obtained from the use of the framework in the specification of the CoLaS model and coordination models like Synchronizers [Frol93a], Composition Filters [Berg94a] and Coordination Policies [Mins97a], the most related approaches to our work. We also show in the second part how the framework was used to compare the semantics of the rules specified in these three models with the semantics of the rules specified in the CoLaS model. At the end, we try to provide answers to the following questions in the CoLaS coordination model: Why these coordination rules and not others? Where do these coordination rules come from? Are all these coordination rules necessary?

Chapter 6 introduces a methodology to formally validate properties (i.e., safety and liveness properties) of CoLaS coordination groups. Our approach is based on the transformation of the coordination groups into Predicate-Actions Petri Nets. Structural and reachability analysis techniques in Petri Nets are used then to perform the verifications. A tool called TINA is used to perform the automatic validation of properties in the Petri Nets. At the end of this chapter we evaluate the limitations of our approach and we point out ideas about how to improve it.

Chapter 7 illustrates how the CoLaS coordination language is used to specify the coordination of a set of concurrent object-oriented systems. The examples selected cover the most important coordination problems in concurrent systems identified in Chapter 2 of this thesis: transfer of information, allocation/access of/to shared resources, simultaneity constraints, condition synchronizations, execution orderings, task/subtask dependencies, group decisions and global constraints. The variety of systems presented and their relevance as representative of the different types of coordination problems in concurrent systems demonstrates the expressive power of the CoLaS model. For each one of the systems specified we show the advantages of using the CoLaS coordination model for the specification of the coordination with respect to the use of a simple concurrent object-oriented language as Smalltalk, some of the examples additionally use Actalk[Brio89b], a support library for the specification of active objects.

Chapter 8 presents our general conclusions about this thesis pointing out our main contributions. We analyse the advantages and disadvantages of using the CoLaS model in the specification of the coordination in concurrent object-oriented systems. The evaluation of CoLaS is done based on the list of requirements identified in Chapter 2 and considered as fundament for an ideal coordination model and language for concurrent object-oriented systems. At the end, we point out the limitations of the CoLaS coordination model and language and we show some clues for future work.

Additionally we include two Appendixes:

Appendix A presents a survey of coordination abstractions in existing coordination models and languages. The coordination abstractions we include are those that we consider to be the most interesting, representatives and related to our work. We present their most important characteristics and we illustrate their use with examples.

Appendix B presents a short introduction to Petri Nets, the formalism used in this thesis to realize the verification of formal properties in CoLaS coordination groups. Petri Nets are a graphical and mathematical modeling tool used to describe and study systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. We also present different verification techniques in Petri Nets based on structural and reachability analysis. At the end we define the list of the most important safety and liveness properties that can be verified in Petri Nets.

CHAPTER 2

Requirements for a Coordination model and language for Active Objects

The goal of this thesis is to specify a coordination model and language for concurrent object-oriented systems based on active objects. We claim that by separating the specification of the coordination aspect from the computation aspect in the concurrent object-oriented systems we simplify their specification, understanding, construction, evolution and validation of properties.

The first step in the specification of a new coordination model and language for concurrent object-oriented systems based on active objects consists of understanding the problems that existing programming languages have in supporting the specification of the coordination aspect in those systems. In the introduction of this thesis we have identified five: 1) lack of high level coordination abstractions, 2) lack of coordination abstractions for complex interactions, 3) lack of separation of computation and coordination concerns, 4) lack of support for the evolution of the coordination code and 5) lack of support for the validation of the coordination code.

The second step consists of studying existing approaches for the specification of the coordination in those systems. The last decade has been rich in the number of coordination models and languages proposed [Papa98a], they differ basically in: the kinds of entities they coordinate, the underlying architecture of the models, the coordination media they use to coordinate and the semantics to which the models adhere to. Linda [Ahuj86a][Carr89a] was the first coordination model and language proposed. Despite of the importance that the understanding of the characteristics of the existing coordination and models and languages has in the specification of new coordination models and languages, few works have been done until now in this direction [Papa98a][Cian01a][Mukh95a]. From our point of view the understanding of the characteristics of existing coordination models and languages passes through the study of the coordination abstractions introduced by those models and languages.

In Appendix A of this thesis we include a survey of coordination abstractions in existing coordination models and languages. The coordination abstractions included are those that we consider to be the most interesting, representatives and closely related to our work. For each coordination abstraction we present its most important characteristics and we illustrate its use with a representative example. The survey of coordination abstractions is used in this chapter for the identification of the requirements we consider to be fundamental for the specification of an ideal coordination model and language for concurrent object-oriented systems, the goal of this thesis. For each coordination abstraction included in the survey we expose their positive and negative aspects for an ideal coordination model and language.

But, before working into the specification of the requirements for an ideal coordination model and language for concurrent object-oriented systems, we propose to analyze some aspects of the definition of what is coordination and which is its significance. We believe that even if most of the people working in the co-

ordination agree with the definition of coordination proposed by Gelernter [Gele92a] (i.e., coordination is the "the glue that binds the separate activities of a system into an ensemble") very few works introducing new coordination models and languages have tried to understand what is behind this definition. From our point of view it is important to have clear answers to the following fundamental questions if we want to define a new coordination model and language: What is coordination? Why is important to coordinate? What should be coordinated? Which are different ways to coordinate? This chapter of the thesis starts with the presentation of different works done in the domain of coordination theory in disciplines like: sociology, political science, management science, economics etc. to try to provide answers to the specific questions mentioned before. All these disciplines have dealt in a way or another with the same questions.

We have divided the presentation of this chapter into six parts:

In the first part of this chapter we introduce the classical definition of coordination introduced by Gelernter [Gele92a]. We also introduce the work done by Ciancarini [Cian96a] in the specification of the elements that compose a coordination model.

In the second part of this chapter we introduce the work done in the area of coordination theory by Malone and Crowston [Crow91a][Malo93a]. Coordination theory concerns the study of how to represent what entities (i.e., people, computer processes, economic markets, etc.) do to coordinate their actions when they work in groups in order to achieve common goals and which are the different alternative approaches to achieve those goals. These works are not specifically related to computer science, they are the result of an interdisciplinary study on coordination in different disciplines. We believe the work done by Malone and Crowston in coordination theory is extremely useful for understanding the meaning, the implications and the different approaches used to manage coordination in software systems.

In the third part of this chapter we present some coordination problems in concurrent systems. We use the results introduced in the second part of this chapter in coordination theory to identify these problems. We propose in this part of the chapter a generic method to identify coordination problems in concurrent systems. The approach is based on the identification of dependencies between the activities performed by the entities that compose those systems. The coordination problems identified are important because they justify the case studies selected in Chapter 7 of this thesis to show the relevance and the expression power of our approach.

In the fourth part of this chapter we present our analysis of the characteristics of coordination abstractions in existing coordination models and languages considering the specification of an ideal coordination model and language for concurrent object-oriented systems. We use in the analysis the survey of coordination abstractions included in Appendix A of this thesis. For each coordination abstraction we expose its advantages and disadvantages.

In the fifth part of this chapter we present our list of requirements for an ideal coordination model and language for concurrent object-oriented systems. We have included in our list of requirements thirteen aspects that we consider to be fundamental and that we will use in this thesis to evaluate our proposal. The requirements were identified taking into account the most important problems that existing concurrent object-oriented programming languages have in supporting coordination.

Finally at the end of this chapter we present our conclusions about the work presented here and we point out the main contributions of this chapter to the thesis.

2.1 Coordination Models and Languages

According to Carriero and Gelernter [Gele92a] a complete programming model can be built out of two separate pieces: the computation and the coordination model. The computation model specifies single computational activities and the coordination model *the glue that binds the separate activities into an ensemble.* A coordination language defines the linguistic embodiment of a coordination model.

A coordination model can be viewed as a triple (E,M, L) [Cian96a] where:

- E are the coordinated entities: these are the entities which are coordinated (e.g. agents, processes, tuples, atoms, etc.).
- M is the coordinating media: this is the media enabling the coordination of the entities (i.e. channels, shared variables, tuple spaces, bags, etc.)
- L are the coordination laws. They represent the semantics framework the model adheres to (i.e. associative access, guards, synchronization constraints, etc.).

A coordination model (E,M,L) represents an abstract framework useful to study and understand problems in designing concurrent and distributed systems. It provides the way to express the interaction of individual entities and the constraints imposed over their interaction.

2.2 Coordination Theory

Although most of the researchers working in the coordination area agree with the definition of coordination proposed by Carriero and Gelernter [Gele92a] few works have been done in the understanding of this definition and in its implications. What is coordination? Why is important to coordinate? What should be coordinated? Which are different ways to coordinate? are fundamental questions that must be answered at the beginning of any work in the specification of a new coordination model and language. We will introduce here different works done in what is called coordination theory [Malo93a] in disciplines including sociology, political science, management science, economics, etc. We believe that even if these works were done in areas completely different to computer science, they can help us to find answers for the questions we have formulated before.

Coordination problems arise in the organization of interactions of a group of entities that cooperate to accomplish some task and to satisfy some goals. It is because entities cooperate that they can perform more elaborated actions, but it is also because of their multiplicity that they must coordinate their actions and resolve conflicts. Coordination theory is defined as the body of principles about how entities can be coordinated to perform their tasks harmoniously.

The problem of the interaction of a group of entities not only concerns the description of the mechanisms that allow entities to interact, it also concerns the study of the different forms of interaction that entities could practice to accomplish their tasks and to satisfy their goals (i.e. cooperation, collaboration, competition etc.) [Ferb99a]. Cooperation is the most common form of interaction. It includes the resolution of all the subproblems occurring during the cooperation: coordination of actions, resolution of conflicts, etc. These subproblems are basically related with determining who makes what, when, how, which whom and with which resources. Coordination in this context concerns the organization in time and in space of the behavior of a group of entities in other either to improve their collective results, or to reduce their conflicts. It is interesting to remark that this definition of coordination does not specify the reasons why the multiple cooperating entities need to be coordinated. According to [Jenn96a] there are basically three reasons:

- There are dependencies between the activities performed by the multiple entities: interdependencies arise either when decisions made by one entity have impact on the decisions of other entities, or when it is possible to have harmful interactions.
- There is a need to meet global constraints: global constraints are conditions imposed on the way in which solutions must be implemented by the entities. If individual agents act in isolation trying to exclusively optimize their local performance, then it is almost unlikely that such global constraints will be satisfied.
- No individual entity has enough competence, resources or information to solve the entire problem.

Malone and Crowston [Malo93a] define coordination as: *the act of working together harmoniously.* This definition of coordination implies that there is work to be done (i.e., a work can be considered as a physical or mental effort or activity directed toward the production or accomplishment of something). The work is done by an actor of a group of actors. Actors that perform activities which are directed toward some ends. Those ends are called the goals. The word harmoniously in the definition means that the activities are not independent and that they must be performed in such a way that displeasing outcomes should be avoided. Malone and Crowston call the goal relevant relationships between activities interdependencies. The work of Malone and Crowston in coordination theory is basically oriented to the study of the different kinds of interdependencies and the way to manage them. Using all the elements introduced before they proposed a more precise definition of coordination:

> *Coordination is the act of managing interdependencies between activities performed by entities in order to achieve some goals.*

The main difference between the two definitions of coordination proposed by Malone and Crowston is that the first definition includes the organization of the behavior of the entities as a basic coordination activity in the coordination, while the second definition concerns uniquely the activity of managing possible conflicts (i.e., interdependencies) that occur once the behavior of the group has been defined. Both definitions agree on the final goal of the coordination which is to improve the collective results of the entities that cooperate. Malone and Crowston have identified the following kinds of basic interdependencies:

- Shared Resource: a resource is required by multiple entities in different activities.
- Prerequisite: an activity must be completed before another activity can begin.
- Transfer: an activity produces something that is required by another activity.
- Usability: something produced by an activity should be usable by another activity.
- Simultaneity: some activities need to occur (or can not occur) at the same time.
- Task/Subtask: a group of activities are all subtasks (subactivities) of an activity.
- Group Decisions: decisions are taken collectively by a group of entities.

What it is also interesting in the work done by Malone and Crowston is that they do not only propose a list of interdependencies but also the way in which these interdependencies can be managed (**Figure 2.1**). For the shared resources interdependency for example, they propose four different ways to manage this interdependency: 1) first come first serve, 2) priority order, 3) managerial decisions and 4) market like bidding. In the first come first serve approach for example the assignment of the shared resource is done on the basis of the arrival order of the requests for the use of the resource. The first entity that request for the use of the resource will be the first to be granted to have it and to modify it.

Dependency	**Examples of coordination processes for managing dependency**
Shared resources	"First come/first serve", priority order, budgets, managerial decision, market-like bidding
Task assignments	(same as for "Shared resources")
Producer / consumer relationships	
Prerequisite constraints	Notification, sequencing, tracking
Transfer	Inventory management (e.g. "Just In Time", "Economic Order Quantity")
Usability	Standardization, ask users, participatory design
Design for manufacturability	Concurrent engineering
Simultaneous constraints	Scheduling, synchronization
Task/subtask	Goal selection, task decomposition

Figure 2.1 : Malone and Crowston's dependencies management examples

The list of interdependencies presented by Malone and Crowston is not intended to be exhaustive, new interdependencies can be defined (i.e., dependencies related with time constraints), existing can be generalized and specialized (i.e., dependencies depending on the number of activities involved in the dependency); what is important about this list is that these interdependencies can be used to propose a systematic approach to identify coordination problems and new interdependencies in systems. The approach can defined as: 1) take the list of interdependencies presented before, 2) identify concurrent activities in the system you built (i.e., communication, resource management, etc.) specially activities implicating multiple entities, 3) determine the existence of interdependencies in the system from the list of activities and 4) identify and add to the list possible new kinds of interdependencies. Each interdependency identified in the system defines a potential coordination problem.

Finally, we will focus on the work made by Mintzberg [Mint92a] in the specification of different ways to handle the coordination. In this work Mintzberg considers three fundamental forms of coordination:

- Mutual Adjustment: this form of coordination occurs whenever two or more entities agree to share resources during the process of achieving some goal. The entities must exchange information to make adjustments in their behavior depending on the behavior of other entities. In this form of coordination no entity has any prior control over other entities.
- Direct Supervision: this form of coordination occurs when two or more entities have an already established relationship in which one entity has some control over the others. Commonly this relation-

ship have been established by mutual adjustment. In this form of coordination the supervisor controls the use of common resources and prescribes certain aspects of the behavior of its subordinates.

- Standardization: this form of coordination occurs when the entities follow pre-established standard procedures in a number of situations. In this form of coordination little coordination is needed, until the procedure itself needs to change. We assume that there are not conflicts in those standard procedures.

Mutual adjustment defines a form of coordination particularly well adapted to a distributed systems. The fact that no entity has a prior control over the others avoids the existence of a centralized controller susceptible of failures that might become a performance bottleneck. Nevertheless, direct supervision has the advantage that the coordination process is simpler, less messages need to be exchanged between the entities and no group decisions are necessary. Finally standardization can be used in both cases. The problem with standardization is that it is not well adapted to the evolution of the coordination requirements. When a systems changes all the standard procedures need to be modified and adapted.

2.2.1 Classification of Coordination Models and Languages

According to Papadopoulos and Arbab [Papa98a] coordination models and languages can be classified in two categories: *data-driven* and *control-driven.*

In data driven models the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components. In the data driven model there is always a coordinator process responsible for manipulating the data being received or sent and coordinating itself other processes. The coordination is done by the invocation of the coordination mechanisms provided by each language. Most of the time, in data driven coordination languages the coordination primitives are mixed to the computation code, it is the responsibility of the programmers at least at a design level to enforce the separation of coordination and computation concerns. Almost all of the coordination models belonging to the data-driven category are based on the notion of a shared dataspace. A shared dataspace is a common, content addressable data structure. All the processes involved in some computation communicate among themselves only and indirectly via the shared dataspace. Processes post information into the shared dataspace and retrieve information by copying or removing information from the shared dataspace. The shared dataspace represents the coordination medium of the models. In this category we find coordination languages like: Linda [Carr89a], Gamma [Bana86a], LO [Andr96b] just to mention some of them. Data-driven coordination languages tend to be used mostly to parallelise computations problems based on their data (data-parallelism).

In control-driven models the state of the computation at any time is defined in terms of the coordinated patterns of processes involved in some computation flow. The coordination evolves because the state of the processes change, or because events are generated. In opposition to the data driven approach the data manipulated by the processes in control driven models is almost never involved in the coordination. In the control driven coordination languages we have an almost complete separation of coordination and computation concerns. Processes (or program modules) can be separated into those related with computation and those with coordination. In this category we find coordination languages like ConCoord [Holz96a], Manifold [Arba93a] just to mention some of them. Control-driven coordination languages tend to be used primarily for modelling systems.

Arbab in [Arba98b] suggested another way to classify coordination models, he suggests that coordination models and languages can be classified in two groups: *endogenous* and *exogenous*. Endogenous models and languages provide primitives that are incorporated within the computation for its coordination. Exogenous models and languages in contrast provide primitives that support the coordination of entities completly separate from computation. In the endogenous category we find languages like Linda [Carr89a] and in the exogenous category we find languages like Manifold [Arba93a].

Another interesting work on the classification of coordination models and languages is Mukhjeri and Kafura [Mukh95a]. They suggest that coordination models and languages can be classified in three groups based on their architecture: *centralized*, *decentralized* and *hybrid*. In centralized coordination models the coordination is performed by a central agent, in decentralized models the participant entities (i.e., the coordinated entities) coordinate themselves to perform the coordination and in hybrid models the responsibility of the coordination is shared between the participant entities and a central agent.

All the classification works introduced before are important because they allow us to identify “groups” of coordination languages. Nevertheless, we believe that it will more interesting for the understanding of existing coordination models and languages to realize a categorization based on the characteristics of the coordination abstractions they introduce. We do not include here such categorization of existing coordination abstractions, but we do a first step in such a direction: we include in this chapter a survey of coordination abstractions in existing coordination models and languages. For each coordination abstraction we show its advantages and disadvantages for an ideal coordination model and language. At the end of the chapter, we use this survey to define the requirements for an ideal coordination model and language specifically adapted for the coordination of active objects - the main goal of this thesis. In Appendix A of this thesis we include the complete specification of the different coordination abstractions introduced in this chapter.

2.2.2 Importance of Coordination Models and Languages

The main advantage of using coordination models and languages in building concurrent and distributed systems results from the separation of the coordination and computation aspects in the systems. It is important to understand that the coordination languages approach do not try to solve “in principle” any new problem from the concurrency or the distribution point of view, existing concurrent object-oriented languages like Java can be used to build concurrent systems instead of using a coordination language. The point is that the lack of separation of computation and coordination concerns, the lack of high level coordination abstractions and the lack of coordination abstractions for complex interactions in languages like Java make difficult their use in building concurrent object-oriented systems, only experts are able to handle their abstractions correctly and mainly relying in design patterns. What makes unique coordination models and languages is that the separation of the computation and coordination concerns is done at the language level. The separation of computation and coordination concerns at the language level promotes:

- Reusability: both coordination patterns and coordinated entities can be reused independently from each other. The coordination patterns that specify the coordination of the coordinated entities can be defined independently of the specification of the computational behavior of the coordinated entities and vice-versa. .
- Understanding: designers and programmers can understand how systems work by studying the coordination specified in the coordination abstractions. It must be clear for a designer how the coordinated entities communicate and how they are synchronized even without knowing exactly the computational behavior of the coordinated entities.

- Evolution: the way a system works can be modified by changing the coordination patterns. Programmers modify the coordination code without modifying the code of the coordinated entities. Similarly, programmers modify the computational code of the coordinated entities without affecting the way they are coordinated.

2.3 Coordination Problems in Concurrent Systems

In the identification of coordination problems in concurrent and distributed systems we use the definition of coordination introduced by Malone and Crowston [Malo93a]: "coordination is the act of managing interdependencies between activities performed by entities in order to achieve some goals". As we already mentioned this work is not only important because of the list of interdependencies it identifies but also because it can be used to define an approach to systematically identify coordination problems in systems. The approach consists of the following steps: 1) take the list of interdependencies presented before, 2) identify concurrent activities in the system (i.e. communication, resource management, etc.), specially activities which imply the work of multiple entities, 3) determine the existence of interdependencies in these activities and 4) identify and add to the list of interdependencies possible new kinds of interdependencies. Using this approach we have identified the following coordination problems:

- Transfer of information: this problem occurs when some activity needs information from other activity (or activities) in order to continue. The information needs to be transported from an activity to another. We can view an information transfer dependency as a producer/consumer relationship. The coordination solution to this problem must take care of the physical transfer of the information from one activity to another; control their synchronization; in case of replicated transfer (i.e., multicast or broadcast) control the replication and transfer of information; and if needed guarantee the atomicity of the transfer (i.e., all or none of the entities will receive the information) and the order of arrival of the information.
- Allocation/Access of/to shared resources: this problem occurs when a group of entities sharing a resource needs part or the whole resource to perform some activity. In a fair system for example the allocation/access of/to a shared resource must be coordinated to avoid the starvation of the entities that compose those systems. The resource can be allocated for example assigning the same allocation time to each entity or the same size of the whole resource to each entity. The shared resource can be for example: the cpu time, the system's memory, the disk space, etc. An example of this coordination problem occurs when multiple users use the same printer to print documents in a networked system. The system must coordinate the allocation/access to the printer and serialize the printing process.
- Simultaneity constraints: this problem occurs when two or more activities need to occur or cannot occur at the same time (i.e., mutual exclusion). Modifications to a database for example must be serialized when two or more modification operations occur at the same time on the same row of a table. It is well known that problems like operations lost may occur when concurrent modifications are not coordinated [Coul94a].
- Condition synchronizations: this problem occurs when an activity must be delayed until some condition is satisfied. An example of this coordination problem occurs for example in a producer-consumer problem [Andr91a] when two processes a producer and a consumer synchronize their executions through the use of a bounded buffer. The consumer can only consume if the buffer contains data to be consumed and the producer can only produce if the buffer has space to put the produced data.

- Execution orderings: this problem occurs when two or more activities need to occur in a certain order in the system. An example of this coordination problem occurs when we write to a file, before to be able to write into a file, the file must be open. The two actions must be executed in this order to avoid potential problems, most of the time the file is lock during the opening of the file to avoid other users to write at the same time in the file.
- Task/Subtask dependencies: this problem occurs when the activity to be done is too big or too complex to be done by only one entity. The entities composing the system may decide to decompose a goal/task in several sub-goals/subtasks in which entities can participate according to their expertise. Usually this coordination problem is addressed since the design phase of the systems, nevertheless it is possible to find dynamic decomposition of goals and activities in systems that support for example load-balancing.
- Group decisions: this problem occurs when a group of entities needs to take a decision. Group decisions are necessary when no single element has a complete view of the whole system. An example of this coordination problem occurs when a server fails and a new server must be chosen. One of the entities that participate in the system must assume the role of new server.

To this list we can also add:

- Global constraints: this problem occurs when global constraints must be respected by all the entities during the execution of their activities. An example of this coordination problem occurs for example in a multi-user system in which different users with different execution priorities execute processes. The operation system must respect the execution constraints imposed by the different users taking always into consideration the global constraints imposed over the set of all users: administrator processes run first that user processes.

The coordination problems identified in this section are so general that they can be found in a large number of computer systems. We used these problems in the specification of a coordination component framework for Open Distributed Systems [Tich97a].

2.4 Coordination Abstractions

In Appendix A of this thesis we include the specification of twenty four coordination abstractions introduced by existing coordination model and language selected from the coordination literature. The coordination abstractions we include are those that we consider to be the most interesting, representatives and related to our work. For each coordination abstraction we show its most important characteristics, pointing out its advantages and disadvantages for an ideal coordination model and language. The coordination abstractions will be introduced alphabetically.

2.4.1 Abstract Communication Types [Aksi92a][Berg94a]

In the ACT model composition filters are applied to abstract communication among objects. The basic object model is extended to introduce input and output composition filters that affect the sent and the reception of messages respectively. Depending on the method invoked, the filters can take actions which extend/modify the original semantics of the object.

Advantages:

- It is possible to define multiparty coordination.

- The filters used in the ACT (e.g., dispatch, meta, error, wait, etc.) allow the specification of different communication and synchronization coordination patterns.
- The ACT coordination abstraction is integrated transparently into the object-oriented model, the enforcement of the coordination is based on the application of filters to method invocations received and sent by the participant objects. It is important to say that the object model proposed in ACTs corresponds to an extension of the basic object model including input and output filters. We believe that an ideal coordination model for objects must be defined without modifying the basic object model.

Disadvantages:

- The ACTs are not defined completely independent of the classes of their participants. The filters which control the coordination are specified in the inputFilters and/or outputFilters of the participant classes. The ACTs can not be reused to coordinate different kinds of participants, they are attached to the participant classes where they are defined.
- It is not possible to specialize the ACTs. Although it is possible to simulate the specialization using delegation.
- It is not possible to compose ACTs.
- It is not possible to validate the coordination code specified in the ACTs.
- It is not possible to dynamically modify the ACTs. Evolution is purely static.
- The coordination is purely reactive, triggered by the reception and the sent of method invocations. It is not possible to define proactive coordination in the participants independent of the reception and the sent of method invocations.

2.4.2 Activities [Kris93a][Kris97a]

Activities are abstractions to model the interplay between groups of objects over a given time. An activity is defined by specifying its participants and a directive. The participants specify the objects that participate to the activity and the directive the actions that compose the activities.

Advantages:

- It is possible to define multiparty coordination. Each activity defines the different classes of the participants of the activity. Multiple participants of the same classes may play at the same time in the activity.
- It is possible to specialize the activities. An activity can be defined as subclass of another activitiy.
- It is possible to compose the activities. The composed activities are called part-activities.
- The activities are defined independently of the participants. The coordination specified in the activities refers to the participants by their names. Each name corresponds to either an object or a group of objects of the same class.

Disadvantages:

- The coordination specified in the activity’s directive concerns exclusively method activations. It is not possible to define for example synchronizations.

- The coordination specified in the activities can not be reused to coordinate objects that are not instances of the classes specified in the participant names of the activities.
- It is not possible to validate the coordination specified in the activities.
- The coordination is not introduced transparently into the object model. In this approach the activities become the execution units of the programs. From our point of view an ideal coordination model for objects must define a coordination model without modifying the basic object model. Participants must not be aware of the coordination.

2.4.3 Activities and Environments [Arap91a]

Activities and Environments are used to formally describe dynamic evolution of object behavior and interactions of collections of cooperating objects. The activities describe interactions of collections of objects and the environments describe the coordination of a set of activities. The notions of object, activity and environment are formally specified using the language of first-order temporal logic FTL [Abad89a].

Advantages:

- The coordination specified in the activities and in the environments concerns the exchange of messages between the group of agents (i.e., objects) and condition synchronizations in the execution of those messages (i.e., temporal constraints).
- The coordination is specified declaratively using temporal constraints written in FTL.
- It is possible and under certain assumptions to test the consistency of a given coordination specification.

Disadvantages:

- The coordination specified in the activities can not be reused to coordinate objects that are not instances of the classes specified in the types of the activities agents.
- Only one object can play an agent role in an activity. To define the coordination of a group of objects playing the same agent role it is necessary to create several activities one per object.
- The coordination is not transparent to the participant objects in the activities, the coordination is specified based on the messages received and sent to the activities. The coordinated objects know about the coordination. The separation of computation and coordination concerns is not respected.
- It is not possible to specialize neither the activities nor the environments.
- It is not possible to compose neither the activities nor the environments.

2.4.4 Cast [Var99a]

A Cast is a hierarchical group actors. Each cast is coordinated by a single director. Coordination in the casts is accomplish by constraining the reception of messages sent to particular actors. An Actor can only receive a message when the coordination constraints associated with the reception of such a message are satisfied

Advantages:

- The coordination is separated from the computation in the cast directors.

- According to the authors this model of hierarchical coordination avoids the need of reflective capabilities. To their point of view coordination models that require reflective capabilities to intercept and control base-level actors (i.e., objects, etc.) complicate the semantics of the languages and require a specialized run-time system.

Disadvantages:

- The coordination is limited to customizing the communication.
- According to the authors the Cast model does not support the level of transparency that can be afforded by the definition of coordination abstractions using reflective architectures. The hierarchical model is limited in the customization of the communication and thus less flexible than a reflective model.
- If we suppose that the kinds of constraints associated with the reception of message are the same of those specified in synchronizers [Frol93a], this approach presents the same disadvantages of this approach. The synchronizers approach is also evaluted in this survey.

2.4.5 Connectors - FLO [Duca97a][Duca98a]

A Connector is a special object that connects components. A connector specifies how message exchanges influence the behavior of the connected components. The behavior of a connector is defined by means of a set of interaction rules which specify how the messages received by the participant objects should be controlled.

Advantages:

- It is possible to determine the compatibility of a participant to participate in a connector. Interface compatibility in roles is required to allow components participation in the connectors.
- The connectors are transparently included in the object model given that they based the interception of messages.
- The coordination is specified separately of the computation of the participants.
- It is possible to specialize connectors to define new connectors.
- What it is interesting about this work is that the connectors abstraction it is not only an architectural abstraction (as in most of the related work in connectors) specifying how objects are connected, but also as an entity in charge of enforcing the connection.

Disadvantages:

- The coordination concerns exclusively the reception of messages by the components. It is not possible to define coordination for example based on the coordination history of the connector (i.e., the number of times some action has occurred or actually occurs in the system)
- It is not clear which is the semantic difference between implies and corresponds operators, both operators can be used to delegate and to propagate messages to other objects.
- The coordination is purely reactive, it is not possible to specify proactive behavior independent of the reception of the messages.
- It is not possible to dynamically modify the coordination specified in the connectors.
- It is not possible to formally verify the connectors specifications.

2.4.6 Connectors - ArchJava [Aldr03a]

ArchJava is an extension of Java that allows programmers to specify the architecture of an application within the source code using Connectors. ArchJava adds new language constructs to support components, connections and ports.

Advantages:

- Even if it is possible to define multiple participant components in the connectors, only one component per class of component can be connected at the same time. It other words, it is not possible using a unique component reference to define connectors where multiple components of the same type are connected at the same time.
- The connections (i.e., coordination) specified in the connectors can be verified at compile time. It is possible to determine connection compatibility (i.e., required methods without provides, etc.) using the default type checking of the system or a type checking defined by user.

Disadvantages:

- The coordination specified in the connectors concerns exclusively how the components are connected. It is not possible for example to define synchronizations constraints in the connector. The coordination defined in the connectors specifies simply the flow of information between components.
- The connections specified in the connectors can not be modified dynamically.
- Because the connectors specify the type of the components that they connect, it is impossible to reuse them to connect other type of components. An ideal coordination model should completely separate the specification of the coordination from the specification of the participants. Objects participation must be defined in terms of participant interfaces and exclusivelly based on the characteristics of the participants. It should not be based neither on their names nor on their types.

2.4.7 Contracts [Helm90a]

A group of cooperating objects is called a behavioral composition. Contracts are constructs for the explicit specification of behavioral compositions. A Contract specifies: the participants in the behavioral composition and their contractual obligations, the invariants that participants cooperate to maintain, the preconditions on the participants to establish the contract and the methods which instantiate the contract.

Advantages:

- It is possible to specify multiparty coordination.
- It is possible to refine (i.e., to specialize) the contracts. The refines statement defines a contract as an specialization of another type of contract.
- It is possible to include (i.e., to compose) sub-contracts. The include statement identifies a subset of contract participants and how they participate in the sub-contract.
- Contracts are defined independently of the classes of their participants. The separation of computation and coordination concerns is respected.

- Type obligations allow to determine the conformance of the participants to participate in the contracts.

Disadvantages:

- The coordination specified in the contract's casual obligations concerns exclusively the exchange of messages. It is not possible for example to specify synchronization constraints in the contract.
- One of the most important critics found in the literature on this approach of contracts concerns the fact that the contracts are purely design abstractions, the contracts are not enforced in the participants. We believe that in an ideal coordination model for objects the coordination specified must be enforced by the system. If not, there is not any guarantee that the objects will be coordinated.
- It it not possible to validate the coordination code specified in the contracts.

2.4.8 Collaborations [Yell97a]

Collaborations are enhanced interface specifications defining the rules governing message exchanges between two components.

Advantages:

- It is possible to determine the compatibility of two components to be coordinated.
- It is possible to define adaptors to avoid the incompatibility of two components to be coordinated.

Disadvantages:

- The coordination can only be specified between two components at the time.
- The coordination is defined based on the states of the components. Any modification to the states of the components affects the coordination specified in the collaborations. The separation of the coordination and the computation concerns is not respected.
- The collaborations are tied to the components to which they belong. They can not be reused to coordinate other components.
- The coordination concerns exclusively the exchange of messages. It is not possible to specify for example synchronization constraints in the collaborations.
- It is not possible to validate the coordination code specified in the collaborations.
- The protocol semantics is synchronous. Components must wait when their mates does not find in a state that enables them to receive messages from such components. We believe an ideal coordination model for objects must support the specification of coordination with different protocol semantics. At least it must possible to specify synchronous, asynchronous and multicast communication in the coordination.

2.4.9 Coordination Contracts [Andr99a][Barr02a]

A Coordination Contract specifies the interaction between objects. A coordination contract superposes a behavior over the direct interaction of its partners by intercepting their interaction. The interaction is ex-

pressed in the form of rules, the events that triggers such rules correspond to the reception of method invocations.

Advantages:

- It is possible to specify multiparty coordination. Nevertheless, only one object of each participant class can be coordinated at the same time by the contract. It is not possible to coordinate within a same contract multiple objects of the same class using the same reference.
- The coordination is specified separated from the computation. The contracts encapsulate all the coordination specification.

Disadvantages:

- Because the contracts specify the classes of the objects that they connect, it is impossible to reuse them to connect objects instances of classes different to those specified in the contracts. An ideal coordination model should completely separate the specification of the coordination from the specification of the participants. Objects participation must be defined in terms of participant interfaces and exclusivelly based on the characteristics of the participants. It should not be based neither on their names nor on their types.
- It is not clear in this work what happens with method invocations when the guards are not valid. It seems that the method invocation is just not executed. If this is true, it is not clear if the sender of the method invocation receives any exception.
- The coordination specified in the contracts is exclusively reactive. The behavioral rules are trigger only by the reception of method invocations. It is not possible to define proactive coordination in the participants independent of the reception of method invocations.
- It is not possible to validate the coordination specified in the contracts.
- It is not possible to dynamically modify the coordination specified in the contracts.

2.4.10 Coordination Environments [Mukh95a]

Coordination Environments specify non-intrusive coordinators that impose collaborative behavior on a set of objects called autonomous objects. The coordinators use special methods called coordinating behavior methods that implement and structure coordination actions. The coordination actions are triggered by the occurrence of events related both with the acceptance of a request message and with the termination of a method that was scheduled by the coordinator.

Advantages:

- Coordination is defined transparently from the set of autonomous objects.
- The coordination actions include the synchronous and asynchronous sent of messages to other objects.
- It possible to define coordination based on the coordination history of the group using local variables.

Disadvantages:

- The coordination refers exclusively to events related to the reception of a method request (i..e, a message) and the termination of a scheduled method. It is not possible for example to define coordination based on events related to the sent of request messages to other objects.
- It is not possible to define synchronizations constraints in the coordination environments.
- Because the coordination environments specify the classes of the objects that they connect, it is impossible to reuse them to connect object instances of classes different to those specified in the coordination environments. An ideal coordination model should completely separate the specification of the coordination from the specification of the participants. Objects participation must be defined in terms of participant interfaces and exclusivelly based on the characteristics of the participants. It should not be based neither on their names nor on their types.
- It is not possible to validate the coordination specified in the coordination environments.
- It is not clear whether it is possible to specialize and compose coordination environments.

2.4.11 Coordination Policies [Mins97a]

Coordination Policies establish the set of rules regulating the exchange of messages between the members of a group. A coordination policy determines the treatment of the messages by specifying what should be done when such messages are sent or received by the members of the group.

Advantages:

- The coordination specified in the coordination policies is defined independently of the computation specified in the agents. The set of rules regulating the exchange of P-messages specify how to coordinate the messages sent and received by the agents.
- The coordination includes time obligations independent of the reception and the sent of messages by the participants. It is possible to define with the time obligations proactions in the participants.
- If we consider a coordination police to be a unit, it is not possible to reuse the coordination policies to coordinate other participants because the coordination policies specify the group of agents G which is coordinated. But if we consider each one of the rules that compose a coordination policy as independent, it is possible to reuse the coordination specified in the laws of the policy to coordinate different group of agents. The rules regulating the coordination does not specify the names of the agents they coordinate. The sender and the receiver of the messages which appear in the rules are specified as variables which are instantiated at run time. The sender and the receiver variables refer to agents in G sending and receiving messages.

Disadvantages:

- It is not possible for example to define condition synchronizations on the messages received by the agents. In the coordination policies the coordination specifies either, the deliver of the message received by an agent, the sent to the communication media of a message sent by an agent, or the modification of the internal state of the CS of the agent.
- Although each agent has a CS state, and even if it is possible to modify in the laws the CS state of an agent, it is not possible to define coordination based on the coordination history of the

group of agents. The problem with the coordination policies approach is that it is not possible to have a global view of the coordination. One agent can not specify coordination actions based on the state of other agents because it can not access the CS state of other agents.
- The coordination describes exclusively cooperation patterns, synchronization patterns can not be specified.

2.4.12 Coordination Types [Puti97a]

Coordination Types define a type model for object-oriented systems based on a process calculus. A type specifies all possible sequences of messages accepted by an object as well as type constraints on the messages parameters. A type checker ensures statically that users of an object are coordinated so that only messages specified by the object's type are sent to the object in an expected order.

Advantages:

- The main advantage of a type approach for coordination like this is that a type checker can ensure statically that users of an object are coordinated. In other words that only messages specified by the object's type are sent to the object in an expected order. Validation of the coordination is thus possible.

Disadvantages:

- The main disadvantage of type approach for coordination like this is that the coordination aspect is only partially specified in the type of each one of the interacting objects. The coordination specified in the coordination types specifies a one to one relation between an object and a client. It is not possible in this way to have a general of view of the coordination of a group of interacting objects.
- The coordination types constrain which messages are allowed to be received by an object and when they are received, it does not defines explicitly the interaction occurring between a group of interacting objects.
- The coordination specified in the type of the object defines the coordination from a point of view of a client interacting with the object. Because each object defines in its type part of the coordination, the coordination is mixed to the computation code of the object.
- The coordination specified in the object types can not be reused separately from the object in which it is defined.
- It is not possible neither to specialize the types nor to composed them to define new coordination types.
- The coordination types abstractions can not be integrated transparently in the basic object model. A coordination types type system extension is needed.

2.4.13 Darwin - Ports [Mage95a]

Darwin is a configuration language that allows distributed programs to be constructed from specifications of components instances and their interconnections. Components are defined in terms of both the services they provide and the services they require. Composite components are defined by declaring both the instances of the components they contain and the bindings between those components. The bindings associate the services required by one component with the services provided by others.

Advantages:

- The architecture of an application is explicitly defined in Darwin. In particular the connections between the different components. It is important to remark that in the coordination community architectural languages are also considered as coordination languages. First because they separate the computation from the connection aspect in the architectures and second because the connection aspect is also part of coordination.
- In Darwin it is possible to validate the connection of the components. It is possible to verify if the components provide the methods that other require in the connections.

Disadvantages:

- The coordination refers exclusively to connections between components.
- The coordination is mixed to the computation code of the components. It is not possible to reuse the connection specified in the components to connect other components.
- It is not possible to dynamically modify the connections specified in the components.

2.4.14 Event Notifications [Papa94a][Papa96a][Hern96a]

Event Notifications synchronize the activity of an object with a number of events occurring in the execution of other objects. Each object has associated an object-manager that monitors its execution and ensures local synchronization constraints. The object-manager is triggered by events occurring in the execution of the object (i.e, internal events) such as the termination of a thread executing a method and external events such as the request for a method execution.

Advantages:

- It is possible to define multiparty coordination.
- It is possible to specify synchronizations based on events occurring in other objects (i.e. changes in objects state, thread events and method execution events). The notifications can be done synchronously or asynchronously.

Disadvantages:

- The coordination specified in the object managers concerns exclusively synchronizations based on events occurring in other objects. It is not possible for example to define proactive coordination independently of events occurring in other objects.
- The coordination can not be reused to coordinate other objects, the coordination is specified within the class definitions of the participants.
- It is not possible to validate the compatibility of the participants to be coordinated. The class constructors are used to instantiate the participants of the coordination, no conditions are imposed on their participation.
- It is not possible to validate the coordination code specified in the classes.
- It is not possible to dynamically modify the coordination specified in the classes.

2.4.15 Finesse - Bindings [Berr98a]

A Binding is an abstract entity that encapsulates communication between distributed software components participating in an application. A binding describes a configuration of components and their allowed or expected interactions.

Advantages:

- It is possible to define multiparty coordination.
- A role can be played by multiple participants at the same time. Nevertheless, it is possible to constrain the number of participant playing a role.
- The coordination is specified separately from the computation. The bindings encapsulate the specification of the communication between the different participant components.
- The coordination is specified independently of the coordinated components. The role defined in the bindings are used to abstractly refer to the participant components without precisely specifying who they are. Furthermore, components have interfaces that allow them to interact with their environment without exposing implementation details.
- It is possible to specialize bindings defining subtype relationships.

Disadvantages:

- The coordination specified in the bindings concerns exclusively execution constraints related with events occurring in roles. It is not possible for example to specify coordination based on the coordination history of the group (i.e., the number of times some event has occurred in the system).
- It is not clear in this work what is an event, the author mentions that there are two types of events: input and output. It seems that the Finesse events are events related to the reception and the sent of messages in the participants playing the roles. If this is true, it means that the number of events is very limited. It is not possible for example to define coordination based on the end of the execution of a message in a participant.
- The coordination is very reactive, trigger basically by events occurring in the participants. It is not possible in this approach to define proactive coordination in the participants.
- It is not possible to validate the coordination specified in the bindings.
- It is not possible to dynamically modify the coordination specified in the bindings.

2.4.16 Formal Connectors [Alle94a]

Formal Connectors are used in the specification of the architecture of systems. A connector is described by specifying process descriptions for each of the roles that components may play and the glue used to bind them.

Advantages:

- It is possible to validate the compatibility of components to be connected.
- The connectors does not specify the type of the components they connect. Because they refer to component by they roles, it is possible to reuse the connectors to connect different types of components.

Disadvantages:

- This approach suffers from the same problems that all the works in the specification of architectural connectors. The most important is that the coordination specified in the connectors concerns exclusively how the components are connected. The coordination defined in the connectors specifies simply the flow of information between components. It is not possible for example to define coordination based on the coordination history (i.e., like the number of times some action occurred in the system) or coordination related with synchronizations.
- Another problem with connectors in general is that it is not clear whether the connectors are only design abstractions. We believe that in an ideal coordination model for objects the coordination specified must be enforced by the system. If not, there is not any guarantee that the objects will be coordinated.
- The coordination is purely reactive based exclusively on the events related with the reception and the sent of messages in the components. It is not possible to define proactive coordination in the participants.
- It is not possible to dynamically modify the connectors.

2.4.17 GAMMA - Multi-Set Rewriting [Bana96a]

A multi-set is a space containing elements. A program in GAMMA is composed of pairs reaction-condition -> action and its execution implies the replacing of those elements in the multiset satisfying the reaction-condition by the products of the action.

Advantages:

- It is possible to specify cooperation patterns where coordination can not be specified as a pre-establish sequence of actions but as a repetitive process of reactions.

Disadvantages:

- It is not clear in this model (even if the coordination appears separated from the computation) what will the result of the coordination. It is not possible to be sure in advance whether the result of coordination will be the one wished. The coordination process finish when no more reactions are possible, it could be that the coordination goals are never achieved.
- We believe that the main problem of this approach is that it does not fit into the basic object model. The coordination in an object model concerns the interaction between the objects. Objects interact by exchanging messages, the coordination targeted by this approach is purely data coordination not based on the exchange of messages.
- It is not possible to validate the coordination code specified. It depends of the kinds of elements appearing in the multi-set at a given time.

2.4.18 Gluons [Pint95a]

Gluons are special kinds of objects responsible for the cooperation of software components. A gluon is an object that handles a finite state automaton with output to control the execution of a protocol's interplay relation. The finite state automaton is composed of states and state transitions.

Advantages:

- It is possible to define multiparty coordination.
- It is possible to reuse the gluons to coordinated different kinds of components. The participants in the gluons are referred by the roles they play in the gluons and not by their names.

Disadvantages:

- A big problem with this approach is that the participants are not coordinated if they do not communicate with the gluons. We believe that in an ideal coordination model for objects the coordination specified must be enforced by the system independently of the participant objects.
- The coordination concerns exclusively the exchange of messages. It is not possible neither to define synchronization constraints in actions executed by the participants nor to define proactive coordination in the participants.
- State transitions are triggered because of messages sent to the gluons, this implies that the coordinated entities must know about the existence of the gluons to coordinate. The coordination is not transparent to the coordinated components and the separation of computation and coordination concerns is not respected.
- It is not possible neither to specialize nor to compose gluons to define new gluons.
- It is not possible to dynamically modify the gluons.
- It is not possible to validate the coordination specified in the gluons.

2.4.19 Linda - Tuple Spaces [Gele85a][Carr94a] + Linda Extensions: Bauhaus Linda [Carr94a], Bonita [Rows97a], Law Governed Linda [Mins94a], Objective Linda [Kiel96a], JavaSpaces [Sun03a]

Linda is coordination model based on the so-called generative communication paradigm. In a generative communication paradigm processes communicate by exchanging data (tuples) through a shared dataspace known as tuple space. Process can read from and write to the tuple space tuples. The tuples are retrieved from the tuple space by means of pattern matching mechanism.

Advantages:

- It is possible to specify cooperation patterns where processes can coordinate independently of their identity and where processes does not need to be alive at the same time.

Disadvantages:

- The coordination is mixed to the computation code of the coordinated entities. It is not possible to reuse coordination patterns to coordinate different entities.
- The coordination is based on the explicit exchange of data through a shared tuple space, the coordination is not transparent for the coordinated entities.
- The coordination refers exclusively to events related with the presence of some existing data in the tuple space. This model does not fix as a coordination model for objects, because objects communicate exchanging messages. We believe that an ideal coordination model for objects must focus on coordinating the interaction of the objects and not the data they produce.

2.4.20 Manifold - IWIM [Arba96a][Arba98a]

Manifold is a coordination language based on the IWIM (Idealized Worker Idealized Manager) model. The basic concepts in the IWIM model are processes, events, ports and channels. A process is a black box with well defined connection ports used to exchange information with other processes. Events are broadcasted by their source in their environment as the result of the occurrence of certain events. The processes decide which events they react to.

Advantages:

- The coordination in the IWIM model is separated from the computation code. The coordination is specified in process called managers and the computation in processes called workers. Manager processes are responsible for connecting worker processes (i.e., providing inputs and directing outputs) and react to event occurrences.
- To coordinate via events the workers must raise the events. Workers do not need to know the identity of the processes with which they exchange information, the coordination can be reused separately of the coordinated processes (i.e., workers).

Disadvantages:

- The broadcast of events from a worker is limited to the all the processes in its environment. This implies that not all the workers will receive all the events generated by the group of worker processes.
- It is not possible to dynamically modify the specified coordination.

2.4.21 Piccola - Scripts [Ache00a]

The core abstractions of the Piccola model are forms (i.e., immutable extensible records), agents (i.e., communicating processes) and channels (i.e., locations where agents asynchronously exchange forms). Forms are used to build higher-level abstractions to define composition and coordination styles. The coordination styles are implemented as component algebras. A script, is an expression of the algebra that specifies how the components are plugged together.

Advantages:

- It is possible to define multiparty coordination.
- The coordination is specified in the coordination styles separately from the computation specified within the components.
- The coordination specified in the coordination styles refers exclusively to the kinds of components they coordinate. The behavior of the components is specified by the set of services they provide and require. To our point of view, a such specification of components allows one to reuse the coordination specified in the coordination styles to coordinate different types of components.
- It is possible to define different coordination styles. In other words different ways of coordinate.
- It is possible to combine coordination styles to define new coordination styles.

Disadvantages

- The coordination specified in the coordination style defines exclusively the composition between the components. It is not possible for example to specify synchronizations constraints.
- It is not possible to dynamically modify the specified coordination.

2.4.22 Rules and Constraints [Andr96a][Andr96b]

A Rule specifies the coordination steps needed to go from one global state to another. Constraints define restrictions over the domain of interpretation of the rule.

Advantages:

- It is possible to define multiparty coordination.
- It is possible to specify synchronizations based on messages received by participant objects.
- It is possible to specify pro-active coordination behavior to make tokens appear in the pool of tokens.
- The coordination is specified in rules separately of the computation code of the participant objects.

Disadvantages:

- The coordination specified in the object managers concerns exclusively synchronizations based on messages received by the participant objects. It is not possible for example to define coordination based on the history of the coordination (i.e., events which have happened).
- The coordination is not transparent for the participant objects, they know of the existence of the coordinator because they participate to inquiry-reservation-confirmation/cancellation protocols used by the coordinator to realize coordination.
- Is not clear in this approach how objects do to propose the sequences of actions necessary to make tokens appear in the pool of tokens. The capacity to determine the sequence of actions implies some basic “intelligent” capacities which make the active objects more than simpler objects.
- It is not possible to dynamically modify the coordination specified in the rules and in the constraints.

2.4.23 Synchronizers [Frol93a]

Synchronizers are special objects that specify multi-object constraints. A synchronizer observes and limits the message invocations accepted by a set of objects, whether or not an object process a message invocation depends on the current status and invocation history of the group of constrained objects.

Advantages:

- It is possible to define multiparty coordination.
- The coordination specified in the synchronizers includes conditional (i.e.,disable constraint) and mutual exclusion (i.e.,atomic constraint) synchronizations. Additionally the coordination allows to refer to the coordination state of the participants and to the coordination history of the synchronizer.

- The synchronizer coordination abstraction is integrated transparently to the object-oriented model, the enforcement of the coordination is based on the constraint of the method invocations received by the participant objects.
- The synchronizers observe and enforce the coordination.
- The coordination is specified declaratively using rules and pattern matching.

Disadvantages:

- The coordination is based exclusively on the constraint of the methods invocations received by the participants of the synchronizer and on the messages sent by the participants to other objects are coordinated. It is not possible to define coordination based on the state of the coordination history.
- It is not possible to specialize the synchronizers.
- It is not possible to compose existing synchronizers to specify new synchronizers.
- The coordination is defined exclusively based on the reception of method invocations received by the participants of the synchronizers, it is not possible to define proactive coordination independent of the method invocations.
- The synchronizers can not be modified dynamically. Evolution support is very restricted.
- It is not possible to validate the coordination code specified in the synchronizers.

2.4.24 Wrappers [Ciob05a]

Wrappers specify the integration of components and their coordination. Components are described as objects and coordination is defined as a process.

Advantages:

- It is possible to validate the coordination.
- It is possible to define multiparty coordination specifying different arguments in the coordination wrapper.
- The coordination is specified in the coordination processes independently of the participants. An interaction wrapper describes an implementation of a coordination process.

Disadvantages:

- It is not possible that multiple participant objects play the same “role” in the coordination processes. Each argument in the interaction wrappers specifies a participant an only one.
- It is not possible neither to dynamically modify the coordination specified in the coordination processes nor to modify the participants playing “role” in the interaction wrappers.
- The coordination specified exclusivelly describes the interaction of the different participants. It is not possible neither to define synchronizations nor to define proactive coordination.
- It seems to us that is not possible neither to specialize nor to combine coordination processes.

2.4.25 Related Work - Summary

A large number of existing coordination models and languages specify a shared tuple space as a means of coordination: Linda [Ahu86a] [Carr89a], Bauhaus Linda [Carr94a], Bonita [Rows97a], Objective Linda

[Kiel96a], Law-Governed Linda [Mins94a] and Laura [Told96a]. Linda was the first coordination model and language created, its coordination model is based on the so-called generative communication paradigm [Gele85a]. In the generative communication paradigm processes communicate by exchanging data through a share data space (known as a tuple space). Generative communication decouples communicating processes in space and in time: processes do not need to know their identities in order to communicate, and they do not need to be alive at the same time in order to communicate. Additionally, the tuple space can also contain active tuples representing processes which after completing their execution turn into ordinary passive tuples. The Linda tuple space coordination model has been integrated in object-oriented programming languages [Kiel96a] and recently in Java™ in the form of a package (i.e., library of classes) JavaSpaces [Sun03a]. From our point of view the main problem with a tuple space coordination approach is that the specification of the coordination is not transparent to the coordinated entities. In a tuple space coordination model the coordinated objects are aware of the existence of a virtual share space with which they must communicate in order to coordinate. Even more,we believe that the idea of communicating by exchanging data through a shared communication media does not fix into the object-oriented model, in an object-oriented model objects communicate via the exchange of messages and not of data.

An second big group of existing coordination models and languages use a reflective approach to manage the coordination. Reflective coordination models perform coordination by intercepting and controlling base operations in the system (i.e., messages exchanges by the objects in object-oriented systems): Contracts [Helm90a], Synchronizers[Frol93a], Abstract Communication Types (ACTs) [Berg94a], Coordinating Environments [Mukh95a], Rules and Constraints [Andr96a], Coordination Policies [Mins97a], Connectors-FLO [Duca98a] and Casts [Vare99a]. The CoLaS coordination model introduced in this thesis belongs also to this group. In opposition to tuple space based coordination models, reflective coordination languages support transparent specification of the coordination. In these models the coordinated entities are not aware of the existence of coordination, the coordination patterns are specified independently of the coordinated entities. Reflective coordination models promote reuse of both the coordinated entities and the coordination patterns. Even if the coordination languages introduced before belong to the same group, there are a lot of differences between them. Contracts [Helm90a] for example are simple design specifications. In CoLaS (our approach) the coordination groups do not only constrain the treatment of messages as in Synchronizers [Frol93a] and ACTs [Berg94a] but they also enforce coordinated actions in the participants as in Rules and Contraints [Andr96a]. The enforcement of coordinated actions is done by reacting to certain specific messages or by self initiating actions in participants depending on the coordination state. In CoLaS the enforcement of coordinated actions is done at five specific moments during the processing of method invocation in the active objects: 1) at the reception of a message, 2) when a message is selected for execution, 3) before a message is executed, 4) before a message is sent to another object and 5) after the execution of a message by an active object. In Coordination Policies [Mins97a] coordination actions are also enforced as in CoLaS, but the actions can only be specified at two moments during their processing: at their reception and before they are sent to other objects. Additionally in Coordination Policies [Mins97a] the policies refer only to the local control state of the object who has received the message and in Synchronizers [Frol93a] they only refer to the state of the synchronizer. In CoLaS the coordination policies not only refer to the state of the coordination group but also to the state of the participants. One of the most important differences between CoLaS and its related coordination models and languages is its support for dynamic evolution of the coordination, in CoLaS a coordination group is a complete dynamic entity that can be created and destroyed and in which coordination rules can be added and removed at any time. Furthermore, in CoLaS new active objects can join the coordination groups at any time and existing participants may leave the coordination

group. The coordination rules only apply during the time the active objects participate in the groups. Approaches like Synchronizers [Frol93a], ACTs [Berg94a] and Coordination Policies [Mins97a] do not manage full dynamic changes of the coordination.

A third group of existing coordination models and languages define the coordination within a temporal context. The coordination abstractions composing these models are formally specified using temporal logics. The most important advantage of coordination models and languages using temporal specification of the coordination is that it is possible to test the consistency of a given coordination specification. We find in this group: Activities and Environments [Arap91a]. In CoLaS we validate our coordination specifications by transforming them into Petri Nets. The main problem with the Activities and Environments approach is that the coordination aspect is reduced to the specification of the temporal constraints associated to the execution of messages.

A fourth group of existing coordination models and languages define coordination abstractions based on the specification of interaction protocols. The coordination abstractions specify the set of messages that can be exchanged and the set of sequencing constraints imposed on them. We find in this group: Gluons [Pint95a], Collaborations [Yell97a], Activities [Kris93a] and Coordination Contracts [Barr02a]. In CoLaS we also specify the interaction protocols but we are not limited to only this aspect of the coordination. Additionally in CoLaS we specify synchronizations constraints on the execution of the messages exchanged.

A fifth group of existing coordination models and languages define the coordination based on events occurring in the systems. We find in this group: Event Notifications [Papa94a], Manifold - IWIM[Arba96a] and Finesse - Bindings [Berr98a]. Part of the coordination specified in CoLaS concerns events related with the reception of method invocations by the active objects and by the sent of method invocations to other objects.

A sixth group of existing coordination models and languages define coordination abstractions as architectural connectors. Most of the time the architectural connectors are specified as process descriptions and the components they connect by the list of services they provide and they require. We find in this group: Formal Connectors [Alle94a], Darwin - Ports [Magg95a], Connectors ArchJava [Aldr03a] and Connectors - FLO [Duca98a].

Finally we have two coordination models and languages which are very difficult to categorize, they are very unique in their approach: GAMMA - Multiset Rewriting [Bana96a] and Piccola - Scripts [Ache00a]. GAMMA specifies coordination in the form of rules reaction-conditions->action applied in a multiset. The execution of a GAMMA program implies the replacement of those elements in the multiset satisfying the reaction-condition by the products of the action. This form of coordination does not fix into the object-oriented model in which objects communicate via the exchange of messages. Piccola [Ache00a] on the other hand does not specify a specific and unique coordination style, but a set of coordination styles. The abstractions introduced in Piccola are so flexible that they can be used to specify different forms of coordination. Furthermore, the different forms of coordination can be combined, in what they call multi-styling coordination.

2.5 An Ideal Coordination Language for Active Objects

We already mentioned in the introduction of this thesis some of the problems which we believe existing object-oriented programming languages have in supporting the specification and implementation of the coor-

dination aspect in concurrent object-oriented systems. The specification of an ideal coordination language for active objects should take these problems into account. They are basically:

- Lack of high level coordination abstractions.
- Lack of coordination abstractions for complex interactions.
- Lack of separation of computation and coordination concerns.
- Lack of support for the evolution of the coordination code.
- Lack of support for the validation of the coordination code.

Requirements

The list of requirements for an ideal coordination model and language that we will introduce below correspond to design decisions taken on five aspects of a coordination language: specification, properties, behavior, evolution and validation. For each aspect we analyze different choices and we select the ones that we consider to be the most appropriate, always justifying our choice. We will always compare the chosen solution which other solutions in coordination models and languages included in our survey.

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

It must be possible for programmers to define new coordination policies within the system as in Coordination Policies [Mins97a] and their specification must be user-defined. Contrary to Synchronizers [Frol93a] that do not support incremental definition of the synchronization policies, the coordination policies must be defined incrementally from others policies like in ACTs [Aksi94a], Coordination Environments [Mukh95a], Contracts [Helm90a],and Connectors-FLO [Duca98a]. Finally, as proposed in Synchronizers [Frol93a], Rules and Constraints [Andr96a] and Coordination Policies[Mins97a] the policies must be defined declaratively to avoid programmers deal with the low-level details of how the coordination must be done.

Coordination Properties**:** Is the coordination data driven or control driven [Papa98a]? Transparently integrated in the host languages? Non-intrusive? Two-party or multi-party? Is the coordination centralized, decentralized or hybrid [Mukh95a]?

Because concurrent object-oriented languages promote data encapsulation and behavior over data, the coordination in concurrent object-oriented systems must be control driven as in Synchronizers [Frol93a], Coordination Environments [Mukh95a], Manifold [Arba96a] and Coordination Policies [Mins97a].

Contrary to Linda based approaches like Objective Linda [Kiel96a] where the coordinated objects are aware of the virtual shared space to which they communicate, the coordination must be transparent from the point of view of the coordinated objects as in Synchronizers[Frol93a], Coordination Environments [Mukh95a] and Coordination Policies[Mins97a]. Moreover, the coordination must be non-intrusive: based on the public interfaces of the coordinated objects and not relying on their internal representation.

Contrary to Collaborations [Yell97a] where the coordination is specified only between two components, the coordination must allow the specification of multi-party policies as in Synchronizers [Frol93a].

Finally, the coordination must be based on a hybrid model as in Synchronizers [Frol93a], ACTs [Aksi94a], Coordination Environments [Mukh95a] and Coordination Policies [Mins97a]. The problem with centralized models like Gluons [Pint95a], Rules and constraints [Andr96a] is that objects are forced to interact with a coordinator agent and the problem with decentralized models like Event Notifications

[Papa96a] is that the objects must know the other objects to perform the coordination. The reusability of objects and coordination is limited in both cases.

Coordination Behavior: Is the coordination limited to the synchronization of actions? or Can actions be enforced and/or be initiated by the system? What kind of information should be referred to by the coordination policies?

Coordination must not be limited as in Synchronizers [Frol93a] and ACTs [Aksi94a] to the synchronization of actions, it must be possible to enforce actions in the coordinated objects independently of the actions occurring in the system. Moreover, it must be possible to initiate actions (i.e., proactive actions [Andr96a][Mins97a]) as in Coordination Policies [Mins97a] based on the state of the coordination. The state of coordination must include the state of the coordinated objects and the history of the coordination.

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

The coordination must be highly dynamic. Objects must be able to join and/or leave the coordination at any time. It must be possible to dynamically modify existing coordination policies and to create new ones at run-time [Andr96a]. A highly dynamic system must be able to dynamically adapt to new coordination requirements.

Coordination Validation: Can we prove that the behavior 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 a formal model must be fully integrated to the coordination language to check the ability of the objects to be coordinated. Furthermore, we would like to be able to prove certain safety and liveness properties of the coordination like deadlock free and termination. The formal model must not be limited to the specification and the verification of the coordination as in Formal Connectors [Alle94a] but causally connected to the language in the sense of an “executable specification”.

We can summarize the requirements for an ideal coordination model and language in the following list:

- The coordination policies must be defined independently of the coordinated entities: the coordination model must enforce the separation of the coordination and the computation aspects. It must be possible to define coordination policies independently of the specification of the coordinated entities.
- It must be possible to define new coordination policies in the coordination model: the coordination model must allow programmers to define their own coordination policies and do not constrain them to use fixed coordination policies.
- It must be possible to incrementally define new coordination policies in the coordination model: the coordination model must allow programmers to use existing coordination policies in the specification of new coordination policies.
- The coordination policies must be multi-party: the coordination model must allow the specification of coordination policies referring to different types of coordinated entities. Furthermore, not only it should be possible to coordinate different types of coordinated entities but also several entities of the same type.
- The coordination policies must be declaratively defined in the coordination model: the coordination model must allow the specification of the coordination in a declarative way avoiding the program-

mers the details of how the coordination must be done. High level coordination abstractions should be used to hide the details about how the coordination is done.

- The coordination policies must be control-driven defined in the coordination model: the coordination model must respect and adapt to the basic object model to specify the coordination. No new abstractions must be added to the object model to specify the coordination.
- The coordination model must be transparently integrated into the host language: the coordination model must integrate into the host language without imposing any constrain to the host language. The coordinated entities must not be aware of the existence of the coordination layer in the systems.
- The architecture of the coordination model must be hybrid: the enforcement of the coordination in the coordination model must be shared between the coordinated entities and a central coordinator. It must be possible to get advantage of the computing power of the entities being coordinated in the enforcement of the coordination and do not convert the coordinator in a bottleneck for the system.
- The coordination policies must include the possibility to define proactions in participants: the coordination model must not be exclusively reactive waiting for events or actions occurring in the system. It must specify proactive coordination in the coordinated entities.
- The coordination policies must include the possibility to refer the state of the participants and to the coordination history of the system: the coordination model must allow the specification of coordination referring to the state of the participants and the history of the coordination.
- It must be possible to dynamically modify the coordination policies: the coordination model must allow the dynamic modification of the coordination. It must be possible to easily adapt the coordination policies to new requirements in the systems.
- It must be possible to prove the capability of the coordinated entities to be coordinated: the coordination model must to allow the system to validate whether potential coordinated entities are capable to participate in the coordination.
- It must be possible to validate basic safety and liveness properties of the coordination: the coordination model must allow programmers to validate formal properties in the coordination specified.

2.6 Conclusions and Contributions

We propose in this thesis, the use of active objects and coordination models and languages for the specification and construction of concurrent object-oriented systems. We believe that by separating the specification of the coordination aspect from the computation aspect in concurrent object-oriented systems we simplify their specification, understanding, construction, evolution and validation of properties We have identified that the most important problems that existing programming languages have in supporting the specification of the coordination aspect in object-oriented systems are: 1) lack of high level coordination abstraction, 2) lack of coordination abstractions for complex interactions, 3) lack of separation of computation and coordination concerns, 4) lack of support for the evolution of the coordination code and 5) lack of support for the validation of the coordination code. Our goal in thesis is to specify a coordination model and language for concurrent object-oriented systems that tackles all these problems.

A large number of coordination models and languages exist [Papa98a], they differ basically in: the kinds of entities they coordinate, the underlying architecture assumed by the models, the coordination media they use to coordinate and the semantics to which the models adhere to. We include in Appendix A of this thesis a survey of coordination abstractions in existing coordination models and languages. From our point of view none of the coordination models and languages included in our survey fully satisfies the list of require-

ments we have identified as fundamental for the specification of a coordination model and language for concurrent object-oriented systems. The requirements can be summarized in the following list:

- The coordination policies must be defined independently of the coordinated entities.
- It must be possible to define new coordination policies in the coordination model.
- It must be possible to incrementally define new coordination policies in the coordination model.
- The coordination policies must be multi-party.
- The coordination policies must be declaratively defined in the coordination model.
- The coordination policies must be control-driven defined in the coordination model.
- The coordination model must be transparently integrated into the host language.
- The architecture of the coordination model must be hybrid.
- The coordination policies must include the possibility to define proactions in participants.
- The coordination policies must include the possibility to refer the state of the participants and to the coordination history of the system.
- It must be possible to dynamically modify the coordination policies.
- It must be possible to prove the capability of the coordinated entities to be coordinated.
- It must be possible to validate basic safety and liveness properties of the coordination.

We believe and we will be prove it all along this thesis that our approach CoLaS, a coordination model and language based on the notion of coordination groups and specially adapted to specify the coordination in concurrent object-oriented validates all of these requirements. The CoLaS coordination model will be introduced in the next chapter of this thesis.

The main contributions of this chapter to the this thesis are:

- We provide an introduction to what is coordination and its significance. We provide answers to important questions about coordination: What is coordination? Why is important to coordinate? What should be coordinated? Which are different ways to coordinate? In the coordination theory coordination can be defined as the act of managing interdependencies between activities performed by entities in order to achieve some goals. The goal of the coordination is to make entities work harmoniously. Forms of coordination are: 1) mutual adjustment, 2) direct supervision and 3) standardization.
- We provide an approach to identify coordination problems in concurrent systems. The approach is based on the identification of dependencies between the activities performed by the entities that compose those systems. We have identified eight coordination problems: 1) transfer of information 2) allocation/access of/to shared resources, 3) simultaneity constraints, 4) condition synchronizations, 4) execution orderings, 5) task/subtask constraints, 6) group decisions and 7) global constraints.
- We provide an analysis of the advantages and disadvantages for the twenty four coordination abstractions included in our survey of coordination abstractions in Appendix A of this thesis. The analysis is made considering the specification of an ideal coordination model for concurrent object-oriented systems. The coordination abstractions included in the survey are those that we consider to be the most interesting, representatives and related to our work. The result of the analysis is used in the specification of the requirements presented just before. We consider these requirements as fundamental for the specification of an ideal coordination model for coordinating concurrent object-oriented systems. They will guide the specification of our coordination model and language CoLaS.

CHAPTER 3

The CoLaS Coordination Model and Language

In the introduction of this thesis we pointed out that one the most important problems in building and maintaining concurrent object systems results from the fact that the functionality of the active objects that compose these systems and they way they cooperate and synchronize are mixed within the objects code. The mixing of concerns makes the concurrent object systems built difficult to understand, modify and customize. We also pointed out that such a problem can be tackled by managing separately the two different aspects as proposed by the so called coordination models and languages [Gele92a]. According to the coordination model and languages approach a complete programming model can be built out of two separate pieces: the computation model and the coordination model. The computation model concerns the specification of the elements that compose the systems and the coordination model the specification of the glue that binds all the elements together.

In Chapter 1 of this thesis, we concluded from the analysis of existing concurrent-object programming languages that the most important problems they have to support the specification and abstraction of the coordination aspect are:

- Lack of high-level coordination abstractions.
- Lack of coordination abstractions to express complex coordination patterns.
- Lack of separation of the computation and the coordination aspects.
- Lack of support for the evolution of the coordination requirements.
- Lack of support for the validation of the coordination code.

In Appendix A of this thesis we include a survey of coordination abstractions in existing coordination models and languages. The coordination abstractions we include come from coordination models and languages that we considered are the most interesting, representatives and related to our work. We present for all the coordination abstractions their most important characteristics illustrating their use with examples. In chapter 2 of this thesis we went through all the coordination abstractions included in our survey analyzing their advantages and their disadvantages which respect to characteristics of an ideal coordination model and language for concurrent object-oriented systems. We identified from this work the list of requirements we consider to be fundamental for a coordination model and language for concurrent object-oriented systems. The requirements can be summarized in the following list:

- The coordination policies must be defined independently of the coordinated entities.
- It must be possible to define new coordination policies in the coordination model.
- It must be possible to incrementally define new coordination policies in the coordination model.
- The coordination policies must be multi-party.
- The coordination policies must be declaratively defined in the coordination model.

- The coordination policies must be control-driven defined in the coordination model.
- The coordination model must be transparently integrated into the host language.
- The architecture of the coordination model must be hybrid.
- The coordination policies must include the possibility to define proactions in participants.
- The coordination policies must include the possibility to refer the state of the participants and to the coordination history of the system.
- It must be possible to dynamically mofiy the coordination policies.
- It must be possible to prove the capability of the coordinated entities to be coordinated.
- It must be possible to validate basic safety and liveness properties of the coordination.

In this chapter we introduce CoLaS, our approach to the specification of the coordination aspect in concurrent object-oriented systems. The CoLaS model is based on the notion of coordination groups. A coordination group is an entity that specifies, controls and enforces the coordination of a group of collaborating concurrent objects. We consider that the main tasks of the coordination in concurrent object systems are: 1) to support the creation of the objects, 2) to enforce cooperation actions between the objects, 3) to synchronize the occurrence of those actions and 4) to enforce proactive behavior in the objects [Andr96a] based on the state of the coordination. The CoLaS coordination model supports the four types of tasks in the specification and construction of concurrent object-oriented systems.

The CoLaS coordination model uses a reflective approach to manage the coordination aspect. Reflective coordination models perform coordination by intercepting and controlling base operations in the system. CoLaS is based on the interception of the messages exchanged by the group of collaborating objects within the coordination groups. Coordination rules define actions to perform when the messages are intercepted.

We believe, and we will show all along this thesis, that the CoLaS coordination model and language satisfies all the requirements we have identified as ideal for a coordination model and language for concurrent object-oriented systems. We will use the CoLaS examples introduced in this thesis to illustrate concretely how these requirements are satisfied in the coordination solutions specified.

We have divided the presentation of this chapter into three parts:

In the first part of this chapter we introduce the CoLaS coordination model, using the example “Subject and Views” [Helm90a]. Using this example we illustrate the basic elements that compose the model. We try to remain very abstract in this presentation, our goal is to give the reader a simple idea about how CoLaS can be used to model and specify coordination problems.

In the second part of this chapter we use the Electronic Vote [Mins97a] and the Electronic Agenda [Bosc97a] examples to illustrate in detail all the different elements that compose the model. During their presentation we build step by step the specification of the coordination groups containing the coordination specification of the two problems. Again we will explain concretely in the examples how CoLaS satisfies the list of requirements for an ideal coordination model and language.

Finally in the third part of this chapter we evaluate the CoLaS coordination model and language with respect to the list of requirements identified as ideal for a coordination model for concurrent object-oriented systems. We conclude this chapter with an evaluation of the pro and the cons of the CoLaS coordination model, pointing out its main contributions and some future work.

3.1 The CoLaS Coordination Model

CoLaS is a coordination model based on the notion of *Coordination Groups*. A coordination group specifies, encapsulates and enforces the coordination of a group of collaborating participants. The CoLaS model is built out of two kinds of entities: the participants and the coordination groups (**Figure 3.1**).

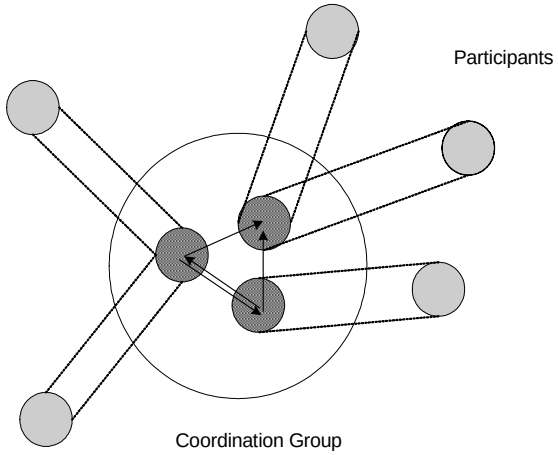


Figure 3.1 : Entities composing the CoLaS model

3.1.1 The Participants

In CoLaS the participants are *active objects*: concurrent objects that have control over concurrent method invocations. In an active object, incoming method invocations are stored into a mailbox until the object is ready to process them. Whether an object is ready or not to process a method invocation depends on the *synchronization policy* associated with the active object [McHa93a]. A synchronization policy defines which methods invocations can be executed concurrently by the object, its purpose is to ensure the consistency of the object state. In CoLaS participants treat the incoming method invocations in a sequential way (i.e., one at the time) following a mutual exclusive synchronization policy.

In CoLas the participants communicate by exchanging messages in an asynchronous way. A message sent from a participant to another participant represents a request for a method invocation in the other participant. The fact that the communication is asynchronous implies that participants are not blocked while their requests are processed by the other participants, they may continue working until they receive their replies from the other participants. The replies are managed using explicit futures. Every message sent to another participant generates a reply, it is up to the participant who receives the future to decide to request or not the reply to the future.

3.1.2 The Coordination Groups

A coordination group is an entity that specifies, controls and enforces coordination between groups of collaborating participants. The primary tasks of a coordination group are: (1) to enforce cooperation actions

between the participants, (2) to synchronize the occurrence of those actions and (3) to enforce proactive actions (proactions in the following) [Andr96a] in the participants based on the state of the coordination.

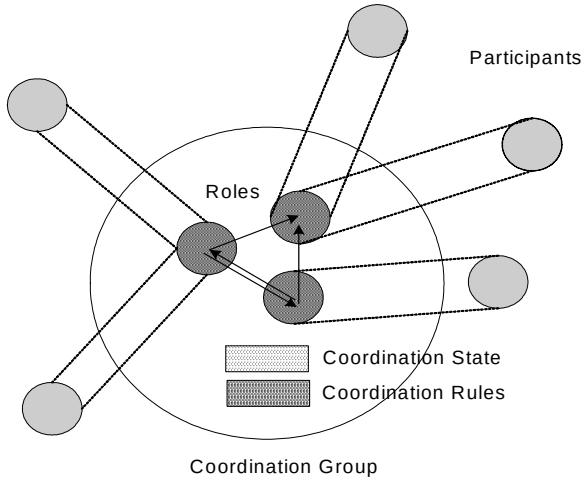


Figure 3.2 : Coordination Group

Coordination Groups Specification

Coordination Groups (i.e.,only groups in the following) are composed of the following three elements (**Figure 3.2**): *the Roles Specification*, *the Coordination State* and the *Coordination Rules*.

- *The Roles Specification*: defines the different roles that the participants may play in a group. Participants playing the same role in a group behave in the same way from the coordination point of view. For each role it is possible to specify a role interface with the minimum requirements for an active object to play the role. Role interfaces are specified by sets of method signatures.
- *The Coordination State*: defines general information needed to perform the coordination. It concerns information like: whether some action has occurred in the system (i.e., historical information), the number of times some action has occurred or actually occurs in the system (i.e., historical counters) and in general information useful to perform the coordination and related with the state of the group and the state of the participants. The coordination state is specified in the form of variables. The CoLaS model defines three types of state variables: *Group Variables* (i.e., variables shared by all the participants of the group), *Role Variables* (i.e., variables shared by all the participants of a role) and *Role Participant Variables* (i.e., private variables associated with each participant of a role).
- *The Coordination Rules*: defines the different rules that govern the coordination of the group. They are associated with the roles and regulate the coordination of all the participants playing the roles. There are three different kinds of coordination rules: *Cooperation Rules* (i.e., specify cooperation actions between the participants), *Reactive Rules* (two sub-types: *Interception Rules* and *Synchronization Rules*. They specify mainly synchronizations over the occurrence of actions in the participants) and *Proactive Rules* (i.e., specify proactions [Andr96a] in the participants).

Active Objects Group Participation

Active objects join coordination groups by joining group roles. To play a role in a group, an active object should at least have the functionalities required by the role. The functionalities required by a role to its participants are specified in the role interface. A role can be played by more than one participant and a participant can play more than one role. Active objects join and leave the groups at any time without disturbing other participants.

Coordination Enforcement

Cooperation rules defines new behaviors for the participants, they are executed by the participants when they receive method invocation requests related to the behaviors specified in the rules. Reactive rules are enforced at four different moments (i.e.,evaluation points) during the processing of the method invocations received by the participants. The reactive rules are checked to verify whether they apply to the request, if so, the rules are enforced (e.g., messages are sent to other participants, the execution of the request is delayed or ignored, etc.). The four evaluation points defined are: at the arrival of the method invocation request, before the execution of the method invocation, before the sent of a message to another participant and after the execution of the method invocation). On the other hand, Proactive rules do not depend on the messages received or processed by the participants but on the state of the coordination of the group. They are enforced non deterministically by the group.

3.1.3 A first View of CoLaS - Subject and Views [Helm90a]

To illustrate the basic characteristics of the CoLaS coordination model we will use the “Subject and Views” coordination problem [Helm90a]. The Subject and Views coordination problem appears when a *Subject* object containing some data and a collection of *View* objects which represent that data graphically (e.g., as a dial, a histogram, or as a counter) cooperate so that all times each *View* always reflects the current value of the *Subject*. The “Subject and Views” coordination problem can be solved using the Observer pattern [Gamm95a]. The Observer pattern defines a design solution to a one-to-many dependency relation between objects so that when one object changes its state, all its dependents are notified and updated automatically. They key objects in this pattern are the subject and the observers. A subject may have any number of dependent observers. All the observers are notified whenever the subject undergoes a change in its state. In response, each observer queries the subject for its state to synchronize its state. The dependency relation specified in the Observer pattern is also known as publisher-subscriber relationship.

Coordination Problems

- Synchronization Constraints: all the different observers reflect the current value of the subject. The state of the observer remains synchronized with the state of the subject.
- Transfer of information between entities: all the dependent observers are notified when the state of the subject changes, the observers then request the value of the new state to the subject and update their states.

Structure

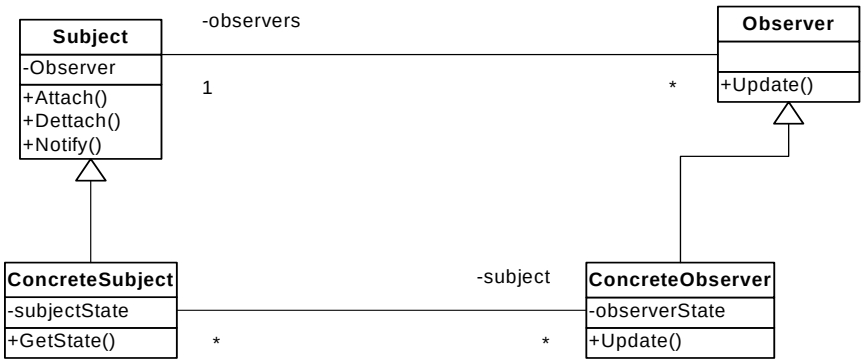


Figure 3.3 : Observer Pattern structure

In (**Figure 3.3**) we show the structure of the Observer pattern [Gamm95a]. The class Subject represents a a subject and the class Observer the observers (i.e., the views) of the subject. Any number of observer objects may observe a subject. The Subject class provides an interface for attaching and detaching observer objects. The Observer class defines an update interface to update the observer state when the observer receives a notification of a change in the state from the subject. The class ConcreteSubject stores the state of the subject and specifies the interface of a method called GetSate to request for the value of the subject's state. The class ConcreteObserver maintains the reference to the ConcreteSubject object and a copy of the subject's state in the observerState variable.

Solution

In (**Figure 3.4**) we create a coordination group named ObserverPattern to encapsulate the coordination described in the pattern. The coordination group specifies the two roles Subject (line 3) and Observer (line 7) representing the two types of objects subject and observers in the pattern. The role Subject constrains the number of participants playing the role to only one (line 4). Only one object may play the role Subject in a ObserverPattern coordination group. The role interface of the role Subject (line 5) specifies that only objects who know how to react to the behaviors getObjectState and setObjectState: can play the role Subject and the role interface of the role Observer (line 8) that only objects who know how react to the behavior doSpecificAction: can play the role Observer. The getObjectState and setObjectState: behaviors allow to access and to modify the state of the object playing the role Subject and the doSpecificAction: behavior to perform any specific action in the observers related with the change in the subject's state (i.e., to redraw a view).

The role Observer defines two participant variables subject (line 9) and observerState (line 10). The subject variable is used to keep the reference to the subject in the observers and the observerState variable to keep a copy of the current state of the subject. The value of the observerState variable in each observer is maintained synchronized with the value of the subject's state by the coordination.

The ObserverPattern coordination group specifies the following coordination rules (only Cooperation Rules in this case):

Rule 1 (line 12): specifies how observers are attached to the subject. A reference to the object playing the role Subject is stored in the subject's variable of the observer.

Rule 2 (line 16): specifies for a subject that whenever the state of the subject changes a notify message is synchronously sent (i.e., message send to self) to the same object (line 18).

Rule 3 (line 20): specifies for a subject that an update message is sent to all the observers of the subject when a notify message is received by the object.

Rule 4 (line 23): specifies for a subject that the current state of the subject is returned when a getState message is received from a observer.

Rule 5 (line 26): specifies for an observer that the value of the observerState variable in the observer is updated when an update message is received from the subject. The current state of the subject is requested directly to the subject (line 27) using the subject participant variable in the observer. The reply to the request is received through a future (result keyword, line 27), the execution of the behavior update in the observer is blocked until the reply is sent by the subject and received by the observer. The doSpecificAction: corresponds in the "Subject and Views" problem to the redraw of the view.

```
1.CoordinationGroup createCoordinationGroupClassNamed: #ObserverPattern.
2.
3.ObserverPattern defineRoleNamed: #Subject.
4.Subject maxNumParticipants: 1.
5.Subject defineInterface: #(#getObjectState #setObjectState:).
6.
7.ObserverPattern defineRoleNamed: #Observer.
8.Observer defineInterface: #(#doSpecifAction:).
9.Observer defineParticipantVariable: #subject.
10.Observer defineParticipantVariable: #observerState.
11.
12.[1] Subject defineBehavior: 'attach: anObserver' as:
13.      [Observer addParticipant: anObserver.
14.      anObserver subject: self receiver].
15.
16.[2] Subject defineBehavior: 'setState: aState' as:
17.      [self setObjectState: aState.
18.      self notify].
19.
20.[3] Subject defineBehavior: 'notify' as:
21.      [Observer update].
22.
23.[4] Subject defineBehavior: 'getState' as:
24.      [^self getObjectState].
25.
26.[5] Observer defineBehavior: 'update' as:
27.      [self observerState: (self subject getState result).
28.      self doSpecificAction: self observerState].
```

Figure 3.4 : Observer pattern

Analysis

This example illustrates clearly how the coordination aspect of the "Subject and View" problem is specified completely separate from the specification of the computation code of the coordinated objects (i.e., the active objects which play the roles Subject and Observer). We can see in the example that the coordination specified in the coordination group refers exclusively to the coordinated objects by the role they play in the group and not by their names. The separation of the coordination aspect and the implementation of the participants allows the coordination group to coordinate different kinds of participant objects and to the participant objects to participate in different coordination groups. The only constraint imposed on the objects for their participation in the groups is the respect of the role interface defined in the roles they will play. In the example, the only constraint imposed to the participants of the role Subject to participate is to provide methods to access and modify their state and to the participants of the role Observer to provide a method to perform the specific action related with the change in the state in the subject. Some existing coordination models and languages constrain the participation of active objects to only those of the type specified in the roles like in Activities[Kris93a], Activities and Environments[Arap91a] and Coordination Contracts[Andr99a]. We believe that the role interface should strictly specify what is necessary for the coordination and not more, associating types to the roles unnecessarily constrains the kinds of participants that may play the role. We believe that the way in which we specify in CoLaS the roles interfaces is the right solution because it concerns exclusively the behavior that must be known by the participants to participate in the coordination groups and nothing more.

It is also important to remark that the number of roles in the groups is not limited. It is possible to specify complex multiparty coordination patterns. Existing concurrent programming languages like Java and some existing coordination model and languages like Collaborations [Yell97a] limit the coordination to two objects at the time. It is not possible to define multiparty coordination patterns. Furthermore, because in CoLaS there is no limitation in the number of participants that may play a role it is possible to specify using the coordination groups multi party-multi participant coordination patterns (i.e., more than one participant playing the same role at the time). Some coordination models and languages in which the participants are specified by their types like in Activities and Environments[Arap91a], Connectors-FLO[Duca97a], Coordination Contracts[Andr92a] and Gluons[Pint95a] constrain the number of participants to only one per role. The possibility to define multi participant coordination patterns is definitely a plus from the point of view of the simplicity of the specification of the coordination. In approaches where multi participant coordination can not be specified in a single abstraction programmers are forced to specify multiple coordination relations between the multiple participants of the roles.

3.2 The CoLaS Coordination Language - A Detailed View

To illustrate in detail all the different aspects of the CoLaS coordination model we will use as example the Electronic Vote [Mins97a]. We will describe in a first time the problem and then step by step we will build the specification of a coordination group that contains the specification of the coordination of the problem.

3.2.1 A Case Study: The Electronic Vote [Mins97a]

Problem Description

In the electronic vote, an open group of participants is requested to vote on a specific issue. Every participant in the group can initiate a voting process on any particular issue at any time. Participants vote by send-

ing their votes to the participant who initiated the voting process and only in the time frame fixed by the initiator of the voting process. The system must guarantee that the vote is fair: (1) each participant votes at most once and only within the period of time established, (2) that the counting is done correctly and only votes from participants of the group are counted and (3) that the result of the vote is sent to all the participants at the end of the voting period. Initially, the counting policy applied to determine the result of the vote will be consensus (i.e., the result of the vote will be positive if the number of positive votes received is equal to the number of voters, otherwise the result will be negative), however other counting policies may also be specified. In (**Figure 3.5**) we can see the UML class diagram corresponding to the solution of the electronic vote problem.

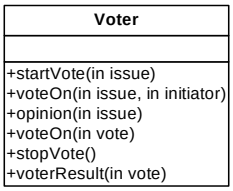


Figure 3.5 : The Electronic Vote - UML Class Diagram

In the Electronic Vote problem we identify only one type of participant: the voter. The UML interaction diagram in (**Figure 3.6**) describes the vote process for the electronic vote. The vote process initiates with a startVote message sent by a voter (a). The initiator of the vote then sends the message voteOn: (b) to all the voters (including himself) to request for their votes. Implicitly the initiator of the vote opens the voting period. The issue of the vote is sent as part of the voteOn: message sent to the voters. Each voter receives then the request for the vote and votes according to its own opinion (c), the value of the vote is then sent to the initiator of the vote process (d).When the initiator of the vote process decides to stop the vote process (e), it closes the voting period and calculates the result of the vote. The result of the vote is then sent to all the voters.

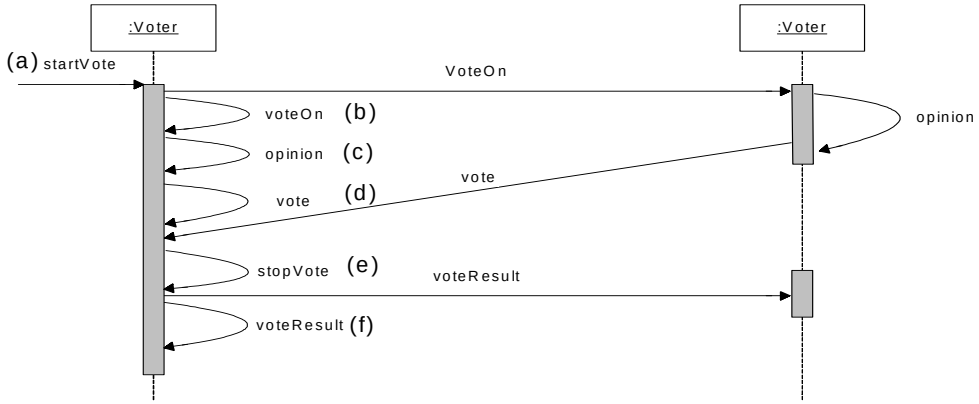


Figure 3.6 : The Electronic Vote - UML Interaction Diagram

The electronic vote example illustrates the following coordination problems:

- Transfer of information: voters communicate with other voters to initiate the voting processes. During a voting process voters communicate with the initiator of the vote to send their votes. The initiator of the voting process stops the vote process, determines the result of the vote and sends the result to all the voters.
- Global Constraints: different global constraints have to be respected: voters can vote at most once, only votes received during the voting period must be taken into account and only one voting process occurs at the same time.
- Dynamic evolution of the coordination: new voters can join the voters group and existing voters can leave the group at any time, the counting policy applied to determine the result of the vote process can be modified at any time by the group.

3.2.2 Roles Specification

In a group, a role specifies a set of participants sharing the same coordination behavior. In the Electronic Vote example only the role Voter was identified. The role Voter refers to all the entities participating in the voting process. Roles are defined by sending the message defineRoleNamed:<Role Name> to the group with the name of role to be created as argument. In (**Figure 3.7** line 3), we show how the role Voter is defined in the group ElectronicVote.

```
1.CoordinationGroup createCoordinationGroupClassNamed: #ElectronicVote.
2.
3.ElectronicVote defineRoleNamed: #Voter.
4.
5.Voter defineInterface: #(#opinion:).
```

Figure 3.7 : Electronic Vote - Coordination Roles

The minimal interface that an active object should have to play a role in a group is specified in the role's *role interface*. The role interface specifies signatures of methods that must be defined in the active objects in order to play the role. The role's role interface is defined by sending the message defineInterface:<Method Signatures Collection> to the role with a collection of method signatures as argument.

In (**Figure 3.7** line 5), we show how we specify the role interface for the role Voter in the ElectronicVote group. The role interface of the role Voter is composed uniquely of the signature of the method opinion:. An active object who wants to play the role Voter must know in advance how to react to the opinion: method invocation. The method opinion: models the opinion of a participant with respect to any particular issue.

Actually in the CoLaS model only methods selectors are specified in the role interface. It would be possible to extend the specification of method signatures with returned values and arguments types without too much work. The idea will be to specify as much as possible the behavior required to participate in the role and thus to avoid possible behavior mismatches. In the example we expect for example that the value returned by the method opinion: be a boolean indicating a positive or negative opinion of a voter on a particular issue, a different result will generate an error in the coordination.

In CoLaS there is in not by default any limitation in the number of participants that can play a role nor in the number of roles that participants can play in a group. Nevertheless, it is possible to constrain for a role the number of participants that may play the role. To constrain the number of participants authorized to play a role the message maxNumParticipants:<Max Number> must be sent to a role with the maximum number of participants as argument. In the electronic vote example the number of participants that may play the role Voter is not limited, the problem statement specifies that the group of voters must be open and that new voters must be allowed to join the group at any time.

Analysis

The most important requirement in the specification of a coordination model and language for concurrent object-oriented systems is to guarantee the separation of the coordination and the computation aspect in the systems. In CoLaS the coordination is specified completely separate of the computation specification of the participants. The only constraint imposed on the participants for their participation in the groups is the respect of the role interface defined in the roles they will play. In the example, the only constraint imposed to the participants of the role Voter is to provide a method opinion:. We can see in the example that we do not constrain the participation of the active objects in the role Voter based on their types as several coordination model and languages do [Arap91a][Puti97a][Duca98a][Andr99a][Aldr03a]. Another requirement that the CoLaS model satisfies is the possibility to define multi-party coordination. In CoLaS each role specifies a type of participant in the coordination, it is possible to define as much roles as necessary. Even more it is possible to coordinate groups of participants where several participants play the same role at the same time. Some existing coordination models and languages either constrain the number of roles to only two as in Collaborations [Yell97a], or constrain the number of participants in each role to one as in Activities and Environments [Arap91a] Connectors-FLO, [Duca98a]. With the exception of CoLaS, no other coordination model and language allows the dynamic specification of new types of participants (i.e., roles) in the coordination.

3.2.3 Coordination State

The coordination state in a group is specified by declaring variables (state variables in the following). The coordination state specifies information needed to perform the coordination. It concerns information like: whether some action occurred or actually occurs in the system (i.e., historical information). the number of times some action occurred or actually occurs in the system (i.e., historical counters), etc.

```
7.ElectronicVote defineVariables: #(#numYes #numNot) initialValues: #(0 0).
8.ElectronicVote defineVariable: #voteInProgress initialValue: false.
9.ElectronicVote defineVariable: #votePeriodExpired initialValue: false.
10.
11.Voter defineParticipantVariable: #hasVoted initialValue: false.
```

Figure 3.8 : Electronic Vote - Coordination State

CoLaS specify three types of state variables:

- *Group Variables*: are state variables shared by all the participants of a group. Group variables are defined by sending the message defineVariable:<Variable Name> to a group with the name of the

variable as argument. In (**Figure 3.8** lines 7, 8 and 9), we can see how we specify the group variables: numYes, numNot, voteInProgress and votePeriodExpired in the ElectronicVote group. The variables numYes and numNot are used to count the number of positive and negative votes received by the initiator of the voting process. The variable voteInProgress is used to control that only one voting process occurs at a time in the group, when the value of the variable is true no other voting process can be started in the group. The variable votePeriodExpired is used to control the duration of the vote, the value of the variable remains set to false until the initiator of the voting process decides to stop the process. When the voting process is stopped the value of the variable votePeriodExpired is set to true.

- *Role Variables*: are state variables associated with the roles. They are shared by all the participants playing the role in which they are defined. Role variables can only be accessed and modified by the participants of the role where they are defined and only during the time they play the role. Role variables are defined by sending the message defineVariable:<Variable Name> to a role with the name of the variable as argument. In the electronic vote example we do not specify any role variable.
- *Participant Variables*: are state variables associated with the participants of the roles. Each participant playing a role in which a participant variable is defined has its own instance of the participant variable. Participant variables can only be accessed and modified by the participants in which the variables are defined and only during the time the participants play the role in the group. Participant variables are defined by sending the message defineParticipantVariable:<Variable Name> to a role with the name of the variable as argument.

  In (**Figure 3.8** line 11), we can see how we specify the participant variable hasVoted in the ElectronicVote group. The hasVoted variable is used to control that each voter votes at most once as specified in the problem statement. When a participant votes, the value of its hasVoted participant variable is set to true. Votes coming from participants where the hasVoted participant variable was already set to true are not taken into account in the counting of the vote result.

Accessing And Modifying State Variables

To refer and to modify the different state variables defined in a group or in a role, we use directly as accessors the name of the variables. In (**Figure 3.9**) we can see how to access and to modify group and role variables. For the group variable we use the pseudo-variable group to access to the variable and for the role variable the pseudo-variable role. It is also possible to refer to a role variable through the role.

```
group <Group Variable>          /* returns the value of a group variable
group <Group Variable>: <Value>/* sets the value of a group variable
role <Role-Variable>            /* returns the value of a role variable
role <Roles Variable>: <Value>  /* sets the value of a role variable
Role <Role-Variable>            /* returns the value of a role variable
                                /* through the role Role
```

Figure 3.9 Accessing and Modifying State Variables

Analysis

The coordination state in the CoLaS model allows the specification of coordination related with the state of the participants and with the history of the coordination. When a coordination model and language does not offer to programmers the possibility to define the state specifically related to the coordination in the coordination abstractions, programmers start to define the coordination state within the computation code of the participants violating the most important coordination requirement: the separation of the coordination and computation concerns. Most of the coordination models and languages allow the specification of the coordination state in their coordination abstractions [Helm90a][Mukh95a][Mins97a][Duca98a][Barr02a]. There are two aspects that are new in the specification of the coordination state in CoLaS: 1) the specification of different accessibility constraints to the different types of variables and 2) the possibility to dynamically define new variables in the coordination state. From our point of view both new aspects are important, the first because it allows programmers to define specifically who can access and modify the variable and the second because it makes possible the evolution of the coordination when the requirements change.

3.2.4 Coordination Rules

The coordination rules specify the different rules governing the coordination of a group. They specify cooperation actions between participants, synchronizations over the occurrence of actions in participants and proactions in participants. We define in CoLaS three types of coordination rules: Cooperation Rules, Reactive Rules and Proactive Rules.

3.2.4.1 Cooperation Rules

The Cooperation Rules are rules that define implications between participant actions. They specify which actions should be done by the participants of a role when they receive messages corresponding to the coordination behaviors specified in the rules. They have the form <Role> defineBehavior: <Message> as: [<Coordination Actions>]. The Cooperation Rules allow a clear separation of the coordination and computation aspects in a system, the specification of the defineBehavior rules contain coordination behavior that is added "dynamically" to the participants when they join the roles in the groups.

In the Electronic Vote example, the problem statement (subsection **3.2.1**) specifies coordination behavior specifically related to the vote process: the initiator of the voting process sends a vote request to all the voters, the voters return their votes to the initiator of the voting process, the votes are counted and the result sent to all the voters. Active objects that want to participate in the ElectronicVote group and play the role Voter do not need to "know" these coordination behaviors in advance, they will "learn" them when they will join the role Voter in the group. In (**Figure 3.10**) we can see the generic specification of a cooperation rule in CoLaS. The <Role> specifies the role to which the cooperation rule is associated, the <Message> specifies the signature of the behavior specified by the rule and the <Coordination Actions> corresponds to a block of coordination statements. We will explain below all these elements in detail.

```
Cooperation Rule      = <Role> defineBehavior: <Message> as:
                            [ <Coordination Actions> ]
```

Figure 3.10 Cooperation Rules BNF

In the ElectronicVote group we define four cooperation rules (**Figure 3.11** lines 14, 19, 22 and 28), they specify the vote process described in the problem statement.

```
14.[1] Voter defineBehavior: 'startVote:anIssue' as:
15.      [group voteInProgress: true.
16.      Voter voteOn: anIssue.
17.      group votePeriodExpired: false].
18.
19.[2] Voter defineBehavior: 'voteOn:anIssue' as:
20.      [self sender vote:(self opinion: anIssue)].
21.
22.[3] Voter defineBehavior: 'vote: aVote' as:
23.      [aVote
24.          ifTrue: [group numYes++]   /* vote is positive
25.          ifFalse: [group numNot++]. /* vote is negative
26.      self sender hasVoted: true ].
27.
28.[4] Voter defineBehavior: 'stopVote' as:
29.      [(group numYes = Voter size)   /* vote result policy
30.          ifTrue: [Voter voteResult: 'Yes']
31.          ifFalse: [Voter voteResult: 'No']].
```

Figure 3.11 : Electronic Vote - behavioral Rules

Rule 1 (**Figure 3.11** line 14): the vote process is initiated with a startVote message sent by a voter. The startVote: behavior specified in the rule defines that a message voteOn:<anIssue> must be sent to all the voters. The argument <anIssue> specify the issue of the vote. Before the voteOn: message is sent to all the voters the group variable voteInProgress is set to true to indicate that a vote process has been started. After the message is sent the group variable votePeriodExpired is set to false to indicate that the voting period is open.

Rule 2 (**Figure 3.11** line 19): the voteOn: behavior specified in the rule defines that each voter must send the message vote:<Vote> to the initiator of the voting process with the result of its vote <Vote> as argument. The vote sent by the voters depend on their personal opinions about the vote's issue. The method opinion:<anIssue> returns true or false depending of the opinion of the voter on the issue. It is important to remember that the method opinion: appears in the voters role interface. Participants must know in advance how to react to this method in order to play the role voters.

Rule 3 (**Figure 3.11** line 22): the vote behavior specified in the rule defines the counting of the received votes. When the vote received is positive we increment the counter of positive votes (i.e. the numYes group variable) otherwise we consider the vote as negative and we increment the counter of negative votes (i.e. the numNot group variable). Once the corresponding vote counter has been increased we set to true the value of the participant variable hasVoted for the participant who sent the vote. The participant variable hasVoted is used to control that voters vote at most once during the voting process.

Rule 4 (**Figure 3.11** line 28): the stopVote behavior specified in the rule defines the counting police used to calculate the result of the vote. In this case the policy used is consensus (i.e. the number of

positive votes should be equal to the number of voters to obtain a positive result). The result of the voting process is sent then to all the voters.

Coordination Actions

The <Coordination Actions> that appear in the specification of the cooperation rules include:

- Manipulations to the coordination state: actions that access or modify the value of the state variables. In (**Figure 3.11** lines 15, 17, 24, 25, 26 and 29) we can see how some state variables are accessed and modified in the ElectronicVote example. Access to state variables is done synchronously.
- Synchronous recursive method invocations: actions to send messages synchronously to the same participant who received the method invocation. As in Actalk [Brio89a] where active objects may send synchronous messages to themselves, CoLaS uses the pseudo-variable self to send synchronous recursive method invocations. In (**Figure 3.11** line 20) we can see how the method opinion: is called using the self pseudo-variable. The method is executed synchronously in the voter who receives the voteOn: message. Coordination rules are not enforced during the execution of synchronous recursive method invocations.
- Method invocations to other participants and to other roles: actions to send messages asynchronously to other participants or to other roles (**Figure 3.11** lines 16, 20, 26, 30 and 31). When messages are sent to roles the messages are multicasted to all the participants of the role. It is not possible to send messages to roles when a reply value is expected.
- Method invocations information extraction: actions to extract information related to the message received by the participant like: the selector, the arguments and the identity of the sender and the receiver of the message.
- Role operations: actions in roles (i.e,. detect:<Condition> -detects the first participant that validates some condition, select:<Condition> -select all the participants that validate some condition, etc.), actions to verify if participants play roles (i.e., includes:<Active Object>), do: <Actions>-perform some actions in each one of the participants, actions to determine the number of participants playing a role (i.e., size and numParticipants) and actions to obtain the unique participant playing the role when the role is played by a unique participant (i.e., unique).

Replies

If the cooperation rule specifies a reply this is indicated with the keyword ^. As we already mentioned before, every request for a method invocation in another participant generates implicitly a reply. When the reply value is not explicitly indicated in the cooperation rule we consider the result of the evaluation of the last action specified in the <Coordination Actions> in the cooperation rule to be the reply.

3.2.4.2 Reactive Rules

Reactive Rules are rules that depend for their application on the messages exchanged by the participants. The CoLaS model defines actually two types of Reactive Rules: Interception Rules and Synchronization Rules. Both types of reactive rules are evaluated at specific points during the processing of the method invocations received by the participants in the group. CoLaS defines four evaluation points:

- *atArrival:* when a method invocation is ready to be received by the participant.
- *atSelection:* when a method invocation is ready to be executed by the participant.

- *atSent:* when a method invocation is ready to be sent to another participant.
- *atEnd*: when a method invocation has finished to be executed by the participant.

Interception Rules

```
Interception Rule          = <Role> <Interception Operator> <Message> do:
                                [<Coordination State Actions>]
Message                    = <Method Signature>
Interception Operator      = interceptAtArrival | interceptAtSelection |
                             InterceptAtSent  | interceptAtEnd
```

Figure 3.12 : Interception Rules BNF

Interception Rules are rules that change the normal processing of the method invocations in the participants to perform actions that modify the coordination state. In (**Figure 3.12**) we can see how Interception rules are specified in the CoLaS model. We define four types of interception rules: *interceptAtArrival*, *interceptAtSelection*, *InterceptAtSent* and *interceptAtEnd*. Each interception rule as indicated by its name corresponds to one of the evaluation points defined in the model.

```
31.[5] Voter intercepAtSelection: 'stopVote' do:
32.      [group votePeriodExpired: true]
33.
34.[6] Voter interceptAtEnd: 'stopVote' do:
35.      [Voter do:[:each | each hasVoted: false].
36.      group voteInProgress: false.
37.      group numYes: 0.
38.      group numNot: 0].
```

Figure 3.13 : Electronic Vote - Interception Rules

In (**Figure 3.13** lines 31 and 34) we can see the specification of two interception rules defined in the electronic vote example. Both rules are related to the stopVote behavior but they differ in the interception point in which they are evaluated.

Rule 5 (**Figure 3.13** line 31): The rule specifies that the voting period is closed before the counting process is done. To indicate the end of the voting period we set to true the value of the group variable votePeriodExpired (line 32). The votePeriodExpired group variable is used to control that only votes arrived during the voting period are counted.

Rule 6 (**Figure 3.13** line 34): The rule prepares the state variables of the group for a new voting process after the execution of the stopVote behavior. In (**Figure 3.13** line 35) the participant variable hasVoted is reinitialized to false in each voter and in (**Figure 3.13** lines 36, 37 and 38) all the group variables are reinitialized: the voteInProgress is set to false to indicate that no voting process is actually occurring, the numYes and numNot variables are reset to zero the initialize the counting of votes.

Coordination State Actions

The <Coordination State Actions> include exclusively operations that modify the state variables. As we already mentioned before the state variables can be accessed and modified using the variables names. In (**Figure 3.13** lines 32, 35, 36, 37 and 38) we can see how the set of group variables defined in the Electronic Vote example are modified.

Synchronization Rules

Synchronization Rules specify synchronization constraints in the execution of the method invocations received by the participants. The CoLaS model defines two forms of Synchronization Rules (**Figure 3.14**): *Ignore* and *Disable.*

```
Synchronization Rule       = <Role> <Synchronization Operator> <Message> if:
                                [<Synchronization Condition>]
Message                    = <Method Signature>
Synchronization Operator   = disable | ignore
```

Figure 3.14 : Synchronization Rules BNF

The Ignore rule specifies that method invocations corresponding to the message <Message> must be ignored when received (i.e., not stored into the participant's mailbox) if the condition specified in the <Synchronization Condition> validate to true. Ignore rules are evaluated at the atArrival validation point in the CoLaS model.

The Disable rule specifies that the execution of the method invocations corresponding to the message <Message> must be delayed (i.e., reinserted in the participant's mailbox) if the condition specified in the <Synchronization Condition> validates to true. It is important to remember that the selection for execution of a method invocation stored in the participant's mailbox depends exclusively of the internal synchronization policy defined in the participant. Our active objects select method invocation on the basis of first come first executed. Disable rules are evaluated at the atSelection validation point in the CoLaS model after that the method invocation associated with the rule has been selected and validated against the synchronization policy of the object. It is important to remark that the specification of a different synchronization policy in the objects may imply that the policy takes into account the coordination behaviors specified in the groups. We will return later during the evaluation of the CoLaS model on this point, possible violations to the separation of the coordination and the computation in the participants may appear. It is also important to remark that we have not defined in CoLaS multi-party coordination rules, rules that depend for their applicability on multiple invocation requests occurring in different participants. Multi-party coordination rules is a future work that we consider important in the CoLaS coordination model. Multi-party coordination rules will allow for example the specification of mutual exclusions of actions occurring in several participants.

In the electronic vote example we define two synchronization rules (**Figure 3.15**), they constrain the execution of the vote and startVote coordination behaviors in the Electronic Vote example.

```
40.[7] Voter ignore: 'vote:aVote' if:
41.      [group votePeriodExpired or: [sender hasVoted]].
42.
43.[8] Voter disable: 'startVote:anIssue' if:
44.      [group voteInProgress ].
```

Figure 3.15 : Electronic Vote - Synchronization rules

Rule 7 (**Figure 3.15** line 40): defines that votes received after the end of the period of vote or votes received from voters that have already voted must be ignored. The system guarantees that the vote is fair: voters vote at most once and only within the voting period defined by the initiator of the vote. The <Synchronization Condition> associated with the rule combines the values of the votePeriodExpired group variable and the hasVoted participant variable.

Rule 8 (**Figure 3.15** line 43): defines that requests for starting new vote processes are disabled if there is actually one voting process occurring in the system. The <Synchronization Condition> associated with the rule uses the value of the group variable voteInProgress to determine whether there is currently a voting process in progress. The group variable voteInProgress is set to true each time a new vote process starts (rule 1, in **Figure 3.11** line 14) and set to false each time the vote process is stopped (rule 6, in **Figure 3.14** line 34).

Synchronization Condition

The <Synchronization Condition> corresponds to a boolean expression (i.e. and, or) referring to:

- Method invocations information: the selector, the arguments, the identity of the sender and the receiver of the method invocation received by the participant. This information is accessed using the predefined variables: selector, arguments, sender and receiver (**Figure 3.15** line 41)
- The coordination state: the values of the state variables (**Figure 3.15** lines 41and 44).
- The keyword true: always true. The true keyword is used to specify rules that always apply.
- The keyword now: the current value of the time is obtained using the keyword now. It is possible to specify time conditions.
- Role operations: actions in roles (i.e. detect:<Condition> -detects the first participant that validates some condition, actions to verify if participants play roles (i.e. includes:<Active Object>), actions to determine the number of participants playing a role (i.e. size) and actions to obtain the unique participant playing the role when the role is played by a unique participant (i.e., unique)

3.2.4.3 Proactive Coordination Rules

Until now the coordination specified in the ElectronicVote group has been purely reactive, the coordination rules specify actions that must be done during the processing of the method invocations received by participants playing the role Voter. Those actions are not be initiated by the participants themselves, they depend for their application on the messages received and exchanged by the participants. To define a richer coordination model we have introduced in CoLaS proactive behavior [Andr96a] in the form of proactive rules. Proactive rules are rules that depend for their application exclusively in the coordination state of the group

and not in the method invocations received by the participants. In (**Figure 3.16**) we can see the specification of the unique proactive rule in the CoLaS model.

```
Proactive Rule            = <Group> validate: <Coordination State Condition>
                                do: [<Coordination Actions>]
```

Figure 3.16 : Proactive Rules BNF

Proactive rules guarantee that certain actions are carried out by the group if a certain condition concerning mainly the coordination state validates to true. In (**Figure 3.17**) we illustrate how the specification of the ElectronicVote group presented in (**Figure 3.11**, **Figure 3.13** and **Figure 3.15**) was modified to introduce proactive rules. We have redefined the rule 4, eliminated the rules 5 and 6 and added a new rule Rule 9 specifying the proaction.

```
1.[1] Voter defineBehavior: 'startVote:anIssue' as:
2.       [group voteInProgress:true.
3.       Voter voteOn: anIssue.
4.       group VotePeriodExpired: false].
5.
6.[2] Voter defineBehavior: 'voteOn:anIssue' as:
7.       [sender vote:(self opinion: anIssue)].
8.
9.[3] Voter defineBehavior: 'vote:aVote' as:
10.      [aVote
11.          ifTrue: [group numYes++]   /* vote is positive
12.          ifFalse: [group numNot++]. /* vote is negative
13.      sender hasVoted: true ].
14.
15.[4] Voter defineBehavior: 'stopVote' as:
16.      [group votePeriodExpired: true ].
17.
18.[7] Voter ignore: 'vote:aVote' if:
19.      [group voterPeriodExpired or: [sender hasVoted]].
20.
21.[8] Voter disable: 'startVote:anIssue' if:
22.      [group voteInProgress ].
23.
24.[9] ElectronicVote
25.      validate: [group voteInProgress and:[group votePeriodExpired]] do:
26.      [(group numYes = Voter size)   /* vote result policy
27.          ifTrue: [Voter voteResult: 'Yes']
28.          ifFalse: [Voter voteResult: 'No'].
29.      Voter do:[:each | each hasVoted: false].
30.      group voteInProgress: false.
31.      group numYes: 0 ].
32.      group numNot: 0].
```

Figure 3.17 : Electronic Vote - Proactive behavior

Rule 4 (**Figure 3.17** line 15): the rule specifies that when the initiator of the vote process decides to stop the voting period the group variable votePeriodExpired is set to true. The votePeriodExpired group variable is used to control that voters vote only once during the voting period.
Rule 9 (**Figure 3.17** line 24): the rule specifies a proaction with a condition based on the values of the group variables voteInProgress and votePeriodExpired. The group variable voteInProgress determines whether a voting process is occurring in the system and the group variable votePeriodExpired determines whether the voting period has expired. When both conditions in the rule are true, the <Coordinations Actions> actions of the rule are executed. The <Coordination Actions> actions in the rule (**Figure 3.17** lines 26 to 32) specify the counting process of the vote result and the sent of the result to all the voters. The policy applied to calculate the result of the vote is consensus: the number of positive votes must be equal to the number of voters. The <Coordination Actions> in the example, include the reinitialization of the group variables to prepare the group for a new voting process. The voteInProgress and the votePeriodExpired state variables are set to false to indicate that no vote process occurs actually in the group and that a new voting period can start, the numYes and numNot variables are set to zero the reinitialize the counted votes.

The <Coordination Actions> shown in the specification of the proactive rules correspond to the same coordination actions specified in Cooperation Rules and the <Coordination State Condition> corresponds to the same coordination state condition specified in the Synchronization Rules (excluding of course conditions concerning information about the received method invocations which in proactive rules do not have any sense given that they do not depend for their application of the reception of method invocations in the participants).

Proactive Rules Enforcement

The evaluation of the proactive rules is done in a indeterministic way by the coordination groups. The group evaluates the <Coordination State Condition> conditions associated with all the registered proaction rules, if the evaluation of the conditions evaluate to true the group forces the execution of the <Coordination Actions> specified in the proactive rules. It is not possible to precisely know when the proactive rules will be evaluated by the coordination groups.

3.2.4.4 Pseudo-Variables

There are four pseudo-variables that can be used within the specification of the coordination rules, they are: group, role, sender and receiver. The group pseudo-variable refers to the group in which the rule is defined, the role pseudo-variable refers to the role to which the rule is associated, the sender pseudo-variable refers to the participant who sent the method invocation associated with the enforced rule and the receiver pseudo-variable to the participant actually processing the method invocation associated with the enforced rule. In (**Figure 3.11** lines 15, 17, 24, 25 and 29; **Figure 3.13** lines 32, 36,37 and 38; and **Figure 3.15** lines 41 and 44) we find references to the group pseudo-variable and in (**Figure 3.11** lines 20 and 26 , **Figure 3.15** line 41) we find references to the sender pseudo-variable. In the ElectronicVote group we do not have references to the pseudo-variables role and receiver.

Analysis

Again the specification of the coordination rules in CoLaS respect the most important requirement in the specification of a coordination model and language for concurrent object-oriented systems which is the sep-

aration of the coordination and computation aspects in the systems. In CoLaS the coordination specified in the coordination rules is independent of the computation specification in the participants. The rules are associated to roles and not to specific participants. They specify cooperation actions between participants (Cooperation rules), synchronizations over the occurrence of actions in participants (Reactive rules) and proactions in the participants (Proactive rules). Cooperation rules define implications between participant actions. Reactive rules depend for their application on the messages exchanged by the participants. And, Proactive rules depend for their application exclusively in the coordination state of the group. The specification of the coordination in CoLaS in the form of rules allows programmers the specification of the coordination policies in a declarative way, as specified in the requirements of an ideal coordination model and language for concurrent object-oriented systems. The coordination rules are high level coordination abstractions encapsulating the coordination, programmers do not care about how the coordination specified in the rules is enforced, they focus exclusively on specifying the type of coordination they want. Additionally the specification of coordination rules in CoLaS allows programmers to specify their own coordination policies. Several existing coordination models have recognized the importance of using rules in the specification of the coordination [Frol93a][Berg94a][Andr96a][Mins97a][Duca98a][Berr98a], most of them include some form of cooperation rules [Frol93a][Duca98a][Mins97a], other some form of synchronization rules [Frol93a], but only CoLaS and Rules and Constraints [Andr96a] include proactive rules.

3.2.5 Dynamic Aspects

One of the most important characteristics of the CoLaS model is its capacity to dynamically adapt the coordination specified in the groups. The CoLaS model support three types of dynamic coordination changes: (1) new participants can join and leave the groups at any time, (2) new groups can be created and destroyed dynamically and (3) new coordination rules can be added and removed from the groups.

Joining and Leaving Groups

New participants can join and leave the groups at any time. To join a group, an active object must join one of the roles specified in the group. To join a role the message addParticipant:<Active Object> toRoleNamed:<Role Name> must be sent to a group. The <Active Object> argument refers to the active object that wants to join the group and the argument <Role Name> to the name of the role it wants to play. It is also possible to join a role by directly sending the message addParticipant: <Active Object> to the role.

In (**Figure 3.18** line 1) we can see how a Person active object is created. We assume the existence of a class Person used to create persons that will play the role Voter in the group. In (**Figure 3.18** line 8) we can see how the person object 'Andrew Peterson' joins the role Voter in the AdminVote group instance. It is important to remember that only active objects satisfying the role interface of the role Voter may play the role. In this case we assume that the class Person defines a method called opinion: <anIssue> which returns true or false according to his personal opinion on the issue anIssue received as argument.

We show additionally in (**Figure 3.18** line 10) how the same participant is removed later from the same role Voter. To remove a participant from a role the message removeParticipant: <Active Object> fromRole:

<Role Name> must be sent to the group. The arguments in the remove of a participant operation correspond to the same type of arguments specified in the addition of a participant operation presented before.

```
1.andrewPeterson := Person
2.                    firstName: 'Andrew'
3.                    familyName: 'Peterson'
4.                    eMailAddress: 'andrew.peterson@iam.unibe.ch'.
5.
6.adminVote := ElectronicVote createCoordinationGroupNamed: #AdminVote.
7.
8.adminVote addParticipant: andrewPeterson toRoleNamed: #Voter.
9.       ....
10.adminVote removeParticipant: andrewPeterson fromRoleNamed: #Voter
```

Figure 3.18 : Dynamic addition and removal of Participants

Dynamic Creation of Groups

Groups can be created at any time, in (**Figure 3.19** lines 7 and 8) we can see how two new groups Students and Citizens are created. Both groups are instances of the ElectronicVote group. In (**Figure 3.19** lines 10 and 11) we can see how the same Person 'Ralph Stevenson' joins both groups.

```
1.ralphStevenson := Person
2.                    firstName: 'Ralph'
3.                    familyName: 'Stevenson'
4.                    id: 2002013467
5.                    eMailAddress: 'ralph.stevenson@cs.unibe.ch'.
6.
7.students := ElectronicVote createCoordinationGroupNamed: #Students.
8.citizens := ElectronicVote createCoordinationGroupNamed: #Citizens.
9.
10.students addParticipant: ralphStevenson toRoleNamed: #Voter.
11.citizens addParticipant: ralphStevenson toRoleNamed: #Voter.
```

Figure 3.19 : Dynamic creation of Groups

Modification of the Coordination behavior

The coordination behavior of the group can be modified by adding, redefining and removing coordination rules. In the ElectronicVote group specified until now (**Figure 3.17**) we do not control the identity of the voter who decides to stop the vote process nor we do control that only the participants registered in the group are the only ones who vote. We will modify the group specification to manage these problems.

```
1.ElectronicVote defineVariable: #voteInitiator.
2.
3.[10] Voter interceptAtSelection 'startVote:anIssue'as:
4.       [group voteInitiator: sender].
5.
6.[11] Voter ignore:'stopVote' if:
7.       [sender ~= group voteInitiator]. /* ~= means different
8.
9.[7] Voter ignore: 'voteOn:aVote' if:
10.      [(Voters includes: sender)or:
11.       [group voterPeriodExpired or:
12.       [sender hasVoted]]].
```

Figure 3.20 : Dynamic modification of the Coordination behavior

To solve the first problem, we define a new group variable called voteInitiator in the group ElectronicVote (**Figure 3.20** line 1) to keep the reference to the initiator of the vote process. We also define a new InterceptAtSelection interception rule associated with the startVote: behavior (**Figure 3.20** line 3) to save the reference to the initiator of the voting process in the voteInitiator group variable. Additionally, we add an Ignore reactive rule (**Figure 3.20** line 6) to discard stopVote: messages received from participants different to the initiator of the voting process.

To solve the second problem we redefine the rule 7 in (**Figure 3.17** line 18) to include in the <Synchronization Condition> condition an extra condition (i.e.,Voter includes: sender) which validates whether the sender of a voteOn: message plays the role Voter in the group (**Figure 3.20** line 9).

Analysis

The capacity of the CoLaS coordination model to dynamically adapt the coordination specified in the groups makes it particularly interesting for the specification and construction of modern concurrent object-oriented systems. In those systems evolution is the most difficult requirement to meet since not all the application requirements can be known in advance. It is extremely important that the coordination model supports the modifications and thus the evolution of the coordination in those systems. The CoLaS model support three types of dynamic coordination changes: (1) new participants can join and leave the groups at any time, (2) new groups can be created and destroyed dynamically and (3) new coordination rules can be added and removed from the groups. We showed in this section how the CoLaS coordination model supports the requirement related to the support of the evolution of the coordination. We showed how new users did join groups and played roles, how the coordination specified in the groups was modified by adding new coordination rules and how new groups were created dynamically to enforced new coordination patterns. No other existing coordination model and language in our survey of existing coordination models and languages support the dynamic evolution of the coordination.

3.2.6 Groups Composition - The Electronic Agenda

Solutions to complex coordination patterns must be defined as the composition of small coordination solutions. A model that ignores the need for composability will not be sufficiently scalable to deal with real

problems [Kafu96a]. We will illustrate how in the CoLaS model existing groups specifications can be used in the specification of new groups. We will use as example a simplified version of the electronic agenda problem [Bosc97a].

Problem Description

The electronic agenda assists in the management of meetings in a conference room. Several users may view and modify the contents of the agenda simultaneously while preventing conflicts (i.e., planning of overlapping events). The electronic agenda supports the following operations: consult the events programmed for a day, add a new event to the agenda and to cancel an event from the agenda. An event is composed of: the day in which the event happens, the beginning and ending time at which the event starts and finishes and a comment line describing the event. The only constraint imposed on the system is that the modifications to the agenda must be accepted by all the members of the group.

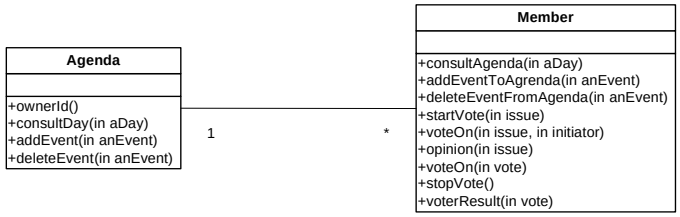


Figure 3.21 : The Electronic Agenda - UML Class Diagram

In (**Figure 3.21**) we can see the UML class diagram corresponding to the electronic agenda problem. We have identified two kinds of entities in the problem description: the agenda and the members of the group (only members in the following). We can see in the specification of the Agenda class the three basic possible operations that can be executed in the agenda: to consult the events occurring some specific day (i.e., consultDay:), to add an event to the agenda (i.e., addEvent:) and to delete an event from the agenda (i.e., deleteEvent:). The Member class on the other hand represents the different users of the agenda.

We propose a decentralized solution to the electronic agenda in which each member of the group maintains a copy of the agenda. Each time that the agenda is modified we modify all the copies of the agenda maintained by the members. A decentralized solution as we propose increases the tolerance to faults of the system because we do not have a single point of failure, if one of the members of the group leaves or has a problem the system will continue working without any problem. A decentralized approach supposes that each time that a new member joins the group he or she receives a copy of the agenda from one of the members of the group. We will not present in the specification of the group the aspects related to joining of new participants to the group we will focus exclusively on the specification of the operations on the agenda.

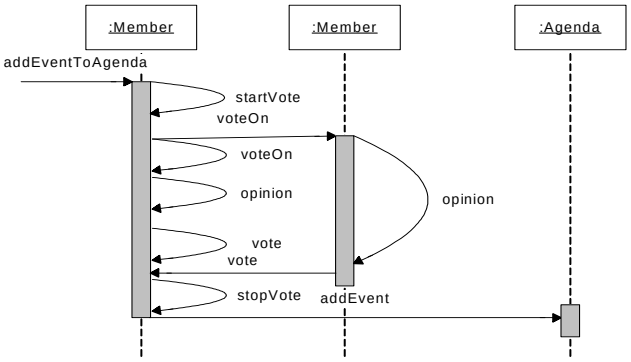


Figure 3.22 : The Electronic Vote - UML Interaction Diagram

In (**Figure 3.22**) we can see the UML interaction diagram of operation addEventToAgenda:. The execution of the addEventToAgenda operation is preceded by the vote of all the members on whether the modification must be applied to the agenda. Only if the result of the voting process is positive the event is added to the agenda. We will not show the UML interaction diagrams corresponding to the deleteEventFromAgenda: and consultAgenda: operations, they are very similar to the addEventToAgenda: UML's diagram. The important now it is not show a complete specification of the problem but to show how the composition facilities in CoLaS coordination model can be used in the definition of new coordination groups.

3.2.6.1 Coordination Roles

```
1.CoordinationGroup createCoordinationGroupClassNamed: #ElectronicAgenda.
2.
3.ElectronicAgenda defineRole: #Member.
```

Figure 3.23 : Electronic Agenda - Roles Specification

In (**Figure 3.23** line 1) we case see how a new group ElectronicAgenda is created. The ElectronicAgenda group specifies a unique role: Member (**Figure 3.23** line 3) representing the users of the agenda.

3.2.6.2 Coordination State

The ElectronicAgenda defines a group variable named issue (**Figure 3.24** line 4), this variable is used to store the issue of each voting process. The issue includes the type of modification to be done to the agenda (i.e., to add or to delete an event) and the specific information related to the event. The second variable defined in the group specifies a participant variable named agenda (**Figure 3.24** line 6). Each member of the group keeps an instance of this variable. The participant variable agenda is initialized with an instance of an

Agenda class. We assume the existence of an Agenda class containing all the methods specified in the Agenda class in (**Figure 3.21**). The Agenda class specifies the computation aspect of the electronic agenda.

```
4.ElectronicAgenda defineVariable: #issue.
5.
6.Member defineParticipantVariable: #agenda initialValue: Agenda new.
```

Figure 3.24 : Electronic Agenda - Coordination State

3.2.6.3 Reusing Existing Coordination Groups

```
7.ElectronicAgenda includeCoordinationGroupSpecification: ElectronicVote
8.                 mappingRoles: (List with: (#Voter -> #Member)).
```

Figure 3.25 : Electronic Agenda - Including Specification

The electronic agenda problem statement requires the positive vote of all the members of the group before a modification to the agenda to be done. The vote process corresponds to the behavior specified in the ElectronicVote group (subsection **3.2.1** through **3.2.5**). In (**Figure 3.25** line 7) we can see how the specification of the ElectronicVote group is used in the specification of the ElectronicAgenda group. When the specification of an existing group is included (i.e., includeCoordinationGroupSpecification: mappingRoles:) in a new group all the state variables and coordination rules specified in the existing group are included into the specification of the new group.

In the specification of the new group it is possible to define a mapping between roles names in the existing group and roles names in the new group. In (**Figure 3.25** line 8) we can see how the role Voter defined in the ElectronicVote group is mapped to the role Member specified in the ElectronicAgenda group. All references to the role Voter in the coordination rules included into the ElectronicAgenda group are replaced by references to the role Member.

It is also possible to define during the composition of coordination groups mappings for message signatures specified in the behavioral rules. The mapping of message signatures is specified by sending the message includeCoordinationGroupSpecification:<Coordination Group> mappingSignatures: <Signatures Mapping List> to the new group. The list of signatures mapping <Signatures Mapping List> contains a list of the form (old-method-signature-> new-method-signature). All the references to old method signatures (i.e., old-method-signature) are replaced by references to new method signatures (i.e., new-method-signature) in the new group.

Analysis

The electronic agenda example illustrates how the coordination aspect of the Electronic Agenda problem is specified completely separate from the specification of the computation code of the coordinated objects. The coordination specified in the coordination group refers exclusively to the coordinated objects by the role they play in the group. In the example the coordination refers to the unique role Member. Different types of participants may play the role Member in the group. The only constraint imposed on the participation of the active objects to the group is the respect of the role interface defined in the roles they will play. In

the example, the only condition imposed to the participants of the role Member corresponds to the same condition imposed to the role Voter inherited from the inclusion of the ElectronicVote coordination group in the ElectronicAgenda group. In the ElectronicVote coordination group participants playing the role Voter must define the behavior opinion: which models the opinion of a voter regarding a particular issue.

The electronic agenda example illustrates also how the CoLaS coordination model allows the specification of new coordination policies from existing coordination policies. The ElectronicVote coordination group in the example is used in the specification of a new coordination group ElectronicAgenda. All the rules specified originally in the ElectronicVote coordination group are included in the specification of the ElectronicAgenda group and the role names mapped to the role names specified in the new coordination group.

Few coordination models and languages offer the possibility to define new coordination policies from existing coordination policies. Most of them use inheritance as a mechanism to refine the specification of the coordination policies [Helm90a][Kris97a][Duca98a]. CoLaS proposes a simple inclusion mechanism to reuse the specification of the coordination specified in existing coordination groups. The inclusion mechanism allows one to inherit the coordination state and the coordination rules specified in the original coordination group. It is possible afterwards to modify the coordination rules and to add new policies if needed. A coordination model that do not support the incremental specification of the coordination policies limits the scalability of the coordination specifications.

3.2.6.4 Coordination Rules

In (**Figure 3.26**) the rules 1, 2 and 3 define specific coordination behavior for the electronic agenda problem, the three rules specify the basic operations in the electronic agenda: to consult the agenda, to add a new event to the agenda and to delete an event from the agenda. The rules 4, 5, 6 complete the coordination rules included from the specification of the electronic vote group (**Figure 3.25**).

Rule 1 (**Figure 3.26** line 9): the rule defines that when a member decides to consult the electronic agenda for a particular day, the member's agenda copy is selected and all the events scheduled for that day are returned.

Rule 2 (**Figure 3.26** line 12): the rule defines that when a member decides to add an event to the electronic agenda, a vote process is started in that issue (i.e. the addition of the new event to the agenda). The initiator of the voting process decides when to stop the voting period. The specification of the voting process in the ElectronicAgenda group is done by the coordination rules specified in the ElectronicVote group (**Figure 3.17**) and included in (**Figure 3.25** line 7). The coordination rules corresponding to the ElectronicVote do not appear explicitly in the example but they make also part of the specification of the ElectronicAgenda group.

Rule 3 (**Figure 3.26** line 19): the rule defines that when a member decides to delete an event from the agenda, a vote process is started in that issue (i.e. the deletion of an event from the agenda). The initiator of the voting process decides when to stop the voting period. The specification of the voting process in the electronic agenda is done as described in the rule 2.

Rule 4 (**Figure 3.26** line 25): the rule defines that when a vote process is started we store in the issue group variable the issue of the vote. For the addition of a new event to the agenda the subject of the issue is #addEvent and for the deletion of an event from the agenda #deleteEvent. For each event we keep additionally to information related to the event received as argument: the subject, the day and time of the event and a small comment about its purpose.

Rule 5 (**Figure 3.26** line 28): the rule defines that when the members agree on the modification of the agenda, the modification should be applied to the agendas of all the members (i.e., the contents of all the copies must be synchronized). The value of the group variable issue determines the modification that must be done to the agendas.

Rule 6 (**Figure 3.26** line 37): the rule defines that when the result of the vote is negative (i.e., members do not agree with the modification of the agenda) the actions specified in the voteResult behavior are executed.

```
9.[1] Member defineBehavior: 'consultAgenda:aDay' as:
10.      [^self agenda consult:aDay ].
11.
12.[2] Member defineBehavior: 'addEventToAgenda:anEvent' as:
13.      [| anIssue |
14.      anIssue := Issue subject:#addEvent args: anEvent.
15.      self startVote: anIssue.
16.      Delay forSeconds: group MaxVotePeriod. /* wait some time
17.      self stopVote]/* stops voting process
18.
19.[3] Member defineBehavior: 'deleteEventFromAgenda:anEvent' as:
20.      [anIssue := Issue subject:#deleteEvent args: anEvent.
21.      self startVote: anIssue.
22.      Delay forSeconds: MaxVotePeriod.
23.      self stopVote] /* stops voting process
24.
25.[4] Member interceptAtSelection: 'startVote:anIssue' do:
26.      [group issue: anIssue ].
27.
28.[5] Member defineBehavior: 'voteResult:aVote'as:
29.      [event := group issue args.
30.      (aVote
31.          ifTrue:
32.              [(group issue subject = #addEvent)
33.                  ifTrue: [ self agenda addEvent: event ]
34.              (group issue subject = #deleteEvent)
35.                  ifTrue: [ self agenda deleteEvent: event]]].
36.
37.[6] Member ignore: 'voteResult:aVoteResult' if:
38.      [aVoteResult not ].
```

Figure 3.26 : Electronic Agenda - Coordination Rules

3.2.7 Groups as Participants

In CoLaS groups can also play the role of participants in other groups. As a participant a group is able to receive and process method invocation requests received from other participants. To transform a group in a participant it is necessary to specify for the group a facade. A *Group Facade* defines an actions mapping list <Actions Mapping List> between method signatures and coordination behaviors specified in the group.

```
<Coordination Group> defineGroupFacade: <Mappings Actions List>

<Actions Mappings List> := List(<method-signature> -> <actions-list>)
```

Figure 3.27 Group Facade Specification

Each method signature <method-signature> that appears in the <Actions Mapping List> have associated an actions list <actions-list> that specifies what to do with the received message. The simplest action that can be specified in the actions list <actions-list> is to forward the received message to a role. The message sent to the role is then multicasted to all its participants. It is also possible to redefine the received message before sending the message to the role. The only constraint imposed to the actions specified in the actions list <actions-list> is that the group must include cooperation rules associated with the method selectors specified in the actions list.

In the Electronic Agenda example only the members of the group can do operations in the agenda. It will be interesting for example to let other active objects not participating in the electronic agenda to consult the agenda for simple information purposes. Suppose for example that there are people in charge of cleaning the rooms in which the meeting events are programmed, they are interested in consulting the use of the rooms in order to plan their work. A natural way to consult the agenda without giving the possibility to modify it (i.e., without being a member) is to define a group facade with a unique operation consultAgendaForCleaning:. In (**Figure 3.28** line 1) we can we see how to specify a group facade for the ElectronicAgenda coordination group. The group facade specifies that whenever the ElectronicAgenda group receives a message consultAgendaForCleaning:<aDay> the same message is forwarded to one of the members to return the agenda plan for the day aDay. The result of the execution of the consultAgendaForCleaning: method is returned to the active object that requested to consult the agenda for cleaning purposes.

```
1.ElectronicAgenda defineGroupFacade:
2.           List with:
3.                ( 'consultAgendaForCleaning:aDay'
4.                ->
5.                [| aMember |
6.                 aMember := Members selectAParticipant. /* randomly
7.                ^(aMember consultAgenda: aDay)result ]).
```

Figure 3.28 : Electronic Agenda - Group Interface

Analysis

The possibility to transform coordination groups in participants in other coordination groups allows programmers to incrementally define new coordination policies. This requirement that appears in our list of requirements for an ideal coordination model and language for object systems, facilitates the scalability of the coordination specifications. In coordination models and languages that do not support the incremental specification of the coordination, the specification of complex coordination policies becomes easily a big problem. The advantages of the separation of computation and coordination concerns loses with the increase in

the complexity of the specification of the coordination. Few existing coordination models and languages support the incremental specification of the coordination, we have Contracts [Helm90a] and Connectors - FLO [Duca98a].

3.3 Evaluation of the CoLaS model

In chapter 2 of this thesis we identified a series of requirements we believe characterize an ideal object-oriented coordination model and language for active objects. We will evaluate the CoLaS coordination model and language with respect to these requirements. They are:

- Clear separation of the computation and the coordination concerns: in CoLaS the coordination and computation aspects are specified separately in two distinct entities: the coordination groups and the participants. The coordination groups are specified independently of the participants they coordinate and the participants are specified independently of the coordination groups which coordinate them.
- Encapsulation of the coordination behavior: in CoLaS the coordination of a group of collaborating participants is encapsulated inside coordination groups. The specification of a coordination group includes: the role specification, the coordination state and the coordination rules.
- Support multi-object coordination: in CoLaS the coordination specified in the coordination groups is not limited to two participants but to group of participants. The coordination groups specifies abstractly the coordination of groups of participants in terms of the roles they play in the coordination and their respective interfaces. The role abstraction allows the specification of the coordination independently of the effective number of participants participating in a group, we talk in this case of a coordination specified intentionally and not extensionally.
- High-level coordination abstractions: in CoLaS programmers do not focus on how to perform the coordination but on how to express it. All the low-level details concerning how the coordination is done are managed internally by CoLaS. For example programmers do not care about locking and unlocking state variable to guarantee their consistency during the coordination. The coordination groups internally serialize the access to the state variables.
- Support evolution of the coordination: in CoLaS the coordination behavior is not fixed. It can change over the time. CoLaS support dynamic coordination changes in three distinct axes in coordination groups: (1) new participants can join and leave the coordination groups at any time, (2) new coordination groups can be created and destroyed dynamically and (3) coordination rules can be added to and removed from the coordination groups.
- Promote the reuse of coordinations abstractions: in CoLaS the coordination groups are specified independently of the participants they coordinate. They can be used to coordinate different groups of participants. Similarly, the participants can be reused in different coordination groups. The minimum requirements imposed to participants to play the roles are specified in the roles interfaces.
- Declarative specification of the coordination: in CoLaS the coordination is specified in a declarative way using rules. The Coordination rules specify: cooperation actions between participants, synchronizations over the occurrence of actions occurring in participants and proactions in participants. The advantage of using rules in the specification of the coordination is that the coordination becomes explicit.
- Incremental specification of the coordination: in CoLaS existing coordination groups specifications can be composed to specify new coordination groups. Complex coordination schemes can be built from simpler coordination specifications.

- Support validation of formal properties: in CoLaS we use Petri Nets to formally validate properties of the coordination layer. In chapter 7 of this thesis we present a methodology to transform CoLaS coordination groups in Predicate-Action Petri Nets. Reachability analysis techniques are then used to validate formal properties.

3.4 Conclusions and Contributions

We propose in this thesis to tackle the complexity of the specification and construction of concurrent object-oriented systems using a coordination models and languages approach. Coordination models and languages promote the separation of the computation and the coordination aspect in the systems. The computation model concerns the specification of the active objects that compose the concurrent object-oriented systems and the coordination model the specification of the glue that binds all them together. Our thesis is that by separating the specification of the coordination aspect from the computation aspect in concurrent object-oriented systems and by the specification of the computation in active objects we simplify the specification, understanding, construction, evolution and validation of properties in this kinds of systems.

We presented in this chapter CoLaS a coordination model and language to perform coordination in concurrent object-oriented systems based on active objects. The CoLaS coordination model and language is based on the notion of coordination groups, entities that control and enforce the coordination of groups of collaborating concurrent objects. A coordination group is a high-level coordination abstraction that specifies, encapsulates and enforces the coordination of a group of collaborating participants. Coordination groups support the dynamic evolution of the coordination requirements in concurrent object-oriented systems.

The CoLaS coordination model tackles the most important problems that existing concurrent object-oriented programming languages have in supporting the specification of the coordination aspect in those systems: 1) lack of high level coordination abstractions, 2) lack of coordination abstractions for complex interactions, 3) lack of separation of computation and coordination concerns, 4) lack of support for the evolution of the coordination code and 5) lack of support for the validation of the coordination code.

The approach used in CoLaS to perform the coordination is the reflective approach. In the reflective approach messages exchanged by the participants in the coordination groups are intercepted at different points during their evaluation and execution to perform coordination actions specified in form or coordination rules. In the CoLaS model the coordination is done on active objects, called participants in the model. Active objects are objects that have control over concurrent method invocations and which communicate asynchronously.

The Coordination Groups are composed of three elements: the Roles Specification, the Coordination State and the Coordination Rules. The Roles Specification defines the different roles that participants may play in the group. The Coordination State defines general information needed to perform the coordination and the Coordination Rules defines the different rules governing the coordination of the group.

The CoLaS coordination model and language fully satisfies the requirements identified as ideal for a coordination model and language introduce in Chapter 2 of this thesis. They are:

- Clear separation of the computation and the coordination concerns: the coordination is encapsulated in the coordination groups and the computation in the participants.

- Encapsulation of the coordination behavior: all the coordination behavior is specified inside the coordination groups. Participants does not need to know in advance anything related to the coordination of the groups where they participate.
- Support multi-object coordination: there is not limit in the number of participants that can participate in the coordination groups, the role abstraction allows one to refer to a group of participants without specifying their number.
- High-level coordination abstractions: the coordination is specified in the form of coordination rules, programmers do not care about the details how they are enforced by the coordination group.
- Support evolution of the coordination: participants join and leave groups at any time and coordination rules can be added and removed.
- Promote the reuse of coordination abstractions: coordination patterns specified in coordination groups can be reused independently of the participants they coordinate.
- Declarative specification of the coordination: the coordination is specified in the coordination groups in the form of rules.
- Incremental specification of the coordination: new coordination groups can be defined from existing coordination groups.
- Support validation of formal properties: formal properties can be verified in CoLaS coordination groups by applying a technique which transform CoLaS coordination groups in Predicate-Action Petri Nets. Reachability analysis are used in the obtained Petri Nets to validate safety and liveness properties.

Contributions

The main contributions of this chapter to the thesis are:

- We introduce CoLaS a group based approach for the coordination of concurrent objects systems. The CoLaS coordination model is based on the notion of coordination groups. A coordination group is an entity that specifies control and enforces the coordination of groups of collaborating active objects. The primary tasks of the coordination groups are: 1) to support the creation of active objects, 2) to enforce cooperation actions between active objects, 3) to synchronize the occurrence of those actions and 4) to enforce proactive behavior on the systems based on the state of the coordination.

  The CoLaS coordination model is built out of two kinds of entities: the participants and the coordination groups. The participants are the entities to be coordinated and the coordination groups are the entities that control and enforce the coordination of the participants. The participants in the CoLaS coordination model are active objects: objects that have control over concurrent method invocations. A coordination group itself is composed of three elements: the roles specification, the coordination state and the coordination rules. The roles specification defines the different roles that participants may play in the group. Each role specifies the minimum requirements it imposes to an active object to play the role. The coordination state defines general information needed to perform the coordination and the coordination rules define the different rules governing the coordination of the group. The coordination rules specify: cooperation actions between participants, synchronizations on the execution of participants actions and proactions or actions that are initiated by the participants independently of the messages they exchange.

  One of the most important characteristics of the CoLaS coordination model and language is its capacity to dynamically adapt the coordination specified in the coordination groups. No other coordi-

nation model and languages in our survey of existing coordination models and languages supports the dynamic modification of the coordination. The CoLaS model support three types of dynamic coordination changes: (1) new participants can join and leave the groups at any time, (2) new groups can be created and destroyed dynamically and (3) new coordination rules can be added and removed from the groups. The capacity of CoLaS to dynamically adapt the coordination specified in the groups at run time makes it particularly interesting for the specification and construction of modern concurrent object-oriented systems. In those systems evolution is the most difficult requirement to meet since not all the application requirements can be known in advance.

- We provide an evaluation of the CoLaS model with respect to the list of requirements we identified as fundamental for the specification of a coordination model and language for active object systems. From our point of view CoLaS fully support all the requirements specified in this list:

  1) Clear separation of the computation and the coordination concerns: In CoLaS the coordination is encapsulated in the coordination groups and the computation in the participants; 2) Encapsulation of the coordination behavior: in CoLaS all the coordination behavior is specified inside the coordination groups; 3) Support multi-object coordination: in CoLaS there is not limit in the number of participants that can participate in the coordination groups, nor in the number of roles that can be specified; 4) High-level coordination abstractions: the coordination is specified in the form of coordination rules, it defines what to do and not how to do it; 5) Support evolution of the coordination: participants join and leave groups at any time, coordination rules can be added and removed and new coordination groups created on the fly; 6) Promote the reuse of coordination abstractions: coordination patterns specified in coordination groups can be reused independently of the participants they coordinate; 7) Declarative specification of the coordination: the coordination is specified in the coordination groups in the form of rules; 8) Incremental specification of the coordination: new coordination groups can be defined from existing coordination groups and coordination groups may play the role of participants in other coordination groups; 9) Support validation of formal properties: formal properties can be verified in CoLaS coordination groups. We transform CoLaS coordination groups in Predicate-Action Petri Nets where we apply reachability analysis techniques to validate safety and liveness properties.

CHAPTER 4

CORODS: A Coordination Programming System for Open Distributed Systems

Software development of distributed systems has changed significantly over the last two decades. This change has been motivated by the goal of producing Open Distributed Systems (ODS in the following) [Crow96a]. ODS are systems made of components that may be obtained from a different number of sources which together work as a single distributed system. OSD are basically "open" in terms of their topology, platform and evolution: they run on networks which are continuously changing and expanding, they are built on top of a heterogeneous platform of hardware ad software pieces and their requirements are continuously evolving. Evolution is the most difficult requirement to meet since not all the application requirements can be known in advance. ODS are a dominating intellectual issue of the search in distributed systems. Figuring out how to build and to maintain those kinds of systems is a central issue in the distributed systems research today.

In 1998 the International Standard Organization (ISO) began a project for preparing standards for Open Distributed Processing (ODP). These standards have now been completed. They define the interfaces and protocols to be used in the various components of an ODS. The ODP standards provide a framework within which ODS may be built and executed. One of the most (if not the most) popular specification for some parts of the ODP is the Common Request Broker Architecture (CORBA)[OMG95a]. The CORBA middleware provides a standard for interoperability between independently developed components across networks of computers. Details such as the language in which components are written or the operating system in which they run is transparent to their clients. The OMG focused on distributed objects as a vehicle for system integration. The key benefit of building distributed systems with objects is encapsulation: data and state are only available through invocation of a set of defined operations. Object encapsulation makes system integration and evolution easier: differences in data representation are hidden inside objects and new objects can be introduced or replaced in a system without affecting other objects.

Although the CORBA middleware seems to provide all the necessary support for building and executing ODS it only provides a very limited support for their evolution. From our point of view the main problem with CORBA systems is that the description of the elements from which systems are built and the way in which they are composed are mixed within the application code. This problem makes those systems difficult to understand, modify and customize. From our point of view the introduction of the so called coordination models and languages into the CORBA model represents a possible solution to this problem. The main goal of a coordination model and language is to separate computation and coordination aspects in concurrent and distributed systems. Separation of concerns facilitates abstraction, understanding and evolution

of concerns. We propose in this chapter to introduce the CoLaSD [Cruz99b] coordination model into the CORBA framework in the form of a coordination service called CORODS. The CoLaSD coordination model is an extension of the CoLaS coordination model presented in chapter 3 of this thesis to perform coordination of distributed active objects. The CoLaSD model takes into account the possibility of failures in the participants common to distributed systems. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network.

We have divided the presentation of this chapter into three parts:

In the first part of this chapter we introduce CoLaSD, a coordination model to manage coordination in distributed object systems. The CoLaSD coordination model corresponds to an extension of the CoLaS coordination model introduced in chapter 3 of this thesis to support coordination of distributed object systems. The CoLaSD coordination model and language is based on the notion of coordination groups, entities that control and enforce the coordination of groups of collaborating distributed objects. Basically we show how the basic asynchronous communication protocol used among the participants to communicate is replaced by the ACS protocol to tackle consistency problems introduced by the distribution. We illustrate the CoLaSD model using as example a simplified version of an architectural pattern used in distributed systems "The Administrator" [Papa95a]. The administrator is an object that uses a collection of "worker" objects to service requests received from clients.

In the second part of this chapter we introduce CORODS, a coordination service for distributed objects based on the CoLaSD coordination model. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network. We show how the CORODS service is integrated into DST (Distributed Smalltalk) [Cinc94a], a middleware framework that provides an advanced object oriented environment for prototyping, development and deploying of CORBA 2.0 applications. We divide into two the presentation of the basic operations specified in the CORODS service. First the lifecycle operations and then the reference operations. The lifecycle operations concern the operations related to the creation, coping, moving and destruction of groups. The reference operations concern operations that allow users to obtain references to groups. The group references are used to manipulate and modify the specification of the groups.

Finally in the third part of this chapter we present some related work in the definition of a coordination service for CORBA, we present our conclusions and we point out the main contributions of this chapter to the thesis.

4.1 Related Work

The idea of using object groups [Guer98a] to perform the coordination of distributed systems has been around for long time. Object groups have proven to be very convenient for distributed programming, particularly for achieving fault-tolerance through replication. Dollimore and Coulouris [Doll92a] have pointed out three aspects in which the object groups and multicast invocations have proven to be useful for the construction of object based platforms for building multi-user applications. The first aspect concerns the issue of informing users when other users have altered shared objects. The second aspect concerns the design of an optimistic form of concurrency control for replicas of shared objects and the third aspect concerns the distribution of capabilities to groups of users.

Several works have been done in the specification and construction of object groups [Mish89a][Wood93a][IONA94a][Land97a]][Guer98a][Neli01a][Baud02a]. In general in all these works the term object group refers to a logical name for a set of objects whose membership may change from time to time. All invocation messages to a group are propagated to its members. The term multicast refers to an invocation message that is sent by one object to a group of objects. An unreliable multicast provides no guarantee about message delivery and ordering. A reliable multicast is either received by all live members of the group or by none of them. An ordered multicast is a reliable multicast in which the messages arrive to all the recipients in the same order.

What makes our approach different from all the approaches mentioned before is that we do not focus on CoLaS exclusively in the communication aspect of the coordination. All the works in object groups mentioned before focus exclusively on providing and integrating multipoint to multipoint communication as a way to define and coordinate parallel activities in distributed systems. We have also integrated group communication in the CoLaS model to multicast messages to all the participants of a role, nevertheless, we have not focus exclusively on this aspect of the coordination, we consider that other coordination aspects like the specifications of the synchronizations in the systems, the control of the lifecyle of the coordinated entities, etc. must also be addressed by the coordination mechanism in a system. We believe that coordination mechanisms focusing exclusively in the specification of the communication (i.e., the interaction) partially fail in supporting the coordination aspect. It is important to remember that the main advantage of using a coordination model and languages like CoLaS for specifying and building distributed system results from the possibility to separate the coordination and computation aspects in such systems and thus simplify their specification, construction and evolution. This is also an important aspect does not supported by the object groups approaches mentioned before.

Concerning related work in introducing the so called coordination models and languages into the CORBA model, our work is very new [Cruz99a][Cruz01a]. To our knowledge the only work that could be consider as related in this domain concerns the introduction of a cooperation service for CORBA based on graph grammar techniques [Drid99a]. The main differences with respect to our approach are: (1) they coordinate sequential objects (2) coordination is specified as graphs transformations and (3) they do not manage the evolution of the coordination rules that are applied over the coordination graphs. Most of the work done in coordination in CORBA concerns the introduction of an object group communication service. For examples, see Electra[Land97a], Orbix+Isis [IONA94a] and Object Group Service [Guer98a]. Furthermore, group communication systems has been recently identified as a key tool for supporting fault tolerance in CORBA: the new fault-tolerance specification [OMG00a] recommends that a view-oriented group communication systems be used to support active object replication in CORBA.

4.2 Motivation - The Administrator Pattern [Papa95a]

Before to present the extensions made to the CoLaS coordination model to support distribution and to present the implementation of the CORODS coordination service in CORBA, we would like to motivate our work with an example. Consider the architectural pattern called Administrator introduced by Papathomas in [Papa95a]. This pattern is used in distributed systems basically to structure load balancing between different machines. The administrator is an object that uses a collection of “workers” objects to service requests received from clients. The administrator application consists of three kinds of entities: (1) the clients that issue requests to the administrator; (2) the administrator that accept the request and distributes the re-

quests to the workers; and (3) the workers that handle the administrator requests and send back the results to the clients.

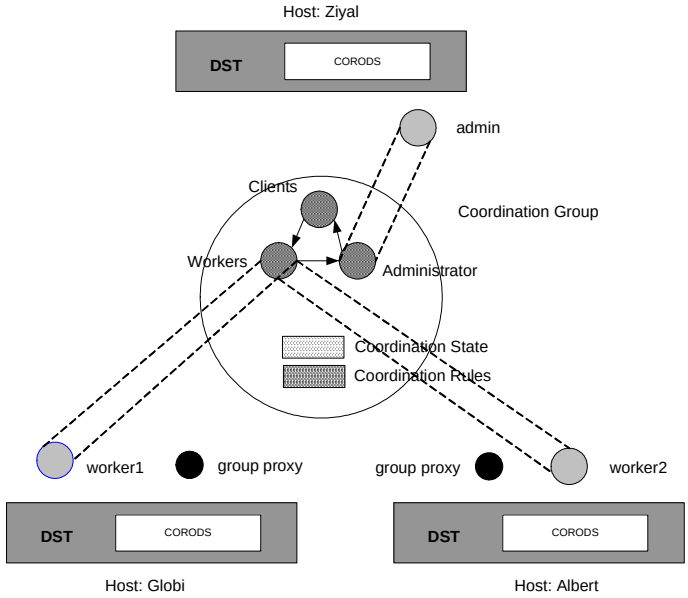


Figure 4.1 : A Distributed Administrator Pattern

Consider now, a specific scenario composed by one administrator and two workers, all the three running on three different machines in a local network: Ziyal, Albert and Globi as illustrated in (**Figure 4.1**). The first worker will run in Globi, the second worker in Albert; and the administrator in Ziyal. CORBA (in our case DST) provides the support to communicate the different participants. It is possible to specify in their code the different remote calls and reception of replies described in the pattern. And, because of the specification of IDLs in CORBA it is possible to make the participants communicate even if they are written in different programming languages. We say in this case that CORBA guarantees the interoperability of the systems. As we already mentioned in the introduction, CORBA provides in principle all the necessary support for building and executing a system like the one described before. Nevertheless, the main problem with CORBA is that the description of the elements from which systems are built and the way in which they are composed are mixed within the application code of the participants. This problem makes CORBA systems difficult to understand, modify and customize. We propose in this chapter to tackle this problem by introducing the CoLaS coordination model in CORBA in the form of a coordination service. But, first it is necessary to adapt the CoLaS coordination model to support the new requirements imposed by the distribution, in particular to manage the possibility of failures in the distributed participants. Consider now how different it will be the specification of the Administrator Pattern solution if it will be possible to specify separately the coordination aspect in a CoLaS coordination group. All the coordination of the system will be specified in a single entity, controlling and enforcing the coordination of the distributed participants. It will be simple

to modify and to adapt the coordination to changes in the requirements given that all the changes will be done in one place and not all over the participants code.

4.3 CoLaSD: Extensions for Distributed Object Coordination

It is easy to understand why distributed object systems are an important computing technology. They allow the sharing of information and resources (i.e., disks, printers, files, databases, etc.), they increase the computing power of the systems because they can process activities in parallel, they can grow easily over a large range of sizes and they do not necessarily crash at once. Building distributed object systems requires ideally that four main issues be addressed [Schr93a]:

- Independent Failure: because there are several distinct computers involved, when one breaks the others may continue working. It is often necessary that the system continues working after one or more computers have failed.
- Unreliable Communication: because in most of the cases, the interconnections between the computers can not be kept in a controlled environment they will not work correctly all the time. Connections may be unavailable, messages may be lost or garbled. One computer can not rely on being able to communicate all the time with another, even if both are working.
- Insecure Communication: the interconnection among the computers can be exposed to unauthorized intrusions and message modifications. It is hard to know what is being trusted and what can be trusted.
- Costly Communication: the interconnections among the computers user provide lower bandwidth, higher latency and higher cost communication that available within a single machine.

Building distributed object systems is still difficult today, not only because they require that engineers address the four issues exposed before, but because existing distributed object-oriented languages provided limited support for their specification, construction and evolution. We believe that coordination models and languages have a role to play in the construction of distributed object-oriented systems, the separation of concerns they promote will allow engineers to reduce the complexity of building such systems. How to extend the CoLaS coordination model to support the coordination of distributed objects is the question that we pretend to answer in this chapter. We will not address all the four issues introduced by distributed systems and exposed before, each one of the mentioned problems will motivate a thesis by itself. We will focus exclusively on extending the CoLaS coordination model to support the consistency problems introduced by the distribution aspect in distributed object systems. The consistency of a distributed object system depends on assertions done by the distributed objects about the state of other distributed objects, because the distributed objects execute concurrently at different places, failures in their execution or in the communication system may modify and thus affect the overall consistency of the system. To provide a solution to the consistency problems we propose to modify the CoLaS model introduced in chapter 3 of this thesis and to replace the basic asynchronous communication model in the model by the ACS (Apply, Call, Send) protocol [Rach92a]. The ACS protocol is a communication protocol designed to support reliable distributed object applications. The ACS communication protocol merges the nested actions model proposed by [Moss81a] with the model of nested asynchronous request messages.

4.3.1 Consistency in Distributed Object Systems

One of the main problems in distributed object systems is that their consistency depends on assertions done by the distributed objects about the state of other distributed objects, because the distributed objects execute

concurrently at different places failures in their execution or in the communication system may modify and thus affect the overall consistency of the system. A well know solution to the problem of consistency consists of enclosing related objects executions inside atomic actions resembling transactions [Coul94a]. The atomic actions have the following ACID properties:

- Atomicity: actions are either completely executed or completely undone.
- Consistency: the execution of the actions preserves the invariant properties.
- Isolation: concurrent actions are isolated from each other.
- Durability: the results of the execution of the actions are permanent.

The main problem with the atomic action model concerns the granularity of the recovery to failures. In the atomic action model either all the object executions inside the atomic action are executed completely or none of them are executed at all. If any local failure occurs during the execution of the atomic action, the action is aborted and its effects on all the objects related by the atomic action are discarded. Local failures cannot be masked to take advantage of partial system availability. A powerful extension to the atomic actions model is the nested actions model proposed by [Moss81a]. In the nested actions model an atomic action may be broken into subactions. The idea is to allow each subaction to fail independently of each other without forcing its parent to abort. Both atomicity and isolation are guaranteed for the subactions.

In the concrete case of distributed object-oriented systems several solutions have been proposed, they can be classified in two approaches [Guer92a]: the explicit and the implicit.

- Explicit approach: in the explicit approach the programmers of the distributed object systems specify logical units of computations as atomic actions using linguistic constructs defined in the object-oriented programming languages. An example of a language using this approach is Hybrid [Nier87a]. In Hybrid actions are specified using the atomic statement. Every time a programmer needs to specify a set of statements as an atomic activity, he places the set of statements inside curly braces. In (**Figure 4.2**) a complex graphic object displays its parts within a single atomic action to guarantee that the parts will all be synchronized.

```
1.var n: partRange;
2.var part: array [partRange] of oid of graphicObject;
3.n := partRange.first;
4.atomic {
5.       coloop {
6.            activity {
7.                 delegate (part[n].displaySelf);
8.            }
9.            if (n <? partRange.last) {n+=1;}
10.           else {break;}
11.      }
12.}
```

Figure 4.2 Atomic actions in Hybrid

- Implicit approach: in the implicit approach the nesting of actions and the nesting of method executions are merged. Each request message issued from a client is enclosed inside an atomic action

whose atomicity is assured by the underlying system. If the service is executed successfully then the action succeeds, otherwise the service fails and the corresponding action is aborted discarding its effects on the affected objects. When the service is executed successfully its effects are permanent in the system. The implicit approach gives to the message passing paradigm a powerful semantics for reliable distributed computing. An example of a language using this approach is Argus [Lisk83a]. In Argus a program consists of a set of *Guardians*. Each Guardian communicates with another Guardian by calling the *handlers* (methods) associated with the Guardian through RPC (Remote Method Invocation) calls. When a handler is invoked, a new subaction is created. The subaction encloses the sending of the message, the execution of the handler and the reply message. If there is any system failure, the system replies with a failure exception. The invoked handler can also abort the subaction and terminate in a user defined exception. In (**Figure 4.3**) we can see how a handler is specified in Argus.

```
1.<Guardian>.<Handler>(<arguments>)
2./* specification of the handler <Handler> for the guardian <Guardian>
3.
4.       .....
5.
6.except when failure(why:string)
7./* alternative code in a case of service failure
8.
9.       .....
10.
11.end
```

Figure 4.3 : Guardian specification in Argus

4.3.2 Consistency in CoLaS

In the CoLaS coordination language introduced in chapter 3 of this thesis we guarantee the consistency of the system using an implicit approach which combines asynchronous communication with the model of nested actions. Every method invocation can be seen as composed of subactions where each subaction is an atomic action itself. The subactions may fail independently of each other without forcing a method invocation to abort.

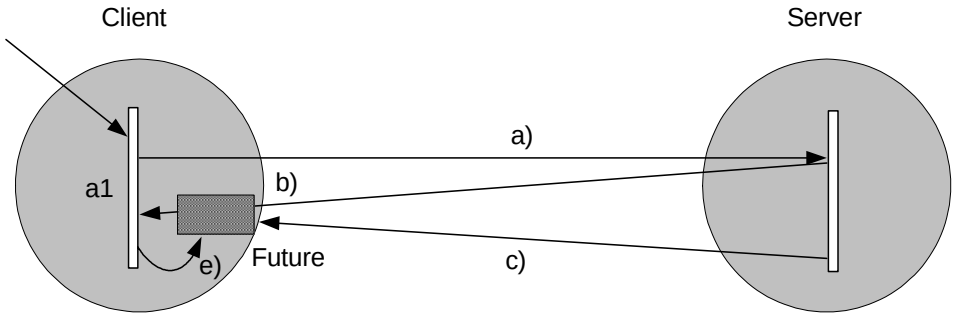


Figure 4.4 : Asynchronous communication in CoLas

In (**Figure 4.4**) we can see the representation of an asynchronous communication between two distributed objects, in (a) a message is sent from the a client object to a server object, in (b) implicitly a future is sent back to the client, in (c) the result of the method invocation (if any) is set by the server object in the future and in (e) the client object requests the future for the result of the method invocation. If the result of the method invocation is not ready in (e) the client object blocks. The advantage of using an asynchronous communication with explicit futures is that the client object does not block while the server object executes the method invocation.

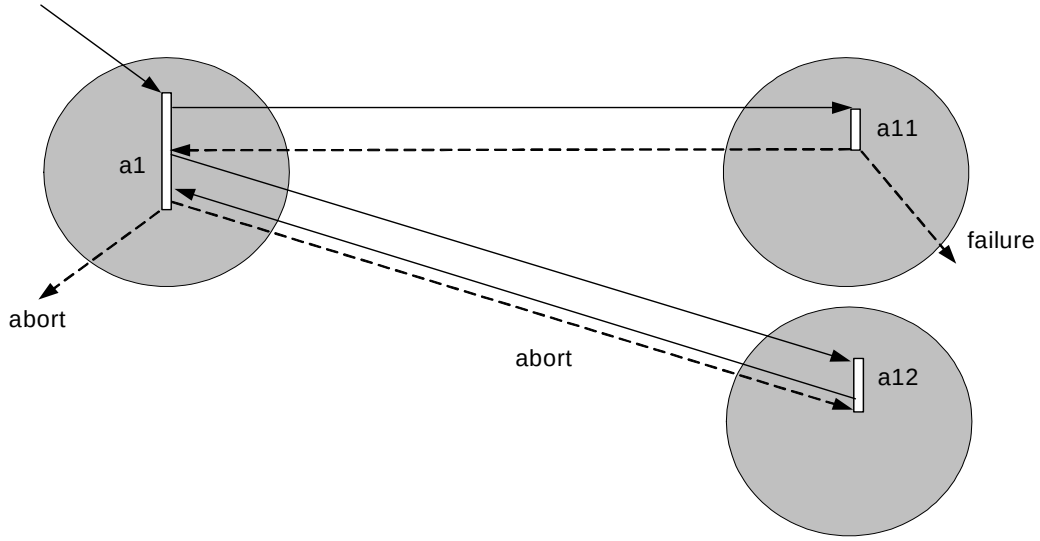


Figure 4.5 : CoLaS nested atomic actions

In (**Figure 4.5**) we can see how a method invocation received by an object generates an atomic action a1. The atomic action a1 is composed of two subactions a11 and a12 corresponding to two messages sent to two other objects during the execution of the received method invocation. A failure during the execution of action a11 produces the abort of the two subactions and of the main action a1. The atomicity model of CoLaS allows a lot of flexibility to manage partial failures in subactions. However, the main problem with the model of nested actions used in CoLaS is that the subactions commit only when the parent action commits, the permanence of the results is guaranteed only for the top parent action. In [Guer92a] a new communication protocol called ACS (Apply, Call, Send) is introduced to tackle this problem. The ACS communication protocol is used for concurrent communication in KAROS [Guer92b] (an exploratory language designed for reliable distributed applications).

4.3.3 The ACS Protocol

In the ACS protocol, distributed objects communicate through three different types of asynchronous message passing: Apply, Call and Send. In an asynchronous communication between two distributed objects we will call the client the distributed object that sends the message to the object and the server the distributed object that receives the message and executes the corresponding method invocation.

4.3.3.1 Apply

In ACS when a client object sends an Apply message to a server object, two concurrent subactions are created (a11 and a12 in **Figure 4.6**). The first subaction a12 corresponds to Apply message sent to the server. It encapsulates the execution of the service related to the message in the server. The second subaction a11 encapsulates the client execution that starts just after that the Apply message is sent to the other distributed object.

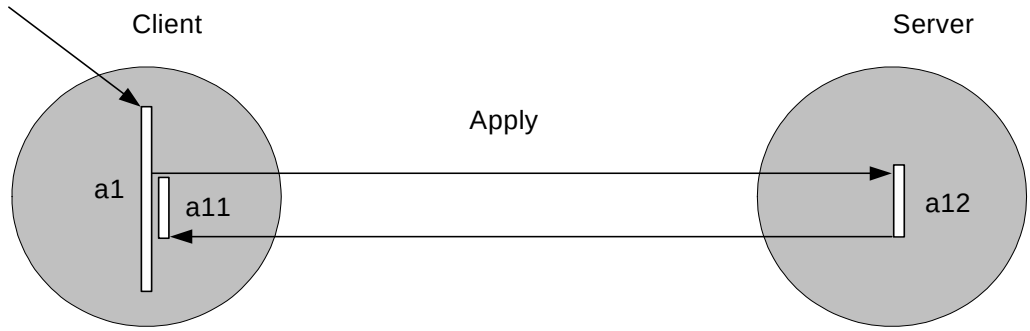


Figure 4.6 : Apply message

If a failure occurs in any of the two subactions a11 or a12 the system aborts the parent action a1 and their effects are discarded. In (**Figure 4.7**) we can see how a failure occurring during the execution of the subaction a12 for example produces the abort of the parent action a1.

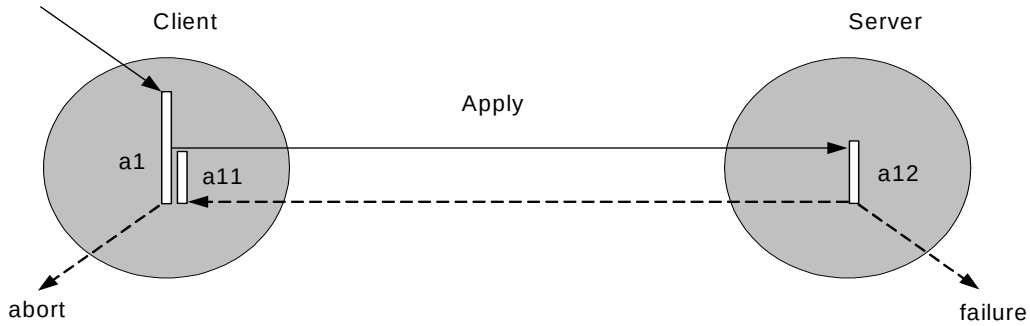


Figure 4.7 : Failure in Apply message

4.3.3.2 Call

In ACS When a client sends a Call message to a server object, two similar concurrent subactions are created (a11 and a12 in **Figure 4.8**). If a failure occurs in the communication system or during the execution of the service, the parent action a1 is not forced to abort like in the Apply message. The client may know that the request has failed and it may choose to abort or to continue its action. Different subactions are thus allowed

to fail independently of each other. A service may be considered as correct even if some of its subactions have not been accomplished.

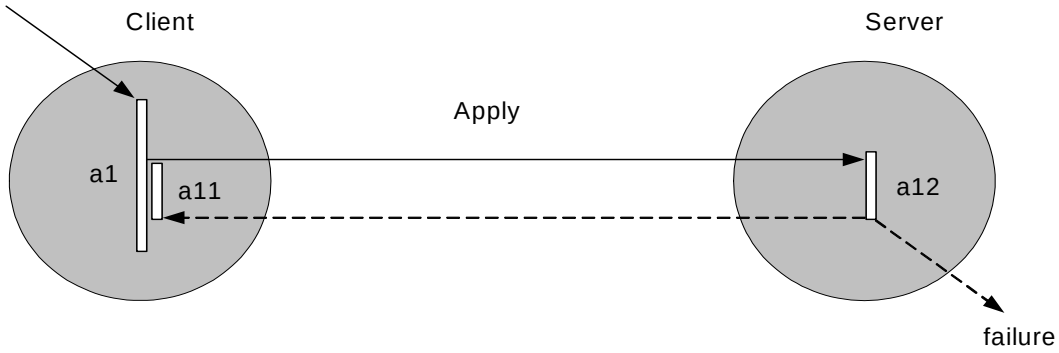


Figure 4.8 : Call Message

In (**Figure 4.8**) we can see how we can see how a failure occurring during the execution of the subaction a12 for example does not causes the abort of the parent action a1. The client decides to continue the action a1 even if the subaction a12 has failed.

4.3.3.3 Send

To provide a simple way for safely breaking atomicity when an independent subaction has to be executed, the ACS protocol introduces a third kind of asynchronous message called Send. When a client sends a Send message to a server object, the client continues executing inside its current action without relying on the server execution and without expecting any reply from the server. In (**Figure 4.9**) we can see how the subaction a12 is executed completely independent from the parent action a1.

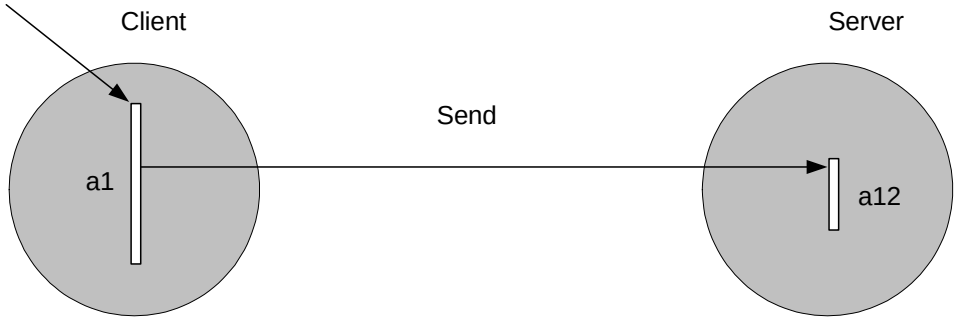


Figure 4.9 : Send Message

4.4 The CoLasD Coordination Model

The CoLaSD model is a coordination model based on the notion of coordination groups. A coordination group specifies, controls and enforces the coordination of groups of collaborating distributed objects. The CoLaSD model is composed of two kinds of entities: the participants and the coordination groups.

4.4.1 The Participants

CoLaSD replaces the basic asynchronous communication model used by the participants in the CoLaS model by the ACS communication protocol. Any message sent by a participant to another participant must be preceded by an ACS protocol message indicating the ACS type of the message. There are three types of ACS protocol messages: apply, call and send. Each ACS protocol message type specifies its corresponding failure semantics in the ACS protocol. When the sender of a message does not specify the ACS type of the message we consider by default the message as a send message.

4.4.2 The Coordination Groups

A coordination group is an entity that specifies, controls and enforces coordination between groups of collaborating participants. The primary tasks of a coordination group are: (1) to enforce cooperation actions between participants, (2) to synchronize the occurrence of those actions and (3) to enforce proactive actions in the participants. The coordination groups (only groups in the following) are composed of the following elements: the roles specification, the coordination state and the coordination rules. The roles specification defines the different roles that participants may play in a group, the coordination state defines general information needed to perform the coordination and the coordination rules define the different rules that govern the coordination of the group. The specification of the three different elements does not differ to much from the specification introduced in Chapter 3 in this thesis. The only difference is that ACS protocol message may appear now in the specification of the coordination rules, indicating specific failure semantics for the messages sent.

4.4.3 CoLaSD - The Administrator Pattern: A Simplified Version

To illustrate the modifications introduced in the CoLaSD model we will use as example a simplified version of “The Administrator” pattern [Papa95a] introduced at the beginning of this chapter. The administrator pattern consists of three kinds of entities: (1) the clients that issue requests to the administrator, (2) the administrator that accept the requests and distributes them to the workers and (3) the workers that handle the administrator requests and send back the results to the clients. Sometimes the administrator must split the requests received from the clients either because the workers do not have all the expertise to manage the whole request or in order to optimize response times. In our example, we assume that each worker is able to manage all the requests received by the administrator. The work of the administrator it to pass the requests to the workers as it receives them from the clients.

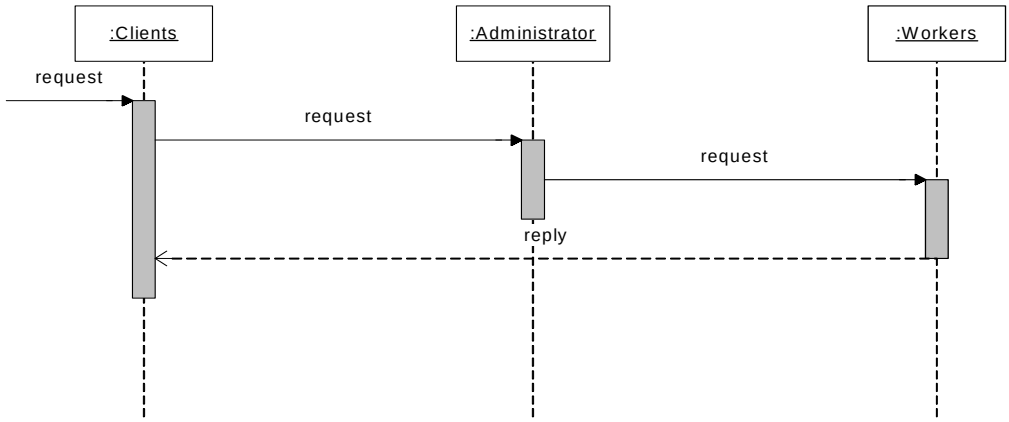


Figure 4.10 : The Administrator Pattern

The Administrator Pattern example illustrates the following coordination problems:

- Transfer of information between entities: clients requests received by the administrator are sent to the workers. The administrator decides which request goes to which worker. The administrator plays the role of a router redirecting requests to workers. The workers receive the requests and execute them.
- Assignment of shared resources: the administrator controls the assignment of requests to workers. The shared resource in this case is the worker processing time. The administrator may apply different assignment policies. In this example we will assume that all the workers have the same capabilities. The administrator selects a worker based on simple “is-free” assignment policy: the administrator chooses in a indeterminist way a free worker between its workers. During the assignment of requests to workers the administrator must prevent the multiple assignment of requests to workers, as well as the assignment of the same request to multiple workers.
- Dynamic evolution of the coordination: the system must be able to scale. New workers can be added to the system and new clients can make requests to the administrator. The assignment policy used by the administrator to allocate requests to workers may also vary during the time. It will be possible for example to allocate client requests to workers based on the execution performances of the workers.

```
1.AdministratorPattern defineRoleNamed: #Client.
2.AdministratorPattern defineRoleNamed: #Administrator.
3.AdministratorPattern defineRoleNamed: #Worker.
4.
5.Worker defineInterface: #(#request:).
6.Client defineInterface: #(#reply:).
7.
8.Worker defineParticipantVariable: #isFree initialValue: true.
9.
10.[1] Client defineBehavior: 'request:args' as:
11.      [Administrator apply request: args].
12.
13.[2] Administrator defineBehavior: 'request:args' as:
14.      [|worker|
15.      worker := Worker detect:[:aParticipant| aParticipant isFree].
16.      worker isFree: false.
17.      worker apply request:args client: sender].
18.
19.[3] Administrator disable: 'request:args' if:
20.      [(Worker detect:[:aParticipant |aParticipant isFree ])isNil].
21.
22.[4] Worker defineBehavior: 'request:args client: client' as:
23.      [client reply: (self request: args)].
24.
25.[5] Worker interceptAtEnd: 'request:args client:client' do:
26.      [receiver isFree:true ].
```

Figure 4.11 : The Administrator Pattern

Role Specification

In the Administrator example the participants play one of the three roles: Client, Administrator or Worker (**Figure 4.11** lines 1,2 and 3). The minimal interface that a distributed object should have in order to play a role in the group is specified by the role interface of the role it wants to play. A role interface specifies signatures of methods used in the specification of the role. The role interface of the role Worker (line 5) specifies that each potential worker must know how to react to the request: method invocation. The request: method models a generic service (i.e., for example a clock service). The interface of the role Client (line 6) specifies that each potential client must know how to react to the reply: method invocation. The reply: method invocation is used in the pattern to return the result of the requested service to the client.

Coordination State

The coordination state specifies information needed to perform the coordination. It may concern information like: whether some action occurred or actually occurs in the system (i.e., historical information), the number of times some action occurred or actually occurs in the system (i.e historical counters), etc. The coordination state is specified by declaring variables. Three types of state variables can be defined in CoLaSD:

group variables, role variables and participant variables. In the Administrator example we define a unique variable called isFree (**Figure 4.11** line 8). The isFree variable is a participant variable, each participant playing the role Worker has a isFree variable associated with it. The isFree variable is used by the administrator to control the assignment of the client requests to the workers. The isFree variable is a boolean variable; when the variable validates true it indicates that the worker is free to execute requests and when the variable validates false it indicates that the worker is busy and can not execute requests. The administrator only assigns jobs to workers which are free.

Coordination Rules

In CoLaSD we define three types of coordination rules as in the CoLaS model: behavioral coordination rules (behavioral rules), reactive coordination rules (reactive rules) and proactive coordination rules (proactive rules). Behavioral rules are rules that define implications between participant actions. Reactive coordination rules are rules that depend for their application on the messages exchanged by the participants of the group. Reactive rules are evaluated at specific evaluation points during the processing of method invocations by the participants. The specification of the CoLaSD Coordination Rules is very similar to the specification of Coordination Rules in the CoLaS coordination model, the only difference finds in the use of the ACS communication protocol in the specification of the rules.

In the Administrator example (**Figure 4.11**) we defined four coordination rules (three behavioral rules: rules 1, 2 and 4) and two reactive rules: (rules 3 and 5):

Rule 1 (**Figure 4.11** line 10): specifies that all clients requests request: are sent to the administrator (line 11). The apply ACS protocol message that appears before the request: message sent to the administrator specifies that, whenever a failure occurs during the execution of the request: method invocation in the administrator the execution of the request: behavior in the client is aborted.

Rule 2 (**Figure 4.11** line 13): defines that a request: message received by the administrator triggers a request:sender: message in a free worker. To select a free worker the role operation detect: is used in the role worker. The detect: operation returns the first participant playing the role worker that validates the condition specified as argument (or nil if none). We can see in (**Figure 4.11** line 15) that the condition specified in the detect concerns the value of the participant variable isFree of the worker. When a free worker is found the variable isFree of the worker is set to false to indicate that the worker is now busy. In line (**Figure 4.11** line 17), we can see how the request: client: message received by the administrator is forwarded to the selected worker. The message includes the identity of the client who made the request. Again, the apply ACS protocol message that appears before the request:client: message sent to the worker specifies that, whenever a failure occurs during the execution of the request:client: method invocation in the worker the execution of the request: behavior in the administrator is aborted

Rule 3 (**Figure 4.11** line 19): defines that a request: message received by the administrator is delayed when there is not worker free to execute the request. To determine whether a worker is free to execute a request we use the participant variable isFree. The role operation detect: is used again in this rule to specify the synchronization condition of the disable rule.

Rule 4 (**Figure 4.11** line 22): defines that a request:sclient: message received by a worker implies (line 23) the execution of the request by the worker and the sent of the reply to the client who sent the request.

Rule 5 (**Figure 4.11** line 25): when a worker finishes to execute a request:client: method invocation, the state of the participant variable isFree is updated to true. The atSent interception rule is evaluated after the execution of the service:client: method invocation by the worker.

Pseudo-Variables

There are three pseudo-variables that can be used within the groups. They are: group, receiver and sender. The group variable refers to the current group, the sender variable refers to the distributed object that sent the message and the receiver variable to the distributed object processing a received message. In the Administrator example we refer to the sender pseudo-variable (line 17) and to the receiver pseudo-variable (line 26).

Failures

In (**Figure 4.11** lines 11 and 17) rules 1 and 3, we can see how the ACS protocol is used in the administrator example: in rule 1 the special apply ACS protocol message precedes the request: message sent to the administrator and in rule 3 the apply ACS protocol message precedes the request:client: message sent to the selected worker. The failure semantics associated with the apply ACS protocol message specifies that whenever a subaction composing a parent action fails the subaction and the parent action are aborted. In the example (**Figure 4.12**) if the execution of the request:client: method invocation fails in the worker the system will abort the execution request: method invocation in the administrator and in the client. The abort of a method invocation in a participant implies the roll back of all the modifications done during the execution of the method invocation until the abort moment.

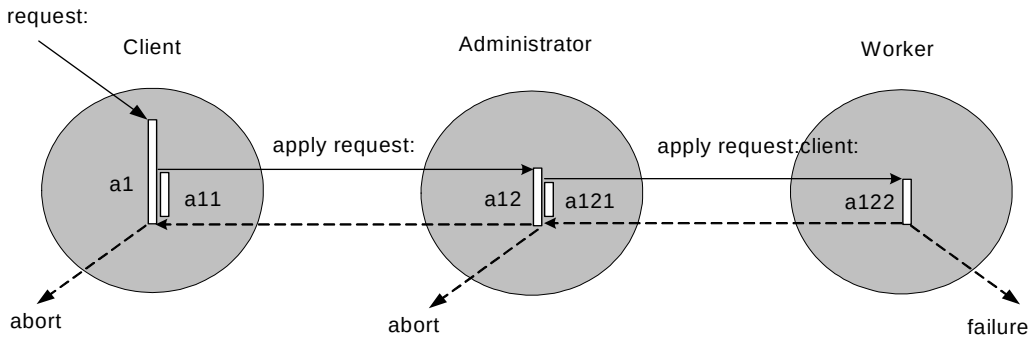


Figure 4.12 Failure of the Apply service:

To avoid the abort of the execution of the method invocation request: in the administrator, it is possible to associate a more flexible failure semantics to the request: behavior. The administrator may decide for example to verify whether the execution of the method invocation request:client: has failed in the worker and decide to select another worker to execute the task. To validate whether the execution of a method invocation has failed we use the future received during the invocation of the request:client: method invocation. It is important to remember that every method invocation in CoLaSD generates a reply and that replies are

managed using futures. The administrator sends the message failed to the future to verify if the request:client: method invocation has failed or not.

In (**Figure 4.13** line 1) we show how the rule 2 can be redefined to implement a completely different failure strategy, the request:client: message (**Figure 4.13** line 7) is preceded by the special call ACS protocol message. The failure semantics associated with the call ACS protocol message specifies that when a subaction composing a parent action fails the parent action should not be necessarily aborted. It is up to the client object executing the parent action to decide whether the parent action should be aborted or not. To verify whether the execution of the service has failed we request the future returned by the method invocation (**Figure 4.13** line 7). In the example if the execution of the request:client: method in the worker fails (in CoLaS a fail corresponds the raise of an error signal by a participant) we select another worker and we retry again to execute the request:client: request in the other worker. If we can not find a worker to execute the request we raise an exception (**Figure 4.13** line 12). For the management of exceptions we use the facilities of the host language in which the CoLaS model is integrated (Smalltalk in our case).

```
1.[2] Administrator defineBehavior:'request:args' as:
2.       [|worker result future |
3.       worker := Workers detect:[:aWorker| aWorker isFree].
4.       worker isFree: false.
5.
6.       [worker notNil and:
7.            [(worker call request: args client: sender) failed ]]
8.            whileTrue:
9.                 [(worker := Workers detect:[:aWorker| aWorker isFree])
10.                     ifNotNil: [worker isFree: false]].
11.
12.      worker ifNil:[InsufficientComputingResourceError raiseSignal ]].
```

Figure 4.13 : Considering failures in workers

Analysis

The example illustrates how it is possible to define a more flexible failure semantics to the coordination specified in the behavioral rules of the AdministratorPattern group. The ACS protocol allows the specification of the communication through three different types of asynchronous message passing: Apply, Call and Send. When using the Apply communication mechanism if a failure occurs all the subactions and the parent action are discarded. When using the Call communication mechanism is used different subactions are allowed to fail independently of each other without affecting the parent action. And, when using the Send communication mechanism is used subactions are executed completely independent from the parent action.

4.5 CORODS - A Coordination Service for CORBA

CORODS is a coordination service for distributed objects based on the CoLaSD coordination model. A prototype of CORODS was built on top of a middleware framework called DST, a CORBA 2.0 compliant framework for Smalltalk. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network. In (**Figure 4.14**) we show how a coordination group created by the CORODS service coordinates participants that are physically distributed between different machines. The CORODS coordination service is integrated in DST as a basic service.

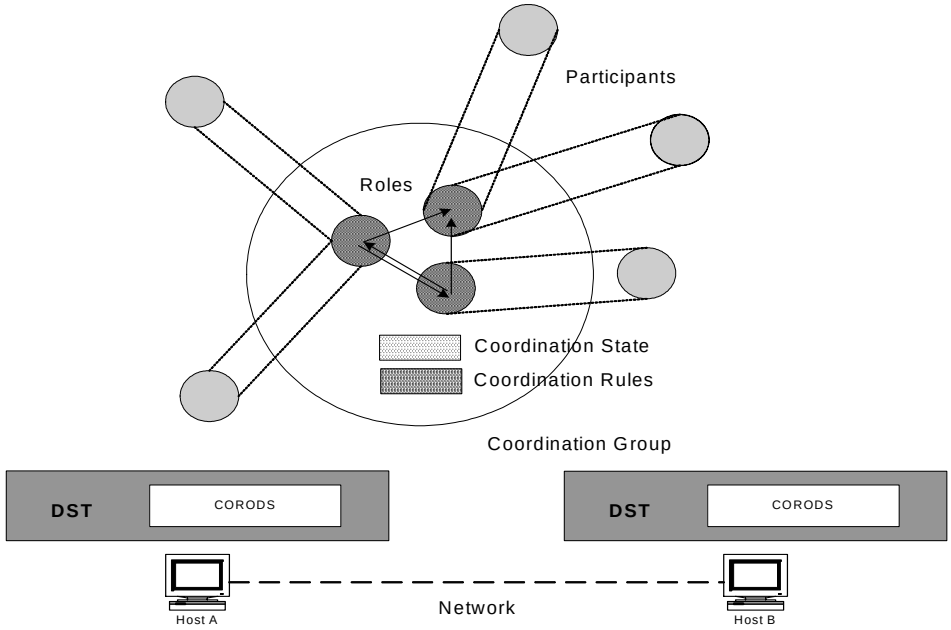


Figure 4.14 : CORODS

4.5.1 The DST Framework

DST is a middleware framework that provides an advanced object oriented environment for prototyping, development and deployment of CORBA 2.0 [OMG95a] compliant distributed applications. CORBA is the standard interface of the central component of the OMA (Object Management Architecture) architecture the Object Request Broker (ORB). The CORBA standard defines common methods of communication between distributed objects on heterogeneous platforms.

The most important function of the ORB is to enable a client to invoke operations on a potentially remote object. To communicate with a remote object, the client must identify the target object by means of an object reference. The ORB is responsible for locating the object, preparing it to receive the request and passing the data needed for the request to the object. If the operation identified by the request implies some reply from the remote object the ORB is responsible for communicating the reply back to the client.

One of the most important features of CORBA is its IDL (Interface Description Language) language. The IDL language is used by the other components of the OMA to specify the services they offer through the ORB. A set of common services have been defined in the OMA architecture. These services represent in general useful services independent of the application domain. They are called Common Object Services (COS) and currently they are 15. DST provides six of them: naming, lifecycle, event notification, transactions, persistence, concurrency control.

From a coordination point of view DST provides all the facilities required to the implementation of the CoLaSD model: it provides remote object interaction facilities, a distributed naming service to locate dis-

tributed objects by names independently of the place where they find, a lifecycle service to control creation and destruction of distributed objects, a concurrency control service to mediate concurrent access to distributed objects and a transactions service to control atomicity of distributed transactions.

4.6 The CORODS Coordination Service

The CORBA specification 2.0 defines the way in which an application can initialize itself in a CORBA environment. It defines interfaces to: initialize the ORB, initialize the Object Adaptor (the OA is the primary mechanism for an object implementation to access ORB services [OMG95a]) and to obtain initial object references. The initial references service is a simplified local version of the naming service, applications use this service to register and to obtain object references which are essential to an application.

To obtain initial object references CORBA defines two operations: list_initial_services and resolve_initial_references. The operation list_initial_services allows an application to return the names of the available objects and the resolve_initial_references operation returns the object reference associated with a name returned by the initial references operation. DST provides by default references for three initial services: the naming service, the factory finder and the interface repository. The naming service is the service used to locate distributed objects by names, the factory finder is the service that allows one to obtain references to factories of a particular class (i.e., a factory is an object that creates objects in response to client service requests) and the interface repository is the place where the IDL definitions are stored. The CORODS service is registered in each ORB as an initial service. Each time the ORB is started a new instance of the CORODS service is created. To obtain a reference to the CORODS service a client must send a resolveInitialReferences: message to the ORB object with the name #CORODS as argument (**Figure 4.15**). In DST the ORB is represented by the class ORBObject.

```
1.corods := ORBObject resolveInitialReferences: #CORODS.
```

Figure 4.15 : Obtaining a reference to the CORODS service

4.6.1 Coordination Groups Lifecycle Operations

The CORODS coordination service specifies basic lifecycle operations for creating, deleting, copying and moving groups locally or remotely; operations required for controlling the population and the migration of groups across the network.

Group Creation

Groups are instances of coordination groups classes (group classes in the following). A group class contains the specification of the roles, the coordination state and the coordination rules that specifies a group. Group classes are created by sending the message *createCoordinationGroupClassNamed:* <Coordination Group Class Name> to the CORODS service. The <Coordination Group Class Name> argument specifies the name of the group class to be created. Groups instances (i.e., groups) are created by sending the message *createCoordinationGroupNamed:* <Coordination Group Name> *forCoordinationGroupClassNamed:* <Coordination Group Class Name> to the CORODS service. The <Coordination Group Name> argument specifies the name of the group to be created. The two operations allows users to create group classes and group instances locally. Two similar operations are specified in CORODS to create group classes and group

instances remotely (i.e., in a remote host), the name of the host (or its IP address) where the group class or group must be created should be specified as an extra argument *inHost:* <Host Name>. It is important to precise that the names of the group classes and the names of the groups must be unique in each machine. We use those names to identify uniquely group classes and groups in machines.

In (**Figure 4.16**) we can see the implementation of the two group classes creation operations (local and remote). In the local case (line 1) we use the CoordinationGroup class to create the group class. The CoordinationGroup class is a basic class containing all the necessary support to specify, create and manipulate group classes and groups. In the remote case (line 5) the creation of the group class is made through the CORODS service in the host where the group must be created (line 10). The reference to the remote CORODS service is obtained through the remote ORB (line 9). In (line 8) we obtain a reference to the remote ORB in the machine named aHostName.

```
1.CORODS >>createCoordinationGroupClassNamed: aCoordinationGroupClassName
2.       CoordinationGroup
3.            createCoordinationGroupClass: aCoordinationGroupClassName.
4.
5.CORODS >>createCoordinationGroupClassNamed: aCoordinationGroupClassName
6.        inHost:aHostName
7.       | orbProxy remoteCORODS |
8.       orbProxy := OrbResolver generateOrbProxy: aHostName.
9.       remoteCORODS := orbProxy resolveInitialReferences: #CORODS.
10.      remoteCORODS
11.           createCoordinationGroupClassNamed: aCoordinationGroupClassName.
```

Figure 4.16 : Group Classes creation

The CORODS service is a groups factory in the sense of CORBA. In the CORBA terminology, factories are objects that create objects in response to clients requests. In the DST framework, a factory is any class that can be instantiated and has interfaces registered for creating objects in the Interface Repository [Cinc94a]. Factories objects are registered during the initialization factories phase of the ORB initialization. For a class to be register as a factory it must have an instance method call abstractClassID. This method returns the appropriate UUID-Universal Unique Identifier which uniquely identifies the class. A UUID is a 16-byte quantity that is guaranteed to be unique. It encodes the local network IP address and a time stamp value indicating the time elapsed since January 1, 1980. One extension made to CoLaSD to support the construction of the CORODS coordination service consists of automatically associating during the creation of group classes an abstractClassID method with a new UUID value to each group class created. Each group class created in CoLaSD becomes in this way a potential groups factory. To locate the correct factory class, the COS specification of the lifecycle service introduces the notion of factories finder. A factory finder is an object at a specific location that helps clients to locate factories of a particular class.

In (**Figure 4.17**) we can see the implementation of the two group creation operations. In the first operation the group is created locally in the place where the create operation is requested (line 12). In the second operation the group is created remotely in the machine specified as argument in the create operation (line 25). In (line 16) we can see how the Factory Finder service is used to obtain a reference to the group class

factory in the local machine. The reference to the group class factory is then used to create a group through the lifecycle service (line 18). The group created is then register into the naming service (line 22).

In the implementation of the remote create group operation (line 25) the group is created through the CORODS service in the remote machine. The name of the remote machine aHostName is used to obtain a reference to the remote ORB object (line 29) and then a reference to the remote CORODS service (line 30). The reference to the remote CORODS service is then used to create the group (line 31). The creation operation returns a proxy to a remote group. All the messages received by the proxy are forwarded to the group in the remote host.

```
12.CORODS >>createCoordinationGroupNamed: aCoordinationGroupName
13.       forCoordinationGroupClassNamed: aCoordinationGroupClassName
14.      | factoryFinder cgFactory cg namingService|
15.      factoryFinder := ORBObject resolveInitialReferences: #FactoryFinder.
16.      cgFactory := factoryFinder
17.                   contextResolve: aCoordinationGroupClassName asDSTName.
18.      cg := cgFactory
19.               createObjectKey: aCoordinationGroupClassName criteria: #().
20.      cg groupName: aCGName.
21.      namingService := ORBObject resolveInitialReferences: #NameService.
22.      namingService contextBind: aCoordinationGroupName asDSTName to: cg.
23.      ^cg
24.
25.CORODS >>createCoordinationGroupNamed: aCoordinationGroupName
26.      forCoordinationGroupClassNamed: aCoordinationGroupClassName
27.      inHost: aHostName
28.      | orbProxy remoteCORODS |
29.      orbProxy := OrbResolver generateOrbProxy: aHostName.
30.      remoteCORODS := orbProxy resolveInitialReferences: #CORODS.
31.      ^remoteCORODS
32.          createCoordinationGroupNamed: aCoordinationGroupName
33.          forCoordinationGroupClassNamed: aCoordinationGroupClassName
```

Figure 4.17 : Groups creation in CORODS

In (**Figure 4.18**) we can see the graphical representation of the sequence of actions that compose the remote create group operation presented in (**Figure 4.17** line 25). The requests for the creation of a remote group named groupX is done in a machine named HostB. The group requested for creation is a group of the class groupClassY whose specification finds in a remote machine named HostA. In (a) the creation request is sent to the local CORODS service in the machine HostB, the request is then forwarded to the remote CORODS service in the machine HostA; in (b) the remote CORODS service contacts the local Factory Finder object to obtain a reference to the group class factory named groupClassY, the reference to the groupClassY group class factory is then used to request the creation of the group; in (d) the newly created group is register in the naming service in the machine HostA; finally in (e) a remote reference (a proxy) to the group is sent back to the user that requested the creation of the group in the machine HostB. In fact the user of the group in the machine HostB does not see that the group finds in the remote machine HostA, all the message sends to the group proxy in the machine HostB are sent automatically to the host HostA.

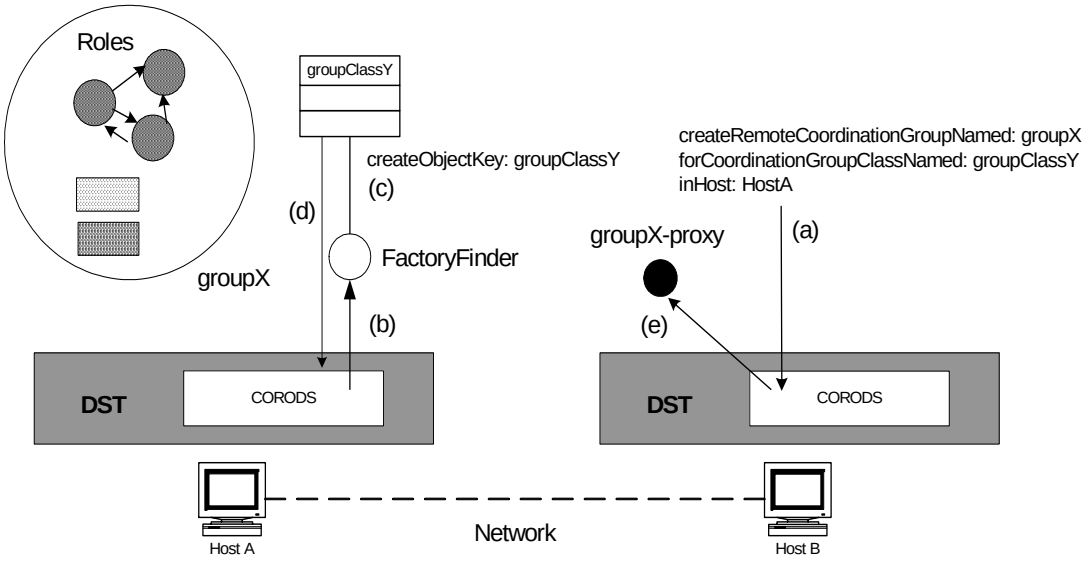


Figure 4.18 : Remote creation of a group

Group Copy

We define three different operations to copy groups in CORODS: the first operation *copyCoordination-GroupNamed:* <Coordination Group Name> *toHost:* <Host Name> (line 1) makes a copy of a local group to the remote machine <Host Name>. By local group we mean a group existing in the machine where the copy request is done. The second operation *copyRemoteCoordinationGroupNamed:* <Coordination Group Name> *fromHost:* <Host Name> (line 11) makes a copy of a remote group from the machine <Host Name> in the local machine. By local host we mean the host where the copy request is done. And, the third operation *copyCoordinationGroup:* <Coordination Group> *toHost:* <Host Name> (line 20) makes a copy of a group (local or remote) to the host <Host Name>. The argument <Coordination Group Name> specifies the name of a group, <Host Name> the name of a machine (or and IP address) and <Coordination Group> specifies a coordination group. In (**Figure 4.19**) we can see the implementation of the three copy operations.

In the implementation of the *copyCoordinationGroupNamed*: <Coordination Group Name> *toHost:* <Host Name> operation (line 1) we obtain first a reference to the local group named aCoordinationGroup-Name that we want to copy. The reference to the group is obtained through the local CORODS service (line 4). We then obtain a reference to the remote ORB object in the machine where we want to make the copy (line 5) and we use such a reference to obtain a reference to the remote Factory Finder object in the machine where we want to copy the group (line 6). Finally we call the copy lifecycle service (line 8) to create a copy of the group in the remote machine. The copy operation returns a proxy to the group copy created in the remote host.

In the implementation of the *copyRemoteCoordinationGroupNamed:* <Coordination Group Name> *fromHost:* <Host Name> operation (line 10) the name of the remote machine aHostName is used to obtain a reference to the remote ORB object in which the group finds (line 13). We use such a reference to obtain

a reference to the remote CORODS service (line 14). The reference to the remote CORODS service is then used to request for a remote reference to the group named aCoordinationGroupName in the remote machine (line 15). Finally we call the copy lifecycle service in the local host to create a copy of the remote group in the local host (line 16). The copy operation returns a local reference to the group copy created in the local host.

In the implementation of the *copyCoordinationGroup:* <Coordination Group> *toHost:* <Host Name> operation (line 18) we receive directly a group reference as an argument. The group reference aGroup can be a proxy to a remote group or a reference to a local group. In both cases the copy operation returns a reference to a group copy created in the machine indicated by the argument aHostName. In (line 20) the name of the machine is used to obtain the reference to the ORB object where we want to make the group copy. The ORB object reference is then used to obtain a reference to the Factory Finder in that machine (line 24). Finally we call the copy lifecycle service to create a copy of the group (line 25).

```
1.CORODS >>copyCoordinationGroupNamed: aCoordinationGroupName
2.         toHost: aHostName
3.       | cg orbProxy remoteFactoryFinder |
4.       cg := self getReferenceToCGNamed: aCoordinationGroupName.
5.       orbProxy := OrbResolver generateOrbProxy: aHostName.
6.       remoteFactoryFinder := orbProxy
7.                                 resolveInitialReferences:#FactoryFinder.
8.       ^cg copyFactoryFinder: remoteFactoryFinder criteria: #()
9.
10.CORODS >>copyRemoteCoordinationGroupNamed: aCoordinationGroupName
11.        fromHost: aHostName
12.      | orbProxy remoteCORODS cg |
13.      orbProxy := OrbResolver generateOrbProxy: aHostName.
14.      remoteCORODS := orbProxy resolveInitialReferences: #CORODS.
15.      cg := remoteCORODS getReferenceToCGNamed: aCoordinationGroupName.
16.      ^cg copyFactoryFinder: ORBObject factoryFinder criteria: #()
17.
18.CORODS >>copyCoordinationGroup: aGroup toHost: aHostName
19.      | orbProxy factoryFinder |
20.      orbProxy := OrbResolver generateOrbProxy: aHostName.
21.      factoryFinder := orbProxy resolveInitialReferences: #FactoryFinder.
22.      ^aGroup copyFactoryFinder: remoteFactoryFinder criteria: #()
```

Figure 4.19 : Copying groups in CORODS

Group Move

Moving a group means to move a group from one place to another across the network. Unfortunately this service has not been completely implemented in CORODS because the move operation is not actually supported by the lifecycle service in DST on top of which we built CORODS. Theoretically the move operation implies that a copy of the group is made at a specified target destination and that the original group is removed from the specified origin destination.

CORODS defines three operations to move groups (similar to those specified for copy groups): the first operation *moveCoordinationGroupNamed:* <Coordination Group Name> *toHost:* <Host Name> operation moves a local group to the remote machine <Host Name>. The second operation *moveCoordinationGroup-Named:* <Coordination Group Name> *fromHost:* <Host Name> operation moves a remote group from the remote machine <Host Name> to the local machine where the request is done. And, the third operation *moveCoordinationGroup:* <Coordination Group> *toHost:* <Host Name> operation moves a group (local or remote) to the machine <Host Name>. The argument <Coordination Group Name> specifies the name of a group, <Host Name> the name of a machine (or an IP address) and <Group> specifies a group. The implementation of the three move operations is very similar to the implementation of the copy operations specified before, basically they differ in the lifecycle operation called by the operation: in the copy operations copyFactoryFinder: and in the move operations moveFactoryFinder:

Group Destruction

The destruction of a group implies the removal of the group from the system. We define a unique group destruction operation: *destroyCoordinationGroup:* <Coordination Group> in CORODS. If references (local or remote) to the group are detected in the system the destroy operation is not executed. During the destroy operation the group is unregistered automatically from the naming service. The group reference sent as an argument can be a local or a remote reference to a group.

```
1.CORODS >>destroyCoordinationGroup: aGroup
2.       | namingService cgName orb |
3.       orb := ORBObject.
4.       cg isRemote ifTrue:[orb:= OrbResolver generateOrbProxy:cg hostName ].
5.       namingService := orb resolveInitialReferences: #NameService.
6.       cgName := cg groupName.
7.       namingService contextUnBind: cgName asDSTName.
8.       ^cg destroy
```

Figure 4.20 : Destroying groups in CORODS

In (**Figure 4.20**) we can see the implementation of the *destroyCoordinationGroup*: operation (line 1). If the group reference is a remote reference we obtain first a reference to the remote ORB object where the group finds (line 4). In this case, we use the name of the name of the machine where the group was created to obtain a reference to the remote ORB (line 4). Each time a group is created we store in the group the name of the machine where the group is created, this is the reason why we send the message hostName to the group. (line 4). In the other hand, if the group reference is local we use the local ORB object (line 3). The reference to the ORB object is used then to obtain a reference to the naming service (line 5). We unregister then the group from the naming service (line 7) and we call the destroy lifecycle service to destroy the group reference (line 8).

4.6.2 References to Coordination Groups

To manipulate or modify an existing group (i.e., modify the coordination state, add coordination rules, etc.) it is necessary to have a reference the group. CORODS provides two operations to obtain references to existing groups: the first *getReferenceToCoordinationGroupNamed:* <Coordination Group Name> returns a reference to a local group. By local group we mean a group existing in the place where the get reference request is done. The second operation *getReferenceToCoordinationGroupNamed:* <Coordination Group Name> *inHost*: <Host Name> returns a remote reference (a proxy) to a group existing in the host <Host Name>. In (**Figure 4.21**) we can see the implementation of both operations.

```
1.CORODS >>getReferenceToCoordinationGrouNamed: aCoordinationGroupName
2.       | namingService |
3.       namingService := ORBObject resolveInitialReferences: #NameService.
4.       ^namingService contextResolve: aCoordinationGroupName asDSTName.
5.
6.CORODS >>getReferenceToCoordinationGroupNamed: aCoordinationGroupName
7.        inHost: aHostName
8.       | orbProxy remoteCORODS |
9.       orbProxy := OrbResolver generateOrbProxy: aHostName.
10.      remoteCORODS := orbProxy resolveInitialReferences: #CORODS.
11.      ^remoteCORODS
12.       getReferenceToCoordinationGroupNamed:aCoordinationGroupNameName.
```

Figure 4.21 : Obtaining references to groups

In the implementation of the *getReferenceToCoordinationGroupNamed:* <Coordination Group Name> operation (line 1) the reference to a local group is obtained using the local naming service (line 3). The name of the group named aCoordinationGroupName is used then to identify the group in the naming service (line 4). It is important to remember that when the groups are created in CORODS they are registered automatically in the naming service in the machine where they are created.

In the implementation of the *getReferenceToCoordinationGroupNamed:* <Coordination Group Name>: *inHost:* <Host Name> operation (line 6) the name of the remote machine aHostName is used to obtain a reference to the remote ORB object (line 9). The reference to the remote ORB is used to obtain a reference to the remote CORODS service (line 10) in the remote machine. The reference to the remote CORODS service is then used to request for a reference to the group named aCoordinationGroupName (line 11). The operation returns a proxy to the remote group in the remote machine. Messages received by the proxy are forwarded to the remote group in the remote machine.

In (**Figure 4.22**) we can see the graphical representation of the sequence of actions that compose the remote getReference operation. The request for the reference of the remote group named groupX is done in the machine named HostB. The getReference operation requests for a reference to a remote coordination group named groupX in the host HostA. In (a) the getReference request is received by the local CORODS service in the host HostB, in (b) the local CORODS service contacts the remote CORODS service in HostA and requests for a reference to the group named groupX, finally in (c) a remote reference (i.e., a proxy) to the remote coordination group is sent back to the user that made the getReference request in the HostB.

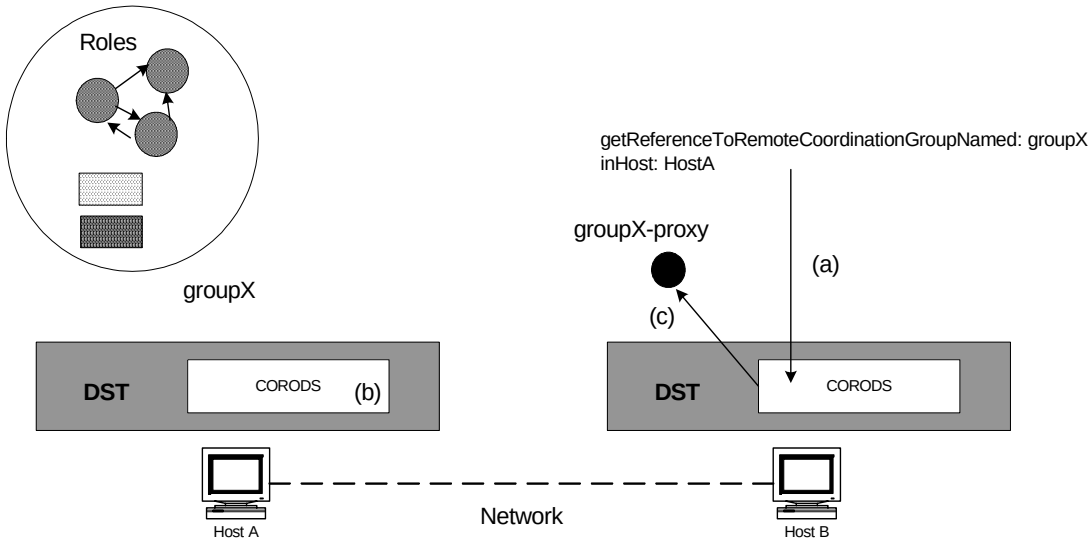


Figure 4.22 : Obtaining a remote reference to a group

Additionally CORODS provides two operations that can be used to locate group classes in hosts, they are: *allCoordinationGroupClassesNamesInHost:* <Host Name> and *allCoordinationGroupNamesInHost:* <Host Name>. The first operation returns the names of all the existing group classes in the host named <Host Name> and the second operation returns the names of all the existing groups created in the host named <Host Name>.

4.6.3 The CORODS service's IDL

The IDL language introduced in CORBA is a neutral declarative language used to describe interfaces that client objects call and object implementations provide. All ORBs independently of the specific language they support (i.e., Java, Smalltalk, etc.) "speak" IDL and use IDL to define interfaces for accessing remotely objects. The interface definition specifies operations the object is prepared to perform, the input and output parameters required and any exception that might be generated. In the IDL language the description

The main elements that constitute the CORBA IDL are [Orfa96a]: modules, interfaces, operations and data types. The modules provide a name space to group a set of interfaces. An interface defines a set of operations that a client can invoke on an object, like a class definition. The IDL defines the operations signatures: parameters and results types. A parameter has a mode that indicates whether the value is passed from the client to the server (in), from the server to the client (out), or both (inout). The parameter also has a type. The operation's signature optionally defines the exceptions that a method raises when it detects an error. An interface may have also attributes, they define accessors and mutators operations for the object. An attribute can be read-only, in which case the implementation only provides an accessor operation. An interface can be derived from one or more interfaces, which means IDL supports multiple inheritance. The operations denote services that clients can invoke

We will present in this section only the IDL specifications of the services related with the creation of groups in CORODS. Our purpose is not to present the complete IDL specification of all the CORODS services but to give an idea to the reader about how these services are specified in the IDL language of CORBA.

Group Creation Operations

In (**Figure 4.23**) we can see the IDL specification of the operations related with the creation of the coordination groups. The IDL specifications of the creation services are defined within a module named CORODS. The CORODS module contains a unique interface named CORODSInterface. The IDL specification contains the specification of the four creation operations defined in **4.6.1**. The first two operations (lines 5 and 9) specify the operations for the creation of groups classes and the last two (lines 14 and 21) specify the operations for the creation of groups. The operations for the creation of groups return as a result an object that implements the GroupInterface interface IDL. The GroupInterface interface IDL contains the IDL specification of all the operations in the CoLaSD model to manipulate and modify the groups (i.e., to define a role, to define a coordination rule, to add a variable, etc.). In the example the pragma selector that appear in the specification of the creation operations indicates a mapping between a selector name and a name used in the IDL specification. In the DST terminology this pragma is called a Selector pragma. In general pragmas are implementation dependent messages to the IDL compiler. The Selector pragma is a DST specific pragma.

```
1.module CORODS {
2.  interface CORODSInterface {
3.
4.  #pragmaselector createCoordinationGroupClassNamed
5.       createCoordinationGroupClassNamed:
6.       void createCoordinationGroupClassNamed (in symbol aGroupName );
7.
8.  #pragmaselector createCoordinationGroupClassNamedInHost
9.       createCoordinationGroupClassNamed:inHost:
10.      void createCoordinationGroupClassNamedInHost
11.                        (in string aGroupName, in symbol aHost);
12.
13. #pragma selector
14. createCoordinationGroupNamedForCoordinationGroupClassNamed
15. createCoordinationGroupNamed:forCoordinationGroupClassNamed:
16. GroupInterface
17.       createCoordinationGroupNamedForCoordinationGroupClassNamed
18.       (in symbol aGroupName, in symbol aCoordinationGroupClassName);
19.
20. #pragmaselector
21. createCoordinationGroupNamedForCoordinationGroupClassNamedInHost
22. createCoordinationGroupNamed:forCoordinationGroupClassNamed:inHost:
23. GroupInterface
24.      createRemoteCoordinationGroupNamedForCoordinationGroupNamedInHost
25.      (in symbol aCoordinationGroupName,
26.      in symbol aCoordinationGrouoClassName,in symbol aHostName);
27. ...
```

Figure 4.23 : Group classes creation's IDL

4.7 CORODS - The Administrator

At the beginning of this chapter we introduced the Administrator pattern [Papa95a] to motivate the work presented in this chapter. We will use the same example to illustrate the use of the CORODS coordination service. We assume the existence of a group class named AdministratorPattern containing the specification of the coordination of the Administrator system. We will define a specific scenario composed by one administrator and two workers, all the three running in three different machines in a local network: Ziyal, Albert and Globi. The first worker runs in Globi, the second worker in Albert; and the administrator in Ziyal. A coordination group named AdminGroup is created in Ziyal. The cordination group enforces the coordination of the distributed workers and the administrator object. We assume of course that the CORODS service was already installed in the three machines. In (**Figure 4.27**) we can visualize the scenario described before.

Group Creation and Enrolment of Participants

Ziyal

```
1.corods := ORBObject resolveInitialReferences: #CORODS.
2.adminGroup := corods createCoordinationGroupNamed: #AdminGroup
3.                    forCoordinationGroupClassNamed: #AdministratorPattern.
4.administrator:= Administrator new.
5.adminGroup addParticipant: administrator toRoleNamed: #Administrator.
```

Globi

```
1.corods := ORBObject resolveInitialReferences: #CORODS.
2.adminGroup := corods getReferenceToCoordinationGroupNamed: #AdminGroup
3.                    inHost: #Ziyal.
4.worker1 := Worker new.
5.adminGroup addParticipant: worker1 toRoleNamed: #Worker.
```

Albert

```
1.corods := ORBObject resolveInitialReferences: #CORODS.
2.adminGroup := corods getReferenceToCoordinationGroupNamed: #AdminGroup
3.                     inHost: #Ziyal.
4.worker2 := Worker new.
5.adminGroup addParticipant: worker2 toRoleNamed: #Worker.
```

Figure 4.24 : The Administrator Pattern Scenario

In (**Figure 4.24** in Ziyal) we can see how the AdminGroup group is created in the machine Ziyal (lines 2 and 3). The AdminGroup group created becomes then potentially accessible to participants running in other machines. To participate in the group the distributed objects must join one of the roles specified in the group (i.e., Administrator or Worker). In line 5 we can see how an administrator participant joins the role Administrator in the group. It is important to remember that each role specified in a coordination group may specify

a role interface. The role interface defines the minimum requirements for an active object to play a role. We assume that the distributed objects created in the example satisfy the role interfaces of the roles they want to play.

To obtain a reference to the AdminGroup group created in Ziyal from a remote machine (i.e., Globi or Albert) we use the CORODS service. In (line 2 - machines Globi and Albert) we can see how the CORODS service is used in Globi and Albert to obtain a remote reference to the AdminGroup group created in the machine Ziyal. In line 5 the two worker objects in the machines Globi and Albert request the AdminGroup group to join the role workers. To enrol in a role the distributed objects send the message *addParticipant:*<Active Object> *toRoleNamed:* <Role Name> to the group with the reference to the distributed object and the name of the role as arguments.

From the users point of view it is not too important to know where the groups are created. The most important is to be able to access and manipulate them as they were local to the machines where the manipulations are done. With the CORODS coordination service it is possible to coordinate objects that find physically distributed through the network benefiting at the same time from the advantages of the use of coordination models and languages. The separation of the coordination and the computation aspects in the specification and construction of the distributed systems built using CORODS facilitate their understanding, modification and evolution.

4.8 CORODS implementation Requirements and Limitations

To make groups and participants remotely accessible through the CORODS service it is necessary to make the CoLaSD model CORBA compliant. To make the CoLaSD model CORBA compliant it is necessary to make all the different elements that compose the CoLaSD model (i.e., groups, roles, coordination rules, participants, etc.) CORBA compliant. For every element in the CoLaSD model we must first define the IDL interfaces for the services they offer and second to make the element a factory. Remember that for a class to be consider as a factory in CORBA it must have an instance method named abstractClassID. This method should return the appropriate UUID-Universal Unique Identifier which uniquely identifies the class.

In the CORBA standard the IDL interfaces are the key element, they allow service providers to specify in a neutral language the interface of the service they provide. An IDL defines a contract binding providers of services to their clients. We will not present the IDL specifications of all the elements that composed the CoLaSD coordination model, what it is important to know is that these IDLs are necessary in order to make the groups remotely accessible. We show below as example the IDL interface specification of the roles in the coordination groups (**Figure 4.25**). In the example the role IDL interface includes the IDL specification of the method *defineVariable:initialValue:* used to specify role variables in a role (line 5).

```
1.interface RoleInterface : CosLifeCycle::LifeCycleObject {
2.
3.       SmalltalkObject defineVariable (in SmalltalkObject aSymbol);
4.
5.       #pragma  selector defineVariableInitialValue
6.                defineVariable:initialValue:
7.                SmalltalkObject defineVariableInitialValue
8.                     (in SmalltalkObject aSymbol,in SmalltalkObject aValue);
9.
10.      SmalltalkObject includesVariableNamed (in SmalltalkObject aSymbol);
11.
12.      SmalltalkObject addParticipant (in SmalltalkObject aParticipant);
13.
14.      ...
15.};
```

Figure 4.25 : Role's IDL Interface

To identify the IDL interface associated with an group element (i.e., a role, a coordination rule, etc.) in CORBA we must specify a method CORBAName in the corresponding class implementing the element. In (**Figure 4.26**) we can see the specification of the CORBAName method for the role element. The method specifies that the IDL interface associated with the role element is the RoleInterface interface that finds in a module called Cords in the IDL repositories.

```
1.CORBAName
2.       ^#'::Corods::RoleInterface'
```

Figure 4.26 : CORBAName method

Dynamicity

Between the benefits that a CORBA ORB has is the possibility to define static and dynamic method invocations. A CORBA ORB lets the users either to statically define method invocations at compile time or to dynamically discover them at run time. For static method invocations it is necessary to compile the IDL specification with an IDL compiler that generates client and servers stubs (called skeletons). The stubs define how clients invoke the corresponding services and how servers process the corresponding invocations. For the dynamic method invocation the ORB provides a run time binding mechanism for servers that need to handle incoming method calls for components that do not have IDL compiled skeletons or stubs.

Because of the dynamic characteristics of the CoLaSD model (i.e., new roles can be added to the groups, new coordination rules can be added or removed, etc.), the static approach of method invocation does not fix well with our approach. The use of the static approach will imply the recompilation of IDLs and the redistribution of the generated stubs to the clients every time a new modification is done to a coordination group. In CORODS we have decided to exclusively use dynamic method invocation to manage changes in the specification of the coordination groups. Whenever the interface of a CoLaSD element is modified at

run time we recompile dynamically the IDL interface and we re-store the new specification in the Interface Repository. We have defined a unique Interface Repository in which we store the IDL specifications of all the groups classes created in the network.

4.9 Conclusions and Contributions

Traditionally the coordination layer of Open Distributed System (ODS) have been built using distributed object-oriented languages. Building distributed object-oriented systems is very complicated. They require ideally that four main issues be addressed: 1) Independent Failure, 2) Unreliable Communication, 3) Insecure Communication and 4) Costly Communication. Building distributed object-oriented systems is still difficult today not only because they require that engineers address the four issues exposed before, but because existing distributed object-oriented languages provided limited support for their specification, construction and evolution. Engineers must take care of connecting the distributed objects and specifying their interactions and synchronizations; and such connections, interactions and synchronizations change when the requirements of the applications change. Evolution is the most difficult requirement to meet since not all the application requirements can be known in advance.

Several solutions have been proposed to tackle the complexity of building such kinds of systems, among them CORBA. The CORBA middleware proposed by the OMG provides a standard for interoperability between independently developed components across networks of computers. The OMG focused on distributed objects as a vehicle for system integration. The CORBA middleware provides the necessary support for building and executing ODS. In performing its task CORBA relies on Object services which are responsible for performing general object management operations such as creation of objects, access control, track of relocated objects, etc. Nevertheless, CORBA has proven its limitation to support the evolution of those systems. One of the main problems is that the computation code of the objects that compose those systems and they way they are composed are mixed within the objects code. This mixing of concerns makes the distributed systems built difficult to understand, modify and customize. The idea of separating the coordination and computation aspects in concurrent and distributed systems introduced by [Gele92a] provides an extremely interesting approach to tackle this problem. We consider the CoLaS coordination model as a good candidate to integrate with the CORBA model. This integration will give CORBA the necessary support to build and evolve ODS. To support the specification of the coordination in distributed systems we extended the CoLaS coordination model to take into consideration the possibility of failures in the participants common to distributed systems. The CoLaSD model (i.e., the new extended model) provides separation of concerns between computation and coordination in ODS simplifying their understanding, modification and customization.

We introduced in this chapter CORODS, a coordination service for distributed objects based on the CoLaSD coordination model. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network. We showed in this chapter how the CORODS coordination service was implemented in top of DST [Cinc94a] a middleware framework CORBA 2.0 compliant. We illustrated the use of the CORODS services using the Administrator example [Papa95a], an architectural pattern used to structure distributed systems. The administrator is an object that uses a collection of “worker” objects to service requests received from clients.

Contributions

The main contribution of this chapter to thesis is:

- We introduce CORODS a coordination service for distributed objects for CORBA[OMG95a]. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network. By using the CORODS service it is possible to perform coordination in distributed object systems guaranteeing at the same time their interoperabililty. The CORODS service supports: a clear separation of the computation and the coordination concerns (i.e., the coordination is encapsulated in the coordination groups and the computation in the distributed participants), the encapsulation of the coordination behavior (i.e., all the coordination behavior is specified in the coordination groups), the specification of multi-object coordination (i.e., multiple distributed participants can participate in the coordination groups, multiple roles can be defined), the specification of high-level coordination abstractions (i.e., rules are used to specify coordination of groups of collaborating distributed objects, the coordination low level details are ignored), the evolution of the coordination (i.e., new coordination patterns can be defined and existing modified), the reuse of coordination abstractions (i.e.,coordination patterns can be reused independently of their distributed participants) and the incremental specification of the coordination (i.e., coordination can be specified using existing coordination).

CHAPTER 5

OpenCoLaS: a Coordination Framework for CoLaS Dialects

Several modifications have been made to the CoLaS coordination model and language since the first time it was introduced in [Cruz99a]. These modifications have mainly concerned the coordination rules that compose the model. Some rules were removed, some others were added and the semantics of some others were changed.

The modifications were always motivated by two main goals: first to obtain a clear separation of coordination and computation concerns in the model and second to define the minimal set of coordination rules necessary to solve the largest number of coordination problems.

We were not been able for long time to justify the existence of the coordination rules that composed the CoLaS coordination model. Why these coordination rules and not others? Where do these coordination rules come from? Are all these coordination rules necessary? These questions appeared all the time. Our main justification to the existence of these coordination rules was that they were purely empirical, the coordination rules that composed the CoLaS coordination model were the result of a long experimental work solving coordination problems in several case study systems we built. No formal justification was proposed until now for the existence of these coordination rules.

Related work to CoLaS like Synchronizers [Frol93a], Composition Filters [Berg94a] and Moses [Mins97a] suffer from the same weakness. These works introduce coordination models based on the same reflective approach like CoLaS and do not provide for a formal specification nor a justification of the existence of their coordination rules. It is important to remember that reflective coordination models perform coordination by intercepting and controlling base operations in the systems. CoLaS is based on the interception of the messages exchanged by the participants of the coordination groups. In order to provide answers to these open questions we developed OpenCoLaS, a framework to experiment in the specification of CoLaS like coordination models and languages. The idea behind the OpenCoLaS framework is to "open" the CoLaS coordination model and language in a way that allows one to experiment with the specification of coordination rules.

We have divided the presentation of this chapter into four parts:

In the first part of this chapter we briefly introduce the different coordination rules that compose the CoLaS coordination model: cooperation, reactive and proactive rules. Cooperation rules are used to specify cooperation actions between participants. Reactive rules are used basically to specify synchronizations over the occurrence of actions in participants. And, proactive rules are used to specify proactions [Andr96a] in groups independently of the messages exchanged by the participants.

In the second part of this chapter we introduce the OpenCoLaS framework, an object-oriented framework that allows users to experiment with the specification of coordination rules in CoLaS like coordination models. To illustrate the structure and the use of the framework we will use as example the Electronic Vote [Mins97a], an example introduced in chapter 3 of this thesis. We will describe a possible solution to this problem using the OpenCoLaS framework. The idea is to show how the Electronic Vote problem can be solved using coordination rules in CoLaS and how the same problem can be solved in the OpenCoLaS framework.

In the third part of this chapter we analyse the evolution of the CoLaS coordination model. We compare the specification of the coordination rules in the three main research publications written in the CoLaS model [Cruz99a][Cruz01][Cruz02]. All along the presentation we expose the reasons that motivated the modifications introduced to the original model [Cruz99a]. The OpenCoLaS framework is used to specify the different coordination rules of the different versions of the model.

In the fourth part of this chapter we use the OpenCoLaS framework to compare the specification of the coordination rules in CoLaS with the coordination rules introduced in similar approaches: Synchronizers [Frol93a], Composition Filters [Berg94a] and Moses [Mins97a]. We illustrate for each related model how their coordination rules can be specified in OpenCoLaS and we compare these results with the specification of the CoLaS model.

Finally at the end of this chapter we present our conclusions pointing out the main contributions of the chapter to the thesis.

5.1 Coordination Rules in CoLaS

The coordination rules specify the rules governing the coordination of a coordination group. CoLaS defines actually three types of coordination rules: cooperation, reactive and proactive rules. We will present here briefly the three different types of coordination rules, for a complete specification of the rules refer to chapter 3 of this thesis.

5.1.1 Cooperation Rules

Cooperation rules are rules that define implications between participant actions. They specify actions that participants must do when they receive method invocations corresponding to the behaviors specified in the rules. Participants react to these method invocations only during the time they play roles in the groups. Cooperation rules have the form <Role> *defineBehavior:* <Message> *as:* <Coordination Actions>. The <Role> specifies the role to which the rule is associated, the <Message> specifies the signature of the method (i.e, behavior) associated with the rule and the <Coordination Actions> specifies a block of coordination statements. The <Coordination Actions> statements include actions that manipulate the coordination state, synchronous recursive method invocations, method invocations to participants and to roles and the manipulation of the participants in roles.

5.1.2 Reactive Rules

Reactive rules are rules that depend for their application on the messages exchanged by the participants of the groups. The CoLaS model defines actually two forms of reactive rules: interception rules and synchronization rules. Both types of rules are evaluated at specific evaluation points defined by the model. CoLaS defines four evaluation points: atArrival (at the arrival of a method invocation), atSelection (after the selec-

tion of a method invocation), atSent (after the sent of a method invocation to another participant) and atEnd (after the execution of a method invocation).

Interception Rules

Interception rules are rules that change the normal processing of the method invocations in the participants. They specify actions that modify the coordination state. They have the form <Role> <*Interception Operator*> <Message> *do:* <Coordination State Actions>. There are four interception operators: interceptAtArrival, interceptAtSelection, InterceptAtSent, interceptAtEnd. Each interception operator corresponds to an evaluation point defined in the model. The <Coordination State Actions> specify actions that modify the coordination state of the group: basically modifications to state variables.

Synchronization Rules

Synchronization rules specify synchronization constraints on the execution of the method invocations received by the participants. There are two forms of synchronization rules: ignore and disable rules. Ignore rules have the form <Role > *ignore:* <Message> *if:* <Synchronization Condition> and disable rules have the form <Role> *disable:* <Message> *if:* <Synchronization Condition>. The ignore rule specifies that method invocations corresponding to the message <Message> must be ignored when received (i.e., not stored into the participant's mailbox) if the condition specified in the <Synchronization Condition> validates to true. The disable rule specifies that the execution of the method invocation corresponding to the message <Message> must be delayed (executed later) if the condition specified in the <Synchronization Condition> validates to true. The <Synchronization Condition> corresponds to a boolean expression referring exclusively to: information related to the received method invocation (i.e. receiver, arguments, etc.), the coordination state and the current time in the system.

5.1.3 Proactive Rules

Proactive rules are rules that depend for their application exclusively on the coordination state of the group and not on the method invocations received or sent by the participants. The CoLaS model defines actually a unique form of proactive coordination rule. The rule has the form <Coordination Group> *validate*: <Coordination State Condition> *do*: <Coordination Actions>. The proactive rule guarantees that certain <Coordination Actions> are carried out by the group if a certain <Coordination State Condition> validates to true. The <Coordination Actions> in the specification of the proactive rule correspond to the same actions that those specified in cooperation rules and the <Coordination State Condition> to the same condition specified in the reactive rules, excluding of course in the condition information related to the received method invocation which in proactive rules do not have any sense because they are not triggered by the reception of method invocations.

5.2 The OpenCoLaS Framework

OpenCoLaS is a framework that allows one to specify coordination rules for CoLaS like coordination models and languages. The OpenCoLaS defines an abstract class named CoordinationRule (**Figure 5.1**) as the root of all the possible types of coordination rules. OpenCoLaS defines three types of coordination rules: behavioral, reactive and proactive rules.

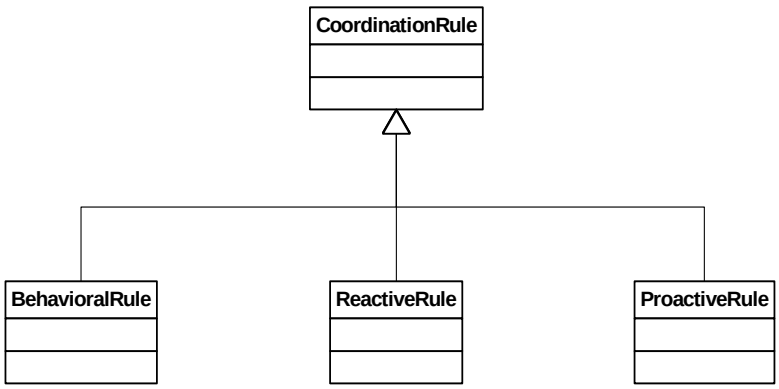


Figure 5.1 : The OpenCoLaS Framework

For each type of coordination rule OpenCoLaS defines an abstract class (subclass of the CoordinationRule class) containing all the necessary support to specify new coordination rules of the corresponding type: BehavioralRule for behavioral rules, ReactiveRule for reactive rules and ProactiveRule for the proactive rules. We will describe in the next subsections how each type of coordination rule is specified in OpenCoLaS.

We will use as example the CoLaS model to show how the coordination rules that compose the model can be specified in OpenCoLaS. Specifically we will use the Electronic Vote problem presented in chapter 3 of this thesis to illustrate how instances of those coordination rules can be created in OpenCoLaS to solve a concrete coordination problem.

5.2.1 The Electronic Vote [Mins97a]

We show in (**Figure 5.2**) a possible solution in CoLaS to the Electronic Vote problem presented in chapter 3 of this thesis. In the Electronic Vote an open group of participants is requested to vote on a specific issue. Every participant in the group can initiate a vote on any issue it chooses. Each participant votes by sending the result of its vote to the participant initiator of the vote. The system must guarantee that the vote is fair: (1) a participant can vote at most once and only within the voting period, (2) the counting is done correctly and (3) the result of the vote is sent to all the participants after the end of the voting period.

```
14.[1] Voter defineBehavior: 'startVote:anIssue' as:
15.      [group voteInProgress:true.
16.      Voter voteOn: anIssue].
17.
18.[2] Voter defineBehavior: 'voteOn:anIssue' as:
19.      [sender vote:(self opinion: anIssue)].
20.
21.[3] Voter defineBehavior: 'vote:aVote' as:
22.      [(aVote = 'Yes')
23.          ifTrue: [group numYes++]
24.          ifFalse: [group numNot++].
25.      sender hasVoted: true ].
26.
27.[4] Voter defineBehavior: 'stopVote' as:
28.      [group votePeriodExpired: true ].
29.
30.[7] Voter ignore: 'vote:aVote' if:
31.      [group voterPeriodExpired or sender hasVoted].
32.
33.[8] Voter disable: 'startVote:anIssue' if:
34.      [group voteInProgress ].
35.
36.[9] ElectronicVote
37.      validate: [group voteInProgress and group votePeriodExpired ] do:
38.      [(group numYes = group size)
39.          ifTrue: [Voter voteResult: 'Yes']
40.          ifFalse: [Voter voteResult: 'No'].
41.      Voter do:[:each | each hasVoted: false].
42.      group votePeriodExpired:false.
43.      group voteInProgress: false.
44.      group numYes: 0 ].
```

Figure 5.2 : The Electronic Vote in CoLaS

5.2.2 Behavioral Rules

The behavioral rules in OpenCoLaS are specified in two steps. In the first step we create behavioral rule classes corresponding to the cooperation rules that we want to specify. In the second step we create instances of the behavioral rules using the created behavioral rule classes.

Rules Class Creation

The BehavioralRule class in the OpenCoLaS framework defines a unique rule class creation method named *defineRule:* <Rule Name> to create behavioral rule classes. The behavioral rule classes created are subclasses of the BehavioralRule class. To illustrate how behavioral rules can be specified in OpenCoLaS we will use as example the cooperation rules specified in the CoLaS model. The CoLaS model specifies one

type of behavioral rule (i.e., cooperation rule) named defineBehavior. In (**Figure 5.3**) we can see how a behavioral rule class named DefineBehavior is created in the OpenCoLaS framework.

```
BehavioralRule defineRule: DefineBehavior
```

Figure 5.3 : CoLaS DefineBehavior rule in OpenCoLaS

Rules Instantiation

The second step in the specification of behavioral rules consists of creating behavioral rule instances using the created behavioral rule class. The BehavioralRule class in the OpenCoLaS framework specifies a unique method *message:* <Message> *actions:* <Coordination Actions> for the creation of behavioral rules instances. The argument <Message> specifies the signature of the method (i.e., behavior) associated with the rule and the argument <Coordination Actions> specifies a block of statements. The statements in the <Coordination Actions> include: (1) actions that manipulate the coordination state, (2) synchronous recursive method invocations, (3) method invocations to participants and to roles and (4) the manipulation of the participants in roles.

Using behavioral Rules

To create behavioral rules we use the DefineBehavior rule class. For each behavioral rule instance to be created we specify the method signature <Message> associated with the rule and the <Coordination Actions> that define the coordination behavior. In (**Figure 5.4**) we show how four coordination behavior rules defined in the Electronic Vote example are defined in OpenCoLaS: 1) startVote: <anIssue> (rule [1]), 2) voteOn: <anIssue> (rule [2]), 3) vote: <aVote>, (rule [3]) and 4) stopVote (rule [4]).

```
1.[1] DefineBehavior message: 'startVote:anIssue'
2.       actions:
3.            [group voteInProgress:true.
4.            Voter voteOn: anIssue initiator: receiver].
5.
6.[2] DefineBehavior message: 'voteOn:anIssue'
7.       actions:
8.            [sender vote:(self opinion: anIssue)].
9.
10.[3] DefineBehavior message: 'vote: aVote'
11.      actions:
12.           [(aVote = 'Yes')
13.               ifTrue: [group numYes++]
14.               ifFalse: [group numNot++].
15.           sender hasVoted: true ].
16.
17.[4] DefineBehavior message: 'stopVote'
18.      actions:
19.           [group votePeriodExpired: true ]
```

Figure 5.4 : Behavioral Coordination Rules Instantiation for the Electronic Vote

5.2.3 Reactive Rules

The reactive rules in OpenCoLaS are specified in two steps. In the first step we create the reactive rule classes corresponding to the reactive rules that we want to specify. In the second step we create the reactive rule instances using the created reactive rule classes. For each reactive rule class created is necessary to specify the semantics of the reactive rule and the validation point at which the rule must be evaluated.

Rules Class Creation

The basic ReactiveRule class in the OpenCoLaS framework defines a rule class creation method named *defineRule*: <Rule Name> *semantics*: <Rule Semantics Actions> *validationPoint:* <Validation Point> to create reactive rule classes (**Figure 5.5**). The <Rule Name> argument specifies the name of the rule, the <Rule Semantics Actions> argument specifies the semantics of the rule and the <Validation Point> argument the validation point at which the rule must be evaluated.

```
ReactiveRule      defineRule: <RuleName>
                  semantics: <Rule Semantics Actions>
                  validationPoint: <Validation Point>
```

Figure 5.5 : Reactive rules in OpenCoLaS

To understand the specification of reactive rules in OpenCoLaS it is important to understand that reactive rules are rules that depend for their evaluation on the method invocations received by the participants in the groups and that the participants are the entities in charge of their enforcement. The <Rule Semantics Actions> specify actions that transform the processed method invocations, like meta-actions in the sense of [Kicz91a] (i.e., actions that allow the modification of the language). It is possible to define three types of semantics actions: 1) actions that modify the arguments of the method invocations, 2) actions that modify the mailboxes of the participants (i.e. to remove a method invocation from a mailbox or to put a method invocation into a mailbox) and 3) actions that transform method invocations into other method invocations, in particular into a NoMessage method invocation.

Concerning the validation point <Validation Point> that appears in the specification of the rule, OpenCoLaS defines actually four evaluation points: atArrival, atSelection, atSent, atEnd. The four validation points correspond to the same validation points defined in the CoLaS model. They specify different moments during the processing of the method invocations in the participants.

To refer to the method invocation currently processed by a participant in the <Rule Semantics Actions> OpenCoLaS defines a pseudo variable called *message*. It is possible to request for the arguments and the selector of the method invocation currently processed by sending the message *arguments* and *selector* respectively to the pseudo variable message.

Concerning the second kind of semantics actions that appear in the rules, they correspond to actions that modify the mailboxes of the participants: (1) to add a method invocation into a participant mailbox and (2) to remove a method invocation from a participant mailbox. OpenCoLaS defines a pseudo variable named *mailbox* to refer to a participant mailbox. To add a method invocation to a participant mailbox one must send the message *addMessage*: <Method Invocation Request> to the mailbox pseudo variable with the method invocation request to add as an argument. To remove a method invocation from a participant mailbox one

must send the message *nextMessage* to mailbox pseudo variable. The *nextMessage* method invocation returns the next method invocation in the participant maibox that validates the internal synchronization policy specified in the participant.

The NoMessage method invocation specified in the third kind of semantics actions, is an special message used internally in OpenCoLaS to indicate that a method invocation must not continue to be processed. It is extremely important to precise that the last action in a <Rule Semantics Actions> specification should always be to return a method invocation: the same received, a new or a NoMessage method invocation. The method invocation returned will continue to be processed by the participant if the method invocation returned is not a NoMessage method invocation

For simplicity OpenCoLaS defines a second rule creation method in the ReactiveRule class named *defineRule*: <Rule Name> *validationPoint:* <Validation Point>. In this rule the semantics associated with the new reactive rule is defined by default as the return of the same method invocation being processed.

To illustrate how reactive rules can be specified in OpenCoLaS we will use as example the reactive rules specified in the CoLaS model. The CoLaS model specifies six types of reactive rules: InterceptAtArrival, InterceptAtSelection, InterceptAtSent, InterceptAtEnd, Disable and Ignore. In (**Figure 5.6**) we can see how the reactive rules of the CoLaS model are specified in the OpenCoLaS framework.

```
1.ReactiveRule
2.       defineRule: Ignore
3.       semantics: [^NoMessage new ]
4.       validationPoint: OpenCoLaS atArrival.
5.
6.ReactiveRule
7.       defineRule: Disable
8.       semantics: [mailbox addMessage: message.
9.                  ^NoMessage new ]
10.      validationPoint: OpenCoLaS atSelection.
11.
12.ReactiveRule
13.      defineRule: InterceptAtArrival
14.      validationPoint: OpenCoLaS atArrival.
15.
16.ReactiveRule
17.      defineRule: InterceptAtSelection
18.      validationPoint: OpenCoLaS atSelection.
19.
20.ReactiveRule
21.      defineRule: InterceptAtSent
22.      validationPoint: OpenCoLaS atSent.
23.
24.ReactiveRule
25.      defineRule: InterceptAtEnd
26.      validationPoint: OpenCoLaS atEnd.
```

Figure 5.6 : CoLaS Reactive Coordination Rules in OpenCoLaS

Ignore (**Figure 5.6** line 1): the rule semantics associated with the rule specifies the return of a NoMessage method invocation. The NoMessage method invocation indicates internally to the participant that the method invocation processed should not continue to be processed. The rule must be evaluated at the atArrival validation point.

Disable (**Figure 5.6** line 6): the rule semantics associated with the rule specifies the return of a NoMessage method invocation after re-inserting the method invocation processed in the participant's mailbox. The NoMessage method invocation indicates internally to the participant that the method invocation processed should not continue to be processed. The rule must be evaluated at the atAccept validation point.

InterceptAtArrival (**Figure 5.6** line 12), InterceptAtSelection (**Figure 5.6** line 16), InterceptAtSent (**Figure 5.6** line 20) and InterceptAtEnd (**Figure 5.6** line 24): the specifications of the four rules do not associate any rule semantics to the rules, by default they return the same method invocation processed. the InterceptAtArrival rule must be evaluated at the atArrival validation point, the InterceptBeforeExecution rule must be evaluated at the atSelection validation point, the InterceptAtSent rule must be evaluated at the atSent validation point and the InterceptAfterExecution must be evaluated at the atEnd validation point.

It is interesting to remark that in the specification of the CoLaS reactive rules two of the rules must be evaluated at the same validation point: the InterceptAtArrival and the Ignore rules at the atArrival validation point. Which rule should be evaluated the first? is a question that must be solved. OpenCoLaS allows users to specify evaluation orders to avoid evaluation conflicts between rules. We will show below how reactive rules are evaluated in CoLaS and how do we can specify evaluation orders between rules in the OpenCoLaS framework.

Rules Instantiation

The second step in the specification of reactive rules is to create reactive rule instances using the created reactive rule classes. The ReactiveRule class in OpenCoLaS specifies a generic creation method message: <Message> *condition*: <Coordination Condition> *actions*: <Coordination State Actions> (**Figure 5.7**). The argument <Message> specifies the signature of the method associated with the rule. The argument <Coordination Condition> specifies a boolean expression referring to information related to the received method invocation and the coordination state. And, the argument <Coordination State Actions> specify actions that modify the coordination state of the group (i.e., modify the values to the state variables).

```
<Reactive Rule >  message: <Message>
                  condition: <Coordination Condition>
                  actions: <Coordination State Actions>
```

Figure 5.7 : Instantiation of reactive rules in OpenCoLaS

For simplicity OpenCoLaS defines two other creation methods for reactive rules, one in which the <Coordination State Action> are not specified and thus by default an empty sequence of actions is assumed and another in which the <Coordination Condition> is not specified and thus by default a boolean expression true is assumed (i.e, the rule always evaluates to true).

In (**Figure 5.8**) we show how a Ignore and a Disable reactive rules defined in the Electronic Vote example are defined in OpenCoLaS. Both rules do not specify <Coordination State Actions> actions, implicitly they have associated an empty sequence of actions.

```
20.[7] Ignore message: 'vote:aVote'
21.      condition:
22.           [group voterPeriodExpired or sender hasVoted].
23.
24.[8] Disable message: 'startVote:anIssue'
25.      condition:
26.           [group voteInProgress].
```

Figure 5.8 : Reactive Coordination Rules Instantiation for the Electronic Vote

The solution to the Electronic Vote problem showed in (**Figure 5.2**) does not include CoLaS Interception rules: InterceptAtArrival, InterceptAtSelection InterceptAtSent, InterceptAtEnd. Interception rules modify the coordination state at different moments during the processing of method invocations by the participants. We illustrate in (**Figure 5.9**) how instances of the interception rules can be created in OpenCoLaS. We do not specify specific messages and actions to these rules, they depend on the particular coordination problem in which they are defined. It is important to remark that we do not specify <Coordination Condition>s during their creation, this means that they always validate to true. The interception rules are evaluated at the evaluation points indicated by their names.

```
InterceptAtArrival message: ..
        actions: [..]

InterceptBeforeExecution message: ..
        actions: [..]

InterceptAtSent message: ..
        actions: [..]

InterceptAfterExecution message:
        actions: [..]
```

Figure 5.9 : Instantiation of Interception Rules

5.2.4 Proactive Rules

The proactive rules in OpenCoLaS are specified in two steps. In the first step we create the proactive rule classes corresponding to the proactive rules that we want to specify. In the second step we create the proactive rule instances using the created proactive rule classes.

Rules Creation

The basic ProactiveRule class in the OpenCoLaS framework defines a unique rule class creation method *defineRule*: <RuleName> to create proactive rule classes. The proactive rule classes created are subclasses of the ProactiveRule class. To illustrate how proactive rules can be specified in OpenCoLaS we will use as example the proactive rules specified in the CoLaS model. The CoLaS model specifies a unique type of proactive coordination rule: Validate. In (**Figure 5.10**) we can see the how a proactive rule class named Validate is created in the OpenCoLaS framework.

```
ProactiveRule defineRule: Validate.
```

Figure 5.10 : CoLaS Proactive Rule in OpenCoLaS

Rules Instantiation

The second step in the specification of proactive rules is to create proactive rule instances using the created proactive rule classes. The ProactiveRule class in OpenCoLaS specifies a generic creation method *condition*: <Coordination State Condition> *actions:* <Coordination Actions> to create proactive rule instances. The <Coordination State Condition> corresponds to the same condition specified in the reactive rules, excluding of course in the condition information related to the received method invocation which in proactive rules do not have any sense. The argument <Coordination Actions> corresponds to the same actions that those specified in cooperation rules. In (**Figure 5.11**) we show how a Validate proactive rules defined in the Electronic Vote example is defined in OpenCoLaS.

```
27.[9] Validate
28.      condition: [group voteInProgress and group votePeriodExpired ]
29.      actions:
30.          [(group numYes = group size)
31.              ifTrue: [Voters voteResult: 'Yes']
32.              ifFalse: [Voters voteResult: 'No'].
33.          Voter do:[:each | each hasVoted: false].
34.          group votePeriodExpired:false.
35.          group voteInProgress: false.
```

Figure 5.11 : Proactive Coordination Rules for the Electronic Vote

5.2.5 Evaluation of Coordination Rules in CoLaS

To complete the presentation of the OpenCoLaS framework we must explain how the coordination rules defined are evaluted in the OpenCoLaS framework. We will first have a look on how and when Coordination Rules are evaluated in the CoLaS coordination model. In CoLaS the method invocations received by participants are stored into their mailboxes until they are ready to process them. The selection of a method invocation from a participant mailbox depends of the internal synchronization policy associated with the participant. Remember that synchronization policies are a mechanism defined in the participant's model to ensure the consistency of the participant state when method invocations are executed concurrently. In Co-

LaS the participants follow a mutual exclusion synchronization policy, only one method invocation is executed at the time by a participant. In other words method invocations are processed by the participants sequentially.

Cooperation Rules Evaluation

Cooperation rules are rules that define coordination behavior for participants in groups. Each cooperation rule specifies the signature of the method (i.e., behavior) to which the rule is associated. Cooperation rules are executed in response to method invocations received by the participants. When a method invocation corresponding to the signature specified in a rule is selected by a participant for execution (i.e., when the rule validates the synchronization policy) the behavior specified in the rule is executed.

Reactive Rules Evaluation

Reactive rules are rules that depend for their execution of the method invocations received by the participants. Each reactive rule specifies the signature of the method to which the rule is associated and a validation point which indicates the moment at which the rule should be evaluated. In CoLaS four different validation points are defined for the evaluation of reactive rules: atArrival, atSelection, atSent and atEnd. At each validation point reactive rules specified as associated with the validation point are matched against the selector of the method invocation processed. Rules matching the method invocation are then executed.

Solving Evaluation Conflicts

The execution of the preselected rules is made on the basis of evaluation priorities associated with the different reactive rules. Rules with higher priorities are executed first than rules with lower priorities. The maximum priority that can be specified for a rule is defined by the OpenCoLas constant maximumEvaluationPriority which is equal to 100 and the minimum priority by the constant minimumEvaluationPriority which is equal to 1. By default when no priority is associated with a reactive rule the evaluation priority associated corresponds to the constant defaultEvaluationPriority which is equal to 50. When several rules have the same evaluation priority at some validation point the reactive rules are evaluated nondeterministically.

Rules Execution

All the preselected reactive rules at some validation point are executed starting with those with the highest evaluation priority. The execution of a rule implies first the evaluation of the <Coordination Condition> specified in the rule. If the <Coordination Condition> validates to true, then the <Coordination State Actions> actions associated with the rule are executed. Finally, the <Rule Semantics Actions> semantics actions specified during the creation of the rule are then applied.

In the CoLaS model two rules: InterceptAtArrival and Ignore must be evaluated at the same validation point (atArrival). To specify the order in which these two rule must be evaluated we must assign different evaluation priorities to these rules during their specification. To specify the evaluation priority for a reactive rule the message *evaluationPriority:* <Evaluation Priority Value> must be sent to the reactive rule during its creation with the priority to assign as an argument. In (**Figure 5.12**) we show how we assign in OpenCoLaS different evaluation priorities to the two reactive rules Ignore and InterceptAtArrival. It is cleat that the Ignore rules must be evaluated first than the InterceptAtArrival rules at the atArrival evaluation point, they have a higher execution priority.

```
Ignore evalutionPriority: OpenCoLaS maximumEvaluationPriority.

InterceptAtArrival evaluationPriority: OpenCoLaS minimumEvaluationPrior
```

Figure 5.12 : Specification of evaluation priorities for CoLaS rules

Proactive Coordination Rules

The proactive rules are enforced and evaluated in the CoLaS model by the coordination groups and not by the participants as in the case of behavioral and reactive rules. It is impossible to determine when a proactive rule will be evaluated nor the order in which they will be selected for evaluation. What is certain is that all the proactive rules associated with the group will be evaluated at that time. If the <Coordination Condition> associated with the proactive rule validates to true the group executes the <Coordination Actions> specified in the rule.

5.3 Evolution of the CoLaS Coordination Model

One of the purposes of specifying and building the OpenCoLaS framework was to provide an experimentation tool to study the specification of existing and new coordination rules in the CoLaS coordination model. The current version of the CoLaS model presented in chapter 3 of this thesis is the result of the work done in the OpenCoLas framework in this direction. To try to illustrate these results we will compare the coordination rules in the CoLaS model with those introduced in two previous versions of the model. The first CoLaS version corresponds to the original CoLaS model introduced in [Cruz99a], the second CoLaS version corresponds to an intermediate version of the CoLaS model presented in [Cruz01a].

5.3.1 Original CoLaS model [Cruz99a]

In the original CoLaS coordination model there were five different types of reactive rules and two types of proactive rules. Four categories of rules composed the reactive rules: synchronization, interception, cooperation and multi-party rules.

Synchronization Rules

Two forms of synchronization rules were specified in the original model: Ignore and Disable rules. Both kinds of rules correspond to the same two synchronization rules that we have in the current CoLaS model.

Interception Rules

Two forms of interception rules were specified in the original model: ImpliesBefore and ImpliesAfter rules. The two rules correspond to the InterceptAtSelection and InterceptAtEnd rules in the current CoLaS model. The only difference between these rules in the two models is that in the current CoLaS model we restrict the kinds of coordination actions specified in the rules to strictly actions that modify the coordination state. In the ImpliesBefore and ImpliesAfter rules the list of coordination actions included for example asynchronous and synchronous recursive message sends. The current CoLaS model provides two more interception rules: InterceptAtArrival and InterceptAtSent. The InterceptAtArrival rule was introduced in the interme-

diate version of the model before we started our work with the OpenCoLaS framework. Both interception rules are evaluated respectively at the atArrival and at the atSent validation points specified in OpenCoLaS.

Cooperation Rules

The original version of the CoLaS coordination model did not specify cooperation rules as in the current CoLaS model (i.e, defineBehavior rules). The cooperation protocols were specified uncleanly using the two interception rules defined in the original model and by defining coordination methods in the participants. The model did not provide a clean separation of the coordination and computation aspects. Coordination code appear mixed within the computational code of the participants. In the current version of CoLaS we have behavioral rules in the groups to specify the coordination behavior. It is not necessary anymore to define coordination methods in the specification of the participants.

Multi-Party Rules

The original version of CoLaS specified an interesting form of multi-party reactive rule called Atomic. The execution of the <Coordination Actions> actions associated with the Atomic rules depended on multiple invocation requests occurring on different participants playing different roles. From our point of view the Atomic rule is not necessary in the current version of CoLaS because the method invocations and rules specify sequences of actions are executed atomically by default. However, we have found some problems not related to the atomicity problems in which some form of multi-party synchronization rules seems to be the most adapted solution. For example, a multi-party condition synchronization implicating multiple participants or multi-party mutual exclusions. From our point of view the specification of multi-party coordination rules is an interesting future work that can be done in the CoLaS model and in the OpenCoLaS framework. The problem with multi-party coordination rules is the specification of a clear semantics considering the fact that they are based on multiple invocation requests occurring in different participants possibly playing different roles.

Proactive Rules

Two forms of proactive rules were specified in the original CoLaS model: Once and Always. For both types of proactive rules it is necessary to specify a <Coordination Condition> and <Coordination Actions> as in the current version of the CoLaS model. The <Coordination Condition> associated with the rule determines whether the <Coordination Actions> associated with the rule must be executed when the rule is evaluated by the group. The semantics of the Once proaction rule ensured that the <Coordination Actions> associated with the proaction were executed only once, the first time the <Coordination Condition> validated to true. The Always rule ensured that the <Coordination Actions> associated with the proactive rule were executed each time the <Coordination Condition> validated to true (and that, during the time of existence of the group). Actually OpenCoLaS specifies only one proactive rule called Validate which corresponds to the second form of proactive rule in the original CoLaS model. It is easy to see that the Once proaction rule is a particular case of the Always proactive rule, the semantics of the rule can be simulated in the Always rule by defining a boolean group variable in the coordination state that indicates whether the proaction was already executed or not by the group.

5.3.2 Intermediate CoLaS model [Cruz01a]

In the intermediate CoLaS coordination model we had one type of behavioral rule, five different types of reactive rules and one type of proactive Rule. Two categories of reactive rules composed the reactive rules: synchronization and interception rules.

Behavioral Rules

One form of behavioral rule was specified in the intermediate CoLaS model: defineBehavior. The behavioral rule corresponds to the same behavioral rule that we have in the current version of the CoLaS model.

Synchronization Rules

Two forms of synchronization rules were specified in the intermediate CoLaS model: Ignore and Disable. Both kinds of rules correspond to the same two synchronization rules that we have in the current version of the CoLaS model.

Interception Rules

Three forms of interception rules were specified in the intermediate CoLaS model: InterceptAtArrival, InterceptBeforeExecution and InterceptAfterExecution. The three kinds of rules correspond to the InterceptAtArrival, InterceptAtSelection and InterceptAtEnd interception rules specified in the current version of the CoLaS model. Actually the CoLaS model provides an extra interception rule InterceptAtSent. The InterceptAtSent interception rules is evaluated at the atSent validation point. The InterceptAtSent rule is an important rule because it allows the specification of coordination based on the messages sent to other participants.

Proactive Coordination Rules

One form of proactive rule was specified in the intermediate CoLaS model: validatesAlways, this proaction rule corresponds to the Validate proaction rule that we have in the current version of the CoLaS model.

5.4 Simplifying the Interception Rules in CoLaS

If we compare the semantics of the different interception rules in the current version of the CoLaS model (i.e., InterceptAtArrival, InterceptAtSelection, InterceptAtSent and InterceptAtEnd) and we analyse the semanctics associated with these rules in the OpenCoLaS framework we can say that they have the “same” semantics (i.e. to execute some actions at some evaluation point during the processing of a method invocation by a participant). They differ exclusively in the validation point at which they must be evaluated. We can deduce that only one generic type of interception rule is necessary in CoLaS and in OpenCoLaS. A possible modification to the specification of interception rules in CoLaS consists of defining a unique type of interception rule name InterceptAt and to specify during its instantiation the validation point at which the rule must be evaluated: atArrival, atSelection, atSent and atEnd. In (**Figure 5.13** line 1) we can see how a generic interception rule InterceptAt is specified in OpenCoLaS, no validation point is associated during its specification. It is only during the instantiation process of the InterceptAt interception rule that we specify the validation point at which the rule must be evaluated.

```
1.ReactiveRule
2.       defineRule: InterceptAt.
3.
1.InterceptAt message: ..
2.       actions: [..]
3.       entryPoint: OpenCoLaS atArrival.
4.
5.InterceptAt message: ..
6.       actions: [..]
7.       entryPoint: OpenCoLaS atSelection.
8.
9.InterceptAt message: ..
10.      actions: [..]
11.      entryPoint: OpenCoLaS atSent.
12.
13.InterceptAt message:
14.      actions: [..]
15.      entryPoint: OpenCoLaS atEnd.
```

Figure 5.13 : Simplifying Interception rules

5.5 Specifying CoLaS like Coordination Models in OpenCoLaS

A second goal of specifying and building the OpenCoLaS framework was to provide a experimental tool to study and compare the specification of Coordination Rules in CoLaS like coordination models and languages. We present in this section our results in the study of the three most important related coordination models and languages to the CoLaS model: Moses [Mins97a], Composition Filters [Berg94a] and Synchronizers [Frol93a].

5.5.1 Moses [Mins97a]

Moses specifies basically two kinds of reactive rules (laws in their terminology): sent(x,m,y):- <Primitive Actions> and arrived(x,m.y):- <Primitive Actions>. The two rules determine what should be done when a specified group of messages is sent and received by the participants of the group. The sent rule specifies that a participant x sends a method invocation request (i.e., a message) m to another participant y and the arrived that a message m sent by a participant x arrives to the participant y. The two reactive rules deal with what Minsky and Ungureanu call *regulated events*. Among the possible <Primitive Actions> we have forward(m,y,x) that emits to the network the message m addressed to the participant y sent by the participant x and deliver(m) that effectively delivers the message m to the participant that received the message. Other possible primitive actions include: modifications to the coordination state (i.e., CS control state of the participant in their terminology). Specifically they propose the following operations: (1) +t add a term to the control state, (2) -t removes a term from the control state, (3) t1<- t2 which change term t1 with term t2; and (4) incr(t(v),x) which increments the value of a term t with some quantity x.

Additionally to the sent and arrived reactive rules, Moses defines two forms of proactive rules (or obligations in the Moses terminology): +obligation(p,dt) and -obligation(p). The +obligation rule causes an ob-

ligation event obligationDue(p) to occur at some participant x (Agent in the Moses terminology) in dt seconds (provided that the obligation has not been repealed in the meantime by the inverse operation -obligation). The occurrence of the obligation event obligationDue(p) forces the participant to evaluate the rule for this event and to carry it on. The rule is thus the action associated with the obligation event.

The Sent rule

When we analyse the semantics of the reactive rules in Moses we can see that when a message m is sent from a participant x to a participant y, there are basically two possible actions over the message m that the <Primitive Actions> may specify: to forward the message m or to do not forward the message m. It does not have any sense to deliver to the participant a message that a participant has requested to send. When the message m is not forwarded the message must be simply ignored (like in the Ignore rule in the CoLaS model) and not sent. When the message is forwarded the message itself is not affected (like in the InterceptAtSent rule in the CoLaS model). In both cases <Primitive Actions> affecting the control state of the participant can be specified.

We propose to specify in OpenCoLaS the sent rule in Moses as two different reactive rules: SentIgnore (with a similar semantics to the Ignore rule specified in the CoLaS model) and SentForward (with a similar semantics to the InterceptAtSent rule specified in the CoLaS model). Both rules must be evaluated at the atSent validation point. In (**Figure 5.14** rules 1 and 2) we illustrate how these two rules are specified in OpenCoLaS. It is important to remark that during the instantiation of the SentIgnore rules (**Figure 5.14** line 10) we specify the <Coordination Actions> that must be executed when the rules are applied and we do not specify a <Coordination Condition> as we do in CoLaS when we create Ignore rules in the examples.

```
1.[1] ReactiveRule
2.       defineRule: SentIgnore
3.       semantics: [^NoMessage new ]
4.       entryPoint: OpenCoLaS atSent.
5.
6.[2] ReactiveRule
7.       defineRule: SentForward
8.       entryPoint: OpenCoLaS atSent.
9.
10.SentIgnore message: <m Moses message>
11.          actions: [<Coordination State Actions>]
12.
13.SentForward message: <m Moses message>
14.          actions: [<Coordination State Actions>]
```

Figure 5.14 : Moses Sent rule in OpenCoLaS

The Arrived rule

The analysis of the arrived rule reveals something similar to what we previously found for the sent rule. We can see that when a message m arrives from a participant x to a participant y, there are also two possible actions over the message m that the <Primitive Actions> may specify: to deliver the message m to the participant or to do not deliver the message to the participant. When the message m it is not deliver the message

is simply ignored (again like in the Ignore rule in the CoLaS model). When the message m is delivered the message itself is not affected (like in the InterceptAtArrival rule in the CoLaS model).

We propose to specify in OpenCoLaS the arrived rule as two different reactive rules: ArrivedIgnore with a similar semantics to the Ignore rule specified in the CoLaS model and ArrivedDeliver with a similar semantics to the InterceptAtArrival rule specified in the CoLaS model. Both rules must be evaluated at the atArrival validation point. In (**Figure 5.15** rules 1 and 2), we illustrate how these two rules are specified in OpenCoLaS. It is important to remark that during the instantiation of the ArrivedIgnore rules we specify the <Coordination Actions> that must be executed when the rules are applied and we do not specify a <Coordination Condition> as we do when we instantiate Ignore rules in the CoLaS examples.

```
1.[1] ReactiveRule
2.       defineRule: ArrivedIgnore
3.       semantics: [^NoMessage new ]
4.       entryPoint: OpenCoLaS atArrival.
5.
6.[2] ReactiveRule
7.       defineRule: ArrivedDeliver
8.       entryPoint: OpenCoLaS atArrival.
9.
10.ArrivedIgnore message: <m Moses message>
11.      actions: [<Modifications to the control state>]
12.
13.ArrivedDeliver message: <m Moses message>
14.      actions: [<Modification to the control state>]
```

Figure 5.15 : Moses Arrived rule in OpenCoLaS

Obligations rules

The obligations rules in the moses model can be specified in the CoLaS model as proactive rules. We can simulate the semantics of the +obligation proaction rule and the generation of the obligation event obligationDue dt seconds later using a coordination state variable of "type" time. We must define a group variable to store the time at which the proactive rule must be evaluated and include a reference to this variable in the <Coordination Condition> associated with the proactive rule. The condition must include a boolean expression which compares the value of the group variable with the current time in the system. The only problem with this approach is that for each proaction specifying a dt value we need to specify a group variable. Concerning the -obligation rule we can simulate the semantics of the rule by specifying another coordination state variable that specifies whether the rule should not be evaluated anymore, again the problem with this approach is that for each -obligation rule a group variable needs to be specified. It will be really interesting to think in the possibility to include in the CoLaS model proaction rules associated with time constraints as in the Moses model, this will avoid the definitions of coordination state variables each time an obligation rule will appear. This could be another idea for future work in the CoLaS coordination model.

```
1.ProactiveRule
2.       defineRule: #+Obligation.
3.
4.+Obligation message: <Moses Obligation Event>
5.            condition: [ timeToExecutionObligation >= Time now ]
6.            actions: [<Coordination State Actions>]
```

Figure 5.16 : Moses +obligation proaction rule in OpenCoLaS

Conclusions

We can conclude from the previous presentation that the two reactive rules specified in Moses correspond to two special forms of the Ignore and InterceptAt reactive rules in the CoLaS model. The SentIgnore and ArrivedIgnore can be instantiated in OpenCoLaS from an Ignore rule specifying atSent and atArrival as the corresponding validation points. The SendForward and ArrivedDeliver can be instantiated from a InterceptAt rule specifying atSent and atArrival as the corresponding validation points.

Concerning the CoLaS model we can say that: 1) the CoLaS model specifies rules at two validation points not considered in the Moses model (atSelection and atEnd), we can say that the CoLaS model is finer than the Moses model; 2) the Moses model does not provide equivalent rules to the Disable, InterceptAtAccept, InterceptAtSelection and InterceptAtEnd, this is a consequence of the previous point. The Disable rules are fundamental to express condition synchronizations in concurrent systems; 3) the Moses model provides an extra Ignore rule at the atSent validation point which the CoLaS model does not provide, the CoLaS model only defines a Ignore rule at the atArrival validation point similar to Moses and 4) The ArrivedIgnore rule in the Moses model does not specify a <Coordination Condition> for the applicability of the rule as the Ignore rule does in the CoLaS model, in the other hand the Ignore rule in the CoLaS model does not specify <Coordination State Actions> as the ArrivedIgnore rules does in the Moses model. It is important to take into consideration that we are comparing both models as they are currently defined, the SentIgnore rule in the Moses model can be easily integrated into the CoLaS model because of flexibility of the OpenCoLaS framework to specify new rules. Concerning proactive rules, both model Moses and CoLaS provides rules to specify proactive behavior, the Moses rules are more general in the sense that they can be associated with time constraints. When time constraints are not defined the Moses proactive rules correspond to the proactive rules specified in the CoLaS model. Nevertheless, as we showed before it is possible to simulate the behavior of Moses proactive rules using coordination state variables in CoLaS.

5.5.2 Composition Filters [Berg94a]

Composition Filters are first class objects used to affect the messages received and sent in the object model. A Composition Filter consists of two parts: an interface and an implementation part. The interface deals with incoming and outgoing messages. The second part corresponds to the implementation part which consists of method definitions, instance variables declarations, definitions of conditions and an optional initialization operation. The Interface part consists of one or more input and output filters, optional internal and external objects and method header declarations. If a message passes through the input filters it can be further delegated to internal objects, methods or external objects that composed the object. All the messages that originates from the method executions within the object and are sent to objects outside the boundaries

of the current object pass through the output filters. Without filters the model is very similar to a conventional object model.

A filter element consists of three parts: 1) a condition, which specifies a necessary condition to be fulfilled in order to continue evaluating a filter element; 2) a matching part, in which the evaluated message is matched against a defined pattern and 3) a substituting part, where parts of the message can be replaced. In the current version of the Sina language (in which the Composition Filters were integrated) we find the following primitive filters: Dispatch, Meta, Error, Wait and RealTime. The Dispatch filter is used to initiate execution of a method when the corresponding message passes successfully through the filters. The Meta filter is similar to the Dispatch filter, but they differ in that if the received message is accepted by the Meta filter the message is first converted to an instanced of class Message and then passed as an argument of a new message to the object. The Error filter is similar to the Dispatch filter but it does not provide method dispatch; it raises an error condition if a message does not pass through the filter. The Wait filter is used for synchronization, in this filter the message is queued as long as the evaluation of the filter condition results in a rejection. The RealTime filter is used for real time computations. These filters can be both input and output filters. An important feature of all these filters is that they are orthogonal to each other and therefore they can be combined. Commonly the last filter to apply is always of class Dispatch which results in the delegation of the request message to its target object.

When a message received by an object is evaluated by a filter, the message is checked against the elements of the filter in the left-to-right order. If the condition associated with the filter validates to true, then the selector received message is matched against the selector of the matching part, when the filter element does not match, the subsequent filter is tried. When both the condition and the matching part validate, the substitution actions described in the substituting part of the filter are applied to the message. The substitution actions specified in the filter include: the rename and the redirection of the message.

The Dispatch Filter

In the CoLaS model we do not have any equivalent rule to this filter, if we do not delay or ignore the execution of a method invocation request, the request will be automatically dispatched. It is not possible to specify that the method invocation request must be dispatched (i.e.executed) at the time of the evaluation of the rule as in Composition Filters.

The Meta Filter

In the CoLaS model we do not have any specific rule equivalent rule to this filter. In all CoLaS rules it is possible to refer to the method invocation request received. It is possible to request the identity of the sender, the identity of the receiver, the selector and the arguments of the method invocation requested. We can say that in some way all our rules are Meta rules because it is always possible to reify the method invocation requested.

The Wait Filter

The semantics of the Wait filter is similar to the Disable rule in the CoLaS model. The method invocation request is delayed internally in the object until that the <Coordination Condition> associated with the rule validates to true. The main difference between the Disable rule and the Wait filter is that the filter may specify transformations of the method invocation requested, we can only do this in OpenCoLaS at the rule class level when we specify the semantics of a rule class. In CoLaS the semantics of the rules is fix.

The Error Filter

In the CoLaS model we do not have any equivalent rule for this kind of filter. We do not consider necessary to include an error rule in the CoLaS model, actually the CoLaS model allows one to raise exceptions in the <Coordination Actions> specified in the rules. We use the exception handling of the language in which CoLaS integrates to raise and to catch exceptions.

The RealTime Filter

In the CoLaS model we do not have any equivalent rule for this kind of filter. From our point of view this is specialized filter useful to solve real time problems, we have not consider until now in the CoLaS model the requirements of specific domains in the specification of rules.

Conclusions

Comparing the Composition Filters and the CoLaS model we can say: 1) the CoLaS model specifies rules at three other validation points not considered in the Composition Filters model (atAccept, atSelection and atEnd), we can say that the CoLaS model is finer than the Composition Filters model; 2) we find only one equivalent rule between the two models: the Disable rule, the other filters can in a way or another be simulated using existing CoLaS rules (with the exception of the RealTime filters), 3) the Composition Filters model allows one to specify transformations to the method invocation requests at the rule level (i.e. there are rules that manipulate and transform the received messages), we can only do this in OpenCoLaS at the rule class level, when we specify the semantics of the rule classes but not in CoLaS. What it is possible in CoLaS is to manipulate the arguments of the invocation requests. We believe that the CoLaS rules are more powerful than the Composition Filters rules: first we are capable of specifying coordination at more different points during the processing of messages by the active objects and second all our rules are meta rules in the sens of the Meta filters in the Composition Filters.

5.5.3 Synchronizers [Frol93a]

Synchronizers are special objects that observe and limit the invocations accepted by a set of ordinary objects. Using the OpenCoLaS terminology we can say that Synchronizers defines basically four kinds of reactive rules: <Pattern> updates <Coordination Actions>, <Pattern> disables <Coordination Condition>, <Pattern> atomic and <Pattern> stops. The rules depend for their application on the matching of the <Pattern> specified in the rule and the method invocation requests received by the participants of the Synchronizers. A<Pattern>specifies defines logical expressions composed of a message or a group of messages (and arguments) associated with a participant or to a group of participants. It is also possible to specify in the <Pattern> some extra condition based on the Synchronizer state.

The updates rule changes the state of the Synchronizer by executing <Coordination Actions> each time a received method invocation request matches the <Pattern>.The disables rule prevents the acceptance of method invocation requests that matches the <Pattern> if the <Coordination Condition> evaluates to true in current state of the Synchronizer. The atomic rule ensures the acceptance of a message of a group of messages specified in the <Pattern> atomically (all or none). The stops rule specifies that the acceptance of a method invocation request matching the <Pattern> terminates the Synchronizer.

The update rule

The semantics of the update rule is similar to that of a InterceptAtArrival rule in the CoLaS model. In the InterceptAtArrival rule some <Coordination Actions> that modify the Coordination State are executed when some method invocation request is received by a participant. Both rules: the update rule and the InterceptAtArrival are evaluated at the arrival of method invocation requests. In (**Figure 5.17**) we can see how the Synchronizers update rule is specified in OpenCoLaS.

```
ReactiveCoordinationRule
        defineRule: #Updates
        entryPoint: OpenCoLaS atArrival.
```

Figure 5.17 : Synchronizer Update rule in OpenCoLaS

The disables rule

The semantics of the disables rule is similar to the Disable rule in the CoLaS model. In the Disable rule method invocation requests are delayed in the participants if the <Coordination Condition> evaluates to true during the evaluation of the rule. In the CoLaS model the Disable rule is associated with the atSelection validation point while in the Synchronizers model the disables rule is associated with the atArrival validation point. According to specification of the Synchronizers model the goal of the disables rule is to prevent the acceptance of method invocation requests, nevertheless because the method invocation requests are delayed in the participants this implies that the method invocation request is received and accepted in some way by the participants, creating confusion. From our point of view the Synchronizers model mixes two different moments in the processing of method invocations in this rule. We will consider in this presentation that the disables rule in the Synchronizers model are associated with the atSelection validation point as the Disable rule in the CoLaS model. In (**Figure 5.18**) we can see how the Synchronizers disables rule is specified in OpenCoLaS.

```
ReactiveCoordinationRule
        defineRule: #Disable
        semantics: [mailbox put: message.
                  ^NoMessage new ]
        entryPoint: OpenCoLaS atSelection.
```

Figure 5.18 : Synchronizer disable rule in OpenCoLaS

The Atomic rule

The original version of CoLaS specified a similar rule to the atomic rule in the Synchronizers model, this rule was eliminated from the current version of the model. The CoLaS model and the work done on OpenCoLaS concentrated basically on the specification of single-party rules: rules in which only one participant and only one method invocation request is taken into account in the specification of rules. From our point

of view the specification of multi-party rules is an interesting future work that can be done in the CoLaS model and in the OpenCoLaS framework.

The stops rule

The CoLaS model does not provide any equivalent rule to this. In CoLaS the application of rules is done in participants during the time they remain playing roles in groups.

Conclusions

Comparing the Synchronizers and the CoLaS model we can say: 1) the CoLaS model specifies rules at two other validation points not considered in the Synchronizers model (i.e. atSent and atEnd), we can say that the CoLaS model is finer than the Synchronizers model; 2) the Synchronizers model does not provide equivalent rules to the Ignore, InterceptAtSelection, InterceptAtSent and InterceptAtEnd, this is a partially a consequence of the previous point; 3) the Synchronizers model provides a Disables rule whose semantics it is not clear. The rule is supposed to be evaluated at the atArrival validation point to avoid the reception of messages when some condition validates to true. Because the message is delayed in the participants when the condition associated with the rule validates to true the message is in some way accepted (the reception is not avoided) by the participants generating then an inconsistency. The rules does not prevent the reception of messages in fact.

It is important to remark that one of the important aspects of the Synchronizers is the possibility to define multi-party coordination rules: rules that depend for their applicability on multiple invocation requests occurring and in different participants. This is something that can not be done in the CoLaS model actually.

Finally we must say that the Synchronizers model is a pure reactive coordination model, it does not provide equivalent rules to the proaction rules specified in the CoLaS model. Synchronizers react exclusively to method invocation received by the participants.

5.6 Conclusions and Contributions

We presented in this chapter OpenCoLaS a framework for experimenting with the specification of CoLaS like rule-based cordination models and languages. CoLaS follows an approach of coordination based on the interception of messages exchanged by the active objects (i.e., reflective approach), each coordination rule specifies coordination actions that must be done in the group at some precise validation point. The OpenCoLaS framework allows the specification of three types of coordination rules: behavioral, reactive and proactive. For each type of coordination rule, the framework defines abstract classes containing all the necessary support to specify new subclasses of coordination rules. The approach used to build the framework is that of meta-languages [Kicz91a] in which the semantics of the rules and their evaluation process are explicitly reified in a framework to facilitate their definition and modification. New coordination models and languages for object systems based on message interception and coordination rules can be created and existing languages compared using the OpenCoLaS framework.

The main goal of this chapter was to provide arguments that justify the choice of the coordination rules in the CoLaS model. Specifically, to provide answers to the following questions: Why these rules and not others? Where do these rules come from? Are all these rules necessary? Concerning the question Why these rules and not others? we can say that basically each type of rule corresponds to a basic coordination need. Cooperation rules are necessary to specify behaviors in the participants exclusively related to the coordina-

tion, synchronization rules (a type of reactive rules) are necessary to specify synchronization constraints and proaction rules are necessary to specify proactive behavior independently of the messages exchanged by the participants. Concerning interception rules (second type of reactive rules) they intercept messages and perform actions that modify the coordination state of the group. At a first view one can think that these rules are not important and they can be eliminated from the CoLaS model because whatever is specified in these rules could also have been specified in the synchronization rules. The truth is that if we would have allowed users to manipulate the normal processing of messages in our rules as we do in the meta specification of rules in OpenCoLaS, we would have only defined interception rules in our model. But, because we do not give all this freedom to our users and because we believe that opening the rules to their manipulation in this way in the specification of the coordination will push programmers to focus more in specifying of how to realize the coordination which is not ideal from the coordination point view. We believe that the coordination rules in the model should have clear and simple semantics, they should keep the specification of the coordination at a high level far from the details of how the coordination is done. Each one of the coordination roles specified in the CoLaS model has its utility no one can be eliminated.

All the rules that make part of the CoLaS models correspond to rules that are evaluated at four different evaluation points during the processing of the messages by the participants (at the arrival of a message, at the selection for execution of a message by the participant, at the send of a message to another participant and at the end of the execution of a message). The first time the CoLaS model was introduced [Cruz99a] the notion of evaluation point was not fundamental in the CoLaS model, it was only until we built the OpenCoLaS framework and that we started to play with the definition of the semantics of the rules, that it appeared as fundamental to clearly specify for each coordination rule a precise moment (evaluation point) in which the rule will be evaluated.

Concerning the question Where do these rules come from? the answer is from the evaluation points. CoLaS defines evaluation points in which the coordination rules are validated and enforced. Such answer raises immediately two new questions: Are the four evaluation points defined in the CoLaS model the only possible/interesting validation points during the processing of the messages by the participants? Which kinds of interesting coordination rules can be defined in each one of these validation points? Concerning the first question we can say that we have experimented with the definition of new validation points in the CoLaS model and the specification of new rules associated with these new validation points. We have found that at the end these new rules can be replaced by combining existing rules. We are almost sure that we do not need to specify more evaluation points in the CoLaS model. Concerning the second question it is difficult to give an answer considering that coordination is something new and we do not know yet if all coordination problems can be solved with the rules that actually we define in the CoLaS model, the only thing we can say is that until now the rules that make part of the CoLaS model seems to be sufficient to tackle a wide range of coordination problems. Nevertheless, the results of the comparison of CoLaS with other similar models done in this chapter shows that it is possible that new rules will need to be defined. If we take for example Moses [Mins97a] we can see that they have a rule with the same semantics that our rule Ignore but evaluated at the atSent evaluation point, some coordination problems are solved using this rule.

Concerning the last question, Are all these rules necessary? the answer is yes. Even if the rules related with the events atSelection and atEnd seems to "violate" the separation of the coordination and the computation aspects in the systems. The atSelection evaluation point corresponds to the moment when a method invocation is ready to be executed by the participant and just after the synchronization policy was validated. The CoLaS models includes a synchronization rule Delay which is evaluated at the atSelection point. The

Delay rule is an important rule because it allows one to specify condition synchronizations [Andr00a]. If we consider a participant as a black box around which the coordination is specified, only the arrival and the departure of messages to and from the participant can be identfied as events from outside of the participant. In other words a pure coordination model for objects must define “in theory” exclusively actions related with these two types of events.

Besides the fact that building the OpenCoLaS framework was fundamental in the understanding of the CoLaS model and in its evolution, the framework represents a powerful (an unique) tool to compare the CoLaS model with related approaches. We presented in this chapter the results of our comparison study of the three most important related coordination models and languages to the CoLaS model: Moses [Mins97a], Composition Filters [Berg94a] and Synchronizers [Frol93a].

Concerning Moses [Mins97a] we conclude that the two reactive rules specified in Moses correspond to two special forms of the Ignore and InterceptAt reactive rules in the CoLaS model. The SentIgnore and ArrivedIgnore can be instantiated in OpenCoLaS from a Ignore rule specifying the atSent and the atArrival as the corresponding validation points. The SendForward and ArrivedDeliver can be instantiated from a InterceptAt rule specifying the atSent and the atArrival as the corresponding validation points. We can say that CoLaS specifies rules at two validation points not considered in the Moses model: atSelection and atEnd. We can also say that Moses does not provide an equivalent rule to the Disable rule important in CoLaS to specify condition synchronization. And, that Moses provides an extra Ignore rule evaluated at the atSent evaluation that CoLaS does not have. Concerning proactive rules, both model Moses and CoLaS provides rules to specify proactive behavior, the Moses rules are more general in the sense that they can be associated with time constraints. When time constraints are not defined the Moses proactive rules correspond to the same proactive rules specified in the CoLaS model. We have shown that it is possible to simulate the behavior of Moses proactive rules including time constraints in CoLaS using state variables. Few coordination models and languages offer the possibiltity to define proactive coordination, CoLaS is one of them [Andr96a][Cruz99a].

Concerning Composition Filters [Berg94a] We conclude that the CoLaS rules are more powerful than the Composition Filters in the sense that it is possible to specify coordination at more different points during the processing of messages by the active objects and second because all our rules are meta rules as in Composition Filters. We can say that CoLaS defines rules at three validation points not considered in the Composition Filters model: atAccept, atSelection and atEnd. Only one Composition Filters rule exist directly in CoLaS: the Disable filter, the other filters can be in a way or another be simulated using CoLaS rules (with the exception of Real Time filters). What is different in Composition Filters is this model allows one to specify transformations to the method invocation requests at the rule level. CoLaS does not. We can not receive a method invocation, transform it in another one and send it to the object. We do not believe that this is fundamental. We do not see cases in which this functionality is needed. Finally the Composition Filters are all reactive related to the arrival of method invocations to the objects. It is not possible to define proactive behavior in this model.

Concerning Synchronizers [Frol93a] we conclude that Synchronizers is a pure reactive coordination model, it does not provide equivalent rules to the proaction rules specified in the CoLaS model. Synchronizers react exclusively to method invocation received by the active objects. We can say that CoLaS defines rules at three validation points not considered in the Synchronizers model: atSent and atEnd; and that Synchronizers does not provide equivalent rules to the Ignore and to our Interception rules. Finally one type of rule that we do not have actually in CoLaS but that Synchronizers has are rules to define multi-party coor-

dination rules (i.e., rules that depend for their applicability on multiple invocation requests occurring in different participants). Multi-party coordination rules are an interesting future work in the CoLaS coordination model.

Contributions

The main contributions of this chapter to thesis are:

- We introduce OpenCoLaS a framework for experimenting with the specification of rule-based coordination models. The idea behind the OpenCoLaS framework is to "open" the CoLaS coordination model and language in a way that allows one to experiment with the specification of coordination rules, possibly also with new coordination rules. The OpenCoLaS framework allows the meta-specification of the coordination rules that compose the CoLaS model.
- We present the semantics of each one of the coordination rules that make part of the CoLaS model. For each rule we clearly specify the moment at which the rule is evaluated and the operational semantics of the execution of the rule. The semantics of the rules are presented using meta operations that alter the normal processing of messages in the active objects, like for example to add a message to the object mailbox or to transform the method invocation in another method invocation.
- We present the results of the comparison of the specification of the coordination rules in CoLaS with the coordination rules introduced in similar approaches: Synchronizers [Frol93a], Composition Filters [Berg94a] and Moses [Mins97a]. We believe that the CoLaS coordination model is a more complete coordination model than the three others presented here. First we are capable of specifying coordination in more evaluation points than the three others and second we have shown that most of the rules (filter in the Composition Filters approach) can be simulated using CoLaS rules. CoLaS is that is the only coordination model and language combining three types of rules: cooperation rules, reactive rules and proaction rules. Each type of rule corresponding to a basic coordination need. Cooperation rules are necessary to specify behaviors in the participants exclusively related to the coordination, synchronization rules (a type of Reactive rules) are necessary to specify synchronization constraints and proaction rules are necessary to specify proactive behavior independently of the messages exchanged by the participants.

CHAPTER 6

Validation

We have presented in this thesis CoLaS, a coordination model to specify the coordination aspect in concurrent object-oriented systems. The CoLaS coordination model is based on the notion of coordination groups, entities that specify control and enforce the coordination of groups of collaborating active objects. The primary tasks of the coordination groups are: 1) to support the creation of active objects, 2) to enforce cooperation actions between active objects, 3) to synchronize the occurrence of those actions and 4) to enforce proactive behavior [Andr96a] on the systems based on the state of the coordination. The CoLaS coordination model follows the coordination model and language approach in which the coordination aspect is specified separately from the computation aspect in the systems. The separation of the specification of the coordination and the computation aspects in concurrent object-oriented systems facilitate their specification, understanding, construction and evolution.

Until now we have mainly focussed our presentation on the software engineering benefits obtained from the separation of the coordination and the computation concerns in concurrent object-oriented systems using CoLaS. We have shown how complex interaction and synchronization patterns which normally are mixed within the computation code of the objects appear now explicitly defined in the coordination groups making those systems easy to understand and to modify. We will focus now in this chapter in providing a methodology to use formal tools specifically Petri Nets for the analysis and verification of the coordination specified in the coordination groups. Petri Nets is a formal modeling language for concurrent systems that has received wide academic and practical interest since its introduction by Carl Adam Petri in 1962 [Petr62a]. Petri Nets are less powerful than Turing Machines, therefore verification of many interesting properties is decidable [Espa94a]. Decidable properties include reachability, a property useful for the verification of safety properties such as deadlock-freedom.

A property of a program is an attribute that is true of every possible history of that program [Andr91a]. Concurrent programs must satisfy two classes of property: safety and liveness [Owic82a]. Safety properties assert that nothing “bad” will ever happen during an execution (a program never enters into a “bad” state) and liveness properties assert that something “good” will eventually happen during the execution. Two important safety properties in concurrent programs are mutual exclusion and absence of deadlock. For mutual exclusion, the “bad” thing is to have more that one process executing critical sections of statements at the same time and for the absence of deadlock is to have multiple processes waiting for conditions that will never occur. Some examples of liveness properties of concurrent programs are [Andr91a]: that a request for service will eventually be honoured, that a message will eventually reach its destination and that a process will eventually enter its critical section. Liveness properties are affected by the scheduling policies, which determine which atomic action is executed the next. If the scheduling does not guarantee fairness (i.e. every process get the chance to proceed regardless of what other processes do).

In this chapter of the thesis we will present our approach to validate safety and liveness properties of CoLaS specifications. We will use Predicate-Action [Kell76a] Petri Nets: Petri Nets with transitions <<if pred-

icate then action>> to formalise the CoLaS groups. We will provide a methodology to transform CoLaS coordination groups into Predicate-Action Petri Nets. We will validate safety and liveness properties using enumeration analysis in the Petri Nets obtained. We will use a tool called Tina: a toolbox for the edition and analysis of Petri Nets and Time Petri Nets, developed in the Software and Tool for Communication Systems group (OLC) of LAAS/CNRS in France. We have additionally included in Appendix B of this thesis a survey on Petri Nets (including Predicate-Action Petri Nets) including formal verification of properties.

We have divided the presentation of this chapter into four parts:

In the first part of this chapter we introduce our methodology to transform CoLaS coordination groups in Predicat-Action Petri Nets. Our methodology consists of defining a mapping function F to transform elements of the CoLaS model into Predicate-Action Petri Nets. We use as example the CoLaS solution to the coordination problem "Subject and Views" presented in chapter 3 of this thesis to illustrate our approach.

In the second part of this chapter we introduce a second example the "The Electronic Vote", to show a complete transformation of a CoLaS group into a Predicate-Action Petri Net. We consider the example to be interesting because the CoLaS group solution to the problem includes almost all the different types of elements that a group may contain.

In the third part of this chapter we specify the different properties of CoLaS coordination groups which can be proved in the transformed Predicate-Action. We show the results of the verification of those properties in the Predicate-Action Petri Nets obtained from the mapping of the "Subject and Views" and the "Electronic Vote" coordination groups. The verification is done using Tina, the toolbox already mentioned before.

Finally in the fourth part of this chapter we present some related work in the use of Petri Nets for the formal verification of properties in coordination models and languages. We conclude this chapter with a presentation of our conclusions, pointing out the main contributions of this chapter to the thesis.

6.1 From CoLaS Groups to Predicate-Action Petri Nets

We will show in this section all the details concerning how to map a CoLaS coordination group into a corresponding Predicate-Action Petri Net. We will start with a brief summary of the CoLaS model and the elements that compose it, then we will show how to map each one the elements that compose CoLaS into a Predicate-Action Petri Net. At the end we will show how all the different Petri Nets obtained must be connected to obtain the final Predicate-Action Petri Net modeling the complete coordination group.

A model is a simplified representation of the real world. It includes only those aspects of the real world system relevant to the problem. Models are used to study the adequacy and the validity of a proposed design. A model can focus on a particular aspect of a problem to perform verifications of properties. In this thesis we will use Predicate-Action Petri Nets in the formalization of the CoLaS coordination model. In Predicate-Action Petri Nets transitions have associated labels of the form "if Condition(X) do Action(X)" where X refers to a set of variables defined in the Petri Net. The Condition(X) specifies a condition to the firing of the transition and Action(X) specifies an action to be executed when the transition is fired. For more information about Predicate-Action Petri Nets refer to Appendix B of this thesis.

6.1.1 The CoLaS model

The CoLaS coordination model is built out of two kinds of entities: the participants and the coordination groups. The participants are the entities to be coordinated and the coordination groups are the entities that control and enforce the coordination of the participants. A coordination group is composed of three elements (**Figure 6.1**): the roles specification, the coordination state and the coordination rules.

The roles specification defines the different roles that participants may play in the group. Each role specifies in a role interface the conditions imposed to the participants to play the role. There is no limitation in the number of participants that may play a role nor in the number of roles that can be played by a participant.

The coordination state defines general information needed to perform the coordination, information like: whether some action has occurred or occurs in the system, the number of times some action has occurred in the system, etc. In general the coordination state contains information about the state of the coordination group and the participants. The coordination state is specified by declaring variables. The are three types of variables: group, role and participant variables. The group variables are shared by all the participants of the group, the role variables are shared by all the participants of a role and the participant variables belong to the participants.

The coordination rules, define the different rules governing the coordination of the group. The coordination rules specify: cooperation actions between participants, synchronizations on the execution of participants actions and proactions or actions initiated by the participants independently of the messages that they exchange. CoLaS defines three types of coordination rules: cooperation, reactive and proactive rules. Cooperation rules specify cooperation actions between participants, reactive rules constrain the execution of actions and proactive rules specify proactions in the participants.

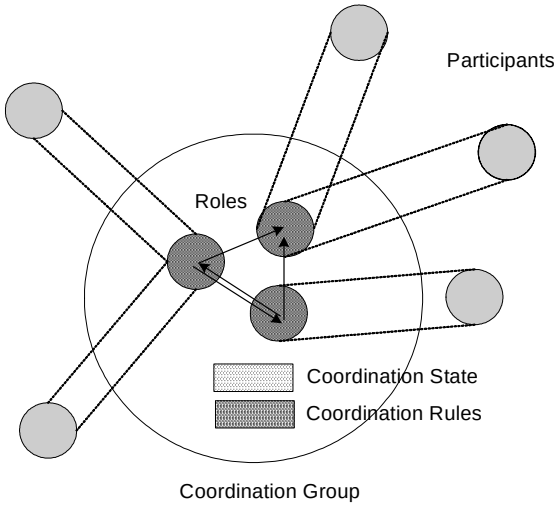


Figure 6.1 : A coordination group

6.1.2 Groups Mapping

The mapping of a coordination group to a Predicate-Action Petri is done by specifying recursively a mapping function F over the elements that compose the group. This technique is inspired in the work of Ayache [Ayac85a] in the modeling and the verification of protocols.

F(<Coordination Group>) = **connect** (F(Role1), F(Role2), ... , F(RoleN)) + F(<Coordination State>)

Every role specified in the group generates itself a complete Predicate-Action Petri Net. All the Predicate-Action Petri Nets obtained are then connected **connect** either directly or indirectly through a virtual medium. The direct connection of the role Predicate-Action Petri Nets models a perfect communication media between the different participants of the group. The use of virtual medium allows one to model asynchronous communication and communication problems like the lost of messages during their exchange. In a first time we will assume a perfect communication media between the different participants, later we will show how to specify different virtual mediums corresponding to different communication problems.

The Coordination State is modelled as variables in the Predicate-Action Petri Net. From our point of view it is not important in the Petri Net to differentiate between the different types of state variables. The different types of state variables define different accessibility constraints on the participants that can not be easily expressed in the Petri Nets and that are not extremely important for the verification purposes. The mapping function will exclusively modify the names of the variables to indicate their type and the role or the group in which they are defined. The function **define** creates a variable in the Predicate-Action Petri Net

F(<Coordination State>) = F(<Group Variables>) + F(<Role Variables>) + F(<Participants Variables>)

F(<Group Variable>) = **define** *groupvar*_<Variable Name>

F(<Role Variable>) = **define** *rolevar*_<Role Name>_<Variable Name>

F(<Participant Variable>) = **define** *partvar*_<Role Name>_<Variable Name>

Message Exchange Mapping

To represent the exchange of messages between participants we have extended the specification of the conditions associated with the transitions in the Predicate-Action Petri Nets with two new conditions: ?m and !m. The condition ?m represents the reception of a message m and the condition !m the sent of message m. The two conditions are used to connect Petri Net places during the generation of the Predicate-Action Petri Nets, they are eliminated at the end when all the connections are done. We will back on this point below in this section.

We will model the asynchronous exchange of messages in a group using a CSP[Hoar85a] similar notation. In our Predicate-Action Petri Nets a transition p?m defines a condition associated with the reception of a message m arriving from a place p and the transition p!m defines a condition associated with the sent of a message m from a place p. In the Petri Net the message exchanged is represented by an intermediate place with the name of the message.

In (**Figure 6.2**(a)) the two places p and q correspond to states in the two participants playing two different roles A and B. At some point in time a participant playing the role A sends a asynchronous message m to a participant playing the role B. The participant playing the role A does not wait until the message is received in the participant playing the role B to continue, we can see in the figure how it is possible that more actions (i.e., a Petri Net sequence) appear in the participant playing the role A after the message m is sent.

To represent in the Predicate-Action Petri Nets the time factor during the exchange of the message we use an approach in which we model the possible causes of the communication problems in a virtual medium connecting the participants. The intermediate place m associated with the name of the message is used to connect to the virtual medium. We will show later in this section how different virtual mediums can be defined and how they can be connected to the participants. We will assume in this work a perfect communication medium connecting the participants, our primary goal is to detect problems in the specification of the coordination and not problems in the communication media.

In the future and for simplicity we will use only ?m and !m to label the transition conditions related to the exchange of messages, we will not include a reference to the place. Furthermore, to reduce the size of the generated Petri Nets we will specify a reduction rule with a special condition representing the combination of the sent and the reception of a message (b). Both the !m and ?m will appear in a same condition !m/?m. The reduction rule will be used when the communication used corresponds to a synchronous recursive sent of a message to the same participant or when the communication media used is consider as perfect.

Finally it is important to remark that in the Petri Nets the tokens represent the messages exchanged by the participants. The flow of a token in a Petri Net represents the flow of a message within and between the participants in the coordination group.

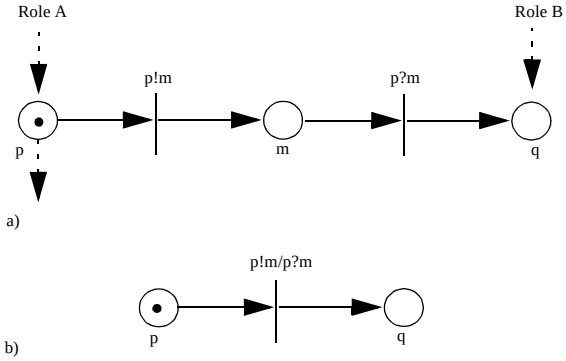


Figure 6.2 Predicate-Action Petri Net for a asynchronous message exchange

In CoLaS every message sent from a participant to another participant generates a reply, replies are sent in the form of futures. The participant who receives the future decides whether to request or not the value returned in the future. To represent the sent of a reply to a participant we include in the representation of the communication a separate message representing the reply (**Figure 6.3**(a)). We add the keyword *ret* to the name of the message in the Predicate-Action Petri Net to indicate that the message corresponds to a reply message. If a participant waits until the reply is received the Predicate-Action Petri Net must include a new

place representing the synchronization (**Figure 6.3**(b)). In CoLaS a participant indicates the wait for a reply by sending the message reply after the name of message sent to the other participant. The message reply is implicitly sent to the future returned from the other participant.

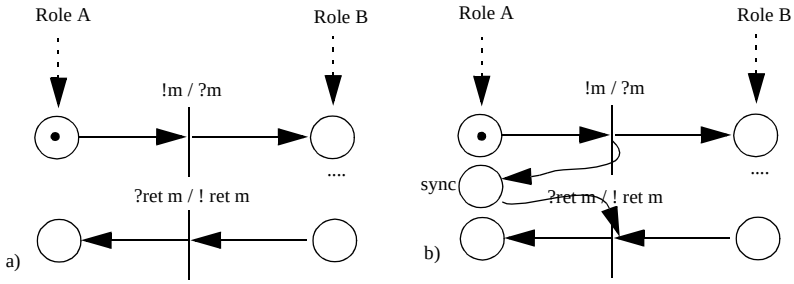


Figure 6.3 Predicate-Action Petri Net for replies

To simplify the graphic representation of the obtained Predicate-Action Petri Net we do not show the values of the input I(p,t) and output O(p,t) functions for a transition t when their values are equal to 1 which is the case in most of the mappings that we show in this section.

Roles Mapping

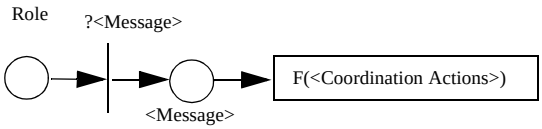
The mapping of roles consists of mapping the different types of coordination rules specified in the group. First the cooperation rules and then the reactive rules. The transformation is done in this order because most of the time the reactive rules refer to behaviors specified in the cooperation rules. The mapping of the reactive rules defines modifications to the Petri Net obtained from the mapping of the cooperation rules.

F(<Role>) = F(<Cooperation Rules>) + F(<Reaction Rules>)

Cooperation Rules Mapping

<Cooperation Rule> :: <Role> ***defineBehavior:*** <Message> ***as:*** [<Coordination Actions>]

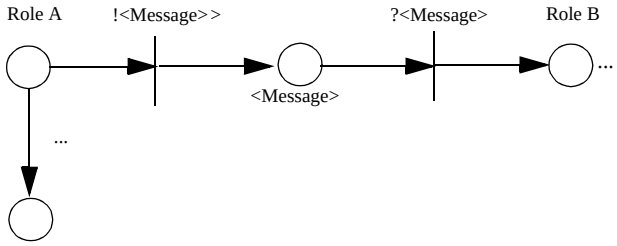
F(<Cooperation Rule>) =



<Coordination Action > :: <Asynchronous Message Send> |
<Synchronous Recursive Message Send> |
<Coordination State Modifications>

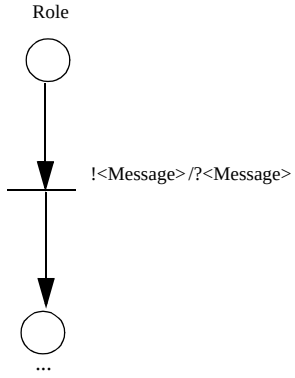
The first type of coordination action corresponds to the asynchronous send of a message to another participant playing a different role. The coordination action is modeled in a Predicate-Action Petri Net simply as a asynchronous message exchange mapping.

F(<Asynchronous Message Send>) =



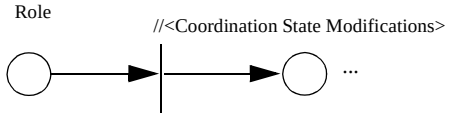
The second type of coordination action corresponds to the send of a synchronous recursive message to the same participant. This coordination action is modeled in a Predicate-Action Petri Net as a reduced message exchange mappings (reduction rule **Figure 6.2**(b)). The condition associated with the transition is !<Message>/?<Message>.

F(<Synchronous Recursive Message Send>) =



The third type of coordination action corresponds to the modification of the coordination state. This coordination action is modeled in a Predicate-Action Petri Net as set of actions to be execute in a transition. We use // to separate the conditions from the actions in a transition label. The <Coordination State Modifications> corresponds to the modification of the values of the variables defined in F(<Coordination State>).

F(<Coordination State Modification>) =



Reactive Rules Mapping

The mapping of the cooperation rules specified in a coordination group continues as follow:

<Reaction Rule> :: <Interception Rule> | < Synchronization Rule>

<Interception Rule>:: <Role> <Interception Operator> <Message> ***do:*** [<Coordination State Actions>]

<Interception Operator> :: ***interceptAtArrival*** **|** ***interceptAtSelection*** **|**
InterceptAtSent **|** ***interceptAtEnd***

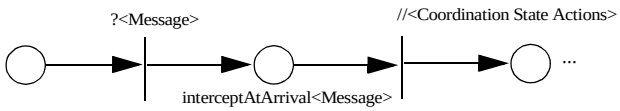
<Synchronization Rule>:: <Role> <Synchro. Operator><Message> ***if:*** [<Synchronization Condition>]

<Synchro. Operator> :: ***disable*** | ***ignore***

Interception rules and Synchronization rules are rules evaluated in the CoLaS model at different moments (i.e., evaluation points) during the processing of the method invocations in the participants. The <Interception Operator> in the Interception rules indicates the precise moment at which the rule is evaluated. The Interception rules (disable and ignore rules) on the other hand are evaluated respectively at the atArrival and at the atSelection evaluation points in the CoLaS model. To model the different Reactive rules in the Predicate-Action Petri Nets we need to model the internal processing of the messages in the participants, particularly the four evaluation points in which these rules are evaluated. It is important to remark that we do not model the behavior of specific participants in our Petri Nets but the behavior of the roles. Implicitly we model the behavior of a kind of unique participant playing the role.

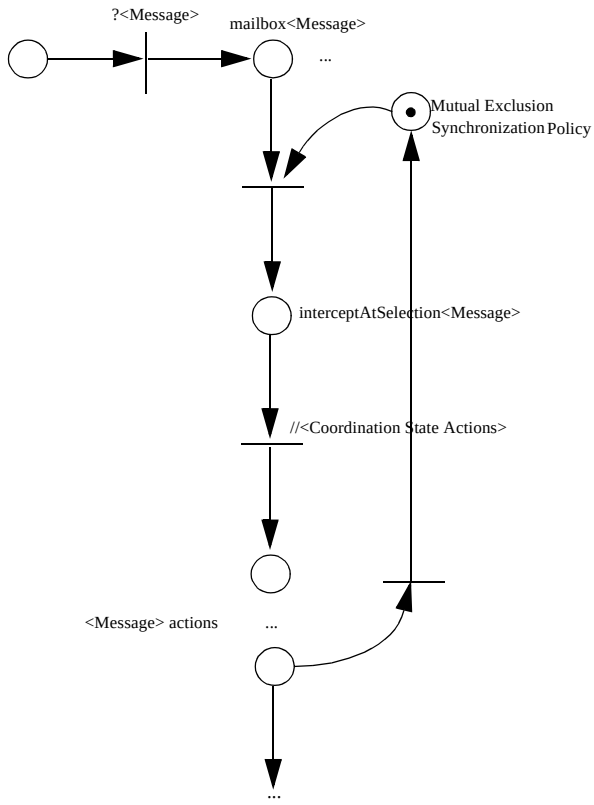
F(<Reaction Rule>) = F (<Interception Rule>) | F (<Synchronization Rule>)

F(<Interception Rule>) = F(***interceptAtArrival*** <Message> do: [<Coordination State Actions>)]=



The mapping function F for an InterceptAtArrival reactive rule specifies a new place named interceptAtArrival<Message> in the Predicate-Action Petri Net after the transition with the condition associated with the reception of the message <Message>. The mapping function specifies also that the <Coordination State Actions> specified in the rule appear as actions associated with a new transition connecting the interceptAtArrival<Message> place which the next place that will be generated from the recursive application of the mapping function to the coordination group. It is important to remember that the <Coordination State Actions> represent actions that modify exclusively the coordination state (i.e, the state variables) of the group.

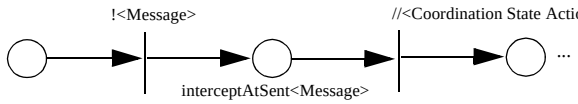
F(<Interception Rule>) = F(***interceptAtSelection*** <Message> do: [<Coordination State Actions>)] =



In the InterceptAtSelection mapping function we model the synchronization policy controlling the execution of messages in the participants. In the CoLaS coordination model participants apply a mutual exclusive synchonization policy: messages are executed sequentially within a participant. The synchronization policy place that appears in the Petri Net is connected to a transition related to the mailbox<Message> place. The mailbox place represents the participant's mailbox and stores messages of type <Message>. Messages of type <Message> can only be executed if there is a token in the place associated with the synchronization policy.

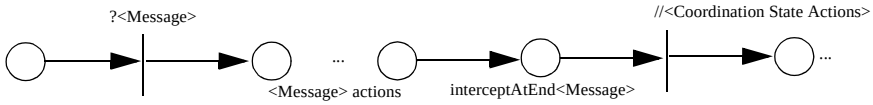
The mapping function shows also that the token associated with the synchronization policy is restored after all the actions associated with the execution of the message are done. We model in this way the atomic execution of actions in the participants in the roles. It is important to remark that in our Predicate-Action Petri Nets we define a mailbox place for each type of message received by the participant. This is done because each message received by a participant may define rules that generate different sequences of Petri Nets after the application of the mapping function F.

F(<Interception Rule>) = F(***InterceptAtSent*** <Message> do: [<Coordination State Actions>)]=



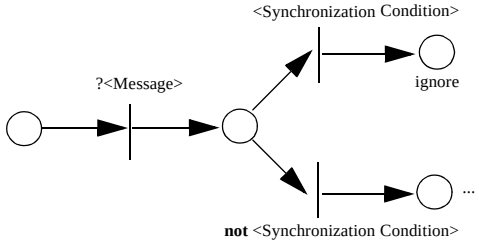
The mapping function F for an InterceptAtSent reactive rule specifies a new place named interceptAt-Sent<Message> in the Predicate-Action Petri Net after the transition with the condition associated with the sent of the message <Message>.

F(<Interception Rule>) = F(***interceptAtEnd***<Message> do: [<Coordination State Actions>)]=



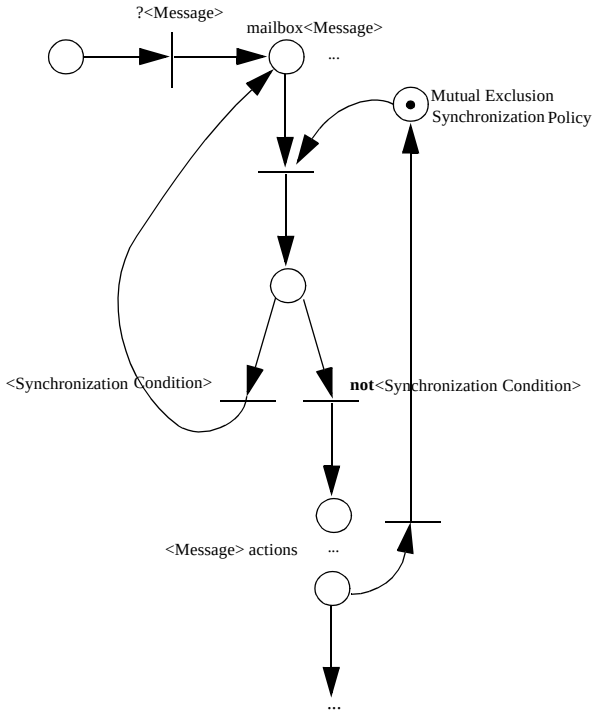
The mapping function F for an InterceptAtEnd reactive rule specifies a new place named interceptAtEnd<Message> in the Predicate-Action Petri Net after all the places and transitions representing all the actions performed during the execution of the message <Message>.

F(<Interception Rule>) = F(***ignore*** <Message> ***if:*** [<Synchronization Condition>)]=



The mapping function F for a Ignore synchronization rule models in the Petri Net the condition and the not condition branches associated with the <Synchronization Condition>. The reason is that for the validation purposes of this chapter it is important to represent all the possible evaluations branches. The Ignore rule is evaluated after the reception of the message <Message>. It is extremely important to remark in the Predicate-Action Petri Net that when the transition associated with the <Synchronization Condition> is fired a token is generated in the place named ignore. The place ignore corresponds to what it is known in the Petri Net language as a dead place. Dead places are interesting places in the validation process because most of the time they are related which possible deadlocks. It is possible already to image what happens in a coordination group when a participant sends a message to another participant and the message is ignored in the other participant. The participant who sent the message may remain blocked if it requires a reply.

F(<Interception Rule>) = F(***disable*** <Message> ***if:*** <Synchronization Condition>)



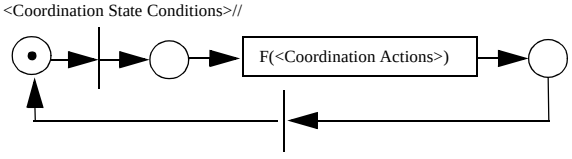
The mapping function F for a Disable synchronization rule models in the Petri Net the condition and the not condition branches associated with the <Synchronization Condition>. The Disable rule is evaluated after the selection of the message <Message> in the mailbox of the participant. In the Predicate-Action Petri Net when the transition associated with the <Synchronization Condition> is fired, a token (i.e., representing the message) is generated in the place corresponding to the mailbox of the message.

Proactive Rules Mapping

Proactive rules are rules that depend for their application exclusively on the coordination state of the group and not on the method invocations received by the participants. Proactive rules guarantee that certain actions are carried out by the group if certain conditions concerning the coordination state validate to true.

Proactive Rule= <Group> ***validate:*** <Coordination State Conditions> ***do:*** <Coordination Actions>

F(<Proactive Rule>) = F(***validate:*** <Coordination State Conditions> ***do:*** <Coordination Actions>)



It is important to remark in the Predicate-Action Petri Net that a token is always present in the initial place in the representation of the rule. The token guarantees that the rule is continuously evaluated. In CoLaS the evaluation of proactions is done indeterministically.

Synchronization Policy

Until now the synchronization policy that controls the execution of messages in participants appeared only in the mapping of the InterceptAtSelection and Disable coordination rules. To obtain an exact representation of the CoLaS groups in Petri Nets, we must modify all the rules related to the reception of messages (in the same role) in which the mailbox associated with the received message does not appear. We must explicitly define mailbox places associated with each possible message received and connect them all to the synchronization policy place. We specify different mailbox places one per each type of message because we need to differentiate the different messages. In (**Figure 6.4**) we show how to connect for a Role A composed of two behaviors messages msg1 and msg2 the reception messages mappings ?m1 and ?m2 to their respective mailboxes places and to the synchronization place. We can see that there is one mailbox place for each kind of message (mailboxmsg1 and mailboxmsg2 places) and that they are all connected to the place representing the synchronization policy named sync. In the initial marking of the Petri Net the synchronization place contains always one token representing the disponibility of the participant to process a message.

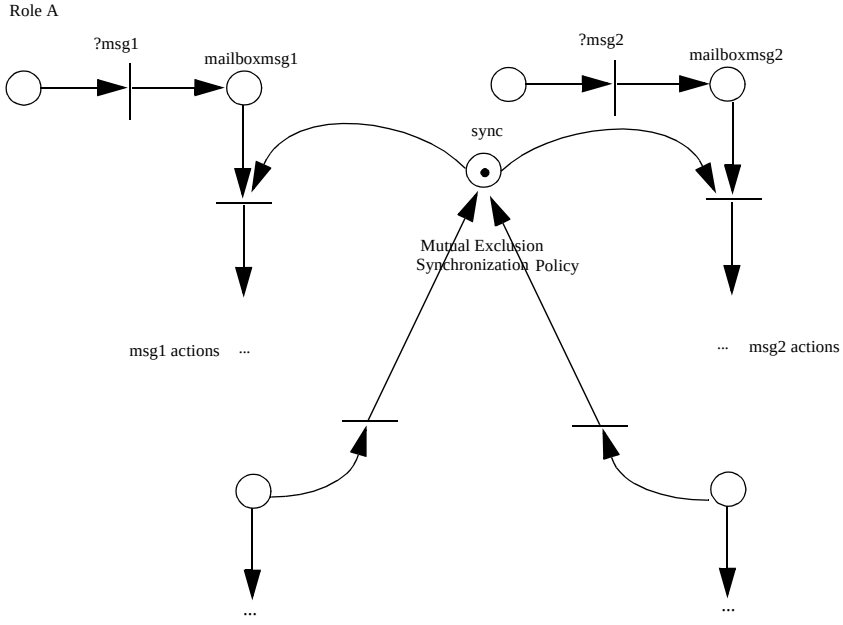


Figure 6.4 Connecting Message receptions

It is also important to define transitions that restore (i.e.,regenerate) the token into the synchronization policy place sync. If the token is not restored the participant will not be able to execute other messages received and stored in the mailboxes. As we mentioned before the synchronization place guarantees the atomic execution of messages in the participants. Different types of synchronization policies must be modelled in a

similar way: first the policy must be modeled, then the Petri Net representing the policy must be connected to all the messages mailboxes and then the transitions which restore the tokens must be added.

6.1.3 Specification of a Virtual Medium

We already mentioned that we will assume a perfect communication medium connecting the participants. Nevertheless, it is possible to define other possible communication mediums to model for example communication problems and delays. The virtual medium is used to connect transitions in the Predicate-Action Petri Nets related to the send of messages !m and the reception of the messages ?m in different participants. Several types of connections can be modelled in the virtual medium, the basic model consists of representing with one (or several) place(s) the transit of a message through the medium. The transit of the message through the medium starts with the fire of the transition labeled !m and finish with the fire of the transition labeled ?m (**Figure 6.5**).

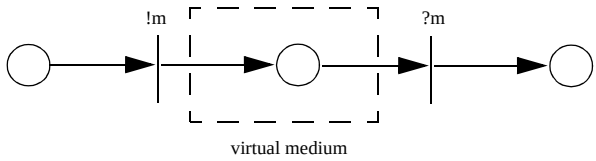


Figure 6.5 : Basic Virtual Medium

In [Ayac85a] an interesting modeling of a virtual medium with lost of messages is presented (**Figure 6.6**). The eventual lost of a message is represented by the fired of the transition labeled lost (a). It is interesting to remark in the model that the number of messages in the virtual medium is not limited. The representation does not constrain the virtual medium to evolve (i.e,. to transfer messages from the sender to the receiver). To solve this problem new transition connections are added (dotted arrows) and the number of tokens in the place UE set to the maximum number of messages that can be in the medium. In this way the receiver is constrained to consume messages because the sender is blocked.

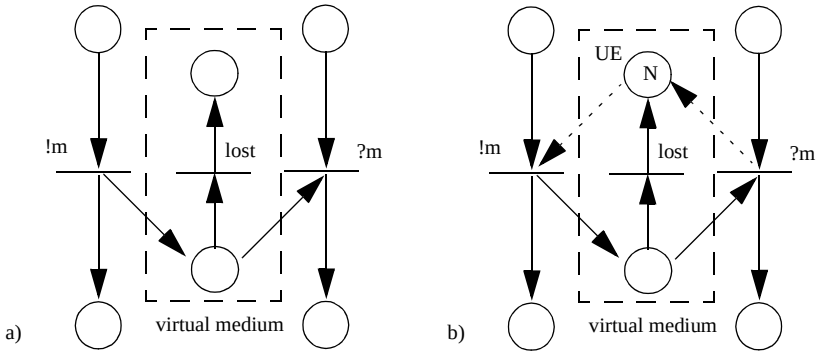


Figure 6.6 : Virtual Medium modeling the lost of messages

Another interesting modeling of virtual medium (**Figure 6.7**) corresponds to a bounded FIFO (i.e. a media containing a limited number of messages and guaranteeing deliver order of the messages). A token in the place ECj indicates that the jth cell of the FIFO is empty. A token in the place M(i,j) indicates that the message Mi finds in the jth cell of the FIFO. The transition lost is associated with the lost of the message Mi and the transitions s(i,j) with the shift of the message Mi from the cell j to the cell j+1 in the FIFO.

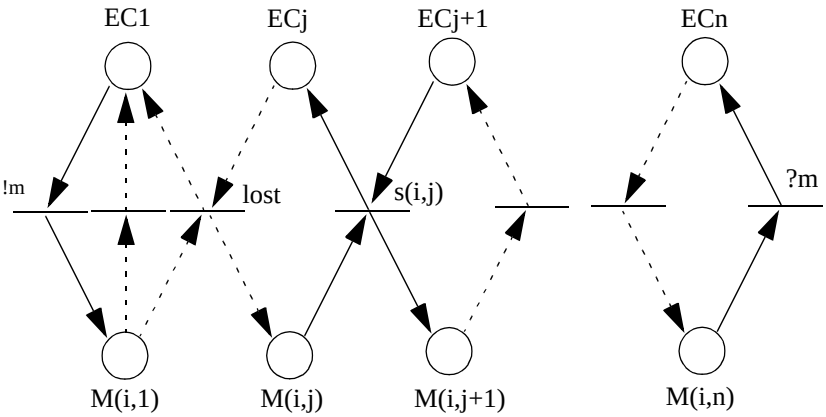


Figure 6.7 FIFO Virtual Medium

6.1.4 From Predicate-Action Petri Nets to Place-Transition Petri Nets

Because mainly all validation techniques available are made on Place-Transition Petri Nets the resulting Predicate-Action Petri Nets must be transformed into simple Place-Transition Petri Nets. The transformation process consists of translating the variables, the conditions and actions defined in the Predicate Action Petri Nets into new places and transitions. In (**Figure 6.8**) we can see how the condition and the action related with the variables N and M in the transition t are translated into a Place-Transition Petri Net. The variables N and M are translated into two new places N and M. The condition N>=4 is translated into a condition related with the number of tokens required to fire the transition t' (i.e., 4) and the number of tokens generated when the transition is fired (i.e.,4). The action M:=M+6 is translated into the number of tokens generated by the transition t' (i.e., 6) in the place M when the transition t' is fired. When all the variables, conditions and actions are translated the transitions t and t' are merged, only the original transition t remains.

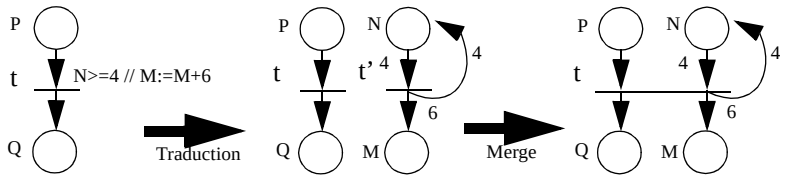


Figure 6.8 Elimination of Predicate and Actions in Predicate-Action Petri Nets

6.2 Case Studies

6.2.1 The “Subject and Views” [Helm90a]

```
1.CoordinationGroup createCoordinationGroupClassNamed: #ObserverPattern.
2.
3.ObserverPattern defineRoleNamed: #Subject.
4.Subject defineVariable: #subjectState.
5.
6.ObserverPattern defineRoleNamed: #Observer.
7.Observer defineParticipantVariable: #observerState.
8.
9.[1] Subject defineBehavior: 'setState: aState' as:
10.      [role subjectState: aState.
11.      self notify].
12.
13.[2] Subject defineBehavior: 'notify' as:
14.      [Observer update].
15.
16.[3] Subject defineBehavior: 'getState' as:
17.      [^role subjectState].
18.
19.[4] Observer defineBehavior: 'update' as:
20.      [self observerState: (Subject unique getState result).
21.      self doSpecificAction].
```

Figure 6.9 Observer pattern group

In the “Subject and Views” example a coordination problem appears when a *Subject* object containing some data and a collection of *View* objects which represent that data graphically (i.e. as a dial, a histogram, or as a counter) cooperate so that all times each *View* always reflects the current value of the *Subject*. The “Subject and Views” coordination problem can be solved using the Observer pattern [Gamm95a]. We show in (**Figure 6.9**) a possible specification of a CoLaS group containing the specification of the coordination of the Observer Pattern. We will focus on explaning how the mapping function F introduced in the previous sections can be used to transform the coordination group into a Predicate-Action Petri net, for any question related to the model refer to chapter 3 of this thesis.

The mapping of the ObserverPattern group into a Predicate-Action Petri Net starts as follows: first the group, then the roles in the group and then the coordination state. The mapping of the coordination state continues with the mapping of the role and participant variables defined in the group.

F(<Observer Pattern>) = connect (F(Subject), F(Observer)) + F(<Coordination State>)

F(<Coordination State>) = F(<Role Variables>) + F(<Participant Variables>)

F(<Role Variable>) = **define** *rolevar*_Subject_subjectState

F(<Participant Variable>) = **define** *partvar*_Observer_observerState

In (**Figure 6.10**) we can see how the different roles specified in the CoLaS ObserverPattern group are transformed into a Predicate-Action Petri Net. We start the transformation with the mapping of the role Subject. The transformation of the roles is done by mapping the coordination rules defined in the role. The role Subject has several cooperation rules associated with it, we start with the mapping to a Predicate-Action Petri Net with the rule setState: rule (line 9) (**Figure 6.10** (1)). The coordination actions specified in the rule setState: includes a modification to the state variable subjectState and the send of a synchronous notify message to the role Observer. The message will be send to all the participants playing the role Observer. In our approach we model a unique participant per role, event if the role Observer may be played by more than one observer, for validation purposes we only represent the behavior of one. We say that we model the behavior of the roles and not the behavior of specific participants.

The transformation of the group continues with the mapping of the cooperation rule notify in the Subject role (line 13). In the example only one coordination action is specified in the notify rule. It specifies the sent of a message update to the participants of the role Observers. We can decide at this point either to start with the transformation of the role Observer or to continue with the transformation of the cooperation rules specified in the role Subject. In the second case, we let open the transitions labeled with messages !m sent to the role Observer and at the end we connect the different open transitions in both roles. The connections are done by connecting !m transitions with ?m transitions of the message m and by adding a place m to the Petri Net corresponding to the message exchanged (if the place do not exists). Because it is possible that a same behavior (message m) appears in the specification of several roles, it is important to indicate in the open transitions (i.e., ?m or m!) the name of the role to which or from which the message was sent or received. When the connection of the open transitions is done we rename again the transitions indicating simply the name of messages exchanged and not anymore the roles. In the example we do not have repeated messages names in the different roles so it is not necessary to modify the names of the transitions in the Predicate-Action Petri Nets generated.

The transformation of the group continues with the mapping of the cooperation rule update (line 19) in the Observer role (**Figure 6.10** (2)). The update rules specifies the sent of the message getState to the Subject role unique participant. We can see in the Petri Net how the reply to the message getState appears in the modeling as a message coming in the opposite direction and the existence of a place named sync (**Figure 6.10** (3)) giving that the observer must wait for the result of the getState message sent to the subject. The transformation of the rest of Observer group continues in the same way until all the coordination rules have been mapped in the Predicate-Action Petri Net. In (**Figure 6.10**) it is possible to see the final result of the mapping process

We can see in the Predicate-Action Petri Net obtained two synchronization policies places one per each role (p3 and p10). In our case their names correspond to the mutual exclusion policy used by the participants to executed messages. All the transitions related with the execution of a received message (t7 and t14 in the Subject role and t6 in the Observer role) are required a token in the corresponding synchronization policy place in order to be fired. The initial marking of the Petri Net assigns one token to each synchronization place.

We can also see in the Predicate-Action Petri Net that there is a place mailbox for a each type of message received. The places p2, p12 and p7 represent the maiboxes of the messages setState:, getState and update messages specified in the cooperation rules 1, 2 and 4 (lines 9, 13 and 19) of the coordination group specification.

Finally it is interesting to remark p6, p11 and p14 represent the messages exchanged by the two kinds of participants: the subject and the observer. We do not connect to the representation of a special virtual medium because we assume a perfect the communication medium between the participants.

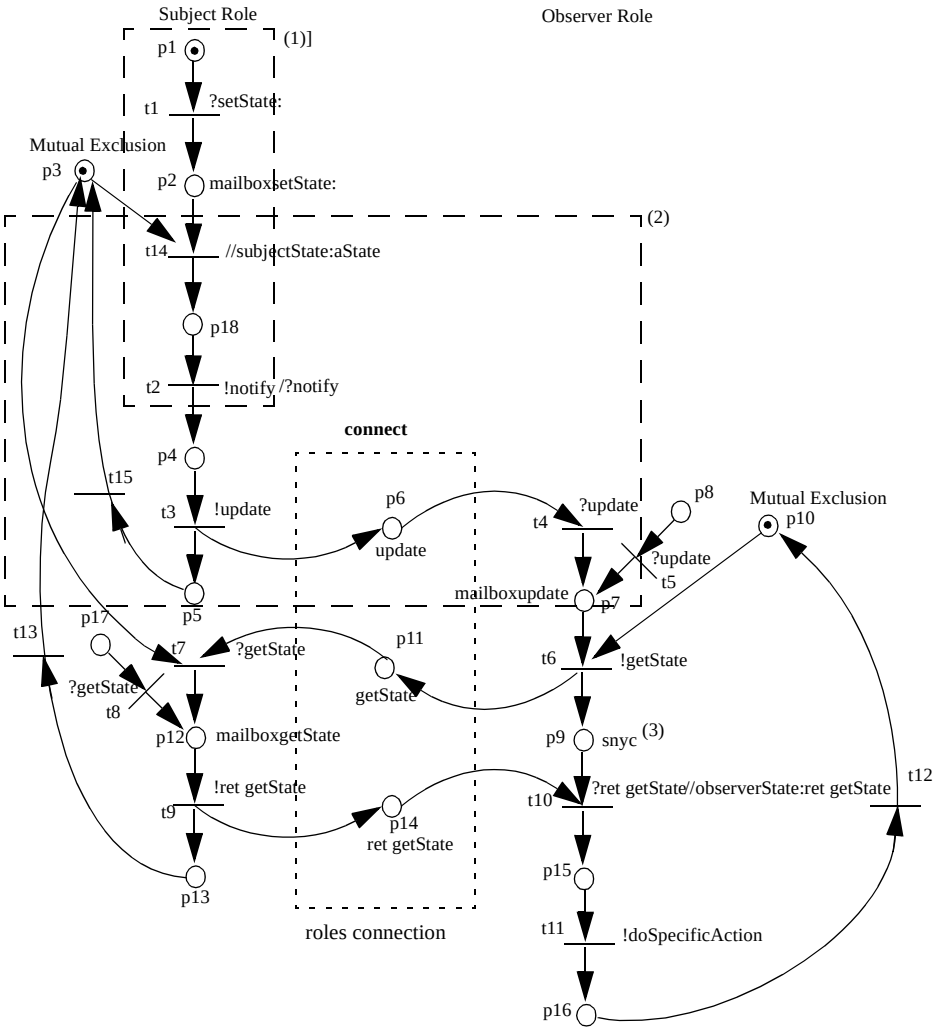


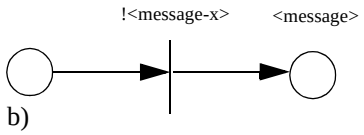
Figure 6.10 Predicate-Action Petri Net for the Observer Cooperation Rules

Some protocol errors in groups can already be detected at the end of the mapping and connection process. The existence for example of places related to messages with no arcs leaving the place indicates that the specification of the group contains undefined behaviors. In (**Figure 6.11** (a)) we can see who the specification of a behavior message-y in a role <Role-A> the sent of a message <message-x> to the role <Role-B>. The problem in the obtained Petri Net (b) is that there is not transition labeled ?<message-x> related to the

specification of the behavior <message-x> in the role <Role-B>. We will come back later during the verification of the Petri Nets in the type of structural errors that can be detected.

```
<Role-A> defineBehavior: message-y as:
[ <Role-B> <message-x> ]
```

a)



b)

Figure 6.11 : A protocol error

6.2.2 The Electronic Vote [Mins97a]

Problem Description

In the electronic vote an open group of participants is requested to vote on a specific issue. Every participant in the group can initiate a vote on any issue it chooses at any time. Participants vote by sending their results to the participant who initiated the vote. We assume that the period of time assigned to vote (i.e., voting period) is defined by the participant initiator of the vote.

The system must guarantee that the vote is fair: (1) each participant votes at most once and only within the voting period established, (2) that the counting is done correctly and only votes from participants of the group are counted and (3) that the result of the vote is sent to all the participants after the end of the voting period. Initially the policy applied to determine the result of the vote (i.e., counting policy) will be consensus (i.e. the number of positive votes should be equal to the number of voters to obtain a positive result otherwise the result will be negative). For simplicity reasons we will add two new requirements, they will simplify the final Petri Net representation obtained: 4) the initiator of the vote must remain neutral so, it does not vote and 5) the result of the vote should not be sent to the initiator of the vote he is the one who counts and does knows the result. These new requirements do not appear in the specification of the CoLas group but in the Petri Net obtained. In (**Figure 6.12**) we can see the CoLaS specification of the ElectronicVote coordination group.

Structural Analysis

In (**Figure 6.13**) we can see the Predicate-Action Petri Net obtained. From the structural representation of the Predicate-Action Petri Net we can immediately see that the Petri Net is composed by two unconnected subnets: one subnet that starts with the reception of the message ?startVote (place p1) and the other that starts with the reception of the message ?stopVote (place p20). We can conclude in this case that the two behaviors are independent (i.e., not related), each one can be executed independently of the other. Semantically we can interpret this as the fact that the vote process can not be stopped if no message stopVote is sent by the initiator of the vote. So, it is possible that the vote process never ends. From our point of view this is a simple example of the advantage of using Petri Nets for the validation of the coordination groups, because Petri Nets are a graphical tool, there are some structural problems that can be immediately detected.

We can also see in the Petri Net obtained that there are some constraints missing in the specification of the CoLaS group, for example:

- We do not control that the voteOn message is received only when a vote process is happening. It will be possible for example to cheat a voter by putting a token in the place p8 and thus to push it to send its vote even if no vote process is actually happening.
- We do not control that the vote message is received only when a vote process is happening and only by the members of the role Voter. If we put a token in place p13 it will trigger a sequence of actions that will modify the counting variables used in the group even if no vote process is happening. We do not control neither the identity of the voters to guarantee that only the voters in the group vote.
- We do not control that the stopVote is sent only when a vote process is happening and only by the initiator of the vote process. If we put a token in place p20 we can stop the vote process event if the message was not sent by the voter initiator of the vote.
- We do not verify the identity of the voter who sends the voteResult to the voters. If we put a token in place 23 for example it will be possible to cheat other voters and made them believe some specific result of the vote.

All the problems mentioned before are related with the reception of specific messages, in general all the places in the Petri Net related with the reception of messages must be analyzed separately to identify possible protocol problems. Of course this implies a certain knowledge of the semantics of the coordination specified in the coordination groups.

Some structural problems that can be easily detected in the Petri Nets obtained are:

- Transitions with no outgoing arcs leaving: as we already mentioned before, this problem implies that some behavior used in the coordination group was not defined.
- Transitions with conditions associated with the reception of a message with more than one entering arc: this problem implies that there exists more than one specification of the same behavior in the coordination group.
- Places with not outgoing arcs: these places do not represent necessary a problem, but they are extremely good candidates to generate deadlocks. A deadlock in a Petri Net occurs when no more transition can be fired at a given time. As we already mentioned before the Ignore rule specified in the CoLaS model define in a Petri Net a place ignore without outgoing arcs. Deadlocks in the coordination appear if another participant waits for the reply of the ignored message.
- Unconnected groups of places: the fact that all the places are not connected does not necessary represent a problem in the protocol, but it implies that it is possible that some places will never be reached when some behaviors are trigger in the coordination group. It is important to identify the potential causes of the unreachability of the places and to connect then to the rest of the net if necessary. In a coordination group this will imply to guarantee that every behavior specified in the group appears in coordination action of another behavior.

```
1.CoordinationGroup createCoordinationGroupClassNamed: #ElectronicVote.
2.
3.ElectronicVote defineRoleNamed: #Voter.
4.
5.Voter defineInterface: #(#opinion:).
6.ElectronicVote defineVariables: #(#numYes #numNot) initialValues: #(0 0).
7.ElectronicVote defineVariable: #voteInProgress initialValue: false.
8.ElectronicVote defineVariable: #votePeriodExpired initialValue: false.
9.Voter defineParticipantVariable: #hasVoted initialValue: false.
10.
11.[1] Voter defineBehavior: 'startVote:anIssue' as:
12.      [group voteInProgress: true.
13.      Voter voteOn: anIssue].
14.
15.[2] Voter defineBehavior: 'voteOn:anIssue' as:
16.      [sender vote:(self opinion: anIssue)].
17.
18.[3] Voter defineBehavior: 'vote: aVote' as:
19.      [aVote
20.          ifTrue: [group numYes++]
21.          ifFalse: [group numNot++].
22.      sender hasVoted: true ].
23.
24.[4] Voter defineBehavior: 'stopVote' as:
25.      [group votePeriodExpired: true.
26.      (group numYes = Voters size)
27.          ifTrue: [Voter voteResult: 'Yes']
28.          ifFalse: [Voter voteResult: 'No']].
29.
30.[5] Voter interceptAtEnd: 'stopVote' do:
31.      [Voter do:[:each | each hasVoted: false].
32.      group voteInProgress: false.
33.      group votePeriodExpired: false.
34.      group numYes: 0.
35.      group numNot: 0].
36.
40.[6] Voter ignore: 'vote:aVote' if:
41.      [group votePeriodExpired or:[sender hasVoted]].
42.
43.[7] Voter disable: 'startVote:anIssue' if:
44.      [group voteInProgress ].
```

Figure 6.12 : The Electronic Vote

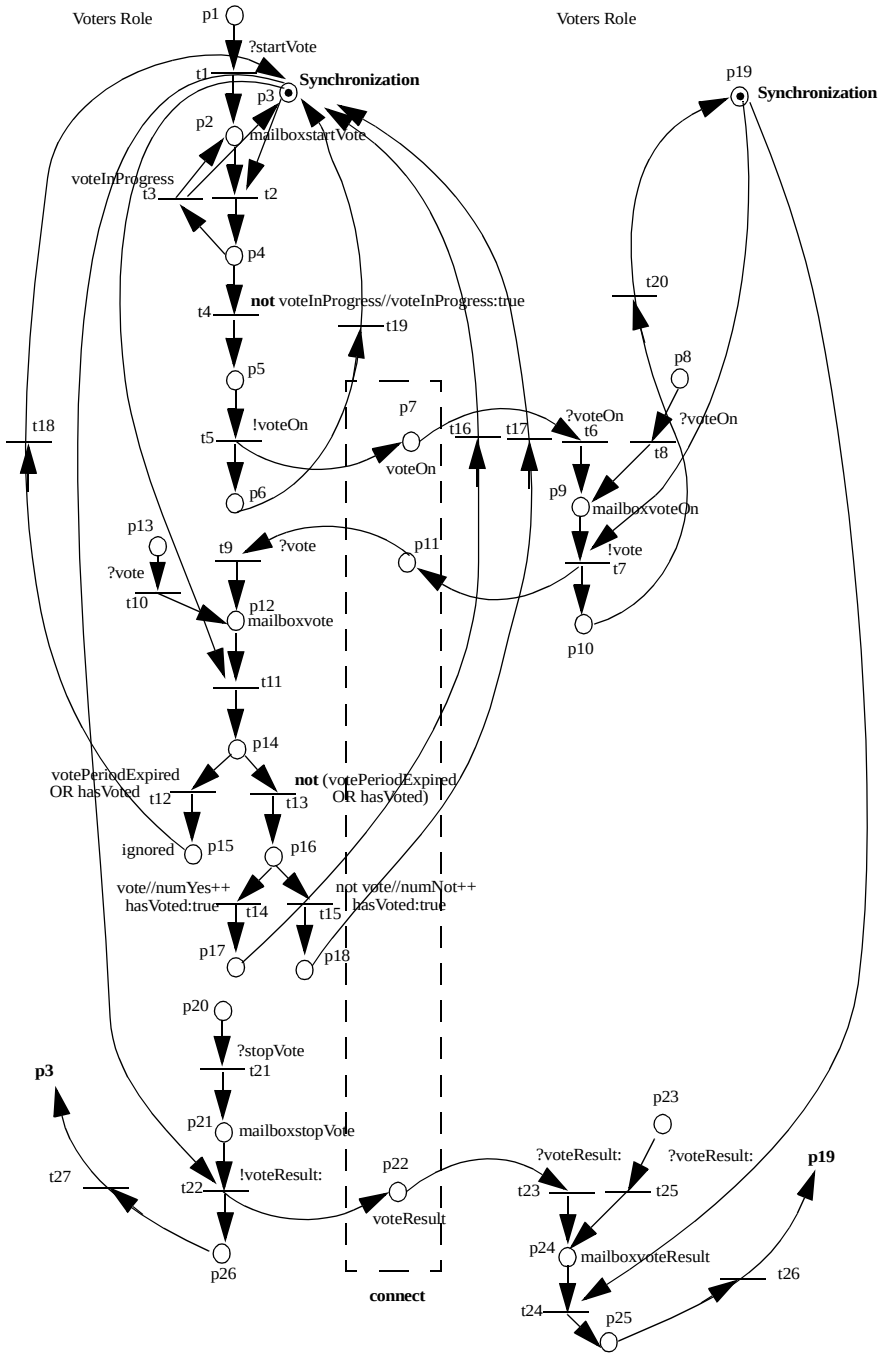


Figure 6.13 Predicate-Action Petri Net for the Electronic Vote

6.3 The Time Petri Net Analyser - TINA

Tina is a toolbox for the edition and analysis of Petri Nets and Time Petri Nets, developed in the Software and Tool for Communication Systems group (OLC) of LAAS/CNRS (http://www.laas.fr/tina/).

The Tina toolbox includes the tools:

nd (NetDraw): An editor for graphically or textually described Petri Nets, Time Petri Net and Automata. Interfaced with analysis tools below and drawing facilities.

tina: Construction of reachability graphs. Inputs nets in textual or graphical format. Outputs graphs in human readable form or in various formats for available model checkers and equivalence checkers. This tool is described in [Bert03a]. Depending on options retained, it builds:

- The coverability graph of a Petri Net, by the Karp and Miller technique.
- The marking graph of a bounded Petri Net, checking boundness on the fly.
- Partial marking graphs of a Petri Net, by the covering steps methods of [Vern96a][Vern97a], the method of persistent sets, or several combinations of them.
- Various state space abstractions for Time Petri Nets (state class graphs)

struct: Structural analysis of Petri Nets (preliminary). Computes generator sets for semi-flows or flows on places and/or transitions of a Petri Net. Also determines the invariance and consistence properties.

Petri Net Description

A net is described by a series of declarations of places and/or transitions and an optional naming declaration for the net. The grammar of a net declaration is the following (we will present here a simplified grammar):

```
<netdesc>          ::= 'net' <net>
<net>              ::= (<trdesc> | <pldesc> )*
<trdesc>           ::= 'tr' <transition> {<tinput> -> <toutput>}
<pldesc>           ::= 'pl' <place> {(<marking)} {<pinput> -> <poutput>}
<tinput>, <toutput> ::= (<place> {'*'<weight>})*
<pinput>, <poutput> ::= (<transition> {'*'<weigth>})*
<weigth>, <marking> ::= INT ---- unsigned integer
```

6.3.1 The “Subject And Views” [Helm90a]

```
Tina version 2.7.4 -- 06/13/05 --
LAAS/CNRS

mode -Ck

INPUT NET

parsed net subjectAndViews

18 places, 15 transitions

net subjectAndViews
tr t1 p1 -> p2
tr t10 p14 p9 -> p15
tr t11 p15 -> p16
tr t12 p16 -> p10
tr t13 p13 -> p3
tr t14 p2 p3 -> p18
tr t15 p5 -> p3
tr t2 p18 -> p4
tr t3 p4 -> p5 p6
tr t4 p6 -> p7
tr t5 p8 -> p7
tr t6 p10 p7 -> p11 p9
tr t7 p11 -> p12
tr t8 p17 -> p12
tr t9 p12 p3 -> p13 p14
pl p1 (1)
pl p10 (1)
pl p3 (1)
```

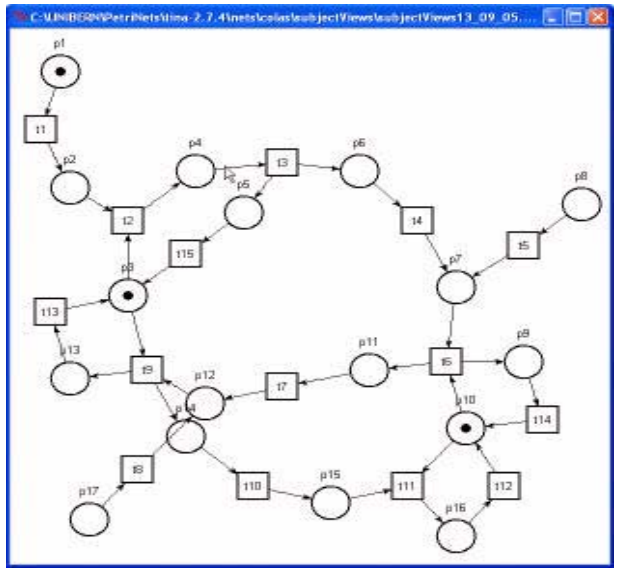


Figure 6.14 “Subject and Views Places-Transitions Petri Net”

We show in (**Figure 6.14**) the Place-Transition Petri Net used to perform the reachability analysis in the TINA tool for the “Subject-Views” example (subsection **6.2.1**). The initial marking used specifies one unique token in places p1, p10 and p3. The presence of a token in place p1 represents the arrival of a message setState to the subject. This event is associated with the modification of the state of the subject playing the role Subject. The two other places p3 and p10 are used to model the mutual exclusion synchronization policy controlling the execution of participants playing roles Subject and Observer in the group respectively. The existence of a unique token in each one of these places models the fact that only one method can be executed by the participants at the same time. The message setState can only be executed by the subject is there is a token in place p3 for example. If another setState message arrives to the subject during the execution of the first setState method the message is stored in place p2 (representing the mailbox of the subject participant) until the first message is executed completely and the token used for the synchronization is restored into the place p3.

Reachability Analysis

```
REACHABILITY ANALYSIS

bounded

20 classe(s), 25 transition(s)

CLASSES:

0 : p1 p10 p3
1 : p10 p2 p3
2 : p10 p18
3 : p10 p4
4 : p10 p5 p6
5 : p10 p3 p6
6 : p10 p3 p7
7 : p11 p3 p9
8 : p12 p3 p9
9 : p13 p14 p9
10 : p13 p15
11 : p13 p16
12 : p10 p13
13 : p10 p3
14 : p16 p3
15 : p15 p3
16 : p14 p3 p9
17 : p10 p5 p7
18 : p11 p5 p9
19 : p12 p5 p9
```

```
REACHABILITY GRAPH:

0 -> t1/1
1 -> t14/2
2 -> t2/3
3 -> t3/4
4 -> t15/5, t4/17
5 -> t4/6
6 -> t6/7
7 -> t7/8
8 -> t9/9
9 -> t10/10, t13/16
10 -> t11/11, t13/15
11 -> t12/12, t13/14
12 -> t13/13
13 ->
14 -> t12/13
15 -> t11/14
16 -> t10/15
17 -> t15/6, t6/18
18 -> t15/7, t7/19
19 -> t15/8
```

```
LIVENESS ANALYSIS

not live

1 dead classe(s), 1 live classe(s)
2 dead transition(s), 0 live transition(s)

dead classe(s): 13

dead transition(s): t8 t5
```

```
STRONG CONNECTED COMPONENTS:

19 : 0
18 : 1
17 : 2
16 : 3
15 : 4
14 : 17
13 : 18
12 : 19
11 : 5
10 : 6
9 : 7
8 : 8
7 : 9
6 : 16
5 : 10
4 : 15
3 : 11
2 : 14
1 : 12
0 : 13
```

Figure 6.15 Reachability Analysis for the Subject-Views Petri Net

The results obtained from the reachability analysis indicate that the Petri Net is bounded and not live. Bounded means that all the time the number of tokens remain finite. This indicates that the coordination rules in the specification of the coordination group do not specify cycles generating infinite number of tokens. What does the “not live” property means? the results show that there is one dead class of transitions 13 composed by the places p3 and p10 and two dead transitions t5 and t8. The fact that p3 and p10 are dead classes means that at some point during the evolution of the Petri Net we find tokens in these two places but

no transition can be fired anymore. If we understand the way we model the CoLaS group in Petri Nets this is normal, even more, these should be the only dead transitions in the Petri Net. In the initial marking of the Petri Net we place a unique token in place p1 representing the arrival of a message setState to the subject participant, the execution of the message requires the existence of a token in the place p3 representing the synchronization policy (similar for place p10 in the role Observer). At the end of the execution of the messages we always restore the tokens in the synchronization policies but not the tokens corresponding to the reception of the messages, it does not have any sense to do it. This is the reason why we reach a state where two tokens are found in places p3 and p10 and everything is blocked. Remember that the validation of behavioral properties depends always of the initial marking and that our initial marking represents a unique message received setState in place p1.

In the other hand, the two transitions t5 and t8 are considered dead because they were never fired. Again this is normal because our initial marking represented the arrival of a message setState and not the arrival of messages update and getState in places p8 and p17. In the Petri Net places p8 and p17 exist because we model the fact that update and getState messages can be received independently of the reception of a setState message in the subject participant. If we add tokens in these places in the initial marking we will be modeling the actual reception of these two messages by the subject and the observer participants.

We mentioned before that in our case the fact that our Petri Net was bounded was normal. Another reason that justifies that justifies this is the fact that we model a perfect communication media, in the case for example we would have modelled differently the communication media the validation of the boundness of the Petri Net will become basic to determine problems related with the spontaneous generation of messages in the medium and problems related with the duplication of messages. A Petri Net not bounded will imply a Petri Net composed of an infinite number of states, this will indicate in our coordination groups potential branches of coordination code with not end. For example possible infinite cycles of coordination behavior.

We will try to complete now the analysis resulting from the TINA tool for the list of behavioral properties listed in **B.2.1**in Appendix B of this thesis. We have:

- Is the Petri Net Safe? Yes, the Petri Net is safe, because from the initial marking (i.e. p1, p3 and p10 containing one token each one) for all possible accessible markings (all different classes in **Figure 6.15**) every place contains at most one token.
- Is the Petri Net Conform? No. In principle not because it is not live. But we already explained why the net is not alive.
- Is the Petri Net free from deadlocks? No. The Petri Net is not free from deadlocks because as we already mentioned before there is a marking (class 13 in **Figure 6.15**) where no transition is enabled. As we already explained this is normal in our modeling because we are modeling the execution of a unique message setState received by the subject participant. We do not model a continuous reception of messages by the participants only the execution flow of one message. To model the reception of several messages for example we will need to set more tokens in the initial marking in places related with the reception of messages (i.e. ?<message> places).
- Is the Petri Net reversible? No. It is not possible from each marking reachable from the initial marking to reach the initial marking. In other words it is not possible to get back to the initial state. This is normal in our case, if not this will means that the received message is executed infinitely.

6.3.2 The Electronic Vote [Mins97a]

```
Tina version 2.7.4 -- 06/13/05 --
LAAS/CNRS

mode -Ck

INPUT NET
parsed net elecvote
26 places, 27 transitions

net elecvote
tr t1 p1 -> p2
tr t10 p13 -> p12
tr t11 p12 p3 -> p14
tr t12 p14 -> p15
tr t13 p14 -> p16
tr t14 p16 -> p17
tr t15 p16 -> p18
tr t16 p17 -> p3
tr t17 p18 -> p3
tr t18 p15 -> p3
tr t19 p6 -> p3
tr t2 p2 p3 -> p4
tr t20 p10 -> p19
tr t21 p20 -> p21
tr t22 p21 p3 -> p22 p26
tr t23 p22 -> p24
tr t24 p19 p24 -> p25
tr t25 p23 -> p24
tr t26 p25 -> p19
tr t27 p26 -> p3
tr t3 p4 -> p2 p3
tr t4 p4 -> p5
tr t5 p5 -> p6 p7
tr t6 p7 -> p9
tr t7 p19 p9 -> p10 p11
tr t8 p8 -> p9
tr t9 p11 -> p12
pl p1 (1)
pl p19 (1)
pl p3 (1)
```

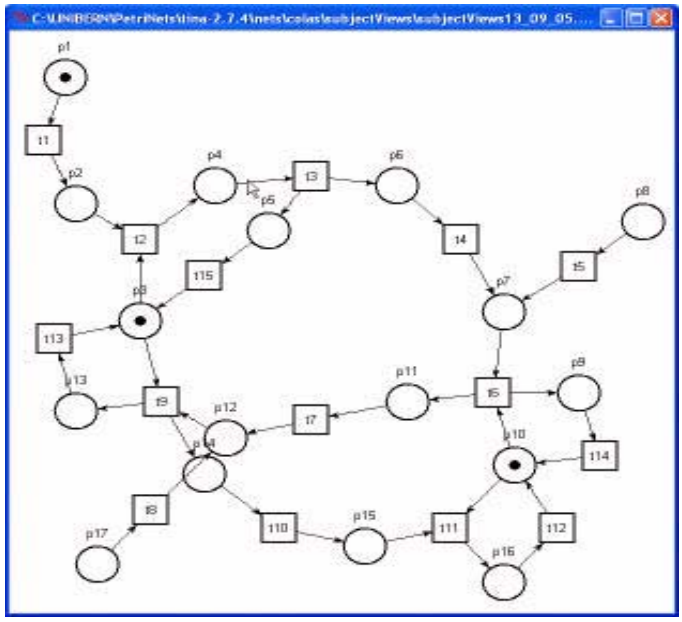


Figure 6.16 Electronic Vote Places-Transitions Petri Net.

We show in (**Figure 6.16**) the Place-Transition Petri Net used to perform the reachability analysis in the TINA tool for the "Electronic Vote" example (subsection **6.2.2**). The initial marking used specifies one unique token in places p1, p3 and p19. The presence of a token in place p1 represents the arrival of a message startVote to a participant playing the role Voters. The two other places p3 and p19 are used to model the mutual exclusion synchronization policy controlling the execution of method invocation requests in the participants. The existence of a unique token in each one of these places models the fact that only one method invocation request is executed by the participants at the same time.

It is possible to see in the Petri Net generated from the ElectronicVote group that we represent "twice" the role Voters specified in the group. Normally a unique Petri Net is generated per roles in our mapping function, in this case and in other two identify clearly the flow of messages and synchronizations described in the group we model a second participant playing the role Voters, different to the voter initiator of the vote.

Reachability Analysis

```
REACHABILITY ANALYSIS

bounded

28 classe(s), 45 transition(s)

CLASSES:

0 : p1 p19 p3
1 : p19 p2 p3
2 : p19 p4
3 : p19 p5
4 : p19 p6 p7
5 : p19 p3 p7
6 : p19 p3 p9
7 : p10 p11 p3
8 : p11 p19 p3
9 : p12 p19 p3
10 : p14 p19
11 : p15 p19
12 : p19 p3
13 : p16 p19
14 : p17 p19
15 : p18 p19
16 : p10 p12 p3
17 : p10 p14
18 : p10 p15
19 : p10 p3
20 : p10 p16
21 : p10 p17
22 : p10 p18
23 : p19 p6 p9
24 : p10 p11 p6
25 : p11 p19 p6
26 : p12 p19 p6
27 : p10 p12 p6


LIVENESS ANALYSIS ---------------------------------
-------------

not live

1 dead classe(s), 1 live classe(s)
9 dead transition(s), 0 live transition(s)

dead classe(s): 12

dead transition(s): t8 t27 t26 t25 t24 t23 t22 t21 t10
```

```
REACHABILITY GRAPH:

0 -> t1/1
1 -> t2/2
2 -> t3/1, t4/3
3 -> t5/4
4 -> t19/5, t6/23
5 -> t6/6
6 -> t7/7
7 -> t20/8, t9/16
8 -> t9/9
9 -> t11/10
10 -> t12/11, t13/13
11 -> t18/12
12 ->
13 -> t14/14, t15/15
14 -> t16/12
15 -> t17/12
16 -> t11/17, t20/9
17 -> t12/18, t13/20, t20/10
18 -> t18/19, t20/11
19 -> t20/12
20 -> t14/21, t15/22, t20/13
21 -> t16/19, t20/14
22 -> t17/19, t20/15
23 -> t19/6, t7/24
24 -> t19/7, t20/25, t9/27
25 -> t19/8, t9/26
26 -> t19/9
27 -> t19/16, t20/26

STRONG CONNECTED COMPONENTS:

26 : 0
25 : 1 2
24 : 3
23 : 4
22 : 23
21 : 24
20 : 27
19 : 25
18 : 26
17 : 5
16 : 6
15 : 7
14 : 16
13 : 17
12 : 20
11 : 22
10 : 21
9 : 18
8 : 19
7 : 8
6 : 9
5 : 10
4 : 13
```

Figure 6.17 Reachability Analysis for the Electronic Vote Petri Net

The results obtained from the reachability analysis indicate that the Petri Net is bounded and not live. As in the first case study, What does the “not live” property means? the results show that there is one dead class

of transitions 12 composed by places p3 and p19; and nine dead transitions t8, t10, t21, t22, t23, t24, t25, t26 and t27. The fact that p3 and p10 are dead classes means that at some point in the time these two places will contain tokens but no transition will be fired. Again, if we understand the way we model the CoLaS group in Petri Nets this is normal. In the initial marking we place a unique token in place p1 representing the arrival of a message startVote to a voter, once the token is consumed, we restore the tokens in the places p3 and p19 (places representing the synchronization policies), but we do not regenerate the token in place p1.

The nine transitions t8, t10, t21, t22, t23, t24, t25, t26 and t27 are considered dead because they were never fired. Again this is normal because our initial marking represented the arrival of a message starVote and not the arrival of the messages voteOn, vote and stopVote in places p8, p13 and p20. In the Petri Net places p8, p13 and p20 exists because we model the fact that also the participants may receive the voteOn, vote and stopVote messages independently of the reception of a startVote message in the voter participant. The result of the reachability analysis confirms what we mentioned before in subsection (**6.2.2**) concerning the fact that if no stopVote message arrives to the participant the vote process could be endless. The second subnet starting in the place p20 and representing the actions executed when the stopVote message arrives to the voter is not connected to the subnet starting in place p1 related to the reception of the startVote message.

We will try to complete now the analysis resulting from the TINA tool for the list of behavioral properties listed in **B.2.1**in Appendix B of this thesis. We have:

- Is the Petri Net Safe? Yes, the Petri Net is safe, because from the initial marking (i.e. p1, p3 and p19 containing one token each one) for all possible accessible markings (all different classes in **Figure 6.17**) every place contains at most one token.
- Is the Petri Net Conform? No. In principle not because it is not live. But already explain why the net is not alive.
- Is the Petri Net free from deadlocks? No. The Petri Net is not free from deadlocks because as we already mentioned before there is a marking (class 12 in **Figure 6.17**) where no transition is enabled. As we already explained this is normal in our modeling because we are modeling the execution of a unique message startVote received by a voter.
- Is the Petri Net reversible? No. It is not possible from each marking reachable from the initial marking to reach the initial marking. In other words it is not possible to get back to the initial state. Again it does not have any sense to reach the initial marking in our case. This will means that received messages are executed infinitely.

6.4 Related Work

Some related work in the formalization of coordination models with Petri Nets have been done by Buffo in [Buff97a]. The subject has become an important topic in the coordination research, in 2004 the first international workshop on coordination and Petri Nets (http://www.cs.unibo.it/atpn2004/pnc04.html) was organized. From our point of view the most important work is SynchNet [Ziae03a] given the similarity of the approach with ours. SynchNet is a compositional meta-level language for coordination of distributed object systems inspired by Petri Nets. The based-object model of SynchNet is inspired by the Actor model [Agha86a]. Each object is identified by a unique reference. Objects communicate by an asynchronous communication mechanism called ARMI (Asynchronous Remote Method Invocation). In ARMI, the source object asynchronously sends a message specifying the method to the invoked in the remote object accompanied by the arguments to be passed.

In (**Figure 6.18**) we can see the example of two transmitters which communicate via asynchronous sending of messages. The delivery of messages triggers invocations of methods in the objects that control the transmitters. Each transmitter is controlled by an object with two methods: an on method that determines transmission power and turns on the transmitter and an off method that turns it off. A global requirement is that no two transmitters may be transmitting at the same time. Turn off messages are sent to turn off the transmitters before the next transmission begins. In Petri Net terms the ob.on may be invoked only when in the state of the TransmitterME there is one ob'.off token available for each object ob' in the group. Once the invocation of an ob.on is decided the state of the generated synchnet is modified by adding one token corresponding to the invoked method ob.on and consuming the tokens specified in the consumes multilist. The only requirement on the invocation of an ob.off method is that the ob is turned on. After consuming the token ob.on, other transmitters may get a chance to be turned on. The "[with fairness]" condition requires that all pending methods to objects in the group must be given a fair opportunity of invocation.

```
1.synchnet TransmitterME (Transmitters: list of TransmistterC)
2.   init = { ob'.off | ob' in Transmitters }
3.   foreach ob in Transmitters [with fairness]
4.       method ob.on
5.            requires { ob'.off | ob' in Transmitters }
6.            consumes { ob.off }
7.       method ob.off
8.            requires { ob.on }
9.            consumes { ob.on }
10.end TransmitterME
```

Figure 6.18 TransmitterME SynchNet specification

In (**Figure 6.19**) we can see the graphical version of the synchnet generated by the expression TransmittersME({t1,t2}), which is an instantiation of TransmitterME on two transmitters t1 and t2.

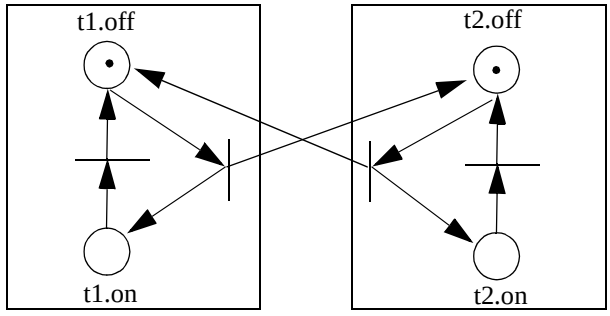


Figure 6.19 Diagram of TransmitterME instantiated on t1 and t2 in its initial state

In the SynchNet work [Ziae03a] it is pointed out the importance of freedom from deadlock in the coordination of a collection of interacting objects. Deadlock is defines as the situation in which the state of the one or more synchnet disables certain methods forever. According to Ziaei and Agha one can verify deadlock-freedom on a synchnet by performing reachability analysis. However, since reachability of Petri nets has

non-elementary complexity, they introduce an alternative formal method for the development of deadlock-free synchnets. They introduce a preorder relation <=that is deadlock-freedom preserving: S <= S' implies that whenever S' does not deadlock in an environment E, using S' in environment E would not result in deadlock either.

6.5 Conclusions and Contributions

We have presented in this chapter of the thesis a formal methodology to verify formal properties in the CoLaS groups. The methodology is based on the specification of a mapping function F transforming CoLaS groups in Predicate-Action Petri Nets. The Predicate-Action Petri Nets are an extension of Place-Transition Petri Nets with transitions labeled with conditions and actions on variables specified in the Petri Net.

F : CoLaS groups- > Predicate-Action Petri Nets

The mapping function F was defined recursively over each one of the elements that compose the specification of a CoLaS group. For each element we showed its corresponding Predicate-Action Petri Net and the way how all the different Petri Nets obtained must be connected to obtain the representation of the group. Different models of connections were illustrated, they differ on the assumptions made on the communication media enabling the communication.

We have selected to use Petri Nets to perform the validation of the CoLaS groups first because by using Petri nets we can benefit from the rich and well studied theory of Petri nets. The theory includes formal characterizations of many interesting properties along with decision algorithms to decide those properties. There exists a lot of analysis tools that made these theories accessible to researchers. And second, because we believe that a graphical representation of the coordination and in particular of the flow of the exchange of messages (i.e, the tokens in the Petri nets) between the group participants facilitates the understanding and the detection of coordination problems in the Petri Nets. This point was illustrated in the examples presented in this chapter, several coordination problems in the coordination specification of the groups were detected by a simple graphical check of the Petri Nets.

We showed concretely using the examples of the "Subject and Views" and the "Electronic Vote" how to use our methodology. For each example we showed the specification of the CoLaS group and the Predicate-Action Petri Net obtained from the application of the mapping function F to the group. We used enumeration analysis techniques in the Petri Nets obtained to verify certain properties. The enumeration analysis technique is based on the construction of an accessibility graph from the initial marking M0. The graph is obtained by firing all the possible transitions until no new transition could be fired. Properties like: boundness, safeness, liveness, reversability and blockings can be tested in nets.

The big problem with the approach developed in this chapter for the validation of group properties concerns the interpretation of the results. This is not a specific problem of this approach but in general a problem of all the approaches that transform the original model in a different formal model to realize the validation of properties. In our case it is difficult to give a generic recipe about how to interpret the results obtained from the validation of properties in the Petri Nets in Tina. We have seen for example in the case of the "Subject-View" and the "Electronic Vote" Petri Nets that even if the results indicated dead classes in the reachability graph this does not necessarily means that there was a problem. We believe that the interpretation of results in the Petri Nets obtained from the groups constitute an interesting future work in the CoLaS coordination model.

We do not pretend that we have found the complete solution to the problem of formally verifying the specifications contained in CoLaS groups specifications. Nevertheless, we believe that the methodology presented here cover the most important aspects of the CoLaS groups. It provides programmers with a simple tool to validate basic properties of group specifications.

There are several modeling aspects which are not cover by this method, for example we only model the existence of a unique participant per role. An idea to represent multiple participants will be to replicate the Petri Nets obtained for the role, one for each participant playing the role and connect them all together.

We do not model the spontaneous generation of the messages received by the participants. In the initial marking we specify the messages received by the participants, the simulation of the reception of different types of messages at different moments during the execution will imply the introduction of temporal constraints in the Petri Net.

We do not model the manipulation of variables in the examples, we showed at the beginning of section **6.1** how this can be done. Using the places associated with the variables it is possible to detect non-structural problems like for example conditions related with variables which never validate to true. Normally if a condition never validates to true this indicates that the transition is never fired. We explained in this chapter how variables in Predicate-Action Petri Nets can be transformed into simple places in Place-Transition Petri Nets to perform the verification of properties.

We do not model all the dynamic aspects of the CoLaS model. For example, we do not model the fact that new participants can join the roles in the groups at any time. Nevertheless, we believe that we can model the modification of the rules and the creation of new groups. The addition of new rule to a group corresponds to the addition and the connection of a new Petri Net representing the new rule. The addition of a new group corresponds to the addition and connection of a new Petri Net representing the new group.

We did not present the result of the validation of properties in the examples using different initial markings. An exhaustive validation of the coordination specified in a group must consider all possible different initial markings. An interesting work would be to determine which initial markings will be sufficient to test in order to conclude that the coordination specified is free of problems (given that generating all possible marking is a problem with exponential complexity, factorial of the number of places). How to select interesting initial markings constitute an interesting future work from our point of view. According to our experience we suggest to consider as initial markings all those including the reception of messages associated with all the possible behaviors specified in the groups (i.e. ?<message> places).

How to test the possible interference problems caused by the execution of more than one message in the same role and in different roles is another interesting and problem. When the number of behaviors (i.e., cooperation rules) defined in a group is small it is still possible to analyze this interference, but as soon as the number of behaviors in a role increases the task become almost impossible to achieve. To analyse the interference of messages in a role or between different roles we must test initial markings in which we take two by two all the different possible combinations of messages that can be received by the participant in a role.

Finally, we believe that it is also important to understand the kinds of things that we can not verify in the CoLaS groups using our approach. Definitely we can not say anything about how behaviors specified in the computational part of the participants or behaviors defined in other groups in which the participants also participate affect the coordination specified in a group. Even if we are able to determine that the specification of a group is deadlock free it does not means that deadlocks will not appear when a participant participates in other groups at the same time.

CHAPTER 7

Case Studies

In the introduction of this thesis we pointed out the limitations that concurrent object-oriented technology has for building and maintaining concurrent object-oriented systems. From our point of view one of the most important problems in building and maintaining concurrent object-oriented systems is that the functionality of the active objects that compose the systems and they way they cooperate and synchronize are mixed within the active objects code. The mixing of cooperation and synchronization concerns makes the concurrent systems built difficult to understand, modify and customize. We also pointed out the importance that coordination models and languages have in the specification and construction of concurrent and distributed systems. Coordination models and languages promote the separation of the computation and the coordination aspects in those systems. The computation aspect concerns the specification of the elements that compose those systems and the coordination aspect the glue that binds all the elements together. We believe, and this is the key point of this thesis, that by separating the specification of the coordination aspect from the computation aspect in concurrent object-oriented systems and by specifying the computation in active objects we simplify their specification, understanding, construction, evolution and validation of properties.

Although coordination is a fundamental aspect of object-oriented programming languages for concurrent systems, existing concurrent object-oriented programming languages provide only limited support for its specification and abstraction. In Chapter 2 of this thesis we identified the most important problems we believe existing concurrent object-oriented programming languages have in supporting the specification of the coordination aspect in concurrent object-oriented systems. They are:

- Lack of high level coordination abstractions.
- Lack of coordination abstractions for complex interactions.
- Lack of separation of computation and coordination concerns.
- Lack of support for the evolution of the coordination code.
- Lack of support for the validation of the coordination code.

The CoLaS coordination model and language that we introduced in this thesis introduces a high level coordination abstraction called Coordination Group that allows programmers to design, to specify and to implement the coordination of groups of collaborating active objects in concurrent object-oriented systems. In Chapter 2 of this thesis we also identified the requirements that we consider to be fundamental for the specification of a coordination model and language for concurrent object-oriented systems. These requirements can be summarized as follows

- The coordination policies must be defined independently of the coordinated entities.
- It must be possible to define new coordination policies in the coordination model.
- It must be possible to incrementally define new coordination policies in the coordination model.
- The coordination policies must be multi-party.
- The coordination policies must be declaratively defined in the coordination model.

- The coordination policies must be control-driven defined in the coordination model.
- The coordination model must be transparently integrated into the host language.
- The architecture of the coordination model must be hybrid.
- The coordination policies must include the possibility to define proactions in participants.
- The coordination policies must include the possibility to refer to the state of the participants and to the coordination history.
- It must be possible to dynamically modify the coordination policies.
- It must be possible to prove the capability of the coordinated entities to be coordinated.
- It must be possible to validate basic safety and liveness properties of the coordination.

We believe and we will show it again in this chapter that our approach CoLaS fully satisfies the list requirements introduced above. The goal of this chapter is to show concretely with six examples how our approach can be used to tackle the complexity of specifying and building concurrent object-oriented systems. Some of the examples were taken from the coordination literature and some others from previous thesis done in the coordination area. The examples selected cover the most important coordination problems in concurrent systems identified in the Chapter 2 of this thesis: transfer of information, allocation/access of/to shared resources, simultaneity constraints, condition synchronizations, execution orderings, task/subtask dependencies, group decisions and global constraints. Not all the examples were implemented in real full-scale. We believe that the diversity of the problems and their relevance as representative of the different types of coordination problems in concurrent systems will be enough to convince the reader that CoLaS is an interesting and effective model to manage coordination problems in concurrent object-oriented systems. We will show in these examples how designers and programmers of concurrent object-oriented systems can get advantage of the separation of the coordination and computation concerns in the specification, construction and evolution of their systems.

We have divided the presentation of this chapter into two parts:

In the first part of this chapter we present the six examples selected. For each example introduced we specify: a short description of the problem; a description of our solution (sometimes we include an interested solution already proposed to solve the problem); a description of the coordination problems that appear in the example and the CoLaS specification containing the specification of the solution to the problem. We do not focus exclusively on the CoLaS solution to the problems, the most important is that for each example we compare our solution with a “classical” solution in a concurrent object-oriented language without coordination abstractions. We use Smalltalk as an object-oriented programming language and we add to the basic core of Smalltalk classes the Actalk framework [Brio89b]: a set of classes specialized in the representation of active objects. We also show for each example how the solution specified in CoLaS satisfies the requirements (not always all at the same time) identified as ideal for a coordination model and language for concurrent object-oriented systems based on active objects. For most of the solutions we complete the presentation of the solution with UML class diagrams and/or UML interaction diagrams describing the most important aspects of the solution.

The selected examples are:

- A Context-Sensitive Help: a system to provide help information in any part of an interface. This example illustrates the following coordination problem: transfer of information.
- The Dining Philosophers: a system simulating a group of philosophers eating and thinking. This example illustrates the following coordination problems: transfer of information, condition synchronizations and allocation/access of/to shared resources.

- The Vending Machine: a system to control a vending machine. This example illustrates the following coordination problems: transfer of information, simultaneity constraints, execution orderings and condition synchronizations.
- The Online-Music Shop: an online music reseller system. This example illustrates the following coordination problems: transfer of information, task/subtask and execution orderings.
- The Ornamental Garden: a system to control the entrance and the number of visitors to a garden. This example illustrates the following coordination problems: global constraints.
- The New Server Election: election of a new replication server. An election is a procedure carried out to choose a process from a group, for example to take over the role of a server that has failed. This example illustrates the following coordination problems: transfer of information and group decisions.

Finally at the end of this chapter we present our conclusions about the work presented here and we point out the main contributions of this chapter to the thesis.

7.1 A Context-Sensitive Help [Gamm95a]

Problem Description

Consider a context-sensitive help facility for a graphical user interface. The user can obtain help information on any part of the interface just by clicking on it. The help that is provided depends on the part of the interface that is selected and its context; for example, a button widget in a dialog box might have different help information than a similar button in the main window. If not specific help information exists for that part of the interface, then the help system displays a more general help message about the immediate context.

Solution: Chain of Responsibility Design Pattern

A natural solution to this problem consists to organize the help information according to its generality (i.e., from the most specific to the most general). The help request needs to be decoupled from the objects that might provide the help information. The Chain of Responsibility design pattern proposes an interested solution to this problem. The pattern avoids coupling the sender of a request to its receiver by giving more than one object a change to handle the request. The pattern chains the receiver objects and pass the request along the chain until an object handles it. Each object in the chain receives the request and either handles it or forwards it to the next object in the chain. The object that made the request has no explicit knowledge of who will handle its request.

Coordination Aspects

- Transfer of information: each object communicates with the next object handler in the chain to pass the requests if necessary. Each object in the chain may decide to handle the request of to forward it to the next object in the chain.

Structure

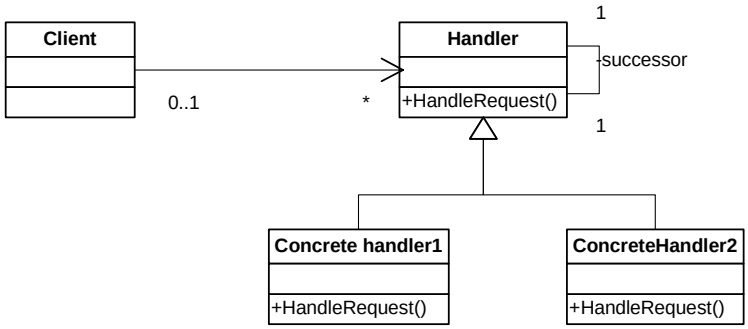


Figure 7.1 : Chain of Responsibility structure

In (**Figure 7.1**) we show the structure of the Chain of Responsibility pattern as presented in [Gamm95a]. The class Handler defines an interface for handling requests and implements the successor link. The class ConcreteHandler handles the requests from which it is responsible. If the ConcreteHandler can handle the request, it does so; otherwise it forwards the requests to its successor. The class Client initiates the request to a ConcreteHandler object in the chain.

Smalltalk Specification

We will present now how the solution to the Context-Sensitive Help problem can be implemented using Smalltalk + Actalk[Brio89a] to represent the active objects. The Actalk framework includes a class called ActiveObject from which our participants in the solution inherit. The ActiveObject class includes all the necessary support to create and manipulate active objects. In our presentation we will precede the specification of methods with the symbol ">>" (only for notation purposes). Active objects communicate asynchronously and replies are send back using futures.

In (**Figure 7.2**) we can see the implementation of the abstract class Handler (line 1). The class specifies the successor instance variable to store the successor handler in the chain of responsibility (line 5). The methods >>successor and >>successor: define accessors methods for the successor instance variable (line 10 and 12). The >>handleRequest: method (line 14) defines the core of the chain of responsibility pattern, when the handler can handle the request (line 15) it calls the executeRequest: method (line 16) to execute the request, when not is the successor in the chain of responsibility (line 19) who will be requested to execute the request (i.e., if there is a successor of course). The class Handler lets the concrete subclasses the responsibility to specify the methods >>canHandle: and >>executeRequest:. The method >>canHandle: validates whether the handler can handle (or must handle the request). It is up to each handler to determine whether it can or not handle the request, in principle the validation is done based in the coordinates of the request. If the coordinates fall within the area graphically cover by the handler the handler must handle the request.

```
1.CaseStudies defineClass: #Handler
2.       superclass: #{Actalk.ActiveObject}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: 'successor '
6.       classInstanceVariableNames: ''
7.       imports: ''
8.       category: 'CR_Pattern'
9.
10.>>successor
11.      ^successor
12.>>successor: aHandler
13.      successor := aHandler
14.>>handleRequest: aRequest
15.      (self canHandle: aRequest)
16.          ifTrue: [ self executeRequest: aRequest ]
17.          ifFalse:
18.              [self successor
19.                  ifNotNil: [self successor handleRequest: aRequest ]]
20.>>canHandle: aRequest
21.      ^self subclassResponsibility
22.>>executeRequest: aRequest
23.      ^self subclassResponsibility
```

Figure 7.2 Handler Class

In (**Figure 7.3**) we can see the implementation of the concrete handler classes. The class View (line 1) represents a graphical view. The class View defines an instance variable named widgets (line 5) containing the list of all the widgets that currently appear in the view. The instance method >>handleRequest: (line 10) specifies that the any request received by the view is sent to the first widget in the list of widgets. The class Widget (line13) specifies an abstract class for all different types of widgets (i.e., buttons, menus, etc.). In the class Widget we specify all the behavior common to all the different types of widgets that we manage in the graphical views. The method >>executeRequest: (line 22) delegates the execution of the request to the method >>displayHelp in the widget. It is the responsibility of each widget to specify the concrete implementation of the method >>displayHelp (line 24). The method >>canHandle: (line 26) specifies that a widget can handle the request if the position of the request (i.e., the coordinates of the mouse click) fall within the coordinates of the current position of the widget. The classes Button and Menu (lines 29 and 38) correspond to concrete implementations of widgets. Because the class Widget is a subclass of the class Handler, buttons and menus behave also as handlers.

```
1.CaseStudies defineClass: #View
2.       superclass: #{Actalk.ActiveObject}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: 'widgets '
6.       classInstanceVariableNames: ''
7.       imports: ''
8.       category: 'CS_Pattern'
9.
10.>>handleRequest: aRequest
11.      self widgets first handleRequest: aRequest
12.
13.CaseStudies defineClass: #Widget
14.      superclass: #{CaseStudies.Handler}
15.      indexedType: #none
16.      private: false
17.      instanceVariableNames: 'position model'
18.      classInstanceVariableNames: ''
19.      imports: ''
20.      category: 'CR_Pattern'
21.
22.>>executeRequest: aRequest
23.      ^self displayHelp
24.>>displayHelp
25.      ^self subclassResponsibility
26.>>canHandle: aRequest
27.      ^self position contains: aRequest position
28.
29.CaseStudies defineClass: #Button
30.      superclass: #{CaseStudies.Widget}
31.      indexedType: #none
32.      private: false
33.      instanceVariableNames: ''
34.      classInstanceVariableNames: ''
35.      imports: ''
36.      category: 'CR_Pattern'
37.
38.CaseStudies defineClass: #Menu
39.      superclass: #{CaseStudies.Widget}
40.      indexedType: #none
41.      private: false
42.      instanceVariableNames: ''
43.      classInstanceVariableNames: ''
44.      imports: ''
45.      category: 'CR_Pattern'
```

Figure 7.3 Concrete Handlers

Analysis

From the coordination point of view we can see in the implementation of the solution how the coordination and computation aspects are mixed within the classes of the participant widgets. The classes Button and Menu inherit all the coordination behavior specific to the implementation of the chain of responsibility pattern from the class Handler. This implies: 1) that the coordinated entities (i.e., the buttons and menus) "know" in advance about the coordination in which they will participate; 2) that if the coordinated entities participate in other coordination solutions they will accumulate more coordination code in their class specifications, code not really related with the functionality of a button or a menu; 3) that the coordination can not be reused to coordinate other kinds of entities different to Widgets; 4) that any modification to coordination code will imply the modification of the different classes that participate in the solution, not only the Handler class will be affected by also the concrete handlers (i.e., buttons and menus); 5) that it is not evident with a simple view of the code to identify which are the classes that participate in the coordination and to understand how they participate; 6) that it is not clear in case of a modification of the coordination to identify which classes will be affected by the changes; 7) that if we want to do not coordinate the menus and the buttons in the example as a chain of responsibilities we will need to modify their inheritance hierarchies and delete some method implementations; 8) that it is not possible to dynamically modify the coordination code if the implementation would have been done in a concurrent object-oriented language with strong typing like Java it would have been necessary the recompilation of the code; 9) that in the case that a new type of participants different to Views will need to be coordinated this will implies the new participants will need to be defined as subclasses of the Handler class.

Another aspect that we should not forget in this example is that we are using active objects to specify the behavior of the different participant classes. There is not reference in the code to the creation of threads or processes like in a typical concurrent program written in Java. The specification of the active objects in the ActiveObject class hides all the low levels details related with the concurrency: creation of process, scheduling of processes, specification of internal synchronization policies, etc. We believe (as we have pointed out several times in this thesis), that not only the specification of the coordination in a coordination model and language is important, both: the specification of the computation related with the concurrency in active objects and the separation of computation and coordination concerns are key elements in the simplification of the complexity of the specification, development and maintenance of concurrent object-oriented systems.

CoLaS Specification

We create a coordination group named ChainRespPattern (**Figure 7.4**) to encapsulate the coordination aspect of the chain of responsibility pattern. The coordination group specifies a unique role named Handler (line 3). The role Handler specifies an interface composed of two signatures executeRequest: (i.e., executes a request) and canHandle: (i.e., returns true if the handler can handle the request false otherwise) (line 5), every object that wants to play the role Handlers must know how to react to both methods. The role Handler specifies additionally a participant variable named successor (line 6). The participant variable successor is used to store for each handler the reference to the next handler in the chain of responsibility. The ChainRespPattern coordination group specifies the following coordination rules:

> Rule1 (line 8): specifies that whenever a request is received by a handler, the handler verifies whether it can handle or not the request. If it can, it executes the request, if not it passes the request to the next handler in the chain of responsibility (if there is one of course).

Rule 2 (line 6): defines how to specify the next handler in the chain of responsibility for a handler. Rule 3 (line 19): validates that a successor handler specified in a setSuccessor: behavior is a valid handler. The handler specified as argument must be a participant of the role Handler. Only objects validating the role interface specified in the role Handles are authorized to play the role.

```
1.CoordinationGroup createCoordinationGroupClassNamed: #ChainRespPattern.
2.
3.ChainRespPattern defineRoleNamed: #Handler.
4.
5.Handler defineInterface: #(#executeRequest: #canHandle:).
6.Handler defineParticipantVariable: #successor.
7.
8.[1] Handler defineBehavior: 'handleRequest: aRequest' as:
9.       [(self canHandle: aRequest)
10.          ifTrue: [self executeRequest: aRequest]
11.          ifFalse:
12.               [self successor
13.                    ifNotNil:
14.                         [self successor handleRequest: aRequest]]].
15.
16.[2] Handler defineBehavior: 'setSuccessor: aHandler' as:
17.      [self successor: aHandler].
18.
19.[3] Handler ignore: 'setSuccessor: aHandler' if:
20.      [(Handler includes: aHandler)not ].
```

Figure 7.4 : Chain of Responsibility Pattern

Analysis

From the group specification point of view the ChainRespPattern group illustrates: 1) the creation of a coordination group (line 1); 2) the specification of a role and its role interface (lines 3 and 5); 3) the specification of a participant variable (line 6); 4) the specification of two cooperation rules (lines 8 and 16) and one reactive synchronization rule (line 19) and 5) the specification of synchronous recursive method invocations (line 9).

From the coordination point of view we can see in the example that in the specification of the group the coordination policies are defined independently of the entities that are coordinated. The conditions imposed to the active objects to play the role Handler are specified in the role interface (line 5). To play the role Handler the active objects must be able to respond to two method invocations: executeRequest: and canHandle:. The method executeRequest: executes a received request and the method canHandle: validates whether the handler can handle (or must handle the request). The active objects that play the role Handler does not need to "know" in advance anything about the coordination specified in the group to play the role, they do not need know even that they will coordinated.

If we compare this solution with the solution presented before (i.e., using Smalltalk and active objects) from the point of view of the facility to realize the specification, construction and modification of the solution; the second solution presents a lot of advantages: 1) the coordination code does not appear in the com-

putation code of the participants; 2) the coordination code can be reused independently of the coordinated entities and the coordinated entities independently of the coordination code; 3) we do not need to modify the class hierarchies of the participants to specify and modify the coordination; 4) the participants do not need to accumulate coordination behavior in their code; 5) any modification made to the coordination is done in only one point in the code (i.e., in the coordination group); 6) it is clear which is the coordination relating the different participants, it is clear what are they roles, their obligations and in general how they participate to the coordination; 7) it is possible to dynamically modify the coordination if needed and 8) it will be easy to introduce new participants if needed.

7.2 The Dining Philosophers[Dijk68a]

Problem Description

A number of philosophers are seated around a circular table. Each philosopher spends his life alternatively between two activities: eating and thinking. To eat a philosopher must sit at a table. Between each pair of table positions there is a single fork and there is the same number of forks than philosophers. To eat, each philosopher needs two forks, the two that find in front of the philosopher over the table (i.e., the one at his left and at the one his right). As a consequence a philosopher cannot be eating concurrently with his neighbor.

Solution

We have identified two types of entities in the problem (**Figure 7.5**): the philosophers and the forks. All the interaction starts when a philosopher tries to eat. To eat the philosopher must take the two forks that find in front him over the table: the one to his left and the one his right. The philosopher waits if one or both of the two forks are actually being used by another philosopher. When the philosopher takes both forks he starts to eat. When the philosopher has eaten enough he frees the two forks putting them over the table and sleeps for some time.

Structure

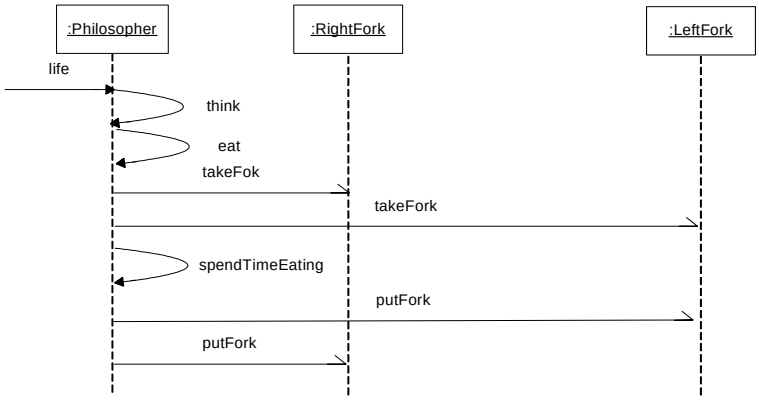


Figure 7.5 The Dining Philosopher's Interaction Diagram

Coordination Aspects

- Transfer of information: there is some basic flow of information between the philosophers and the forks. The philosopher "announces" to the forks his intention to taken them and when he finishes to eat he "announces" to the forks his intention to put them over the table.
- Condition Synchronizations: philosophers can only eat if they can take the two forks that find in front on them over the table. A fork can not be taken if the fork is already being used by another philosopher.
- Allocation/Access of/to Shared Resources: each fork is shared by two philosophers. Only one philosopher can access a fork at the time. Philosophers do not eat if they do not have the two forks that find in front of them over the table.

Smalltalk Specification

In this example we do not use the ActiveObject support of Actalk[Brio89a]. Our purpose is to illustrate how the low details related with the concurrency that appear in the solution of the dining philosophers problem difficult its specification and understanding. In the example we use the basic support for concurrency included in Smalltalk: the creation of a process (line 14), the resuming of the execution of a process (line 15), the yielding of the processor time to another process with same priority (line 14) and the use of a mutual exclusion semaphore to control the access to a shared resource (lines 38 and 40).

In (**Figure 7.6**) we can see the implementation of the classes Philosopher and Fork (lines 1 and 25). The class Philosopher specifies two instance variables leftFork and rightFork (line 5) representing the two forks that each philosopher has in front of him over table. Each instance variable stores the reference to a fork. The method >>life (line 10) specifies the activities that a philosopher does during his whole life: to think (line 12) and to eat (line 13). The method >>think (line 16) simulates the behavior of the philosopher when he thinks (it is also possible to use the Smalltalk class Delay to stop the philosopher for some milliseconds). The method >>eat (line 18) specifies the activity of eating of the philosopher. To eat a philosopher needs his two forks (lines 19 and 20), if he can take both forks he proceeds to eat (line 21). The method >>take (line 39) in the fork uses a mutual exclusion semaphore to control the access to the fork. If the fork is being used when the request for take is received the execution of the calling process is suspended until the fork is put over the table. The method >>put (line 37) in the fork uses the same mutual exclusion semaphore that the take method to indicate that the fork can be used by another philosopher. If a philosopher process was suspended waiting for the fork the process is resumed and the philosopher can take the fork.

```
1. CaseStudies defineClass: #Philosopher
2.       superclass: #{Core.Object}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: 'rightFork leftFork philproc'
6.       classInstanceVariableNames: ''
7.       imports: ''
8.       category: 'Philosophers'
9.
10.>>life
11.      self philproc:[[true] whileTrue:
12.                  [self think.
13.                  self eat.
14.                  self philproc yield]] newProcess.
15.      self philproc resume.
16.>>think
17.      Transcript cr; show: ‘Im thinking’.
18.>>eat
19.      self leftFork take.
20.      self rightFork take.
21.      Transcript cr; show: ‘I spend some time eating’.
22.      self rightFork put.
23.      self leftFork put.
24.
25.CaseStudies defineClass: #Fork
26.      superclass: #{Core.Object}
27.      indexedType: #none
28.      private: false
29.      instanceVariableNames: 'semaphore '
30.      classInstanceVariableNames: ''
31.      imports: ''
32.      category: 'Philosophers'
33.
34.>>semaphore
35.      semaphore ifNil: [ semaphore := Semaphore forMutualExclusion ].
36.      ^semaphore
37.>>>put
38.      self semaphore signal
39.>>take
40.      self semaphore wait
```

Figure 7.6 Philosopher and Fork classes

Analysis

From the coordination point of view we can see in the implementation of the solution how the coordination and computation aspects are mixed within the philosopher and fork classes. The method >>life (line 10) for example which is a behavior exclusively related with the coordination of the philosopher calls the low level

Smalltalk method for the creation of processes >>newProcess. The process encapsulating the >>life method (lines 11 to 14) is defined as an infinite loop composed of two subactivities. think and eat. Only the >>life method executes concurrently in our solution, this means that philosophers execute concurrently among them but sequentially internally. The main advantage of using the ActiveObject classes introduced by Actalk [Brio89a] is that all the computational aspects related with the concurrency can be encapsulated and thus hidden to the programmers within these classes. Programmers define the ActiveObject class or subclass that fits the best to its object model (i.e. active object, actors, etc.) and focus exclusively on the specification of the computation behavior specific to the problem. We can also see in the solution that a mutual exclusion semaphore is used to specify the internal synchronization constraints associated with the execution of the methods >>take and >>put in the forks. In the Actalk framework is possible to specify at a high level for each object model the type of synchronization constraint needed, if we consider for example the case of a multiple readers only one writer synchronization policy, in Actalk programmers will not take care of specifying the low level details of how the policy is applied but only in specifying which behaviors must be considered as readers and which as writers.

Additionally we can see in the solution: 1) that the coordinated entities (i.e., the philosophers and the forks) must "know" in advance about the coordination in which they participate; 2) that the coordination can not be reused to coordinate other kinds of entities different these philosophers and forks; 3) that any modification to coordination code will imply the modification of the philosopher and fork classes in the solution; 4) that it is not easily to dynamically modify the coordination (i.e., even if in Smalltalk this is possible). Consider for example the case in which forks are replaced by chopsticks in the solution. In the new solution it will be necessary to modify the specification of the class philosopher because of such simple modification. We will see below in the CoLaS specification that such a change in the coordination specification does not have any impact in the CoLaS solution to the problem, this is because the specification of the coordination in CoLaS is done based on the roles that participants play in the coordination and not in their identities or their types. If the potential participants satisfy the role interfaces associated to the roles they want to play in the coordination groups they will be authorized to play the roles.

ColaS Specification

We create a coordination group named DiningPhilosophers (**Figure 7.7**) to encapsulate the coordination aspect of a solution to the dining philosophers problem. The coordination group specifies two roles Philosopher (line 3) and Fork (line 7). The role interface of the role Philosopher (line 4) specifies that philosophers must know how to spend their time thinking (i.e., they must be able to react to the method think). The role philosophers specifies two participant variables named leftFork and rightFork (line 5) they model the two forks that find at the left and at the right side of the each philosopher over the table. The role Forks additionally defines a participant variable named isFree used to keep the current state of the fork (i.e., busy or free).

```
1.CoordinationGroup createCoordinationGroupClassNamed: #DiningPhilosophers.
2.
3.DiningPhilosophers defineRoleNamed: #Philosopher.
4.Philosopher defineInterface: #(#think).
5.Philosopher defineParticipantVariables: #(#leftFork #rightFork).
6.
7.DiningPhilosophers defineRoleNamed: #Fork.
8.Fork defineParticipantVariable: #isFree initialValue: true.
9.
10.[1] Philosopher defineBehavior:
11.      'setRightFork:rightFork setLeftFork:leftFork' as:
12.      [self rightFork: rightFork.
13.      self leftFork: leftFork].
14.
15.[2] Philosopher defineBehavior: 'life' as:
16.      [[true] whileTrue: [self think. self eat]].
17.
18.[3] Philosopher defineBehavior: 'eat' as:
19.      [(self rightFork take) wait.
20.      (self leftFork take) wait.
21.      Transcript cr; show: 'I spend some time eating'.
22.      (self leftFork put) wait.
23.      (self rightFork put) wait].
24.
25.[4] Fork defineBehavior: 'take' as:
26.      [self isFree: false].
27.
28.[5] Fork disable: 'take' if:
29.      [self isFree not].
30.
31.[6] Fork defineBehavior: 'put' as:
32.      [self isFree: true].
```

Figure 7.7 Dining Philosophers

The DiningPhilosophers coordination group specifies the following coordination rules:

Rule 1 (line 10): specifies how to assign the two forks associated to a philosopher.

Rule 2 (line 15): specifies the life of a philosopher. A philosopher passes all his life thinking and eating.

Rule 3 (line 18): specifies that when a philosopher wants to eat first he tries to take the two forks that find in front of him over the table. If he gets the two forks then he spends some time eating and then he frees the two forks putting them back over the table. In this rule we can see the use of the wait message sent to the futures received from the sent of the messages take and put to the forks. The wait message blocks the execution of the method >>eat until the execution of the operation that it precedes is done. In this case the philosopher executing the method >>eat blocks if the forks can not be taken (lines 19 and 20) and during the execution of the messages put by the two forks (lines 22 and 23). In other words, the method wait guarantees the synchronic execution of a method.

Rule 4 (line 25): specifies that when a fork is taken by a philosopher the variable specifying the state of the fork isFree is set to false.

Rule 5 (line 28): specifies that a fork can not be taken by a philosopher if the fork is being used. The message take is delayed until the fork is free.

Rule 6 (line 31): specifies that when a fork is free by a philosopher the variable specifying the state of the fork isFree is set to true.

Analysis

From the group specification point of view the CoLaS Dining Philosophers group illustrates: 1) the creation of a coordination group (line 1); 2) the specification of two roles (lines 3 and 7) one with its role interface (line 4); 3) the specification of a participant variable (line 8) and 4) the specification of five cooperation rules (lines 10, 15, 18, 25 and 31; 5) and one reactive synchronization rule (line 28).

From the coordination point of view we can see in the example that in the specification of the group the coordination policies are defined independently of the entities that are coordinated. The conditions imposed to the active objects to play the role Philosopher are specified in the role interface (line 4), different types of "philosophers" may play the role Philosopher as long they implement them method >>think. Also because no role interface is defined for the role Fork, in principle we can model the same problem with different types of utensils (i.e. chopsticks, spoons, etc.). The active objects that play the role Fork do not need to" know" anything about the coordination specified in the group to play the role, they do not need know even that they are coordinated.

If we compare this solution with the solution presented before from the point of view of the facility to realize the specification, construction and modification of the solution; the second solution presents a lot of advantages: 1) the coordination code does not appear in the computation code of the participants, it will be possible to use different kinds of participants to play the roles Philosopher and Fork; 2) the coordination code can be reused independently of the coordinated entities and the coordinated entities independently of the coordination code; 3) we do not need to modify class hierarchies of the participants to specify and modify the coordination; 4) modifications to the coordination are done in one point in the code, in the group specification; 5) it is possible to dynamically modify the coordination if needed, adding new rules and 6) that it is easy to introduce new types participants if needed.

The CoLaS implementation presented in (**Figure 7.7**) of the DiningPhilosophers is not deadlock free. Consider the situation where all the Philosophers become hungry at the same time, sit down at the table and then each philosopher picks up the for to his (or her) right. The system can make no further progress since each philosopher is waiting for a fork held by his neighbor. We propose to modify the existing coordination group to define a deadlock free solution. In this new solution to the dinning philosophers problem we will introduce some asymmetry into the definition of a philosopher. Up to now, each philosopher had the same specification. We will define two types of philosophers: odd-numbered philosophers get the left fork first and even-numbered philosophers get the right fork first.

```
33.Philosopher definesParticipantVariable: #id.
34.
35.[3] Philosopher defineBehavior: 'eat' as:
36.      [|firstFork secondFork|
37.      (self id\\2= 1)           /* \\ represents the module operator
38.          ifTrue:
39.              [firstFork := self rightFork.
40.               secondFork := self leftFork]
41.          ifFalse:
42.              [firstFork := self leftFork.
43.              secondFork := self rightFork].
44.      (firstFork take) wait.
45.      (secondFork take) wait
46.      Transcript cr; show: 'I spend some time eating'.
47.      (secondFork put) wait.
48.      (secondFork put) wait].
```

Figure 7.8 Dining Philosophers deadlock free

In the new solution a new participant variable name id was added to the coordination group. The variable id is used to determine the order in which the forks must be taken by the philosopher. We dynamically modify the exiting coordination group by replacing the eat rule with a new rule that selects forks differently based on the id of the philosophers. This example shows how the coordination specified in a coordination group can be easily modified without affecting the specification of the participants in the coordination. This can be done because of the clear separation of coordination aspect of the problem in the coordination groups. Support for the evolution and the modification of the coordination are two fundamental requirements of an ideal coordination models and languages that the CoLaS models supports.

7.3 The Vending Machine

Problem Description

A vending machine has a number of different parts: a coin accepter into which coins can be inserted and a number of slots each containing a piece of fruit. The parts of a vending machine are subject to a consistency requirement in order for the vending machine to have the desired functionality: insert enough money and get back a piece of fruit from one of the slots. When a sufficient amount of money has been inserted into a coin accepter, one or more of the slots are available opening. Each slot may be priced differently. Opening one of the slots (i.e. taking the items they contain) will remove the inserted money from the coin accepter and prevent other slots from being opened. Pushing a special button on the coin accepter, it is possible to get a refund.

Solution

We have decided to specify three types of entities (**Figure 7.9**): the CoinAccepter, the CoinRefunder and the Slots. When a user of the vending machine inserts money in the CoinAccepter we increase the amount

of money received and when the user request to be refunded we return the amount of money he or she still has in the machine. The Slots contain the different products contained in the vending machine.

Structure

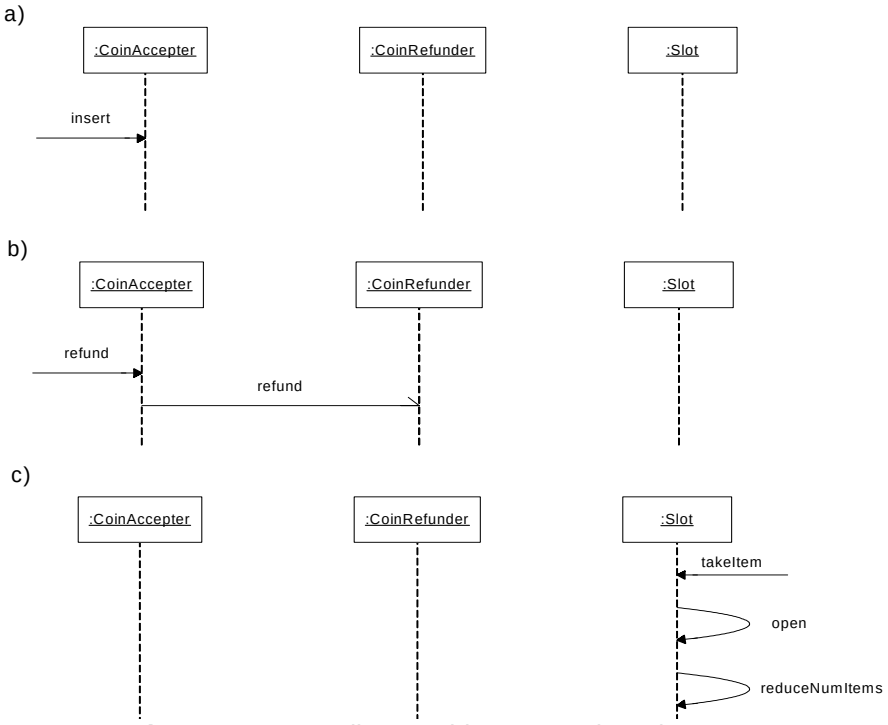


Figure 7.9 : Vending Machine Interaction Diagrams

Coordination Aspects

- Transfer of information: there is flow of information between the different elements. The CoinAccepter accepts the money and increases the amount of money available for the user in the machine. The CoinRefunder returns the money still available and indicates to the CoinAccepter to reinitialize the counter of money introduced by the user. When a user takes an item from one of the slots the slot indicates the CoinAccepter to reduce the amount of money available for the user by item price.
- Simultaneity constraints: the system controls that refunds and the take of fruits from the Slots do not happen at the same time. The system must control also that only one Slot is opened at the time.
- Execution orderings: several execution orderings must be respected in the systems. The CoinAccepter accept the money only when the money is inserted. The reset of the amount of money available for the user is done only after a refund. The number of items contained in a slot is reduced only after the item was taken by the user.
- Condition Synchronizations: the system controls that user take items only if they have inserted enough money and no refund is done to the user if no money was inserted or if not money is still available in the machine.

Smalltalk Specification

```
1.CaseStudies defineClass: #CoinAccepter
2.       superclass: #{Actalk.ActiveObject}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: 'amountOfMoneyInserted'
6.       classInstanceVariableNames: ''
7.       imports: ''
8.       category: 'VendingMachine'
9.
10.>>insert: aFloat
11.      self amountOfMoneyInserted: self amountOfMoneyInserted + aFloat.
12.      self displayTotalInserted.
13.
14.CaseStudies defineClass: #CoinRefunder
15.      superclass: #{Actalk.ActiveObject}
16.      indexedType: #none
17.      private: false
18.      instanceVariableNames: 'coinAccepter slotsManager'
19.      classInstanceVariableNames: ''
20.      imports: ''
21.      category: 'VendingMachine'
22.
23.>>refund
24.      self slotsManager blockSlots result
25.          ifTrue:
26.              [self refund:(self coinAccepter
27.                                  amountOfMoneyInserted wait).
28.              self coinAccepter resetAmountOfMoneyInserted wait.
29.              self slotsManager unblockSlots wait ]
```

Figure 7.10 Vending Machine classes CoinAccepter and CoinRefunder

In this example we use the ActiveObject support introduced in Actalk[Brio89a]. In (**Figure 7.10**) we can see the implementation of the classes CoinAccepter and CoinRefunder (lines 1 and 14). The class CoinAccepter specifies an instance variable named amountOfMoneyInserted (line 5) which contains the total amount of money inserted (and still available) by the user of the vending machine. The method >>insert: (line 10) specifies what happens when the user insert some money in the coinAccepter, basically the counter of the amount of money inserted is increased and total amount of money inserted is displayed. The Class coinRefunder specifies two instance variables named coinAccepter and slotsManager (line 18) which are used to keep references to the corresponding elements of the vending machine. The method >>refund (line 23) specifies what happens when the user request to be refunded; basically the vending machine first blocks the slots to avoid the user to take the fruits, then it returns to the user the amount of money he or she has not used, then the counter of the amount of money inserted is reinitialized to zero and finally the slots are unblocked. The blocking of the slots is done by the SlotsManager, the blocking and unblocking of slots is the mechanism used to guarantee the mutual exclusion of the refund and takeItem operations in the vending machine. Only one of these two operations may occur at the same time in the vending machine.

```
30.CaseStudies defineClass: #Slot
31.      superclass: #{Actalk.ActiveObject}
32.      indexedType: #none
33.      private: false
34.      instanceVariableNames: 'item price numItems coinAccepter
35.                              slotsManager moneyStore'
36.      classInstanceVariableNames: ''
37.      imports: ''
38.      category: 'VendingMachine'
39.
40.>>takeItem
41.      self slotsManager blockSlots result
42.          ifTrue:
43.               [self open. self updateMoneyAndReduceNumItems. self close.
44.               self slotsManager unblockSlots wait].
45.>>updateMoneyAndReduceNumItems
46.      (self coinAccepter reduceAmountOfMoneyInserted: self price) wait.
47.      (self moneyStore addMoney: self price) wait.
48.      self reduceNumItems.
49.
50.CaseStudies defineClass: #SlotsManager
51.      superclass: #{Actalk.ActiveObject}
52.      indexedType: #none
53.      private: false
54.      instanceVariableNames: 'slotsAreBlocked'
55.      classInstanceVariableNames: ''
56.      imports: ''
57.      category: 'VendingMachine'
58.
59.>>blockSlots
60.      ^self slotsAreBlocked      “if the slots are blocked we dont block”
61.          ifTrue: [ false ]
62.          ifFalse: [ self slotsAreBlocked: true ].
63.
64.CaseStudies defineClass: #MoneyStore
65.      superclass: #{Actalk.ActiveObject}
66.      indexedType: #none
67.      private: false
68.      instanceVariableNames: 'totalAmountOfMoneyInserted'
69.      classInstanceVariableNames: ''
70.      imports: ''
71.      category: 'VendingMachine'
72.
73.>>addMoney: aFloat
74.      self totalAmountOfMoneyInserted:
75.                    self totalAmountOfMoneyInserted + aFloat.
```

Figure 7.11 Vending Machine classes Slot, SlotsManager and MoneyStore

In (**Figure 7.11**) we can see the implementation of the classes Slot (line 30), SlotsManager (line 50) and MoneyStore (line 64). The class Slot models a slot of the vending machine. Three of the instance variables specified in the class Slot: item, price and numItems (line 34 and 35) specify information related to the item contained in the slot. The item instance variable defines the name of item contained in the slot, the price instance variable defines the price of the item and numItems instance variable defines the number of items contained in the slot. The other three instance variables: coinAccepter, slotsManager and moneyStore are used to keep the references to the corresponding elements of the vending machine. The method >>takeItem (line 40) specifies what happens when a user request to take an item contained in a slot; basically the vending machine uses the blocking mechanism of the SlotsManager to avoid the user to be refunded, then it opens the slot, it decreases the balance of the amount of money inserted by the user and the number of items in the slots, then it closes the slot and finally it unblocks the SlotsManager. The class SlotsManager is used to control the mutual exclusion in the execution of the >>refund and >>takeItem operations. Each one of the two operations request first to block the slots if the slots are already blocked the operation is not done. The class MoneyStore models the element of the vending machine containing the total amount money received as result of the selling of items in the vending machine. When a user takes an item from the vending machine the counter associated with the total amount of money stored in the machine is increased by the price of the item.

Analysis

From the coordination point of view we can see in the implementation of this solution how the coordination and computation aspects are mixed within the classes of the participants. We can see for example in the specification of the different classes how instance variables are defined to store the references to the objects (i.e., parts of the machine) with which they interact. It is clear that if the coordination needs to be modified to include a new interaction with a different object it will be necessary to modify the specification of the participant classes to define the new references. The consequences of the mixing of coordination and computation are: 1) that the coordinated entities must "know" in advance about the other participants of the coordination; 2) that the coordination can not be reused to coordinate other kinds of entities, consider for example the case of a ticket machine which does not includes slots with items but a new element in which users specify the type of tickets they need. It will be very complicate to reuse even part of the coordination specified in the solution, we will need to define new classes, redefine methods and predefine the relations to the new elements; 3) that any modification to coordination code will imply the modification of the different classes that participate in the solution; 4) that if the coordination must be modified it is not clear how to identify which classes will be affected by these changes and how; 5) that it is not easy to dynamically modify the coordination.

We can also see in the example that is not simple to specify the synchronization constraints specified by the problem, for example the mutual exclusion of the >>refund and >>takeItem operations is done here by using an extra element the SlotsManager. The synchronization code is mixed to the computational code of the three classes. In the specification of the solution we decided to ignore the operations >>refund and >>takeItem when one of the two operations is already occurring in the system, if we want to adopt a different policy and for example just delay their execution we will need to use a Smalltalk mutual exclusion semaphore to guarantee that processes are suspended when the mutual exclusion semaphore is already used. The semaphore will need to be shared by the two classes Slot and CoinRefunder. Again a reference to the semaphore will need to be defined in the two classes as instance variables. The coordination code related to the synchronization will be spread over the two classes and mixed to the computation code. Any modification

to the synchronization code will imply the modification of the specification of the two classes. If the implementation would have been done in a concurrent object-oriented language with strong typing like Java it will have been certainly necessary the recompilation of the code.

Another aspect that we should not forget in this example is that we are using active objects to specify the behavior of the different participant classes. The ActiveObject class hides all the low levels details related with the concurrency: creation of process, scheduling of processes, specification of internal synchronization policies, etc. As we already pointed out in a previous example not only the specification of the coordination in a coordination model and language is important, also the specification of the computation related with the concurrency in active objects is a key element in the simplification of the complexity of the specification, development and maintenance of the concurrent object-oriented systems

CoLaS Specification

We create a coordination group named VendingMachine (**Figure 7.12**) to encapsulate the coordination aspect of a solution to the vending machine problem. The coordination group specifies four different roles representing the different parts of the machine: CoinAccepter (line 3), CoinRefunder (line 8), Slot (line 12) and MoneyStore (line 16). The role CoinAccepter specifies a role variable named amountOfMoneyInsertedByUser (line 5) to count the total amount of money inserted by the user. The role interface of the role CoinAccepter defines a unique signature (lines 6): displayTotalAccepted which displays the total amount of money introduced by the user. The role CoinRefunder specifies a unique signature in its role interface (line 10): refund:. The method refund: models the physical refund of the money to the user. The role Slot defines two signatures in its role interface (line 13): open and close. The open and close methods model the physical opening and closing of the slots where the items are contained. Additionally three participant variables are defined in the role Slot (line 14): item, price and numItems. The three participant variables model the name of item contained in the slot, its price and the number of items contained in the slot. Finally the role MoneyStore defines a role variable named totalAmountOfMoneyInserted (line 18) to count the total amount of money inserted in the machine and obtained from the selling of the different items contained in the machine. The role interface of the role MoneyStore defines a unique signature (line 19): storeMoney:. The storeMoney: method is used to increase the total amount of money stored in the money store.

The VendingMachine coordination group specifies the following coordination rules:

Rule 1 (line 20): specifies that when money is inserted in the coin accepter by a user of the vending machine the counter of the amount of money inserted in the machine is increased and the new total is displayed.

Rule 2 (line 25): specifies that when the user decides to request to be refunded, the machine returns to the user the amount of money he or she has in the machine (i.e., not consumed) and the counter of the amount of money inserted by the user is reinitialized to zero.

Rule 3 (line 29): specifies that when a user decides to take an item from a slot, the machine must open the door that gives access to the item, then the amount of money inserted by the user is reduced by an amount equal to the price of the item, then the number of items contained in the slot are reduced and finally the door of the slot is closed.

Rule 4 (line 37): specifies that a user can not take an item from a slot if the amount of money he has introduced is inferior to the price of the item.

Rule 5 (line 40): specifies that a user is not refunded if the actual amount of money inserted by the user is equal to zero(i.e., the value of the role variable amoutOfMoneyInsertedByUser is equal to zero).

```
1.CoordinationGroup createCoordinationGroupClassNamed: #VendingMachine.
2.
3.VendingMachine defineRoleNamed: #CoinAccepter.
4.CoinAccepter maxNumParticipants: 1.
5.CoinAccepter defineVariable: #amountOfMoneyInsertedByUser initialValue: 0.
6.CoinAccepter defineInterface: #(#displayTotalAccepted:).
7.
8.VendingMachine defineRoleNamed: #CoinRefunder.
9.CoinRefunder maxNumParticipants: 1.
10.CoinRefunder defineInterface: #(#refund:).
11.
12.VendingMachine defineRoleNamed: #Slot.
13.Slot defineInterface:#(#open #close).
14.Slot defineParticipantVariables: #(#item #price #numItems).
15.
16.VendingMachine defineRoleNamed: #MoneyStore.
17.MoneyStore maxNumParticipants: 1.
18.MoneyStore defineVariable: #totalAmountOfMoneyInserted initialValue: 0.
19.MoneyStore defineInterface: #(#storeMoney:).
20.
21.[1] CoinAccepter defineBehavior: 'insert: money' as:
22.      [self amountOfMoneyInsertedByUser+= money.
23.      self displayTotalAccepted: self amountOfMoneyInsertedByUser].
24.
25.[2] CoinRefunder defineBehavior: 'refund' as:
26.      [self refund: CoinAccepter amountOfMoneyInsertedByUser.
27.      CoinAccepter amountOfMoneyInsertedByUser: 0].
28.
29.[3] Slot defineBehavior: 'takeItem' as:
30.      [self open.
31.      CoinAccepter amountOfMoneyInsertedByUser-=: self price.
32.      (MoneyStore unique storeMoney: self price) wait.
33.      MoneyStore totalAmountOfMoneyStored+= self price.
34.      self numItems--.
35.      self close].
36.
37.[4] Slot ignore: 'takeItem' if:
38.      [CoinAccepter amountOfMoneyInsertedByUser < self price].
39.
40.[5] CoinRefunder ignore: 'refund' if:
41.      [CoinAccepter amountOfMoneyInsertedByUser = 0 ].
```

Figure 7.12 : The Vending Machine

Analysis

From the group specification point of view the VendingMachine group illustrates: 1) the creation of a coordination group (line 1); 2) the specification of roles and role interfaces (lines 3, 6, 8, 10, 12, 14, 16 and 19); 3) the specification of role variables (lines 5 and 18); 4) the specification of three cooperation rules (lines 21, 25 and 29) and two reactive synchronization rules (lines 37 and 40) and 5) the specification of synchronous recursive method invocations (lines 23, 26, 30 and 35).

From the coordination point of view we can see in the example that in the specification of the coordination group the coordination policies are defined independently of the entities that are coordinated and that nowhere in the specification of the participants it was necessary to specify their relations to other participants as in the solution introduced previously in this section. If we compare both solutions from the point of view of the facility to realize the specification, construction and modification of the solution; the second solution presents a lot of advantages: 1) the coordination code does not appear in the computation code of the participants; 2) the coordination code can be reused independently of the coordinated entities and the coordinated entities independently of the coordination code; 3) we do not need to modify class hierarchies of the participants to specify and modify the coordination; 5) it is clear which is the coordination relating the different participants, it is clear what are their roles, their obligations and how they participate to the coordination; 6) it is possible to dynamically modify the coordination if needed and 8) new participants and relations can be easily introduced if needed, for example the introduction of new slots in the machine is done simply by enrolling a new participant to the role Slot.

This example is also very interesting example because it shows also the problems that the CoLaS model has to support simultaneity constraints. In the presentation of the CoLaS model in Chapter 3 of this thesis we mentioned that there are not CoLaS synchronization rules to define multi-party coordination rules (i.e., rules that depend for their applicability on multiple invocation requests occurring in different participants). We can see in this example that such kinds of rules will simplify the way the simultaneity constraints identified in the coordination aspects of the example are specified. In the Smalltalk solution a class named SlotsManager is used to manage the mutual exclusion of the execution of the two operations: >>takeItem and >>refund. We use a similar solution in the CoLaS specification but this is not very natural. The ideal will be to be able to define such synchronizations at a high level without introducing any new class. Consider a multi-party synchronization rule of the form:

VendingMachine ***mutualExclution***: #(CoinRefunder refund, Slot takeItem)

The rule specifies that the different behaviors appearing in the list (in the respective roles) are executed mutually exclusive. Such a high level rule will simply the complexity of specifying the solution to the synchronization problem in the CoLaS specification and will avoid the mixing of the synchronization details in the specification of the participants behaviors. A solution in CoLaS to the mutual exclusion problem in the example is proposed in **Figure 7.13** using a similar approach that the one used in the Smalltalk specification, a new type of participant called SlotsManager will be necessary to centralize the control of the mutual exclusion. To our view point the introduction of multi-party synchronization rules become a priority in the future work agenda of the CoLaS coordination model and language.

```
1.VendingMachine defineRoleNamed: #SlotsManager.
1.SlotsManager defineVariable: #slotsAreBlocked initialValue: false.
2.
3.[6] SlotsManager defineBehavior: 'blockSlots' as:
4.       [^self slotsAreBlocked
5.           ifTrue: [ false ]
6.           ifFalse: [ self slotsAreBlocked: true. true]].
7.
8.[7] SlotsManager defineBehavior: 'unblockSlots' as:
9.       [self slotsAreBlocked: false]
10.
11.[2] CoinRefunder defineBehavior: 'refund' as:
12.      [(SlotsManager unique blockSlots result)
13.          ifTrue:
14.              [self refund: CoinAccepter amountOfMoneyInsertedByUser.
15.              CoinAccepter amountOfMoneyInsertedByUser: 0.
16.              SlotsManager unique unblockSlots wait]].
17.
18.[3] Slot defineBehavior: 'takeItem' as:
19.      [(SlotsManager unique blockSlots result)
20.          ifTrue:
21.              [self open.
22.              (CoinAccepter amountOfMoneyInsertedByUser -=self price.
23.              (MoneyStore unique storeMoney: self price) wait.
24.              MoneyStore totalAmountOfMoneyStored += self price.
25.              self numItems--.
26.              self close.
27.              SlotsManager unique unblockSlots wait]].
```

Figure 7.13 Vending Maching using a SlotsManager

7.4 The Online-Music Shop [Pric00a]

Problem Description

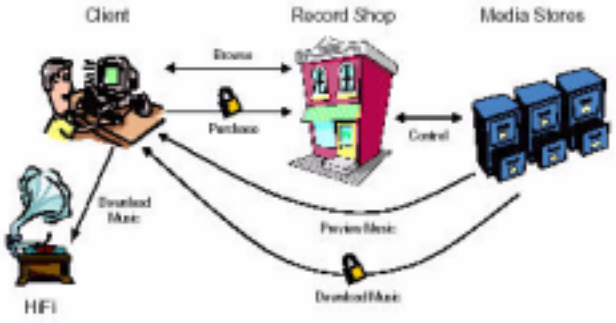


Figure 7.14 Online-Music Shop problem

Multiple service providers maintain databases of digital music tracks. A client that wants to buy music browses the available tracks at the on-line record store and listens to streamed samples of tracks in which he or she is interested before paying for and downloading high-quality versions of the files into his local computer or hi-fi. The music itself is stored in one or more media stores. We assume also that those media stores belong to different record companies.

Solution

We have decided to specify four types of entities (**Figure 7.15**): the OnlineRecordShop which represents the web interface used by the clients, the RecordShop which represents the record shop, the MediaStores which represents the place where physically the music tracks are stored and the Bank which represents the entities in charge of the validation of payments done online. Clients can browse through the titles stored in the shop with the help of keywords, listen previews (i.e., low quality tracks) of selected titles and purchase the titles (i.e., high quality tracks). When the client decides to purchase titles, he provides all the information concerning his credit cards (i.e., only payments with credit cards are accepted) if the bank does not authorize a payment the client does not receive the high quality tracks of the selected songs. We do not focus here on the security aspects related to the online payment we assume everything is done in a secure way for the client.

Coordination Aspects

- Transfer of information: clients browse their favorites titles specifying keywords in their web interface, the titles containing the specified keywords are sent from the record shop to the web interface. Low quality song tracks are sent on demand from the record shop to the online record shop, the tracks find physically stored in the media stores. High quality song tracks can be purchased on demand. The payment information flows between the online record shop, the record shop and the banks. Authorizations are sent from the banks to the record shop to validate the transactions.
- Task/Subtask: when a client request to the record shop for titles containing some specific keyword the record shop must request the different media stores for such titles. The information of the differ-

ent titles is not stored in record shop but spread over the different media stores. A record shop plays the role of an intermediary, it must request the different media stores for the titles related with some keyword, join all the different answers and send back the complete answer to the clients.

- Execution orderings: several execution orderings must be respected in the system. To show client previews of titles containing some specified keywords the record shop request first the different media stores for the titles. To purchase a title a user must first select a title and then introduce the information related to the payment. A confirmation of the payment from the bank is necessary to the record shop before the record shop will send the high quality version of a music-track to the client.

Structure

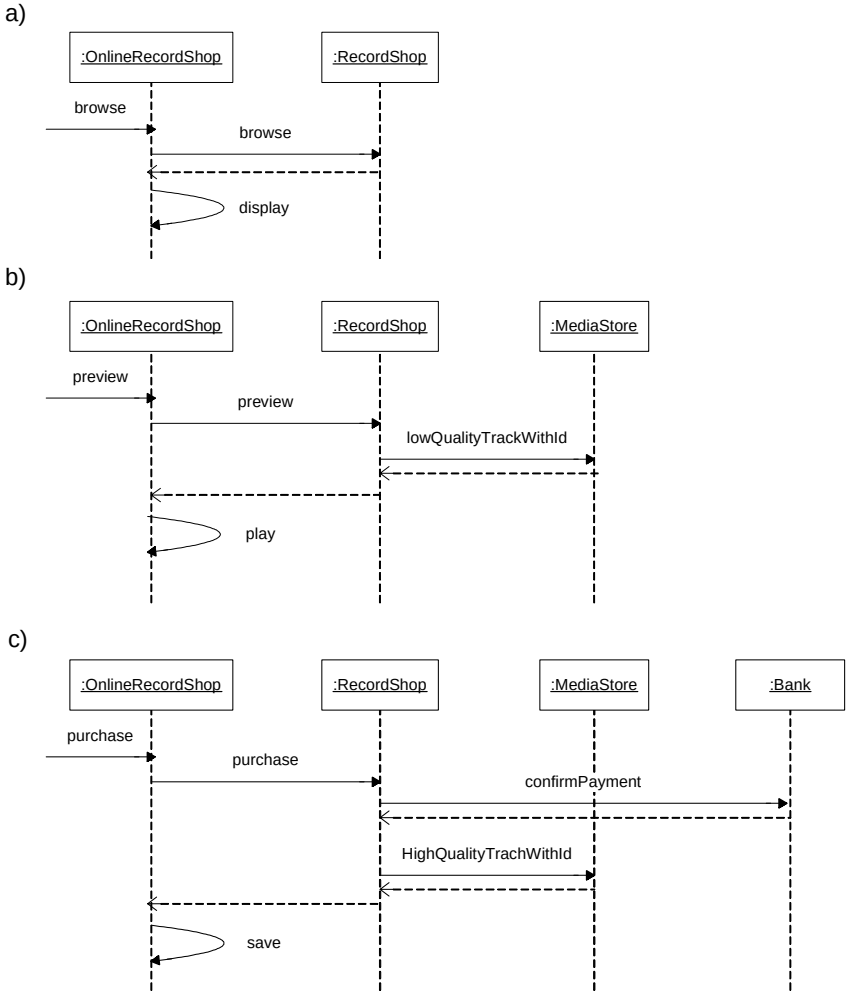


Figure 7.15 Online Music-Shop Interaction Diagrams

Smalltalk Specification

```
1.CaseStudies defineClass: #OnlineRecordShop
2.       superclass: #{Actalk.ActiveObject}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: ''
6.       classInstanceVariableNames: 'recordShop'
7.       imports: ''
8.       category: 'OnlineMusicShop'
9.
10.>>browse
11.      |keyword|
12.      keyword := self requestKeyword.
13.      self display: (self recordShop browse: keyword) result.
14.>>preview
15.      |titleInfo track|
16.      titleInfo := self selectTitle.
17.      track := (self recordShop preview: titleInfo) result.
18.      self play: track.
19.>>purchase
20.      |paymentInfo titleInfo track|
21.      titleInfo := self selectTitle.
22.      paymentInfo := self requestPaymentInformation.
23.      track :=(self recordShop purchase: aTitleInfo
24.                              payment: paymentInfo) result
25.      self save: track.
```

Figure 7.16 Online Music Shop: OnlineRecordShop class

In this example we use the ActiveObject support introduced in Actalk[Brio89a]. In (**Figure 7.16**) we can see the implementation of the class OnlineRecordShop (line 1). The class OnlineRecordShop specifies an instance variable named recordShop (line 6) which is used to keep the reference to the record shop (i.e., or to the record shops in case there will be several). The method >>browse (line 10) specifies what happens when a client of the online record shop decides to browse the music tracks offered by the different media stores (i.e., each media store corresponds to a different record company and thus has a different offer). The client specifies a keyword and the system requests to the record shop for all the titles containing the specified keyword. The titles received are displayed in the web interface. The method >>preview (line 14) specifies what happens when a client of the online record shop decides to request a preview of a title in which he or she is interested. The client selects a title among those previously displayed. The selected title is used by the system to request the record shop for a low quality version of the title. The low level track is then played in the client's machine. The method >>purchase (line 19) specifies what happens when a client of the online record shop decides to purchase a title in which he or she is interested. The client selects a title among those displayed. The selected title is then used by the system to request the record shop for a high quality version of the title. The client specifies also the payment information related to the purchase. The payment information is sent to the record shop and then to the corresponding bank for verification. If the payment information is correct (i.e., authorized by the bank) the high quality music-track is delivered to the client and saved in the client's machine.

```
26.CaseStudies defineClass: #RecordShop
27.      superclass: #{Actalk.ActiveObject}
28.      indexedType: #none
29.      private: false
30.      instanceVariableNames: 'mediaStores banks'
31.      classInstanceVariableNames: ''
32.      imports: ''
33.      category: 'OnlineMusicShop'
34.
35.>>browse: aKeyword
36.      |results|
37.      results := OrderedCollection new.
38.      self mediaStores do:
39.          [:each| results add: (each titlesWithKeyword: aKeyword) result].
40.      ^results.
41.>>preview: aTitleInfo
42.      |mediaStoreId mediaStore|
43.      mediaStoreId := aTitleInfo mediaStoreId.
44.      mediaStore := self mediaStores detect:[:each| each id= mediaStoreId].
45.      ^(mediaStore lowQualityTrackForTitle: aTitleInfo) result.
46.>>purchase: aTitleInfo payment: aPaymentInfo
47.      |mediaStoreId mediaStore bank authorization|
48.      mediaStoreId := aTitleInfo mediaStoreId.
49.      mediaStore := self mediaStores detect:[:each| each id= mediaStoreId].
50.      bank := self banks detect:[:each | each name = aPaymentInfo bank].
51.      authorization := (bank confirmPayment: aPaymentInfo) result.
52.      (authorization ~= -1)
53.          ifTrue:
54.              [self registerAuthorization: authorization
55.                   forPayment: aPaymentInfo
56.              ^(mediaStore highQualityTrackForTitle: aTitleInfo) result].
```

Figure 7.17 Online Record Shop: RecordShop class

In (**Figure 7.17**) we can see the implementation of the class RecordShop (line 26). The class RecordShop specifies two instance variables named mediaStores and banks (line 30) which are used to keep the references to the different media stores and the different banks. The method >>browse: (line 35) specifies that when a browse request is received by the record shop a request for titles related with the keyword received is sent to the different media stores. The results received from the media stores are joined and sent back as a reply. The method>>preview: (line 41) specifies that when a preview: request is received by the record shop first we identify the media store in which the title is stored and then we request the media store for a low quality track of the title. The low quality track is sent back as a reply. The method>>purchase:payment: (line 46) specifies that when a purchase:payment: request is received by the record shop first the bank related with the payment information received from the client is identified, then the payment information is verified with the bank and if the bank authorizes the transaction we request the media store containing the title for a high quality copy of the track of the title. The high quality track of the title is sent back as a reply.

```
57.CaseStudies defineClass: #Bank
58.      superclass: #{Actalk.ActiveObject}
59.      indexedType: #none
60.      private: false
61.      instanceVariableNames: ''
62.      classInstanceVariableNames: ''
63.      imports: ''
64.      category: 'OnlineMusicShop'
65.
66.>>confirmPayment: aPaymentInfo
67.      | authorization |
68.      ((authorization := self validatePayment: aPaymentInfo) ~= -1)
69.          ifTrue:
70.               [self registerAuthorization: authorization
71.                        forPayment: aPaymentInformation].
72.      ^authorization].
```

Figure 7.18 Online Record Shop: Bank class

In (**Figure 7.18**) we can see the implementation of the class Bank (line 57). The method >>confirmPayment: (line 66) specifies that when a payment confirmation is received, the bank verifies internally if the information received is correct and if the balance (or the credit) of the client allows the transaction. If the bank authorizes the transaction the authorization is sent back and the information related to the payment is register by the bank. Internally the bank updates the current balance of the client's account.

Analysis

From the coordination point of view we can see in the implementation of this solution how the coordination and computation aspects are mixed within the classes of the participants. We can see in the specification of the different classes how the classes define instance variables to store the references to the objects with which they interact. It is clear that if the coordination needs to be modified to include a new interaction with a different object it will be necessary to modify the specification of the participant classes to define the new references. Consider for example the case where we must include in the coordination radio stations promoting new songs and singers. Certainly the RecordShop class will be affected and new coordination behavior added. This problem is a consequence of the lack of separation of the coordination and computation concerns in the code. The coordination code and the relations to the participants are specified within the computational code in the classes. This implies: 1) that the coordinated entities "know" in advance about the other participants of the coordination, for example the RecordShop knows about media stores and banks; 2) that the coordination can not be reused to coordinate other kinds of entities, if we want to replace the media stores for example for another storage elements, the RecordShop class will need to be modified and the new relations introduced, 3) that any modification to coordination code will imply the modification of the different classes that participate in the solution, for example the introduction of new ways of payment will imply modifications in the class RecordShop; 4) that it is not clear in case of a modification which classes will be affected by these changes; 5) that it is not possible to easily dynamically modify the coordination.

ColaS Specification

We create a coordination group named MusicShop (**Figure 7.19** and **Figure 7.20**) to encapsulate the coordination aspect of a solution to the music shop problem. The coordination group specifies four different roles representing the most important entities in the problem: RecordShop (line 3) representing the record shop, MediaStore (line 6) representing the different medias where the music is physically stored, OnlineRecordStore (line 10) representing the web application that runs in the browsers of the clients and through which they interact with the music store and Bank (line 16) representing the entity in charge of authorizing the payments of the clients in the online-music shop. Remember that we are interested in the specification the coordination aspect and not in the complete computational specification of the entities that make part of the solution. We will not show all the details related to the web solution like whether we are using applets, or web services, etc.

The role RecordShop specifies in its role interface a unique signature: registerAuthorization:forPayment: (line 4, registers the authorization received from a bank for a payment). MediaStore specifies in its role interface the following signatures: id (line 7, id of a mediastore), titlesWithKeyword: (line 7, determines all titles containing a keyword), lowQualityTrackForTitle (line 8, returns the low quality track of a title) and highQualityTrack (line 9, returns the high quality track of a title). The role OnlineRecordStore specifies a role variable named recordShop (line 12) which is used to keep a reference to the record shop. In its role interface the role OnlineRecordStore specifies the following signatures (lines 13 and 14): requestKeyword (requests to the client for some keyword to use in the search of music titles); display: (displays the list of music titles in the screen. We assume that the web application includes all the functionality to manipulate the information about the titles and tracks, like different types of sorts (i.e., by name, by year, etc.) and the possibility to listen tracks which is normally this is done by a media player; selectTitle (selects a title between those displayed in the screen); play: (plays a music track); save: (saves a music track file) and requestPaymentInformation (request all the payment information to the client). The role Banks specifies in its role interface the following signatures: name (line 17, name of bank), validatePayment: (line 17, validates the information related with a payment, an authorization is generated if the payment is authorized by a bank), registerAuthorization:forPayment: (line 18, registers all the information related with the payment and the authorization generated by then bank when they the payment is approved).

The MusicShop coordination group specifies the following coordination rules:

Rule 1 (line 20): specifies that when a client in the web interface requests to browse the music titles in the shop, the web interface application requests the client for some keyword (or keywords) to be used in the search, then a query is sent to the record shop with the keywords specified and the results returned are displayed in the web browser.

Rule 2 (line 25): specifies how to manage a browse request coming from a client in a online interface. The record shop queries all the mediastores for titles related with the keyword received in the request, all the results (i.e., titles) returned by the mediastores are joined and sent back to the online record shop.

Rule 3 (line 32): specifies that a request for a preview from a client in a web interface implies the selection of the title by the client and then the sent of a request for a preview for that tittle to the record shop. When the low quality track of the song is received as reply from the record shop the online interface plays the track using one of the media players installed.

Rule 4 (line 38): specifies that a request for a preview implies first the identification of the mediastore in which the title is stored (line 40) we assume that this information makes part of the information stored in the

title information received in the browse operation. A request for a low quality track of the song is then sent to the media store. The reply received from the media store is then sent to the online record store. We also assume here that each title is contained in only one media store.

```
1.CoordinationGroup createCoordinationGroupClassNamed: #MusicShop.
2.
3.MusicShop defineRoleNamed: #RecordShop.
4.RecordShop defineInterface: #(#registerAuthorization:forPayment:).
5.
6.MusicShop defineRoleNamed: #MediaStore.
7.MediaStore defineInterface: #(#id #titlesWithKeyword:
8.                              #lowQualityTrackForTitle:
9.                              #highQualityTrackForTitle:).
10.
11.MusicShop defineRoleNamed: #OnlineRecordStore.
12.OnlineRecordStore defineVariable: #recordShop.
13.OnlineRecordStore defineInterface:#(#requestKeyword #display: #selectTitle
14.                              #play: #save: #requestPaymentInformation).
15.
16.MusicShop defineRoleNamed: #Bank.
17.Bank defineInterface: #(#name #validatePayment:
18.                        #registerAuthorization:forPayment:).
19.
20.[1] OnlineRecordShop defineBehavior: 'browse' as:
21.      [|keyword|
22.      keyword := self requestKeyword.
23.      self display: (self recordShop browse: keyword) result ].
24.
25.[2] RecordShop defineBehavior: 'browse: aKeyword' as:
26.      [|results|
27.      results := OrderedCollection new.
28.      MediaStore
29.       do:[:each| results add:(each titlesWithKeyword: aKeyword) result].
30.      ^results ].
31.
32.[3] OnlineRecordShop defineBehavior: 'preview' as:
33.      [titleInfo track|
34.      titleInfo := self selectTitle.
35.      track := (self recordShop preview: titleInfo) result.
36.      self play: track].
37.
38.[4] RecordShop defineBehavior: 'preview: aTitleInfo' as:
39.      [|mediaStoreId mediaStore|
40.      mediaStoreId := aTitleInfo mediaStoreId.
41.      mediaStore := MediaStore detect:[:each | each id = mediaStoreId].
42.      ^(mediaStore lowQualityTrackForTitle: aTitleInfo) result].
```

Figure 7.19 : Online-Music Shop browse and preview specifications

```
43.[5] OnlineRecordShop defineBehavior: 'purchase' as:
44.      [|paymentInfo titleInfo track|
45.      titleInfo := self selectTitle.
46.      paymentInfo := self requestPaymentInformation.
47.      track:=(self recordShop purchase: titleInfo
48.                              payment: paymentInfo) result.
49.      self save: track].
50.
51.[6] RecordShop defineBehavior: 'purchase: aTitleInfo
52.                               payment: aPaymentInfo' as:
53.      [|mediaStoreId mediaStore bank authorization|
54.      mediaStoreId := aTitleInfo mediaStoreId.
55.      mediaStore := MediaStore detect:[:each | each id = mediaStoreId].
56.      bank := Bank detect:[:each | each name = aPaymentInfo bank].
57.      authorization := (bank confirmPayment: aPaymentInfo) result.
58.      (authorization ~= -1)
59.          ifTrue:
60.              [self registerAuthorization: authorization
61.                   forPayment: aPaymentInfo.
62.              ^(mediaStore highQualityTrackForTitle:aTitleInfo) result]].
63.
64.[7] Bank defineBehavior: 'confirmPayment: aPaymentInfo' as:
65.      [| authorization |
66.      ((authorization := self validatePayment: aPaymentInfo) ~= -1)
67.          ifTrue:
68.              [self registerAuthorization: authorization
69.                        forPayment: aPaymentInfo].
70.      ^authorization].
```

Figure 7.20 : Online Music Shop purchase specification

Rule 5 (line 43): specifies that a request for purchase from a client in a web interface implies the request for all the information concerning the payment (i.e., credit card and client information) and the sent of a purchase request to the record shop. The high quality track of the song corresponding to the title purchased is received and stored in the machine of the client.

Rule 6 (line 51): specifies that a request for purchase received by the record shop implies the verification of the payment by the bank and the retrieve and the sent to the online record shop interface of the high quality track corresponding to the title purchased. All the information related to the transaction realized is register by the record shop.

Rule 7 (line 64): specifies that a confirmation of a payment by the bank implies the verification of all the payment information and the sent of an authorization to the record shop. All the information related to the authorized payment is register by the bank.

Analysis

From the group specification point of view the Online-Music shop group illustrates: 1) the creation of a coordination group (line 1); 2) the specification of four roles (lines 3, 6, 11 and 16) and their corresponding role interfaces (lines 4, 7, 13 and 17); 3) the specification of a role variable (line 12) and 4) the specification of seven cooperation rules (lines 20, 25, 32, 38, 43, 51 and 64).

From the coordination point of view we can see in the example that in the specification of the group the coordination policies are defined independently of the identity of the entities that are coordinated and that nowhere in the specification of the participants it was necessary to specify their relations to other participants. When we wanted to refer to the participants we referred exclusively to the roles they where playing in the group. If we compare this solution with the Smalltalk solution presented before from the point of view of the facility to realize the specification, construction and modification of the solution; the second solution presents a lot of advantages: 1) the coordination code does not appear in the computation code of the participants; 2) the coordination code can be reused independently of the coordinated entities and the coordinated entities independently of the coordination code; 3) we do not need to modify class hierarchies of the participants to specify and modify the coordination; 5) it is clear which is the coordination relating the different participants, it is clear which are they roles, their obligations and how they participate to the coordination; 6) it is possible to dynamically modify the coordination if needed and 8) it will be easy to introduce new participants and new relations if needed.

```
71.MusicShop defineRoleNamed: #RadioStation.
72.RadioStation defineInterface: #(#name #topTenTitles)
73.
74.[8] OnlineRecordShop defineBehavior: 'topTenTitlesInRadioStation: aString'
75.      as: [self display:
76.           (self recordShop topTenTitlesInRadioStation: aString ) result].
77.
78.[9] RecordShop defineBehavior: 'topTenTitlesInRadioStation: aString' as:
79.      [|radioStation|
80.      radioStation := RadioStation detect:[:each | each name = aString].
81.      ^radioStation topTenTiles result].
```

Figure 7.21 Dynamic Modification of the Coordination

We will illustrate now with an example how easy is to modify the coordination already specified in the coordination group. We suggests for example the idea of introducing a new type of participant in the coordination. Consider for example the introduction of radio stations as new participants in the coordination. In our new version, the online shop will be used to promote songs that appear in the top ten of the radio stations. In (**Figure 7.21**) (line 71) we define a new role named RadioStation representing the radio stations. The role RadioStation specifies in its role interface two behaviors (line 71): name (the name of the station) and topTenTitles (the list of the ten top titles in the station). To play the role RadioStation an active object must know these two behaviors in advance.

We add two new rules to The MusicShop coordination group:

Rule 8 (line 74): specifies that when a client in the web interface requests for the top ten titles in a radio station, the query is sent to the record shop with the name of the radio station as argument. The result of the request is displayed in the web interface.

Rule 9 (line 78): specifies how to manage a request for the top ten titles in a radio station coming from a client in a online shop interface. The record shop identifies the radio station and sends the request for top ten titles. The result of the request is sent back to the online shop.

7.5 The Ornamental Garden [Burn93a]

Problem Description

A large ornamental garden, probably formerly the grounds of a British stately residence, is open to members of the public, who must pay an admission fee to view the beautiful collection of roses, shrubs an aquatic plant. Entry is gained by two turnstiles (or more). The management of the gardens want to be able to determine at any time, the total number of visitors as they enter and leave the gardens. Additionally we will consider because of protection purposes of the place the number of visitors visiting the place at some time is limited by some predefined number and it must be possible to define several entrances and exists in the park.

Solution

A concurrent program that implement the population count required by the management of the ornamental garden consists of several concurrent turnstiles each incrementing (or decrementing) a shared counter when a person passes through the turnstile.

Coordination Problems

- Controlling access to shared resources: the concurrent access to the global counter of visitors must be controlled.
- Global constraints: a global synchronization must be respected, no more that numMaxVisitors must be authorized to enter the garden through all the turnstiles. If the number of visitors is exceed the turnstiles must avoid users to enter into the garden.

Structure

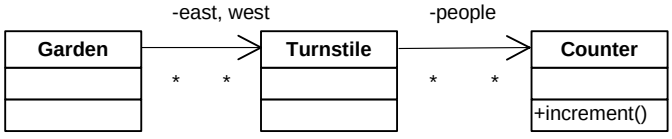


Figure 7.22 :Ornamental Garden structure

The structure presented in (**Figure 7.22**) represents a particular case of the problem in which only two turnstiles are defined in the garden one at the east and the other at the west of the garden.

Smalltalk Specification

```
1.CaseStudies defineClass: #Turnstile
2.       superclass: #{Actalk.ActiveObject}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: ''
6.       classInstanceVariableNames: 'counter'/* share variable
7.       imports: ''
8.       category: 'OrnamentalGarden'
9.
10.>>counter: aTurnstileCounter
11.      counter := aTurnstileCounter
12.>>enterVisitor
13.      self counter
14.          incrementCounterIfDoneDo:[Transcript cr; show:‘Welcome‘]
15.          ifNotDoneDo: [Transcript cr; show:‘Garden is full‘]
16.>>leaveVisitor
17.      self counter decrementCounter
18.
```

```
1.CaseStudies defineClass: #TurnstileCounter
2.       superclass: #{Core.Object}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: ''
6.       classInstanceVariableNames: 'counter counter_sem'
7.       imports: ''
8.       category: 'OrnamentalGarden'
9.
10.>>counter
11.      counter ifNil:[ counter := 0 ].
12.      ^counter
13.>>counter: anInteger
14.      counter := anInteger.
15.>>counter_sem
16.      counter_sem ifNil: [ counter_sem := Semaphore forMutualExclusion ].
17.      ^counter_sem
18.>>incrementCounterIfDo: aDoneBlock ifNotDoneDo: aNotDoneBlock
19.      self counter_sem critical:
20.          [ self counter < 100
21.              ifTrue: [ self counter: self counter + 1. aDoneBlock eval]
22.              ifFalse: [ aNotDoneBlock eval ] ]
23.>>decrementCounter
24.      self counter_sem critical: [ self counter : self counter -1 ]
```

Figure 7.23 Ornamental Garden classes

Analysis

From the coordination point of view we can see in the implementation of this solution how the coordination and computation aspects are mixed within the classes of the participants. The specification of the turnstiles in particular the >>enterVisitor and >>leaveVisitor methods make directly reference to the mechanism used to control the coordination (i.e., the counter). Similar for the value that contains the maximum number of visitors that can enter the park. The constant used finds coded into the specification of the mechanism used to control the number of visitors. This can be partially solved by defining an accessor and by assigning the vale to an instance variable. We can also see in the specification of the Turnstile class also how this class which represents a participant of the coordination defines an instance variable to store the reference to the shared counter. It is clear that if the coordination needs to be modified to include a new interaction with a different object it will be necessary to modify the specification of the Turnstile participant class to define the new references. Another important point that appears in the solution is that it is the responsibility of the user to define how the coordination is done, in this case we use a mutual exclusion semaphore to exclude multiple modifications at the same time of the counter. Only experimented programmers know that the best way to access and modify a shared variable is by encapsulating the calls and modifications in critical blocks. Nevertheless, these details of the coordination are low level details, the ideal will be to define high level abstractions that allow users to define their coordination in a safe way (e.g., for example avoiding possible not liberation of semaphores after their use) at a high level.

CoLaS Specification

We create a coordination group named OrnamentalGarden (**Figure 7.24**) to encapsulate the coordination aspect of a solution to the ornamental garden problem. The coordination group specifies a unique role named Turnstiles (line 5), we do not consider necessary to represent the Garden entity in our solution in some way the group models this class. The coordination group specifies two group variables maxNumVisitors (line 2) and numVisitors (line 3). In CoLaS groups variables are shared by all the participants of the group, they can acceded automatically by using their names. Because they group variables are shared variables the group internally protects their integrity by serializing their accessors. In CoLas this aspect is managed internally by the model.

```
1.CoordinationGroup createCoordinationGroupClassNamed: #OrnamentalGarden.
2.OrnamentalGarden defineVariable: #maxNumVisitors initialValue: 100.
3.OrnamentalGarden defineVariable: #numVisitors initialValue: 0.
4.
5.OrnamentalGarden defineRoleNamed: #Turnstile.
6.
7.Turnstile defineBehavior: 'enterVisitor' as:
8.       [group numVisitors = group maxNumberOfVisitors
9.            ifTrue: [Transcript cr; show: 'Garden is full']
10.           ifFalse: [group numVisitors++. Transcript cr; show: 'Welcome'].
11.
12.Turnstile defineBehavior: 'leaveVisitor' as:
13.      [group numVisitors--].
```

Figure 7.24 Ornamental Garden

The OrnamentalGarden coordination group specifies the following coordination rules:

Rule1 (line 7): specifies that the entrance of a visitor into the garden by a turnstile implies the increase of the number of visitors of the garden. If the number of visitors is already in the garden is equal to the maximum number of visitors authorized to enter the garden the visitor will not be allowed to enter.

Rule2 (line 12): specifies that the exit of a visitor from the garden by a turnstile implies the decrease of the number of visitors of the garden.

Analysis

From the group specification point of view the CoLaS OrnamentalGarden group illustrates: 1) the creation of a coordination group (line 1); 2) the specification of two group variables (lines 2 and 3); 3) the specification of a role (line5) and 4) the specification of two cooperation rules (lines 7 and 12).

From the coordination point of view we can see in the example that in the specification of the group the coordination is defined independently of the identity of the entities that are coordinated. The coordination policies appear clearly defined in the two cooperation rules and the role Turnstile defines the only type of participants. No restrictions are imposed by the role on the participation of active objects. We will use this example to illustrate some problem that the CoLaS coordination model has. The enterVisitor rule behavior specifies that whenever the number of visitors is equal to the maximum number of visitors permitted in the garden a message "Garden is full" is written in the Transcript (i.e., the screen). The message enterVisitor received by the turnstile will be consider as executed once the message is written. The first question that immediately rises is: Would it be possible to delay the execution of the message in case the number of visitors is exceeded? a delayed message will imply that the message will not be consider as consumed and no new message enterVisitor will be need to be sent again to the turnstile. The answer is yes, in principle CoLaS allows to disable messages when conditions are not satisfied (i.e., condition synchronizations). In (**Figure 7.25**) we show a new version of the solution specifying a disable rule in the coordination group.

```
14.Turnstile defineBehavior: 'enterVisitor' as:
15.      [group numVisitors++. Transcript cr; show: 'Welcome'].
16.
17.Turnstile defineBehavior: 'leaveVisitor' as:
18.      [group numVisitors--].
19.
20.Turnstile disable: 'enterVisitor' if:
21.      [group numVisitors = group maxNumberOfVisitors].
```

Figure 7.25 Ornamental Garden with disable rule

Rule3 (line 13): specifies that visitor to the garden are not authorized to enter to the garden if the number of visitors already in the garden is equal to the maximum number of visitor authorized to enter the garden.

7.6 The New Server Election

Problem Description

Many distributed application are easy to implement if there is in the system a dedicated process to administer certain tasks. For example a replica server, replication is the maintenance of on-line copies of data and other resources. Replication it is a key to the effectiveness of distributed systems, in that it can provide enhanced performance, high availability and fault tolerance. A basic architecture model for the management of replicated data is one in which each client's requests are first handled by a component called a front end. The front end then communicates with one or more replica managers, rather than forcing the client to do this itself explicitly. If the front end server fails one of the replica managers must take over the role of front end server. An election is a procedure carried out to choose a process from a group, the main requirement is for the choice of the elected process to be unique, even if several process call election concurrently.

Solution

We will use a a ring-based election algorithm proposed by Chang and Roberts [Chan79a], suitable for a collection of processes that are arranged in a logical ring. The algorithm assumes that the processes do no not know the identities of the others a priori and that each process knows only how to communicate with its neighbor (i.e., let's say the clockwise direction). The goal of the algorithm is to elect a single coordinator, which is the process with the largest identifier. The algorithm assumes that all the processes remain functional and reachable during its operation (i.e., which is our case). Initially, every process is marked as a non-participant in an election. Any process can be begin an election. It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its neighbor. When a process receives an election message, it compares the identifier in the message with its own. If the arrived identifier is smaller and the receiver is not a participant then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant. On forwarding an election message in any case, the process marks itself as a participant. If, the receiver identifier is that of the receiver itself, then this process identifier must be the greatest and it becomes the new elected coordinator process. The new elected process marks itself as a non-participant once more and send an elected message to its neighbor announcing its election and enclosing its identity. When a process other than the elected receives and elected message, it marks itself as a non-participant and forwards the message to its neighbor.

Coordination Problems

- Transfer of information: election messages are exchanged between neighbour processes. The election messages are exchanged clockwise and they contain the identification of process with the highest identification id (initially when an election is started the election message contains the id of the process initiator of the election). The id associated to each process is used to elect a new process. Messages indicating the identity of the new selected process are exchanged between neighbour processes too. They announce the identity and the election of a new process.
- Group decisions: the group of process decides to elect a new coordinator process. The processes exchange election messages choosing the processes with the highest id. In the current example the group decision corresponds to the election of a new server possibly because of the failure of the active one. Whatever the decision will be the algorithm described in the solution can be used to select among the different solutions proposed by the different participants of a group.

Smalltalk Specification

```
1.CaseStudies defineClass: #Server
2.       superclass: #{Actalk.ActiveObject}
3.       indexedType: #none
4.       private: false
5.       instanceVariableNames: ''
6.       classInstanceVariableNames: 'id next participant elected'
7.       imports: ''
8.       category: 'RingBasedElection'
9.
10.>>electNewServer
11.      self participant: true.
12.      self next election: self id.
13.>>election: anInteger
14.      anInteger > self id
15.          ifTrue:
16.               [self next election: anInteger.
17.               self participant: true].
18.      anInteger < self id
19.          ifTrue:
20.               [self participant
21.                   ifFalse:
22.                        [self next election: self id.
23.                        self participant: true ]].
24.      anInteger = self id
25.          ifTrue:
26.               [self participant: false.
27.               self elected: self receiver.
28.               self next elected: self receiver
29.>>elected: aServer
30.      self receiver ~= aServer
31.          ifTrue: [ self next elected: aServer].
32.      self elected: aServer
```

Figure 7.26 Ring Based Election Server class

Analysis

From the coordination point of view we can see in the implementation of this solution how the coordination and computation aspects are mixed within the participant class. The Server class which represents the participants of the new election contains the specification of the coordination about how the new server is elected. It is clear that if the coordination needs to be modified for example to select no the server with the highest id but the server with the lowest we will be force to modify the specification of the Server class. Something similar it will happen it we decide to change the type of participants in the election. Either the new participant class must define (i.e., copy) the specification already contained in the Server class or if the behavior is specified in another special class the new participant class will have to be defined as a subclass of this new class. Few programming languages allow the dynamic change of hierarchies of classes and even if this will be possible the most important is that these changes are temporal changes, only during the time the partici-

pant plays some role in the coordination, afterwards the ancient class hierarchy must need to be restored. Changing and unchanging class hierarchies could generate errors if the work is done carefully.

CoLaS Specification

```
1.CoordinationGroup createCoordinationGroupClassNamed: #RingBasedElection.
2.
3.RingBasedElection defineRoleNamed: #Server.
4.
5.Server defineParticipantVariables: #(id #next #participant #elected)
6.       initialValues: #(0 nil false nil).
7.
8.[1] Server defineBehavior: 'electNewServer' as:
9.       '[self participant: true.
10.       self next election: self id]'.
11.
12.[2] Server defineBehavior: 'election: anInteger' as:
13.      '[anInteger > self id
14.          ifTrue:
15.              [self next election: anInteger.
16.              self participant: true].
17.      anInteger < self id
18.          ifTrue:
19.              [self participant
20.                  ifFalse:
21.                      [self next election: self id.
22.                      self participant: true ]].
23.      anInteger = self id
24.          ifTrue:
25.              [self participant: false.
26.              self elected: self receiver.
27.              self next elected: self receiver]]'.
28.
29.[3] Server defineBehavior: 'elected: aServer' as:
30.      '[self receiver ~= aServer
31.          ifTrue: [ self next elected: aServer].
32.       self elected: aServer]'
```

Figure 7.27 Ring-Based Election group

We create a coordination group named RingBasedElection (**Figure 7.27**) to encapsulate the coordination aspect of the ring based election algorithm specified in the solution of the new server election problem. The coordination group specifies a unique role named Server (line 3). The coordination group specifies four participant variables (line 5): id which specifies a unique id associated to the server, participant is a variable that indicates whether the server has already participated to the election (i.e., in some sort whether it has voted), next is a variable used to keep a reference to the next server in the ring and elected is a variable used to keep a reference to the elected server.

The RignBasedElection coordination group specifies the following coordination rules:

Rule1 (line 8): specifies that when a new election of a server is request by a server, the participant participant variable of the server is set to true and the server sends a message election: to its neighbor server with its id.

Rule2 (line 12): specifies that when a server receives an election: message it compares the id received in the message with its own id. If the received id is greater than its id then the server sends a message election to its neighbor server with the same id it received and set its participant variable to true. If the received id is smaller than its id, then if the server has not already vote (i.e., set its participant variable to true) it sends to its neighbor server its id (remember that the algorithm is based on selecting the server with the highest id). If the received id is equal to the id of the server then this server has the highest id and it considers itself as elected. The server then sends a message elected with its own reference to its neighbor server which will propagate the identity of the new elected server.

Rule3 (line 29): specifies that when a server receives an elected: message if the served elected is different to the one who received the message then the server sends the elected: message to the its neighbor server to propagate the identity of the new elected server.

Analysis

From the group specification point of view the VendingMachine group illustrates: 1) the creation of a coordination group (line 1); 2) the specification of a roles (line 3); 3) the specification of participant variables with initial values (line 5 and 6) and 4) the specification of three cooperation rules (lines 8, 12 and 29).

Although the specification of the coordination group looks very similar to the specification of the Smalltalk solution presented before, from the coordination point of view we can see that in the specification of the group the coordination is defined independently of the identity of the entities that are coordinated. We refer to the participants of the coordination by the role they play in the coordination, in this case the role Server (we could have named ParticipantInElection the role to use a more generic name). Different kinds of server participants can participate in the election. To be more rigorous we could have specified some signatures to filter the kinds of servers that can participate in the election: for example we could have requested them to be able to perform some specific service like to be able to replicate information which is our case. The important is that the coordination can be modified independently of the participants and reused to coordinate different types of participants.

We can also see in the group specification how the coordination policies appear clearly defined in the three cooperation rules. If the policy used to determined the new elected server will need to be changed, it will be necessary to modify only the election behavior rule which specifies the policy. In CoLaS rules can be modified dynamically, this means that new election policies can be defined dynamically for the group. We do not need to modify class hierarchies of the participants to specify and modify the coordination.

Finally another important aspect in the group specification is that there is not restriction in the number of participants that may play the role Server. The election coordination behavior may scale up and adapt to a greater number of participants (i.e., electors).

7.7 Conclusions

We have shown in this chapter with a series of examples that CoLaS fully satisfies the list requirements for an ideal coordination model and language for active objects introduced at the beginning of this chapter. We

consider these requirements to be fundamental for the specification of a coordination model and language for concurrent object-oriented systems. We believe that CoLaS tackles the most important problems existing concurrent object-oriented programming languages have in supporting the specification of the coordination aspect in concurrent object-oriented systems: 1) CoLaS allows the specification of high level coordination abstractions hidden the low level details about how the coordination is done, 2) CoLaS allows the specification of complex interactions concerning more than two participants, even more CoLaS allows the specification of interactions in which more than one participant may play the same role in the coordination; 3) CoLaS supports the separation of coordination and computation, the coordination code refers to the participants exclusively by the role the participants play in the groups and not by their identities or names, similarly the coordination policies are specified in the form of coordination rules defined completely independent of the computation code of the participants and 4) CoLaS supports the evolution of the coordination code, basically new coordination groups can be defined dynamically, new coordination rules can be added and existing removed or redefined and new participants can be added to the group.

The CoLaS coordination model and language uses a high level coordination abstraction called *Coordination Group* that allows programmers to design, to specify and to implement the coordination of groups of collaborating active objects in concurrent object-oriented systems. Designers and programmers of concurrent object-oriented systems get advantage of the separation of the coordination and computation concerns in the specification, construction and evolution of their concurrent object-oriented systems.

We showed in this chapter concretely with six examples how our approach can be used to tackle the complexity of specifying and building concurrent object-oriented systems. The examples selected cover the most important coordination problems in concurrent systems identified in chapter 2 of this thesis: transfer of information, allocation/access of/to shared resources, simultaneity constraints, condition synchronizations, execution orderings, task/subtask dependencies, group decisions and global constraints. The diversity of the problems and their relevance as representative of the different types of coordination problems in concurrent systems show that CoLaS is an interesting and effective model to manage coordination problems in concurrent object-oriented systems.

In most of the showed examples we have used the active object support introduced in Actalk[Brio89a]. to specify a solution. The Actalk framework includes a class called ActiveObject from which participants in the solutions inherit. The ActiveObject class includes all the necessary to support to create and manipulate active objects. Programmers define the ActiveObject class or subclass that fits the best to its object model (i.e., active object, actors, etc.) and focus exclusively on the specification of the computation behavior specific to the problem. We believe that not only the specification of the coordination in a coordination model and language is important to tackle the complexity of building concurrent object-oriented systems, both the specification of the computation related with the concurrency in active objects and the separation of computation and coordination concerns in the coordination model and language are key elements in the simplification of the complexity of the specification, development and maintenance of concurrent object-oriented systems.

We also included in this chapter an example in which the active object support of Actalk is not used, we illustrate with this example how the low level details of the concurrency of the participants appear in the specification of the coordination code. Coordination is specified in CoLaS at a high level.

We also showed examples in which we illustrate some of the problems that CoLaS have in supporting simultaneity constraints. In the presentation of the CoLaS model in Chapter 3 of this thesis we mentioned that there are not CoLaS synchronization rules to define multi-party coordination rules (i.e., rules that de-

pend for their applicability on multiple invocation requests occurring in different participants). We believe that such type of rules are important and that they are an interesting future work .

Finally, we also showed examples in which the CoLaS specification was modified to adapt to changes in the coordination: we introduced new types of participants in the coordination, we specified new roles and we defined new rules. It is important to remember that one of the most important characteristics of the CoLaS model is its capacity to dynamically adapt the coordination specified in the coordination groups. The CoLaS model support three types of dynamic coordination changes: (1) new participants can join and leave the groups at any time, (2) new groups can be created and destroyed dynamically and (3) new coordination rules can be added and existing removed from the groups

CHAPTER 8

Conclusions

We have proposed in this thesis to tackle the complexity of the specification and construction of concurrent object-oriented systems based on active objects using the coordination models and languages approach. The coordination models and languages approach, which appeared in the beginnings of the 90s, promotes the separation of the computation and coordination aspects in the building and the specification of concurrent and distributed systems. According to the coordination models and languages approach a complete programming model can be built out of two separate pieces: the computation model and the coordination model. In our case, the computation model concerns the specification of the active objects that compose those systems and the coordination model the specification of the glue that binds all them together.

Our claim in this thesis is that by separating the specification of the coordination aspect from the computation aspect in concurrent object-oriented systems and by specifying the computation in active objects we simplify the specification, understanding, construction, evolution and validation of properties in this kind of systems. What is new in our approach is the way in which we specify the coordination aspect of a group of collaborating active objects in an abstraction called coordination group. We introduced in this thesis a new coordination model and language called CoLaS specifically adapted to the specification and the programming of the coordination aspect of concurrent object-oriented systems based on coordination groups.

We have identified that the most important problems that existing programming languages have in supporting the specification of the coordination aspect in concurrent object-oriented systems are five: 1) lack of high level coordination abstractions, 2) lack of coordination abstractions for complex interactions, 3) lack of separation of computation and coordination concerns, 4) lack of support for the evolution of the coordination code and 5) lack of support for the validation of the coordination code.

There exists a large number of coordination models and languages [Papa98a], they differ basically in: the kinds of entities they coordinate, the underlying architecture assumed by the models, the coordination media they use to coordinate and the semantics to which the models adhere to. We have included in Appendix A of this thesis a survey of coordination abstractions in existing coordination models and languages. From our point of view none the coordination models and languages included in our survey fully satisfies the list of requirements we have identified to be fundamental for the specification of a coordination model and language for concurrent object-oriented systems. We can summarized the identified requirements as follows:

- The coordination policies must be defined independently of the coordinated entities: the coordination model must enforce the separation of the coordination and the computation aspects. It must be possible to define coordination policies independently of the specification of the coordinated entities.
- It must be possible to define new coordination policies in the coordination model: the coordination model must allow programmers to define their own coordination policies and do not constrain them to use fixed coordination policies.

- It must be possible to incrementally define new coordination policies in the coordination model: the coordination model must allow programmers to use existing coordination policies in the specification of new coordination policies.
- The coordination policies must be multi-party: the coordination model must allow the specification of coordination policies referring to different types of coordinated entities. Furthermore, it should be possible to coordinate not only different types of coordinated entities but also several entities of the same type.
- The coordination policies must be declaratively defined in the coordination model: the coordination model must allow the specification of the coordination in a declarative way avoiding the details of how the coordination is done. High level coordination abstractions should be used to hide the details about how the coordination is done.
- The coordination policies must be control driven defined in the coordination model: the coordination model must respect and adapt to the basic object model to specify the coordination. No new abstractions must be added to the object model to specify the coordination.
- The coordination model must be transparently integrated into the host language: the coordination model must integrate into the host language without imposing any constraint to the host language. The coordinated entities must not be aware of the existence of the coordination layer in the systems.
- The architecture of the coordination model must be hybrid: the enforcement of the coordination in the coordination model must be shared between the coordinated entities and a central coordinator. It must be possible to get advantage of the computing power of the entities being coordinated in the enforcement of the coordination. The coordinator must not be a bottleneck for the system.
- The coordination policies must include the possibility to define proactions in participants: the coordination model must not be exclusively reactive waiting for events or actions occurring in the system. It must specify proactive coordination in the coordinated entities.
- The coordination policies must include the possibility to refer the state of the participants and to the coordination history of the system: the coordination model must allow the specification of coordination referring to the state of the participants and the history of the coordination.
- It must be possible to dynamically modify the coordination policies: the coordination model must allow the dynamic modification of the coordination. It must be possible to easily adapt the coordination policies to new requirements in the systems.
- It must be possible to prove the capability of the coordinated entities to be coordinated: the coordination model must allow the system to validate whether potential coordinated entities are capable of participating in the coordination.
- It must be possible to validate basic safety and liveness properties of the coordination: the coordination model must allow programmers to validate formal properties in the specified coordination.

We have shown all along this thesis that our approach CoLaS, a coordination model and language based on the notion of coordination groups (and specially adapted to specify the coordination in concurrent object-oriented), fully supports the list of requirements defined above. The CoLaS coordination model and language introduces a high level coordination abstraction called *Coordination Group* that allows programmers to design, to specify and to implement the coordination of groups of collaborating active objects in concurrent object-oriented systems. A coordination group is an entity that specifies, controls and enforces the coordination of groups of collaborating active objects. The primary tasks of the coordination groups are: 1) to support the creation of active objects, 2) to enforce cooperation actions between active objects, 3) to syn-

chronize the occurrence of those actions and 4) to enforce proactive behavior on the systems based on the state of the coordination.

The CoLaS coordination model is built out of two kinds of entities: the participants and the coordination groups. The participants are the entities to be coordinated and the coordination groups are the entities that control and enforce the coordination of the participants. The participants in the CoLaS coordination model are active objects: objects that have control over concurrent method invocations. A coordination group itself is composed of three elements: the roles specification, the coordination state and the coordination rules. The roles specification defines the different roles that participants may play in the group. Each role specifies the minimum requirements it imposes to an active object to play the role. The coordination state defines general information needed to the coordination and the coordination rules define the different rules governing the coordination of the group. The coordination rules specify: cooperation actions between participants, synchronizations on the execution of the participants actions and proactions or actions that are initiated by the participants independently of the messages they exchange.

One of the most important characteristics of the CoLaS coordination model and language is its capacity to dynamically adapt the coordination specified in the coordination groups. The CoLaS model supports three types of dynamic coordination changes: (1) new participants can join and leave the groups at any time, (2) new groups can be created and destroyed dynamically and (3) new coordination rules can be added and removed from the groups. The capacity of CoLaS to dynamically adapt the coordination specified in the groups at run time makes it particularly interesting for the specification and construction of modern concurrent object-oriented systems. In those systems evolution is the most difficult requirement to meet since not all the application requirements can be known in advance. No other existing coordination model and language in our survey of existing coordination models and languages in AppendixA of this thesis supports the dynamic evolution of the coordination. It is precisely because the CoLaS coordination model and language supports the dynamic evolution of the coordination aspect in concurrent object-oriented systems that we have suggested in this thesis to use it in the specification and construction of Open Distributed Systems (ODS). We introduced the CoLaSD coordination model into the CORBA framework in the form of a coordination service called CORODS. The CoLaSD coordination model is an extension of the CoLaS coordination model to realize the coordination of distributed active objects. The CoLaSD model takes into account the possibility of failures in the participants common to distributed systems. The CORODS coordination service supports the creation, the moving, the copying, the referencing, the modification and the destruction of coordination groups across the network. Although the CORBA middleware seems to provide all the necessary support for building and executing ODS, the truth is that it provides a very limited support for their evolution. From our point of view the main problem with CORBA is that it does not enforce the separation of the description of the elements from which systems are built and the way in which they are composed. This problem makes those systems difficult to understand, modify and customize. Coordination models and languages in particularly CoLaS may help CORBA to become the right tool to build ODS.

8.1 Evaluation of the CoLaS Model

The CoLaS coordination model and language satisfies all the requirements in the list of requirements identified to be fundamental in the specification of a coordination language for concurrent object-oriented systems:

- Clear separation of the computation and the coordination concerns: in CoLaS the coordination and computation aspects are specified separately in two distinct entities: the coordination groups and the

participants. The coordination groups are specified independently of the participants they coordinate and the participants are specified independently of the coordination groups which coordinate them.

- Encapsulation of the coordination behavior: in CoLaS the coordination of a group of collaborating participants is encapsulated inside coordination groups. The specification of a coordination group includes: the role specification, the coordination state and the coordination rules.
- Support multi-object coordination: in CoLaS the coordination specified in the coordination groups is not limited to two participants but to a group of participants. The coordination groups specifies abstractly the coordination of groups of participants in terms of the roles they play in the coordination and their respective interfaces. The role abstraction allows the specification of the coordination independently of the effective number of participants participating in a group, we talk in this case of a coordination specified intentionally and not extensionally.
- High-level coordination abstractions: in CoLaS programmers do not focus on how to perform the coordination but on how to express it. All the low-level details concerning how the coordination is done are managed internally by CoLaS. For example programmers do not care about locking and unlocking state variable to guarantee their consistency during the coordination. The coordination groups internally serialize the access to the state variables.
- Support evolution of the coordination: in CoLaS the coordination behavior is not fixed. It can change over the time. CoLaS support dynamic coordination changes in three distinct axes in coordination groups: (1) new participants can join and leave the coordination groups at any time, (2) new coordination groups can be created and destroyed dynamically and (3) coordination rules can be added to and removed from the coordination groups.
- Promote the reuse of coordinations abstractions: in CoLaS the coordination groups are specified independently of the participants they coordinate. They can be used to coordinate different groups of participants. Similarly, the participants can be reused in different coordination groups. The minimum requirements imposed to participants to play the roles are specified in the roles interfaces.
- Declarative specification of the coordination: in CoLaS the coordination is specified in a declarative way using rules. The Coordination rules specify: cooperation actions between participants, synchronizations over the occurrence of actions occurring in participants and proactions in participants. The advantage of using rules in the specification of the coordination is that the coordination becomes explicit.
- Incremental specification of the coordination: in CoLaS existing coordination groups specifications can be composed to specify new coordination groups. Complex coordination schemes can be built from simpler coordination specifications.
- Support validation of formal properties: in CoLaS we define a simple methodology that we can use to formally validate properties in the coordination layer. In chapter 6 of this thesis we present such a methodology. The basic idea of this methodology is to transform CoLaS coordination groups in Predicate-Action Petri Nets. Reachability analysis techniques are then used to validate formal properties.

8.2 The Good, The Bad and The Ugly of the Model

We believe that showing only the good aspects of the model will not be useful for learning from the experience of defining a coordination model and language for active objects. It is also important too to show some

problems and ideas of the coordination model that we did not mention during its presentation and during its evaluation.

8.2.1 The Participants

There are three different ways one could structure a concurrent object-based system in order to protect objects from concurrency [Papa95a]: the orthogonal approach, the homogenous approach and the heterogeneous approach. In the orthogonal approach concurrency execution is independent of objects. In the homogenous approach all objects are considered as "active" entities that have control over concurrent invocations. And, in the heterogeneous approach both active and passive objects are provided. Our participants follow the active object approach; themselves they have the responsibility to schedule concurrent requests. The main advantage of the active object model is that programmers do not perform synchronizations at a thread level, the synchronization is done at the object level, most of the time based on the semantics of the methods specified in the object classes. The synchronization is specified in policies that "in principle" can be reused and modified separately of the objects themselves. Although all this sounds easy and simple, the specification of synchronization policies when the number of behaviors to control increases becomes as complex as trying to specify synchronizations with threads. Furthermore, the synchronization policies become very difficult to modify and verify. From our point of view the synchronization policies approach suffers from a scalability and the active objects approach suffers from this problem.

Another problem related with the specification of synchronization policies for active objects concerns the impossibility to combine different synchronization policies within the same object. In some cases we would like to combine different synchronization policies to control different kinds of methods, sometimes we would like to include even class methods and not only instance methods. In all the research works done in synchronization policies we have read it was never mentioned how to combine different synchronization policies, nor how class methods can be combined in the synchronization policies with instance methods. In other words active objects are not the panacea, similarly to concurrency itself there are advantages using them but also there are disadvantages. There is an interesting paper written by Milicia and Sassone [Mili04a] analyzing what is the current situation of modern concurrent object-oriented languages like Java and C# related with synchronization policies. The conclusion is that still today those languages suffer from the same problems related with the specification of synchronization policies identified in [Mats94a]. They suggest that the separation of concerns promoted by Aspect Oriented programming may finally solve these problems. They mention also that coordination languages, in particular the Composition Filters approach [Berg94a] are an interesting way to solve these problems too.

Finally, the last important point we wanted to mention here is that even if theoretically the goal behind the specification of the coordination models is to separate coordination from computation, as soon as one associates a different synchronization policy to the participants (i.e., different to mutual exclusion which the one that we use) the details of the synchronization policy appear in the specification of the coordination language. Remember for example that the cooperation rules (i.e. defineBehavior rules in CoLaS) define behaviors that participants "learn" when they join roles in the groups, their execution depends on whether they validate or not the synchronization policy associated with the participants. We experimented with different synchronization policies in this work in order to increase the internal parallelism of the model. If we consider for example a Multiple readers only one Writer synchronization policy (i.e., object methods are divided into two categories: readers and writers; readers methods executed concurrently if not writer method is running and writers methods executed mutual exclusively) it is necessary to specify for the cooperation rules

whether the rules should be consider as reader or writer methods. In “theory” the specification of the behavior of the participant must be independent of the coordination specified in the group, sometimes this is not possible like in the example.

Communication

We already said that participants communicate by exchanging messages asynchronously in CoLaS. Our experience in the specification of coordination groups has shown us that most of the time the participants communicate synchronously. Any solution to the specification of a coordination model must include at least these two forms of communication. In CoLaS we use the futures generated by the method invocation requests in other participants to synchronize the execution of messages in the participants. When a reply is expected we sent the message *value* to the future to receive the result and when no value is expected but we want only to synchronize the execution of messages we sent the message *wait*. In both cases either because the result is not ready of because the other participant have not finished to execute to method invocation the participant who sent the request blocks in the future.

Another important type of communication used in the CoLaS model is group communication. In the CoLaS model it is possible to send a message to a all the participants of a role (i.e., multicast message) [Coul94a]. One of the problems of the CoLaS model is that the communication model used to communicate between participants and between a participant and a role are not the same. We do not manage replies when a message is sent to all the participants of a role. (i.e., multicasted). It will be interesting to extend the communication model used in CoLaS to communicate with roles managing multiple-replies. This can be an interesting future work.

8.2.2 Role Specification

We already mentioned in this thesis that we believe that our role concept in very weak, in particular the role interface definition. Even if an active object implements the behaviors specified in the role interface, there is not guarantee that the coordination will not break. It will be interesting to extend the specification of method signatures with returned values and argument types to obtain a typed interface definition. The advantage of having a typed interface definition is that we can specify more precisely the requirements that we impose to participants to play the roles. In our model the role concept is fundamental in the specification of groups it allows to identify and to specify abstractly the coordination of a group of participants sharing the same coordination behavior.

8.2.3 The Coordination State

Concerning the coordination state the three types of variables defined in CoLaS correspond to three possible accessibility constraints that can be defined on variables specified in a group. Group variables can only be used by all the participants playing roles in the group, role variables can only be used by the participants playing the role and participants variables can only be used by the participant to whom they belong. We do not specify types for the variables, the type of a variable corresponds to the actual value stored in the variable. If the type of the value stored in the variable changes the type of the variable also changes.

Concerning the concurrent access to group and role variables (i.e. shared variables), it is important to know that these two types of variables are stored in the group entity. A request for the value or for the modification of the value of these variables corresponds implicitly to a method invocation request to access or

to modify the variable in the group entity. The concurrent access to variables in groups is controlled by mutual exclusion semaphores implicitly associated to the variables.

8.2.4 The Coordination Rules

The CoLaS model defines three types of coordination rules: cooperation rules, reactive rules and proactive rules. The first two types of rules depend for their evaluation of the messages exchanged by the participants playing roles in the group. Cooperation rules define actions that must be executed when the participants playing some role receive method invocation corresponding to the behavior specified in the cooperation rules. Reactive rules define actions that must executed at some specific points during the processing of the method invocations by the participants. Some reactive rules define additionally conditions that must be validated before the actions associated with the rules be executed (i.e., synchronization rules). The four evaluation points defined in the CoLaS model are: atArrival, atSelection, atSent and atEnd.

The important point here is that, from the separation of concerns point of view only the rules associated with the atArrival and atSent evaluation points respect the separation of concerns promoted by coordination models. If we consider a participant as a black box around which the coordination is specified, only the arrival and the departure of messages to and from the participant can be perceived as events from outside the participant. In other words a pure coordination model for objects must define exclusively actions related with these two types of events.

Why we have defined two more evaluation points in the CoLaS model? the atSelection evaluation point corresponds the to moment when a method invocation is ready to be executed by the participant and just after the synchronization policy was validated. The CoLaS models includes a synchronization rule Disable which is evaluated at the atSelection point. The Disable rule is an important rule because it allows one to specify condition synchronizations [Andr00a]. Andrews specifies that there are two basic kinds of synchronizations in concurrent systems: mutual exclusion and conditions synchronizations. Mutual exclusion is the problem of ensuring that critical sections of statements do not execute at the same time and condition synchronization is the problem of delaying a process until a given condition is true. The “delaying a process” in an active object corresponds to the delaying of the execution of a method invocation by the object. This is the reason why we have defined the atSelection evaluation point and why the rule Disable exists. Concerning the evaluation point atEnd, we do not have a strong justification for this. The CoLaS model defines a unique rule interceptAtEnd associated with this point. The interceptAtEnd rule is basically used to update state variables related with the execution of method invocations, for example: a variable storing the identity of the last method executed, or a variable counting the number of method invocations executed. At a first view, it seems to be possible to include every action specified in the InterceptAtEnd rule in the specification of coordination actions in cooperation rules, remember that cooperation rules define behaviors that are requested by other participants as method invocations. The problem is that the InterceptAtEnd rules are not necessarily associated with behaviors specified in cooperation rules. All the different interception rules can be associated to methods not related with the coordination behaviors specified in the coordination group for example to behaviors appearing in the role interfaces. This is not very common but it happens. This is the reason why we prefer to keep this rule and this evaluation point in our model.

8.2.5 Dynamic Aspects

The CoLaS model supports three types of dynamic coordination changes: first new participants can join and leave the groups at any time; second, new groups can be created and destroyed and third, new coordination

rules can be added and removed from the groups. The problem with the three types of modifications is that it is almost impossible to determine in advance (i.e., without a formal analysis) which are the consequences of these actions for the coordination specified in the group. If we consider the Electronic Vote problem introduced in this chapter for example, What will it happen if for example during a vote process a new voter joins the group after that the vote message is sent by the initiator of the vote to all the voters? the answer is, the new voter will never receive the request to vote. And, because it makes part of the participants playing the role Voter and because the decisions are taken unanimously the result of the vote will be negative even if the others voter voted positively.

The dynamic addition and removal of rules can even have more serious consequences in the coordination specified in the group, they can for example introduce errors or deadlocks in the coordination. Suppose for example a cooperation rule which specifies an action corresponding to the asynchronous send of a message to another participant and where no behavior associated with such a message exists in the other participant, or a cooperation rule for example that introduces a cycle in the communication protocol specified in the group. We need to be sure that whenever we modify the group specifications we do not introduce problems in the coordination. Unfortunately the only way to control that the dynamic modifications of the groups do not introduce problems in the coordination is to formally verify the coordination specification at the time the modifications are made. In our case this is completely unrealistic, we have defined a validation method that transforms CoLaS groups into Petri Nets to validate safety and liveness properties. Actually the transformation process is not automatic and the interpretation of the results requires some knowledge of Petri Nets and some knowledge of the coordination specified in the coordination groups. CoLaS is not the only coordination model and language that suffers from this problem, in general most of the existing concurrent object-oriented programming languages and coordination language suffer from this problem. The real problem here is that in general the validation of properties in programs in not made in the language in which the programs are written. The ideal solution will be to include in the language mechanisms to automatically validate the code. Today, we are still quite far from this ideal solution.

8.3 Some Implementation Concerns

8.3.1 The Role Concept

How to integrate the role concept into object oriented systems is a thesis research subject itself. In CoLaS the fact that a participant plays a role concretely means, that there are methods (cooperation rules) and instance variables (role state variables) related to the coordination that must “make part” of the participant classes (they must appear in the classes definitions). There are two possible ways the role concept can be introduced to an object-oriented model: either the language provides multiple inheritance and then a role can be modeled as a class or the role characteristics are modeled at the instances level using delegation. If we choose the first option this will imply for our model that participant classes hierarchies will need to change dynamically. For example a participant playing a role Voters in the ElectronicVote example will belong at the same time to the class Voters and to the class Person, a new class PersonVoter specifying the multiple inheritance will need to be created. It will be necessary to modify the class hierarchies of the participant classes at runtime. If we choose the second approach this will imply that at some point it will be necessary to introduce in the internal representation of our participants some specific knowledge concerning how to manage the access and execution of the behaviors (i.e, methods) and variables associated with the roles violating in this way the separation of concerns between computation and coordination. In other words, the

introduction of the concept of role into the model introduces a lot of advantages like the possibility to specify multiparty coordination and to abstract the specification of the coordination from the specification of the coordinated entities, but at the same time it introduces a series of problems to be considered. Actually what we do in the implementation of CoLaS is that we define internally for all the participants a variable named *roles* to keep the reference to the roles the participant plays. How clean can be kept the separation between coordination and computation in the implementation of the CoLaS model when the role concept is introduced in a object-oriented language is not clear. All modifications done to the participant hierarchy violates too the principle of separation of concerns promoted by the coordination models and languages. The coordination model must not interfere with the computational specification of the participants. To our point of view, there is not clear solution to this problem, we believe that this can be a thesis subject by itself. We believe the role concept in our model must be considered more in detail. It will be interesting to have a look in different thesis actually working in the integration of the role concept into object-oriented languages. It is important to always analyze the implications of every implementation decision in the model before to implement the model, in particular if they affect the separation of concerns between computation and coordination.

8.3.2 Coordination Enforcement

There are two different architectures that can be used to implement the enforcement of the coordination specified in the CoLaS groups: a centralized architecture where the coordination rules are validated by a central entity representing the group (i.e., group entity) and a decentralized architecture where the coordination rules are enforced by the participants playing the roles in the groups. Our first implementation of the CoLaS model followed a centralized architecture, the participants sent notifications to the group entity to indicate they were at one of the four evaluation points defined in the model, then the group entity evaluated the rules associated with the evaluation point and if the execution condition associated with the rules validated to true, the rules were executed by the group entity. The problem with this architecture is that: first the group entity becomes very easily a bottleneck giving that the execution of all the messages related with the coordination are controlled and executed by the group entity and second that the centralized architecture does not uses all the potential computational power of the participants given that all the coordination work related with the enforcement of the coordination is done in the group entity.

The decentralized approach (the one we have chosen in our current implementation of the CoLaS model) in the other hand, divides the enforcement of the coordination work between the participants and the group entity. Most the information necessary to the enforcement of coordination rules associated with roles can be stored in the participants (i.e., the coordination rules associated with the roles they play and the Participant variables in the Coordination State). The main advantages of the second approach are that the group entity does not represents anymore a bottleneck in the architecture and that the computational power of the participants is also used in the enforcement of the coordination. In the second architecture only the modifications of the group and the role variables in the coordination rules imply the communication between the participants and the group entity during the enforcement of the coordination. The big disadvantage with the second architecture is that whenever the specification of the group changes (i.e., addition and/or removal of rules) the changes must be notified to all the participants and their coordination rules updated.

In general whatever architectural approach is used to build the model it requires that participants include some behavior that allows then to interact with the group entity. The point here is that both implementations impose some conditions in the internal behavior of the participants that make that some aspects of the coor-

dination at the end appear mixed again within the computational aspect of the participants. In CoLaS our participants are subclasses of a class named ActiveObject, a class specially adapted to interact with the group entities to enforce the coordination.

8.4 Future Work

As we already mentioned in this thesis, we consider that there are several aspects in the CoLaS coordination model which deserve special attention and thus some future work.

- [*Coordination model*] we believe it will be interesting to work in a better integration of the role element to the model. For example, it will be interesting to extend the specification of the role interfaces as we mentioned during the presentation of the model. The ideal will be to be able to specify as precisely as possible the requirements imposed on the participants, of course without fixing their types. We have seen that the role interface is an extremely good idea to separate the specification of the participants from the coordination in the coordination groups.
- [*Coordination model*] we believe it will be interesting to work in the specification of multi-party rules, rules that depend for their application on multiple invocation requests occurring in different participants. For example it will be interesting to be able to specify condition synchronizations implicating more that one participant. One interesting work in this direction is Interacting Processes [Fran96a]. We showed in the Vending Machine example introduced in Chapter 7 the utility of such kinds of rules.
- [*Coordination model*] we believe it will be interesting to work in the problems related with the introduction of different synchronization policies into the model. We already mentioned in this thesis that the introduction of new synchronization policies in the objects affects the specification of the coordination. The coordination model has to specify how the new coordination behavior specified will behave with respect to the synchronization policy. How to keep the separation of computation and coordination concerns in this case is a big challenge.
- [*Validation*] we believe it will be interesting to work more in the interpretation of the results obtained from the validation of properties in the Petri Nets in which we transform the coordination groups.
- [*Validation*] we believe it will be interesting to work in tools that automatically transform coordination groups in Predicate-Action Petri nets using the mapping function defined in this thesis. Such a work will avoid to programmes possible errors introduced during the manual transformation of the groups.
- [*Validation*] we believe it will be interesting to work in the validation of properties when multi method invocations are processed at the same time.
- [*Validation*] we believe it will be interesting to work in the validation of properties when the objects participate in different coordination groups at the same time.
- [*Validation*] we believe it will be interesting to be able to determine coordination problems generated by non coordinated behavior. Until now we can not guarantee that a non coordinated behavior in a participant does not affect the coordination behavior of a group, even if the two behaviors are specified separately.
- [*Implementation*] the main challenge from the implementation point of view is to keep the separation of the coordination and the computation in the CoLaS model in the implementation of the model. The biggest challenge is the definition of coordination behavior at the object level and not at the class level in an object oriented language. Normally in an object model the behavior of the objects

is defined in their classes. In CoLaS, coordination behavior is “added” to the participants when they join the roles, this behavior is defined at the object level and not at the class level. If we add the new behavior at the class level other instances of the same class of the participant object will be affected by the coordination even if they do not play roles in a coordination group. We believe it will be interesting for example to define an object model in which not all the behavior is specified at the class level but in which it will be possible to specify behavior specific to objects.

APPENDIX A

Coordination Abstractions

A.1 Abstract Communication Types [Aksi92a][Berg94a]

The Abstract Communication Types (ACT in the following) approach introduces abstractions to structure, abstract and reuse object interactions. In the ACT model *composition filters* are applied to abstract communication among objects. The basic object model is extended to introduce input and output composition filters that affect the sent and received messages respectively.

Composition Filters

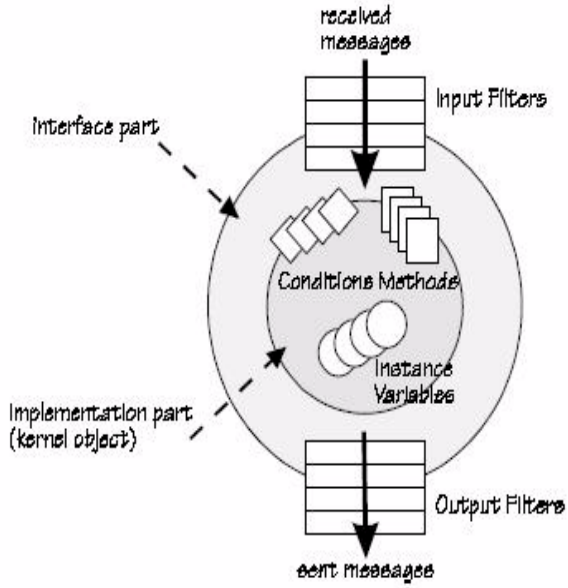


Figure A.1 : Composition Filters

A composition filter object consists of two parts: an interface and an implementation part. The interface part deals with incoming and outcoming messages. It consists of one or more input and output filters, optional internal and external objects and method header declarations. The implementation part contains method definitions, instance variable declarations, definitions of conditions concerning instance variables, an optional initialization operation. The implementation part is fully encapsulated within the object.

There are two types of composition filters: input and output filters. An input filter specifies conditions for message acceptance or rejection and determines the subsequent actions. If a message passes through the input filters it can be further delegated to internal objects, methods or external objects. All the messages that originate from method executions within the object sent to objects that are outside the boundaries of the current object pass through the output filters. Output filters specifies conditions and actions on the sent of messages.

The composition filters model is adopted by the Sina language [Aksi89a]. The current version of Sina provides a number of primitive filters such as: dispatch, meta, error, wait and realtime. These filters can be used as both input and/or output filters. These filters are orthogonal to each other, and therefore they can be combined.

A composition filter consists of a number of filter elements. When a message is to be evaluated by a filter the message is checked against the elements of the filter in left to right order. A filter element consists of three parts:

- A condition, which specifies a necessary condition to be fulfilled to continue with the evaluating of the filter. A condition always results in a boolean value, and is free of side effects.
- A matching part, which specifies a pattern matching expression against which the evaluated message is matched. A pattern matching expression refers to the message's selector.
- A substituting part, which specifies how parts of the message are replaced.

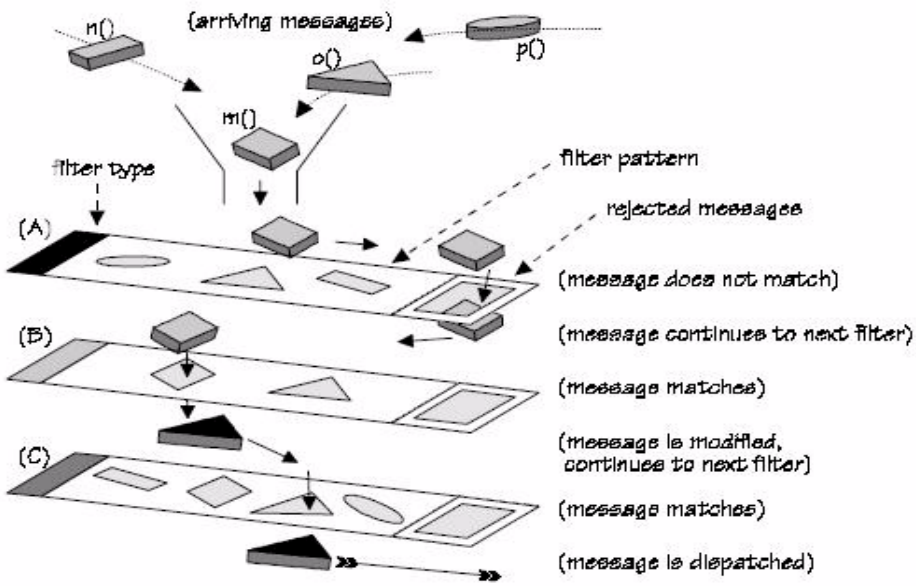


Figure A.2 : Filters evaluation

Evaluation

The selector of the processed message (i.e. received or sent) is matched against the selector of the matching part of each filter element; when the filter element does not match the subsequent filter is tried. When a filter matches the condition associated with the filter, the filter is applied to the message. The type of the filter determines what happens to the message. Commonly the last filter in a sequence filters is a dispatch filter, which results in delegation of the request message to its target object. In (**Figure A.2**), we show the evaluation process of the filters.

Inheritance and Delegation

Input filters can be applied to perform basic object oriented data modeling techniques such as inheritance and delegation. In the composition filters model, inheritance is not directly expressed by a language construct but is simulated by input filters. Inheritance can be simulated by delegating messages to internal objects.

ACT

ACTs are classes that abstract interaction among objects. They operate on first class representations of messages. For converting a message into is first class representation (reification) a new filter class meta-filter is used. The meta-filter has the same structure that the dispatch filter previously described. They differ in that messages accepted by meta-filters are first converted into instances of the class Message and then passed as argument of new messages to the ACT objects. The ACT object can retriever information from the message and modify the contents of the message. The ACT object can convert an instance of Message back to a message execution. ACTs can be further classified as abstract sender types (AST) and abstract receiver types (ART). Both types of ACTs objects are responsible for abstracting one way communication among objects. An AST object is responsible for handling outgoing messages and an ART object for handling incoming messages.

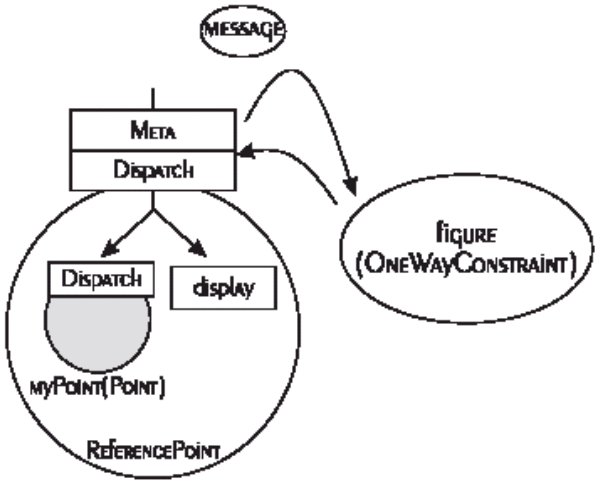


Figure A.3 : ASTs object controlling outgoing messages

In (**Figure A.3**), we show graphically the specification of an ART communication type introduced in [Berg92a]. In the example an instance of Reference Point is supposed to store the reference to the coordinates of a figure. When the references point are changed, then all the dependent graphical objects must be updated accordingly. To compose this constraint behavior with ReferencePoint, the interface of the class ReferencePoint is extended by declaring an object figure of class OneWayConstraint in the externals clause and by adding a new input filter called constraint of class Meta (**Figure A.4**). The class OneWayContraint is an ART which provides the consistency of the dependant variables when the independent variable changes. Whenever the reference point is moved using the method moveTo the applyConstraint method of the OneWayConstraint is applied in the figure. The constraints associated with the figure are specified using the method putConstraints. This method accepts an ordered collection of instances of class Block as argument. Each block defines a constraint expression to be solved.

```
1.class ReferencePoint interface
2.
3.  externals figure: OneWayConstraint;// instance of the ART class
4.  internals myPoint: Point,
5.  methods display returns Nil;// display itself
6.
7.  inputFilters
8.    {constraint: Meta={True=> [*.moveTo]figure.applyConstraint};
9.     disp: Dispatch={true=> myPoint.*, True=>inner.*};}
10.end
11.
12.class OneWayConstraint interface
13.
14. methods
15.   applyConstraint(Message) returns Nil; // independent value
16.   putDependants(OrderedCollection(Any)) returns Nil;
17.   size returns Integer;
18.   putConstraints(OrderedCollection(Block) returns Nil;
19.   getConstraints returns OrderedCollection(Block);
20.
21. inputFilters
22.   {disp: Dispatch = {true => inner.*}
23.
24.end
```

Figure A.4 : ReferencePoint and OneWayConstraint classes specification

A.2 Activities [Kris93a][Kris97a]

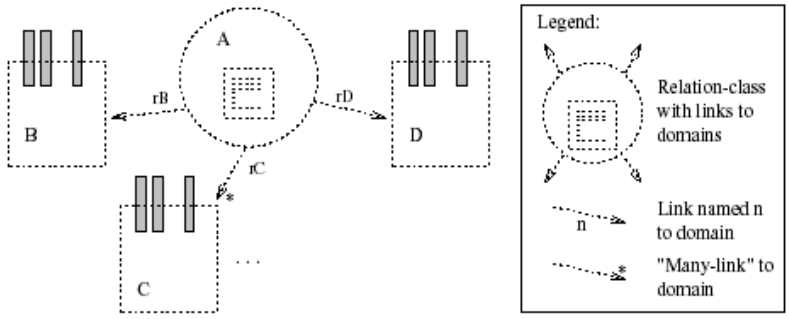


Figure A.5 : Graphic representation of an Activity

Activities are abstractions to model the interplay between groups of objects over a given time. An activity is defined by specifying its participants and a directive. The participants specify the objects that participate to the activity and the directive the actions that compose the activities. A directive may include other activity-objects and method activations of the participants. The interplay between the participants, which is described collectively, specifies “who is doing what to whom”. The atomic elements of an activity’s directive usually comprises three things: subject (who), object (whom) and verb (what is done).

Assuming the existence of classes B, C and D an activity class A is defined as the relation between participant classes B, C and D as: Class A [B, C, D] (...). The activity A defines a relation between three objects corresponding to the domains of the participant classes B, C and D. In the specification of an activity it is possible to give names to the objects that participate in the activity Class A [rb: B, rC: *C, rD: D], the participants in the activity A are named rb, rC and rD. The ’*’ in the specification of the participant domain means that rc may refer to an arbitrary number of C objects. There is not restriction in the number of activities in which participants may play at the same time. In (**Figure A.5**), we can see the graphic representation of the activity A and its participants.

```
CLASS C ( ... )

CLASS R1 ROLE C ( ... )
CLASS R2 ROLE C ( ... )

CLASS A1 [ ... , R1, ... ] ( ... )
CLASS A2 [ ... , R2, ... ] ( ... )
```

Figure A.6 : Roles specification

Because not every aspect of a participant is relevant for every activity in which it participates activities introduces the notion of *role*. A role specifies the different aspects of the participants that are relevant for an activity. Roles are described as role-classes. In (**Figure A.6**), R1 and R2 are role classes for class C. The activity classes A1 and A2 has role classes respectively R1 and R2 as one of their domain classes. An object of class C can acquire role-objects form R1 and R2 during its life cycle and participate in the activities A1 and A2.

Specialization and Aggregation

```
Class G[... Pg: ...](...action(...))

Class S: G [... Ps: ...](...action (...))
```

Figure A.7 : Specialization Mechanism

Activities specialized from another (super) activity are called sub-activities (**Figure A.7**). An activity may be redefined in several ways: by adding more participants classes, by refining the description of existing participants classes (i.e., substituting a participant class by a subclass) and by refining the actions already part of the action sequence.

```
Class P:[... pP: ...](... action (...))
Class W:[...pW:...](...action(...oP..))
```

Figure A.8 : Aggregation Mechanism

Activities can also be aggregated to form larger activities (**Figure A.8**). The aggregated activities are called part-activities. Each activity/part-activity is responsible for managing its associated interplay.

A.3 Activities and Environments [Arap91a]

This work define the notions of *objects, activities* and *environments* within a temporal context. These notions are used to formally describe dynamic evolution of object behavior and interactions of collections of cooperating objects. The objects represent entities of the problem domain. They communicate between them by sending and receiving messages. The activities describe interactions of collections of objects and the environments describe coordination of a set of activities. The notions of object, activity and environment are formally specified using the language of first-order temporal logic FTL [Abad89a].

FTL Syntax - Modal Operators

`[]`(“always in the future”)	[] (“always in the past”)
`Y` (“sometime in the future”)	Y (“sometime in the past”)
`O` (“next”)	O (“previous”)
`u` (“until”)	S (“since”)

Objects

Objects are entities that represent the problem domain. An object communicates with other objects by sending and receiving messages. A message represent a request for the receiver to perform some task or to return to the sender some information. Objects have associated constraints. Object constraints specify temporal orders in which messages are to be sent to and received from an object.

In (**Figure A.9**) we show the specification of the class ControlTower introduced in [Arap91a] and representing a control tower in an airport. The messages section specifies the messages that can be received by the ControlTower instances. The message take_off and landed for example inform the tower which airplanes have taken off and landed. The constraints section specifies temporal constraints associated with these messages. The first constraint for example ensures that whenever a message request_take_off with parameter x is received, sometime in the future the message permission_take_off will be send to x.

```
class ControlTower {
  messages
    request_take_off(x,self,y);
    request_land(x,self,y);
    take_off(x,self,y);
    landed(x,self,y);
    permission_take_off(self,x);
    permission_land(self,x);
  constraints
    (x)[]((y) request_take_off(y,self,x) =>
         O Y permission_take_off(self,x));
  ...
}
```

Figure A.9 : ControlTower class specification

Activities

Activities model interactions of collections of objects. An activity specification is divide in three parts: the agents, the messages and the constraints.

The agents part specifies the different agents which objects may represent. Objects participate in activities by becoming agents (representing an agent). For each agent an object class is specified. Candidate objects for representing a particular agent must be instances of the object class associated with that agent. An object may either decide on its own or be solicited by another object to participate in an activity. Objects are not restricted to represent only one agent in an activity. It is possible for the same object to represent several agents of an activity. An object may also participate in several activities simultaneously.

The messages part contains the names of messages that can be sent to and received from the activity. The communication between activities takes place by exchanging messages like for objects.

The constraints part contains the set of temporal constraints on the messages exchanged by the activity. The constraints associated with the activities differ to the constraints associated with the objects in that they

describe the interaction of a collection of objects. The set of constraints associated with a particular activity are verified only once the message start is sent to that activity. The message start can only be sent to the activity when all the activity agents are represented by some object. A sequence of messages exchanges by an activity is legal if the sequence of messages satisfies the temporal constraints associated with that activity. Objects participating in a particular activity cannot stop their participation unless a legal sequence of messages has been exchanged with respect to that activity. Finally an activity can only be deleted when there are not objects representing agents in that activity.

```
activity TakeOff {
  agents
    ct: ControlTower;
    pl: Airplane;

  messages
  -- related to agent ct
    request_take_off(self, ct, y);
    taken_off(self, ct, y);
    permission_take_off(ct, self);
  -- related to agent pl
    com_take_off(self, pl);
    com_pos_take_off(self, pl);
    set_pilot(self, pl, y);

  constraints
  [](com_take_off(self, pl)  => O Y permission_take_off(ct,self);
  [](taken_off(self, ct, self) => O Y com_take_off(self,pl);
  ...
}
```

Figure A.10 :Take-off activity specification

In (**Figure A.10**), we show an activity introduced in [Arap91a] describing the take off activity of an air traffic control application. It contains two agents: airplane and control tower. Constraints associated with the activity concern the communication between the activity and the control tower and the activity and the airplane. There is not direct communication between the airplane and the control tower. The communication between the activity and the control tower is related to the request to take off. When the permission is granted the activity communicates to the airplane to indicate the authorization to take off.

Environments

An environment defines relationships between activities and relationships between activities and object participating in activities. An environment specification consists of two parts: the first part specifies a set of activities composing the application. The second part a set of constraints concerning object participation in activities, the temporal order of activity executions and object flow between activity executions. The constraints associated with an environment can be classified in five groups: 1) local constraints (constraints which must hold for a set of objects to participate in a particular activity), 2) flow constraints (constraints

that must hold for a set of objects to participate in a particular activity with respect to their participation in previous, current or future activities), 3) message-message constraints (temporal orders between messages exchanged by the different activities), 4) message-activity constraints (temporal orders between activity executions and messages exchanges) and 5) activity-activity constraints (temporal orders between activity executions). The first two classes define constraints expressing conditions for objects to participate in activities, the last three represent temporal relationships between message exchanged and/or activity executions.

Consistency of the Specifications

Testing the consistency of a given specification of an application reduces to testing whether there exists at least one sequence of message exchanges satisfying the specification (constraints). According to Arapis [Arap91a], a general satisfiability algorithm for FTL does not exists. However and under the assumption that at any point of time the domain of interpretation for FTL formulas is finite it seems that is possible to find an algorithm for testing satisfiability of FTL formulas. This work proposes a such algorithm.

A.4 Cast [Vare99a]

In this work coordination is modelled hierarchically by grouping actors [Agha86a] into casts. Each cast is coordinated by a single director. Coordination in the hierarchical model is accomplish by constraining the reception of messages that are addressed to particular actors. Messengers are special migrating actors that represent a message from a remote cast.

An Actor can only receive a message when the coordination constraints associated with the reception of such a message are satisfied. It is not clear in this work the kind of constraints that can be imposed to the messages received, we assume that the constraints correspond to those previously introduced in synchronizers [Frol93a] in which one of the authors was previously implicated. The coordination constraints are checked for conformance by the casts directors.

The director-actor relationship form a set of trees. A message from a sender actor is received by a target actor only after approval by all the directors in the target actor‘s coordination forest path up to the first common director, if such a director exists, otherwise, approval is required of all the directors in the target‘s coordination forest path up to the top level. An actor can have at most one director at a given point in time. However a director may itself belong to a cast and thus be coordinated by another director.

It is important to remark in this model it is also possible to have completely “uncoordinated” actors. By “uncoordinated” we mean actors that do not belong to a cast and which therefore have no external constraints on message reception.

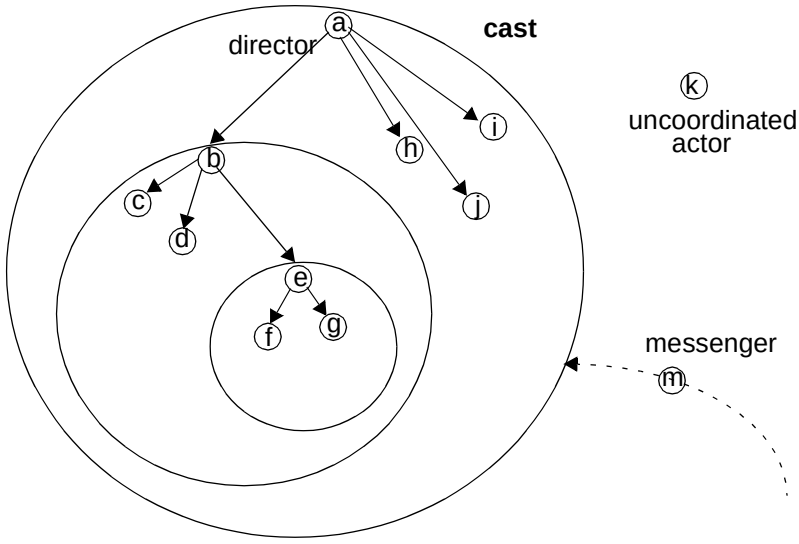


Figure A.11 Coordinated activity with casts, directors and messengers.

In (**Figure A.11**), we show a sample actor configuration. Some examples of valid messages paths are:

- A message from any actor can go directly to actor a, or actor k.
- A message from actor f to actor g has to be approved by director e.
- A message from actor c to actor f has to be approved directors b and e.
- A message from actor k to actors b, c, d,...,j has to be approved by actor a.

A.5 Connectors - FLO [Duca97a][Duca98a]

A connector is a special object that connects components. A component can be an object or a group of objects. In (**Figure A.12**), we show a connector template as introduced in the FLO language [Duca97a]. A connector specifies how message exchanges influence the behavior of the connected components. Components participate in connectors by playing roles. A component can participate in a connection if it provides an interface compatible to a connector's role. Roles are specified by variables names in the connector template declaration. A role specifies the set of method selectors on a component which will be intercepted or invoked by the connector (a subset of the component's interface).

New connectors can be define from existing connectors from existing connector templates by adding new rules or by combining connectors definitions. In (**Figure A.12**), a connector template with the keyword inherit preceding a list of existing connector templates, the template specifies a new connector as a combination of existing connector definitions.

```
(defConnector connectorAB (:roleA :roleB) ; a list of role names
  :inherit ((...)); a list of ancestors
  :var     ; some connector variables
  :behavior; interaction rules of connector
)
```

Figure A.12 : Connector specification

Connector's behavior

The behavior of a connector is defined by means of a set of interaction rules which specify how the messages received by participant objects should be controlled. In (**Figure A.13**), we show the formal specification of the rules. Each rule is defined by a filter, an operator and a context. The filter specifies which messages should be intercepted for which kinds of participants, given by a role name. The operator defines the semantics of the rule and gives meaning to the context of rule. The context of the rule specifies the execution of messages: a list of method invocations on participants.

```
Rule ::= Filter Operator Context
Filter ::= Selector Rolename List-of-Calling-Args
Context::= Message+
Message::= Selector Rolename Args
Operator ::= implies | permitted-if | corresponds
```

Figure A.13 : Filters syntax

There are three types of rule operators: implies, permitted-if and corresponds. The implies filter is used to propagate messages to the sender object or to other objects after the reception and the execution of a messsage. The permitted-if filter inhibits the execution of the received message is a condition, named a guard is satisfied. The corresponds filter, delegates the execution of the received message to some objects. The delegated message can be different to the received message.

```
(defConnector calculator-displayer(:calculator:displayer)
:behavior
  (((compute-new-value: calculator val) implies
      (add-new-value: displayer (convert connector val result)))
   ((compute-new-value: calculator val) permitted-if
      (free-variables? : displayer))))
  ;computing a new value is only possible it the displayer can display it
  ; end of behavior definition

(defmethod convert((conector calculator-displayer) v1 v2)
  (list (from-float-to-pixels v1) (from-float.to-pixels v2))
    ; a conversion from two floats to a list of pixels(list
```

Figure A.14 : A Calculator-Graphic Displayer's Connector

In (**Figure A.14**), we show a calculator-graphic displayer connector introduced in [Duca98a]. The example represents a calculator component that generates new data when the method new-value is invoked. The calculator displays the calculated data on a graph displayer with only displays a limited number of values on x-y axes; and has a method for displaying a value (add-new-value) and one for removing a value by clicking on the display (remove-one-value). The connector specifies that each value computed by the calculator should be displayed by the displayer, in other words that when the graph displayer is full new values should not be computed. Additionally, it specified that when the format of the calculator's result values is not compatible with the displayer's format, the value must be converted.

A.6 Connectors - ArchJava [Aldr03a]

ArchJava is an extension of Java that allows programmers to express the architecture of an application within the source code. ArchJava adds new language constructs to support component, connections and ports.

```
1.public component class PoemPeer {
2.  public port search {
3.    provides PoemDesc{} search(PoemDesc partialDesc) throws IOException;
4.    provides void downloadPoem(PoemDesc desc) throws IOException;
5.  }
6.
7.  public port poems {
8.    requires PoemDesc[] getPoemDesc();
9.    requires Poem getPoem(PoemDesc desc);
10.   requires void addPoem(Poem poem);
11. }
12.
13. public port interface client {
14.   requires client(InetAddress address) throws IOException;
15.   requires PoemDesc[] search(PoemDesc partialDesc, int hops, Nonce n);
16.   requires Poem download(PoemDesc desc);
17. }
18.
19. public port interface server {
20.   provides PoemDesc[] search(PoemDesc partialDesc, in hops, Nonce n);
21.   provides Poem download(PoemDesc desc);
22. }
23.
24. void downloadPoem(PoemDesc desc) throws IOException { ... }
25. ...
26.}
```

Figure A.15 PoemPeer class

A component in ArchJava is a special kind of object that communicates with other components in a structured way. Components are instances of component classes. Components in ArchJava communicate through ports. A port represents a logical communication channel between a component and one or more components that it is connected to. Ports declare two sets of methods, specified using the requires and pro-

vides keywords. A provided method is implemented by the component and it is available to be called by other components connected to this port. Conversely, each required method is provided by some other component connected to this port. A component can invoke a required method declared in one of its ports by sending a message to the port. If a component communicates with multiple different components using the same interface, it can declare a port interface and the create a port of that interface type for each component it needs to communicate with. The goal of the ports is to specify both the services implemented by a component and the services a component needs to do its job, making dependencies explicit. In (**Figure A.15**) we can see an example introduced in [Aldr03a]. A PoemSwap is a simple peer-to-peer program for sharing poetry-online. The PoemPeer component represents the network interface of the PoemSwap application. The proems port requires methods that get descriptions of all the poems in the database, retrieve a specific poem by its description and add a poem to the database.

```
27.public component class PoemSwap {
28.  private final SwapUI = new SwapUI();
29.  private final PoemStore store = new PoemStore();
30.  private final PoemPeer peer = new PoemPeer();
31.
32.  connect pattern SwapUI.poems, PoemStore.poems;
33.  connect pattern PoemPeer.poem, PoemStore.poems;
34.  connect pattern SwapUI.search, PoemPeer.search;
35.
36.  public PoemSwap() {
37.    TCPConnector.registerObject(peer, POEM_PORT, “server”);
38.    connect(ui.poems, store.poems);
39.    connect(peer.poems, store.poems);
40.    connect(ui.search, peer.search);
41.  }
42.
43.  connect pattern PoemPeer.client, PoemPeer.server with: TCPConnector {
44.    connect(sender.client, PoemPeer.server)
45.      with new TCPConnector(address, POEM_PORT, “server”);
46.    }
47.  };
48.}
```

Figure A.16 PoemSwap architecture.

In ArchJava, the set of permissible connections in the architecture is declared using connect patterns. Aconnect pattern specifies two or more port interfaces that may be connected together at run time. Actual connections are made using connect expressions that appear in the methods of the components. A connect expression specifies concrete component instances to be connected in addition to the connected ports. Each connected pattern must provide a connection constructor for each of the required connection constructors declared in the connected ports. Instead of using ArchJava’s default type checking rules, connect patterns can specifiy that a user-defined connector class should be used for type checking. The default implementation of type check returns an error for each required method that has no matching provided method, or has more than one matching provided method. In (**Figure A.16**) we can see a textual description of the Poem-

Swap architecture. The PoemSwap component class contains three subcomponents- a user interface, a poem store and the network peer. Connect patterns show how these components may be connected and the connect expressions in the constructor link the components together following these patterns.

Connector abstractions are defined using the archjava.reflect library. This library defines a Connector class that user-defined connectors classes extend. The class Connector provides a hook for defining customized connectors. Different forms of connectors can be specified: procedure call, event, stream, arbitrator, adaptor and distributed connectors.

A.7 Contracts [Helm90a]

In object oriented systems, groups of related objects cooperate to perform tasks or maintain invariants. In the contracts work a group of cooperating objects is called a *behavioral composition*. Contracts are constructs for the explicit specification of behavioral compositions.

Contract Specification

A *contract* defines the behavioral composition of a set of cooperating participants.A contract specifies: the participants in the behavioral composition and their contractual obligations, the invariants that participants cooperate to maintain and the preconditions on the participants to establish the contract and the methods which instantiate the contract. There are two types of contractual obligations: type and causal. The type obligations specify variables and external interface definitions that participants must support and the causal obligations specify ordered sequences of messages (actions) that participants must perform and certain conditions that participants should make true in response to these messages.

```
Contract SubjectView

  Subject supports [
    value: Value
    SetValue(val:Value) -> Δvalue {value = val}; Notify()
    GetValue(): Value -> return value
    Notify() -> <|| v: v ε Views: v -> Update()>
    AttachView(v:View) -> {v ε Views }
    DetachView(v:View) -> {v not ε Views }

  Views: Set(View) where each View supports [
    Update() -> Draw()
    Draw() -> Subject -> GetValue() {View reflects Subject.value }
    SetSubject(s:Subject) -> {Subject = s}]

  invariant
    Subject.SetValue(val) -> <∀v: v εViews: v reflects Subject.value>

  instantiation
    <||v: v ε Views: <Subject -> AttachView(v) || v->SetSubject(Subject)>>
end Contract
```

Figure A.17 : Contract SubjectView

In (**Figure A.17**), we show the specification of the SubjectView behavioral composition (Observer Pattern) introduced in [Helm90a]. In the SubjectView behavioral composition a Subject object, containing some data and a collection of View objects which represent that data graphically (i.e. as a dial, a histogram, etc.) cooperate so that all times each View always reflects the current value of the Subject.

Refinement and Inclusion

Two important operations on contracts are: refinement and inclusion. Refinement allows the specialization of contractual obligations and invariants in contracts. Contracts are refined by either specializing the type of a participant, extending its actions, or specifying a new invariant. Refinement is expressed in a contract by the refines statement. Inclusion allows contracts to be composed from simpler sub-contracts. The sub-contracts are denoted by the include statement which identifies a subset of a contract's participants and how they participate in the sub-contract. Participation in the sub-contracts impose additional obligation on participants over and above those defined in the contract.

Conformance

Contracts are defined independently of classes of their participants, a class conforms to a participant's definition in a contract only if its methods and instance variables satisfy both the typing and causal obligations required in the participant's definition. Class implementations must be mapped to participant specifications. A conformance declaration specifies how a class, and thus its instances, support the role of a participant in a contract. A conformance declaration contains a set of bindings of the form a: σ <- b: τ, which maps identifier b of type τ defined in a class, to the identifier a of type σ in a participant.

Instantiation

Behavioral compositions are created by instantiating contracts. They require identifying objects as participants in the desired contract, and then establishing the contract via the methods specified in the contract's instantiation statement. Typically the instantiation statement ensures that objects have references to other participants and that the initial conditions required for the contract are valid.

A.8 Collaborations [Yell97a]

The collaborations work assumes a world in which systems are composed of software components interacting with other components via typed interfaces. Each component exposes one or more interfaces through which messages are sent to and received from a potential collaboration mate component. When an interface of component A is bound to and interface of component B, A and B are said to engage in a collaboration: messages sent through A's interface are received at B's interface and vice versa. Each interface has a type, this type is associated with what they call a *collaboration specification:* an enhanced interface specification defining the rules governing message exchange. A collaboration specification (only collaboration in the following) consists of two parts: the *interface signature* and the *protocol*.

```
Collaboration Filter {
  Receives Messages {
    itemToBeFiltered(dataItem: ObjectRef);
    noMoreItems();}
  Send Messages {
    newFilterRequest();
    ok();
    remove();}
  Protocol {
    States {Stable(init), Filter, Respond};
    Transitions {
      Stable: -newFilterRequest-> Filter;
      Filter: +itemToBeFiltered-> Respond;
      Filter: +moreItems    -> Stable;
      Respond: -ok-> Filter;
      Respond: -remove-> Filter;}}
```

Figure A.18 : A Filter's collaboration specification

- The interface signature: describes the set of messages that can be exchanged between the component and its mate. Besides indicating the type of its parameters, each message in a collaboration specification is labeled as a send or a receive message.
- The protocol: describes a set of sequencing constraints. Sequencing constraints define legal orderings of messages by means of a finite state grammar. The finite state grammar is specified by means of a set of named states and a set of transactions. There is one transaction for each message that can be sent or received from a particular state. A transition has the form <state>: <direction> <message> -> <state>, where direction is + (indicating that the message is a receive message) or - (indicating that the message is a send message). Every protocol has a unique initial state init that corresponds to the initial state when the collaboration is established. It is not possible to have two transitions associated with the same state having the same label (i.e. if s1: +M1 -> s2 and s1: +M2 -> s3 are transitions, then M1 must be different to M2. The same is true if the signs + are reversed to - signs in the transitions). A protocol may have final states with no outgoing transitions, or it may be nonterminating.

A component can expose multiple interfaces, allowing it to simultaneously engage in multiple collaborations with multiple components. However any collaboration is always between exactly two components. A collaboration between a component C and multiple other components for example, must be modelled by separate interfaces in C, one interface for each other party which it collaborates with. In (**Figure A.18**), we show a Filter collaboration introduced in [Yell97a] and describing how a filter component interacts with a data server component in the context of a Global Desktop graphical environment. The Receives and Send in the collaboration define that the Filter component sends messages item ToBeFiltered and noMoreItems and receives messages newFilterRequest and ok. The protocol in the collaboration defines 3 different states: Stable, Filter and Respond. In the Stable state for example the collaboration defines that when the Filter component finds in the state Stable, if the component receives the message newFilterRequest the Filter component changes to state Filter in the protocol.

Protocol Semantics

When two components collaborate with each other via interfaces, each component receives and sends messages according to protocol defined in the collaboration. There are two possible semantics one can assign to collaborating components: an asynchronous semantics and a synchronous semantics. Under the asynchronous semantics, a component may send a message m whenever it is in state that enables a send m transition, even if its mate is not in a state that enables it to receive that message. Under the synchronous semantics, a component can only send a message to its mate if the component is in a state that enables it to send them message and if its mate is in a state that enables it to received the message.Collaborations follow a synchronous semantics.

Protocol Compatibility

The collaborations work defines additionally an algorithm to test protocol compatibility. The idea behind a protocol compatibility algorithm is to be able to determine whether two components can collaborate or not on the basis of a collaboration specification. Protocol compatible components are guaranteed to work together free of protocol errors (e.g. messages arriving out of sequence or deadlock).

Adaptors

```
<s1>: + <message> from <component_name> -> <s2>
[, <save_actions> ]
[, <invalidate_actions> ];

<s1>: - <message> to <component_name> -> <s2>
[, <synthesis_actions> ]
[, <invalidate_actions> ];
```

Figure A.19 : Adaptor's transition rules

When two components that have incompatible collaboration specifications wants to collaborate, adaptors need to be specified between the two components. An adaptor is a piece of code that sits between the two components and which compensates their interfaces incompatibilities. An adaptor is specified by a finite-state machine with interfaces to the two collaborating components. In (**Figure A.19**), we can see how an adaptor for t. The semantics of the first rule is as follows: when the component finds in the state <s1> the adaptor can receive (i.e. +) a message of type <message> from <component_name> and then will advance to state <s2>. The semantics of the second rule is as follows: when in state <s1> the adaptor can send (i.e. -) a message of type <message> to <component_name> and will then advance to state <s2>.

A.9 Coordination Contracts [Andr99a][Barr02a]

A coordination contract specifies the interaction between objects based on the separation between structure, (what is stable) and interaction (what is changeable). A coordination contract superposes a behavior over the direct interaction of its partners by intercepting their interaction. The interaction is expressed as rules of the form: when <event> do <reaction> with <guard>. An event is typically a method invocation and the reaction specifies a set of operations of the contract and its partners that take place as long as the guard is true. The whole interaction is handled as an atomic transaction.

```
1.contract <name>
2.  partners <list-of-partners>
3.  invariant <the relation between the partners>
4.  constants ..
5.  attributes ..
6.  operations ..
7.  coordination <interaction-with-partners>
8.  behavior    // the contract's own behavior
9.       <additional behavior being superposed>
10.end contract
11.
12.<interaction-with-partner>
13.      <name> : when <condition> do <set of actions> with <condition>
```

Figure A.20 Coordination Contract specification.

In (**Figure A.20**) we show the form of a Coordination Contract.The condition under when established the trigger of the interaction. Typically the condition is related with the occurrence of actions in the partners. The do clause identifies the reactions to be performed, usually in term of actions of the partners and some of the contract's own actions. The reactions of the partners constitute what is called the synchronization set associated with the interaction. In (**Figure A.21**) we show an example of the specification of a coordination contract of a VIP account in a bank, the coordination specifies the relation between the owner y of the account and the account x.

```
1.contract VIP package
2.  partners x: Account; y: Customer;
3.  constants CONST_VIP_BALANCE: Integer
4.  attributes Credit: Integer;
5.  invariants
6.      ?owns(x,y) = TRUE;
7.      x.AverageBalance() >= CONST_VIP_BALANCE;
8.  coordination
9.      vp:  when y.calls(x.withdrawal(z)) do x.withdrawal(z)
10.          with x.Balance() + Credit() > z;
11.end contract
```

Figure A.21 VIP account package coordination contract.

A.10 Coordination Environments [Mukh95a]

Coordination Environments (CEs in the following) specify non-intrusive coordinators that impose collaborative behavior on a set of objects called autonomous objects. Coordination in the CEs model is enforced by Coordinating Environment objects (CE objects in the following) that are instances of Coordination En-

vironment classes (CE classes in the following). CE objects coordinate collections of autonomous objects that compete and cooperate to achieve a common task or goal.

Autonomous Objects

An autonomous object has a public interface (PI in the following) that is visible to the client and a current public interface (CPI in the following) that is not visible to the client. The PI is determined by the name, return type and types of the arguments of the public methods of an autonomous object. The CPI is a subset of the PI that stores only representations of the method names that the object may execute in its current local state. A client requests service from an object group by explicitly communicating with one or more of the autonomous objects in the group. A client uses a two step procedure to invoke a public method of an autonomous object. The first step is to construct a special object called a request message that stores a representation of the method being invoked and a list of the actual argument values. In the second step, the client object synchronously invokes a special method of the server and supplies the request message as an argument to that method.

Request messages are managed by an autonomous object's Request Handler (RH). The RH uses the CPI interface of the autonomous object to determine whether to execute the request message. Only if the message appears in the CPI the message the message is accepted and executed. Because autonomous objects do not buffer request messages that cannot be processed immediately they defines two models of sending request messages: 1) only-once request where a request message may be sent only once and control returns to the client if the message is not accepted. The client decides what to do when the request fails. 2) until-accepted request where the request messages is sent repeatedly until it is accepted by a sever object.

CE Objects

CE objects coordinate collections of autonomous objects. The CE objects use special methods called Coordinating behavior methods (CBs methods) that implement and structure coordination actions. The coordination actions of a CE are triggered by the occurrence of events related both with the acceptance of a request message and the termination of a method that was scheduled by the CE object. A CE object is informed of the occurrences of acceptance and termination events by a RH using event messages. Event messages are observed by a CE object using event objects. There are two types of event objects (EOs: elementary (EEO) and composite (CEO) events. EEO events objects can be used to observe acceptance and termination of events related to a method in only one component while CEO can be used to observer acceptance and termination events related to a method in more than one component.

On observing an event, a CE object may take one or more of the following actions: update its local variables, schedule the accepted message for execution and either continue immediately or wait for the termination of a method, block or unblock request messages, add default argument values to the argument list in the request message, synchronously and asynchronously invoke methods in coordinated objects, specify a replacement CB method, specify which coordinated object to observe the next event in, or any of the two as the next event and inquire whether the last event observed was an acceptance or a termination. A CE object may also mark one or more acceptance events as unobserved. This action enables CE object ignore those acceptance events that do not play any role in its coordinating activities.

In (**Figure A.22**), we show the CE class MultiButtonPanel defined in [Mukh95a]. The class CoordinationEnvironment implements CE objects, the class ElementaryEvent implements EEOs, the class GroupComponent implements autonomous objects that participate in the object groups, the class Event

implements the common behavior of both EEOs and CEOs and the underlined names specify the coordination actions taken by a coordination behavior.

```
class MultiButtonPanel: public CoordinatingEnvironment {
public:
  MultiButtonPanel (ElementaryEvent* e1, ElementaryEvent* e2) {
    depressedButton=Null;
    depressedButton=e1; undepressButton=e2;};
  virtual void Initiate(){
    Become(&MultiButtonPanel::NoneDepressed);}

  protected:
    GroupComponent* depressedButton;
    ElementaryEvent* depressButton, undepressButton;
  virtual void UndepressButtonCA()=0;
  virtual void ReplacementCB2() {
    Become(&MultiButtonPanel::OneDepressed);}
  virtual Event* ObserveEvent1() {
    return Observe(depressButton)}
  virtual Event* ObserveEvent2() {
    return Observe(depressButton, undepressButton);};

  void NoneDepressed() {
    Event* whichEvent=ObserveEvent1();
    depressedButton=whichEvent->WhichComponent();
    whichEvent->AwaitTermination(THISCE);
    ReplacementCB2();};
  ...
```

Figure A.22 : MultiButtonPanel Coordination Environment

A.11 Coordination Policies [Mins97a]

This coordination model defines a coordination policy P as a triple < M, G, L>.

- M is the set of messages representing primitive operations of the activity to be coordinated (also called P-messages).
- G is a distributed group of agents. They are permitted to send and receive P-messages. They are the participants in the policy P.
- L is the set of rules regulating the exchange of P-messages between members of the group G (also called the law of the policy).

The Law of a Policy

A law L of a policy P determines the treatment of P-messages by specifying what should be done when a such message is sent and when it arrives. More specifically, the law deals with the following two kinds of events that are regulated

sent(x,m,y): occurs when an agent x sends a P-message m addressed to y. If the destination keyword is all m is multicasted to all members of the group. The sender is considered the home of this event.

arrived(x,m,y): occurs when a p-message m sent by x arrives at y. The receiver y is considered the home of the event. The receiver is considered the home of this event.

A law L is a pair <R, CS> where R is a fixed set of rules defined for the entire group G of the policy in question and CS is a mutable set of control states, one per member of the group. The effect of actually any given event is prescribed by the law L of the policy. The prescription of a law consists of a sequence of primitive operations carried out as the immediate response to the occurrence of the event. The operations that can be included in the ruling of the law for a given regulated event are called primitive operations. They are primitive in the sense that they can be performed only if thus authorized by the law. These operations include:

1) operations that change the CS of the home agent (i.e. +t, -t, t1<-t2, incr(t(v),x))

2) the operation forward(m,y,x) emits the network the message m addressed to y, where x identifies the sender of the message.

3) the operation deliver(m) delivers the message m to the home agent. R defines the global set of rules that compose the law L. The function of R is to evaluate a ruling for any possible regulated event that occurs at an agent with a given control-state.

Policies Enforcement

The law for a given policy P = <M, G, L> is enforced as follows: there is a controller associated with each member of group G, logically placed between the agent and the communications medium. All controllers have identical copies of the global set of rules R of L and each controller maintains the control states of the agents under its jurisdiction. When x sends a message m to its assigned controller. The controller evaluates the ruling of the law L for the event sent(x,m,y) and its carries out this ruling. If part of the ruling is to forward a message m to y, x's controllers sends m to the controller assigned to y. When m arrives to the controller of y it generates an arrived(x,m,y) event. The ruling for this event is computed and carried out. The message m is delivered to y if so required by the ruling.

Members Admission

The admission of new members into G and the remotion of existing members from it, is done by a secretary server who acts as a name server for members of its group.

Obligations

An obligation imposed on a given agent serves as a kind of motive force that ensures that a certain action will be carried out at this agent at a specified time in the future provided that certain conditions on the control state of the agent is satisfied at that time. The primitive operation +obligation(p,dt) carried out at agent x would cause the event obligationDue(p) to occur at x in dt seconds. The occurrence of the event obligationDue(p) at x, prompts the controller of x to evaluate the ruling of the law for this event.

```
R1: sent(X,startVote(issue(I),end(ET)),all) :-
  not(voteInProgress@CS),do(+yesVotes(0)),do(+noVotes(0)),
  (do(+voteInProgress)),do(+obligation(sendResults,ET +100)),
  do(forward).

R2: arrived(X,startVote(issue(I),end(ET)),Y) :-
  do(+vote(init(X),end(ET))),do(deliver).

R3: sent(Y,castVote(Val),X) :-
  vote(init(X),end(ET))@CS,clock(T)@CS,T < ET,
  do(-vote(init(X),end(ET))),do(forward).

R4: arrived(Y,castVote(yes),X) :- yesVotes(N)@CS,do(incr(yesVotes(N),1)).

R5: arrived(Y,castVote(no),X) :- noVotes(N)@CS,do(incr(noVotes(N),1)).

R6: obligationDue(sendResults) :-
       yesVotes(N1)@CS,noVotes(N2)@CS,
       do(-yesVotes(N1)),do(-noVotes(N2)),do(-voteInProgress),
       do(forward(results(yesVotes(N1),noVotes(N2)),all)).

R7: arrived(_,results(yesVotes(_),noVotes()),_) :- do(deliver).
```

Figure A.23 : Law L for electronic voting policy

In (**Figure A.23**), we show the electronic vote example introduced in [Mins97a]. Under this law every agent in the group can initiate a vote on any issue he chooses, by sending the message startVote to all members of the group. Agents vote by sending castVote messages to the initiator. The law ensures the following requirements: 1) a member can vote at most once and only within the period allotted for the vote; 2) the counting of the votes is done correctly; 3) the vote is secret, 4) agents are notified of the result of the vote.

A.12 Coordination Types [Puti97a]

In the object-oriented paradigm types specify contracts between objects and their users. Strong typing ensures that the violations of type constraints (type errors) cannot occur during program execution. The problem is that not all the constraints can be checked statistically, sometimes the constraints depend on the object's current state and history. This work proposes a type model for object-oriented systems based on a process calculus. Some actions in the calculus are annotated with type information. A type specifies all possible sequences of messages accepted by an object as well as type constraints on the messages's parameters. A type checker ensures statically that users of an object are coordinated so that only messages specified by the object's type are sent to the object in an expected order. In (**Figure A.24**), we show the syntax of the process calculus. A process specifies the behavior of an object. There are three atomic actions for sending messages, accepting messages and creating new objects. Semicolons separate actions from the process which should be executed after the actions. A message consists of a constant name c (message selector) and a list of arguments a1,...,aN (a).

```
0::= 0 (zero process; no action)
  | x.c[a];q      (send message c with arguments a to x; then execute q)
  | c(x); q       (accept message c with parameters x of types ;then q)
  | (x)$a[][a];   (create new object x that executes a[][a´]; then )
  | a=a´? q.q     (execute if a=a´; otherwise executes q´)
  | q+ q'         (alternatives; execute either orq´)
  | a[][a']       (call a with type arguments and arguments a´)

a ::= x           (parameter or object identifier)
  | (s)(x)q       (closed process; does not contain free variable names)
```

Figure A.24 : Processes Syntax

Static Checking

Static checking is divided into two parts: 1) to check whether objects (behaving as servers) are actually able to accept all messages as promised by the object's type and 2) to check whether objects (behaving as users) send only type-conforming messages.

A.13 Darwin - Ports [Mage95a]

Darwin is a configuration language that allows distributed programs to be constructed from specifications of components instances and their interconnections. Components are defined in terms of both the services they provide to allow other components to interact with them and the services they require in order to interact with other components. Composite components are defined by declaring both the instances of other components they contain and the bindings between those components. The bindings associate the services required by one component with the services provided by others. The bindings are only made between required an provided services with compatible types.

In (**Figure A.25**), we show the specification of a variable length pipeline of filters instances in which the output of each instance is bound to its predecessor's output. The binding between the required input of a filter and the provided output of the precedent filter are declared by the bind statement. In the example the input of each filter component instance F[k+1] is bound to the output of its predecessor filter F[k] by the statement bind F[k+1].input -- F[k].

```
component filter {
  provide output <stream char>;
  require input <stream char>;
}

component pipeline(int n) {
  provide output;
  require input;

  array F[n]: filter;
  forAll k:0 .. n-1 {
    inst F[k]@ k + 1;
    when k < n-1;
      bind F[k+1].input -- F[k].output;
  }
  bind
    f[0].input -- input;
    output -- F[n-1].output;
}
```

Figure A.25 : Specification of a pipeline component

A.14 Event Notifications [Papa94a][Papa96a][Hern96a]

The coordination model introduced in this work is based on the ability to synchronize the activity of an object with a number of events occurring in the execution of other objects. The event notifications model was introduced in the concurrent object-oriented programming language called ATOM [Papa96a].

The event notifications model associates to each object an object-manager that monitors its execution and ensures local synchronization constraints. The object-manager is triggered by events occurring in the execution of the object (internal events) such as the termination of a thread executing a method and external events such as the request for a method execution. The object-manager can undertakes actions like resuming a suspended thread and requesting or queuing a request until the object reaches an appropriate state depending on the local synchronization constraints. Another function of the object-manager is to accept requests from other object-managers allowing a synchronized execution of their respective objects.

Information about an object's state is available to managers and other objects via *state predicates*. A state predicate is used to determine whether or not the object is in a state that satisfies some condition abstracted in the predicate. An object may request for example to get notified when a remote object reaches a state satisfying a state predicate. There are two ways of coordinating the execution of an object with state changes of another object:

- Asynchronous notification of state changes: used to get notified when a target object has reached a state satisfying a given state predicate. To request for a notification of a state change an object must define first an instance of a synchronization event. The event instance is used to synchronize the execution of the object that holds the event and a target object. The object that makes the request for notification specifies additionally whether the object must be suspended or blocked until the occur-

rence of the event in the target object. When the object is suspended only the calling thread is suspended, the object may continue to accept new requests during that time. The thread will resume when the target object will reach the object state specified in the synchronization event. When the object is blocked, no more requests are accepted. In this approach when the thread is resumed, all that may be asserted is that the target object has been in a state satisfying the state predicate associated with the notification. Nothing can be said on whether this is still true.

- Synchronous notification of state changes: used to indicate that a target object has reached a certain state. In this approach the calling thread is suspended until the target object is in a state satisfying a predefined object state. The synchronization mechanism locks the target object when the state is reached. This mechanism ensures that when the thread is resumed, the object is still in the requested state. Requests to the locked object are delayed until the object is unlocked.

It is also possible to get notified and synchronized with events other than changes in the objects state. An object can be notified of the invocation, execution and completion of methods in other objects. The notification may also be synchronous or asynchronous. The target object register with the source object its notification requirement. It specifies the selector of the method invocation, the event type (i.e.invocation, execution and completion), the type synchronization and the method that should be called by the target object in the object when the event occurs.

In (**Figure A.26**), we show an example introduced in [Papa96a]. The example concerns a producer object which produces data packages and stores them in a buffer object. A consumer object retrieves data packages from the buffer and consumes them. The purpose of the rateController coordinator is to ensure, by modifying the rate of the producer, that the buffer will never get empty or full. After initialization, a new thread is created to execute the monitor method of the rateControl object (**Figure A.26** line 2 in the rateControl class). This method loops waiting for notification events. In (**Figure A.26** lines 8 and 9 in the rateControl class) the calls to the notifyRequest methods request the buffer's object manager to notify the rateControl object when the buffer is at the abstract states ('contains', self.low) and ('contains', self.high) respectively.

```
1.class Consumer(Activity)
2.  def _init_(self,ch):
3.    self.c = ch
4.
5.  def stepaction(self):
6.    data = self.c.get().
7.    self.consume(data)
```

```
1.class Producer(Activity)
2.  methods=['changeRate,'getRate']
3.  s= 0; rate = 2; c = None
4.
5.  def _init_(self,ch):
6.    self.c = ch
7.
8.  def stepaction(self):
9.    for i in range(1,self.rate):
10.     data = self.produce()
11.     self.c.put(data)
12.
13. def changeRate(self,r):
14.   self.rate = r
15.
16. def getRate(self):
17.   return self.rate
```

```
1.class rateControl(ActiveObjectSupport):
2.  activities = ['monitor']
3.
4.  def _init_(self,p,c,b,h,l):
5.   self.prod = p; self.cons = c; self.buf=b
6.   self.high = h; self.low = l
7.   self.buf.newPred(contentsPred,['containts']())
8.   self.ishi = self.buf.notifyRequest(('contains',self.high))
9.   self.islo = self.buf.notifyRequest(('contains', self.low))
10.
11. def monitor(self):
12.   while not self.atState((stopped)):
13.   r = self.waitComplexEvent(
14.       'Any', ({'hi':self.ishi,'low':self.islo}))
15.   if r = 'hi' #decrease rate
16.     self.prod.changeRate(int(self.prod.getRate()/2))
17.   else   # increase rate
18.     self.prod.changeRate(int(self.prod.getRate()*2))
```

Figure A.26 : Flow Control Example

A.15 Finnesse - Bindings [Berr98a]

Finnesse is a coordination model and language based on an abstraction called binding. an abstract entity that encapsulates communication between distributed software components participating in an application. Bindings are described in terms of the following concepts:

- binding: describes a configuration of components and their allowed or expected interactions.

- role: a binding has a set of roles that can or must be filled by participating components. One or more components can fulfill a single role. A role definition can be prefixed by a cardinality constraint.
- interface: components have interfaces through which they interact with their environment. Each interface is connected to one or more roles in the binding and must implement the behavior specified by the role it fills.
- events: components participate in a binding by executing events at their interfaces. Events have parameters and direction (in or out).
- event relationships: specify the behavior and interactions of a binding by describing the relationships between events occurring at object interfaces.

In (**Figure A.27**), we show a binding with two roles: client and server as introduced in [Berr98a]. The interactions specifications defines relationship between events occurring in the different roles. Events are referred to by the role name followed by a period '.' and the event name. In the example line 8, the client executes a send event followed by all servers executing the receive event. The symbol '#' in the interactions represents the number of components executing the event. When no cardinality constraint is given, the default cardinality is exactly one. The binding described in reality a reliable multicast.

```
1.Binding Example {
2.  Import ...;
3.  Roles {
4.    Client {send! }
5.    [#>=1] Server { receive? }
6.  }
7.  Interactions {
8.    Client.send -> [#=all] Server.receive
9.
10.}
```

Figure A.27 : Binding describing a reliable multicast

Events are specified by a name, a direction indicator and a parameter list. For example: e! (x: t1; y: t2) defines an event named e, x and y are the event parameters and t1 and t3 are the data type of the parameters. The direction can be '!' to indicate an output event and '?' to indicate an input event.

Events relationships provide the basis for describing behavior in bindings. They capture the relation between events at the interfaces of components participating in a distributed application. There are three type of event relationships:

- Casual relationships: which describes casual dependencies between events.
  e1 ! (x: t1) -> e2 ? (y: t2): specifies that event e1 must complete before event e2 begins.
- Parameter relationships: which describe the relation between parameters of casually related events.
  e1 ! (x:t1; y: t2) -> e2 ? (z: t3) { z = f (e1.x) }: specifies that parameter z in event e2 is a function of parameter x in event e1.
- Timing relationships: which describe any time relationships between events.
  e1 ! () -> [now - end(e1) < 10] e2 ? (): specifies a guard for an event based on a timing constraint.

Inheritance and Subtyping

Finnesse supports inheritance (keyword inherits) and explicit specification of subtype relationships (keyword implements). The inherits keyword instructs Finesse to include the roles and interactions of the parent binding into the child binding. Definitions in child bindings with same named roles and actions that in parent bindings override parent definitions. The implements keyword in the other hand is intended to allow specific implementation of high-level behaviors. High-level bindings can be replaced for specific implementations of the binding.

Interaction Semantics

There are two separate interaction semantics in Finesse: one for dependent (sequential) iteration and one for independent (parallel) interaction. Both take the form of a postfix operator on an action or event. The *+ operator indicates that the action or event should be repeated with a casual dependency on previous executions (sequentially). The *- operator indicates that the actions or event should be repeated with no dependency on previous executions (in parallel). In (**Figure A.28**), we show the specification of a binding using the two interaction semantics. The binding specifies two roles: consumer and producer. The consumer can only consume one data item at a time, while the producer can produce many data items in parallel. The interactions describe that each produce event results in a consume event.

```
1.Binding Example {
2.  Roles {
3.    Consumer { consume?(x:t1) *+ }
4.    Producer { produce!(x:t1) *- }
5.  }
6.  Interactions {
7.    {Producer.produce -> Consumer.consume } *-
8.  }
9.}
```

Figure A.28 : Interaction Semantics

A.16 Formal Connectors [Alle94a]

This work provides a formal system for specifying architectural connector types. The architecture of a system is described in three parts. The first part of the description defines the component and connector types. A component type is described as a set of ports and a component-spec that specifies its function. Each port defines a logical point of interaction between the component and its environment. A connector type is defined by a set of roles and a glue specification. the roles describe the expected local behavior of each of the interacting parts. The second part of the system definition is a set of component and connector instances (actual entities that will appear in the configuration). In the third part of the system definition, component and connector instances are combined by prescribing which component ports are attached as which connector roles.

Connector Specification

```
1.connector Service =
2.  role Client = request!x -> result?y -> Client P
3.  role Server = invoke?x -> return!y -> Server []
4.  glue = Client.request?x- > Service.invoke!x
5.       ->Service.return?y -> Client.result!y -> glue
6.  []
```

Figure A.29 : Service Connector

A connector is described by specifying process descriptions for each of its roles and its glue. The process descriptions is specified using a subset of CSP[Hoar85a] (a process algebra). In (**Figure A.29**), we show the specification of a connector service introduced in [Alle94a]. The server role describes the communication behavior of the server. The server role is defined as a process that repeatedly accepts and invocation ant then returns; or it can terminate with success. The client role describes the communication behavior of the user of the service. The client role is defined as a process that can call the service and then receive the result repeatedly. The glue specification coordinates the behavior of the two roles by indicating how the events of the roles work together. The ([]) represents the alternative operator, P[] Q specifies a process that can behave as P or as Q. The (Π) represents a decision operator, PΠQ specifies a process that can behave non deterministically as P or as Q. The (->) represents the prefixing operator, e->P specifies a process that engages in event e and then becomes process P.

A.17 GAMMA - Multiset Rewriting [Bana96a]

The GAMMA (General Abstract Model for Multiset and mAnipulation) model is based on multiset rewriting. The basic data structure in GAMMA is a multiset (a bag) containing elements. A program in GAMMA is composed of pairs (reaction-condition -> action) and its execution implies the replacing of those elements in the multiset satisfying the reaction-condition by the products of the action. The result is obtained when no more such reactions can take place. In (**Figure A.30**), we show a generator of prime numbers in GAMMA. The program eliminates elements from the multiset {2, ..., N} those x's multiples of y's. The reaction condition R specifies the predicate x is multiple of y and the action A specifies the remove of the element x from the multiset.

```
prime_numbers(N) = T ((R,A))
       ({2,..N}) where
       R(x,y) = multiple(x,y)
       A(x,y) = {y}
```

Figure A.30 : Prime numbers in Gamma

A.18 Gluons [Pint95a]

Gluons are special kind of objects responsible for managing the cooperation among software components. They encapsulate and implement interaction protocols by instantiating an interplay relation for a given protocol.

A gluon is an object that handles a finite state automaton with output to control the execution of a protocol's interplay relation. The finite state automaton is composed of states and state transitions. A gluon contains a start state, any number of intermediate states and many end states. A state transition triggers the execution of an action which is composed of operations. State transitions are fired when the gluons receive messages.

There are three types of operations that compose an action in a gluon: messages sends, object assignments and message selector assignments. A message send action allows a gluon to send a message to a component requesting for a service, an object assignment action allows a gluon to keep a reference to software components and a message selector assignment action allows a gluon to keep a reference to message selectors.

Gluons have roles that store participants references to the software components that are "compatible" with the role. A typical example of roles are client and server roles in a client-server protocol. The compatibility refers to the fact that a component plays a role in the gluon.

Protocol Transitions			*Event/Action*
State	Transition	State	
	0	**Start**	Source:registerServer{server}
			Server := server
Start	**1**	**Start**	<any_obj:<message>
			MessSel := <message> <message> -> Server
Start	**2**	**End**	<any_obj>:exit
			gluonDisconnecting{self}->Server Server := none

Figure A.31 : Protocol transition table for a simple gluon

In (**Figure A.31**), we show a simple gluon that handles an interaction protocol between a server and a client. The protocol handles message forwarding. The association between the server and the gluon is requested by the server component by sending the message registerServer to the gluon. This message triggers state transition 0 which initiates the gluon's protocol. Any client component can then send messages to the gluon and these messages are forwarded to the server with transition 1. Finally, the Gluon can be disconnected from the server by sending to it the message exit.

A.19 Linda - Tuple Spaces [Gele85a][Carr94a]

Linda is coordination model based on the so-called generative communication paradigm. In a generative communication paradigm processes communicate by exchanging data (passive tuples) through a shared dataspace (known as tuple space). The generative communication paradigm decouples processes in both space and time: no process need to know the identity of the other processes, nor is it required all the processed to be alive at the same time. In addition to the passive tuples containing data, the tuple space can also

contain active tuples representing processes which after the completion of their execution, transform into passive tuples.

Linda is composed of a set coordination primitives on the tuple space: in, rd and eval. The primitive out(t) is used to put a passive tuple t in the tuple space, the primitive in(t) retrieves a passive tuple t from the tuple space, the primitive rd(t) retrieves a copy of t from the tuple space (the tuple t retrieved is not removed from the tuple space) and the primitive the eval(p) puts an active tuple (i.e., a process) in the tuple space. The primitives rd and in are blocking primitives and will suspend execution until the desired tuple is found. The primitives out and eval are non-blocking primitives. A process that executes eval(p) will execute in parallel with p, which will turn into a passive tuple when it completes execution. Additional primitives were introduced into the basic model: rdp(t) and inp(t) are not blocking variants of rd(t) and in(t) respectively.

The tuples are sequences of typed fields. They are retrieved from the tuple space by means of pattern matching mechanism. The matching of a tuple t with an actual tuple ta in the tuple space will succeed provided that the number, position and types of the t's fields match those of ta.

```
1.phil(i)
2.  int i;
3.{ while(1) {
4.    think();
5.    in('room ticket');
6.    in('chopstick', i);
7.    in('chopstick', (i+1)%Num);
8.    eat();
9.    out('chopstick', i);
10.   out('chopstick', (i+1)%Num);
11.   out('room ticket');
12.   }
13.}
```

```
1.initialize()
2.{
3.    int i;
4.    for (i=0; i<Num; i++) {
5.      out('chopstick', i);
6.      eval(phil(i));
7.      if ( i<(Num-1))
8.        out('room ticket');
9.    }
```

Figure A.32 : Dinning Philosophers in Linda

In (**Figure A.32**), we show the implementation of the classical problem dinner philosophers in Linda. In the dinner philosophers a group of five philosophers sat around a table try to eat at the same time. Between each pair of table positions there is a single chopstick (i.e., there are five chopsticks in total for the five philosophers). To eat, each philosopher must have two chopsticks, they can only use the two chopsticks on either side of them.

There has been a lot of works done on Linda extensions. We will refer to some of the most important here:

- Bauhaus Linda[Carr94a]: is a direct extension of the Linda model featuring multiple tuple spaces in the form of multiset (msets). Instead of adding tuples to and reading or removing tuples from a single flat tuple space, Bauhaus Linda's out, rd and in operations add multisets to and read or remove multisets from another multiset.
- Bonita [Rows97a]: includes a new set of primitives that provide asynchronous access to the tuple spaces. The new primitives are:

rqid=dispatch(ts,tuple,[template,destructive|nondestructive]): non-blocking primitive which controls all the access to a tuple space. If a tuple is specified then this tuple is placed in the tuple space. If a template is specified this indicates that the tuple is to be retrieved from the specified tuple space. If this is the case and extra field is used to indicate if the tuple retrieved should be removed (destructive) or not removed (nondestructive).

rqid=dispatch_bulk(ts1,ts2,template,destructive|nondestructive): non-blocking primitive which controls the movement of tuples between tuple spaces. The source tuple space is ts2 and the destination tuple space is ts2 and the tuples are either moved (destructive) or copied (nondestructive).

arrived(rqid): non-blocking primitive that detects if a tuple or result associated with a rqid is availble. The primitive either returns true or false.

obtain(rqid): blocking primitive which waits for the tuple or result with rqid to arrive.

- Law-Governed Linda [Mins94a]: extends the Linda model with rules to control events occurring during the interaction of each process with the tuple space. Three classes of events are controlled: invocation events (occur when a process invokes one of the Linda operations out, rd or in), selection events (occur when the template of a linda in or rd operation invoked by a process is matched with some tuple in the tuple space) and asynchronous events (occur asynchronously with respect to the processes). The primitive operations that can be included in a ruling are:

  complete: execute the operation being invoked.

  complete(arg'): like complete, except that the original argument of the operation is replaced with arg'.

  return: delivers a selected tuple to a process.

  return(t'): like return, except that the tuple t' is delivered to the process instead of the matched tuple t.

  out(T): out operation in Linda.

  remove: remove a process from the system.

- Objective Linda [Kiel96a]: introduces a coordination model that adapts the Linda model to object orientation. The objects in the model are instances of abstract data types with are described in a language-independent language called Object Interchange Language (OIL). Object matching is based on object types and the predicates defined by type interfaces. The operation on the object space are:

  out(m:MULTISET;timeout: REAL):BOOLEAN: tries to move the objects contained in m into the object space. Return true if the operation can be done, false otherwise.

  in(o:OIL_OBJECT;min,max:INTEGER;timeout:REAL):MULTISET: tries to remove multiple objects matching o1,..,on matching the template object o from the object space returns a multiset containing at least min at most max objects.

  eval(m:MULTISET;timeout:REAL):BOOLEAN: tries to move the objects contained in m into the object space and starts their activities. Returns true if the operation could be completed successfully, false if not or if the timeout fixed expires.

  infinite_matches:INTEGER: constant that will be interpreted as an infinite number of matches when provided as min or max.

  infinite_time:REAL: constant that will be interpreted as an infinite delay when provided as timeout.

- JavaSpaces [Sun03a]: A JavaSpace service is the equivalent of a tuple space in the Linda model. A JavaSpace service contains entries. An entry is typed group of objects expressed in a class for the

Java platform that implements the interface net.jini.core.entry.Entry. There are four primitives that can be invoked in a JavaSpace service.

write: writes a given entry into the JavaSpace service.

read: reads an entry from a JavaSpace service that matches a given template.

take: reads an entry from a JavaSpace service that matches a given template, removing it from the JavaSpace service.

notify: notify an object when entries that match a given template are written in the JavaSpace service.

A.20 Manifold - IWIM [Arba96a][Arba98a]

Manifold is a coordination language based on the IWIM (Idealized Worker Idealized Manager) model. The basic concepts in the IWIM model are processes, events, ports and channels.

A process is a black box with well defined connection ports used to exchange units of information with other processes. The exchange of information is done in only one direction: either into (input port) or out (output port). Ports have names associated with them, p.i for example refers to the port i of the process instance p.

The interconnections between the ports of processes are made through channels. A channel connects a port in a producer process to another port in a consumer process, p.o -> q.i denotes a channel connecting the port o of the producer process p to the port i of the consumer process q.

Independently of the channels, the IWIM model proposes an event mechanism for information exchange. Events are broadcast by their source in their environment at the occurrence of certain events. Processes decide which events they want to react to. The event mechanism supports anonymous communication: a process does not and need not to know the identity of the processes which it exchanges information.

There are two types of process in IWIM: workers and a managers. The responsibility of a worker process is to perform a computational task. The worker is not responsible for obtaining the proper input it requires to perform its task, nor is it responsible for the delivering the results it produces. It is up to managers processes to coordinate the necessary communication among a set of worker processes.

There are two means of communication available to a worker process: via its ports and via events. The primitives that allow a process to exchange information through ports are: read and write. To communicate using events the worker must raise the events. The events are broadcast to all the processes in its environment.

Manager process can create new instances of processes and broadcast and react on event occurrences. It can also create and destroy channel connections between port of the process instances it knows, including itself. The manager process controls the communication among a number of processes instances. In (**Figure A.33**), we show an implementation of the classical problem of dinning philosophers in Manifold as introduced in [Arba98a]. Upon activation, a Fork instance enters an infinite loop (lines 25 to 29) waiting for a pair of event occurrences (line 11) and reacting to them (lines 12 to 15).

```
1.#define WAIT(preemptall, terminated(self))
2.
3.event request, done.
4.manner Eat(process, process, process) import.
5.manner Think(process) import.
6.manner GetTicket() import.
7.manner ReturnTicket() import.
8.
9.export Fork()
10.{begin: while true do {
11. begin: WAIT.
12. request.*phil & *ready.*phil: {
13.   save *.
14.   begin: (raise(ready), WAIT).
15.   done.phil:.
16.   }
17. }
18.}
19.
20.export Philosopher()
21.{
22.event ready.
23.begin: while true do {
24. begin: Think(self);
25.    GetTicket();
26.    (raise(request,ready), WAIT).
27. ready.*lfork & ready.*rfork: Eat(self, lfork, rfork).
28. end: raise(done);
29.   ReturnTicket().
30. }
31.}
```

Figure A.33 : Dinning Philosophers in Manifold

A.21 Piccola-Scripts [Ache00a]

Piccola is a small “composition language” designed to support software composition. The core abstractions of the Piccola model are forms (immutable, extensible records), agents (communicating processes) and channels (locations where agents asynchronously exchange forms). In top of the Piccola model forms are used to build higher-level abstractions to define composition and coordination styles. Piccola proposes an approach for composing and coordinating software components in which different high-level, algebraic coordination styles may be defined and agents script components according to these styles. The coordination styles are implemented as component algebras. A script, is an expression of the algebra that specifies how the components are plugged together.

Table 1: Stream Style Components (Provided-Required Services)

	Provided Services	*Required Services*
Source		put(X): write element downstream close(): signal end of stream
Filter	put(X): accept a data element close(): close the input stream	put(X): write element downstream close(): signal end of stream
Sink	put(X): accept a data element close(): close input stream	

To illustrate how Piccola can be used to specify coordination styles we will show as example a Push-Flow coordination style introduced in [Ache00a]. In this style an individual component pushes data downstream to another component to which it is connected. The style includes three kinds of components: a source (produces data and pushes it downstream), a filter (accepts pushed data, process it and pushes the result further downstream) and a sink (accepts pushed data and represents the end of the stream). In (**Table 1**), we show the provided and required services for the three different elements that compose the Push-Flow coordination style.

```
Source | Sink -> () : connect stream s to the sink
Source | Filter -> Source : manipulate stream s using filter
Filter | Filter -> Filter : compose two filters
Filter | Sink -> Sink: build new sink using filter
Source + Source -> Source: concatenate streams (sequential composition)
Source & Source -> Source: merge streams (parallel composition)
Sink + Sink-> Sink: multiplex a stream to two sinks
```

Figure A.34 : Push Stream Signature

In (**Figure A.34**), we show the set of composition rules (signature) of the stream style. The signature of the style specifies the correct bounds between the different streams components. The operators (i.e. |, + and &) in the stream style are specified using scripts. In (**Figure A.35**), we can see the specification of the | operator. The asSource abstraction specifies the binding between a source component S and the component Right appearing in the right side of the operator.

```
1.asSource(S).
2.  S
3.  _ |(Right): #define the | connector
4.    S.reqPut.bind(Right.put)
5.    S.reqClose.bind(Right.close)
6.    return asEmptyOrSource(Right)
```

Figure A.35 : The | Operator

In order to plug a source component mySource for example into a filter the asSource is applied to it. In (**Figure A.36**), we show how to a source component is plugged to a filter component. The coordination between the two components is performed as procedure calls.

```
s := asSource (mySource)
s | filter | ...
```

Figure A.36 : Source-Filter plugging

A.22 Rules and Constraints [Andr96a][Andr96b]

The coordination model introduced in this work is based on the use of rules and constraints, constructs that come from the tradition of declarative (rule-based) programming languages. A rule specifies the coordination steps needed to go from one global state to another. Constraints define restrictions over the domain of interpretation of the rule; they can be used for capturing restrictions over general coordination schemes. Two kinds of rules are specified in this model to perform coordination: reactive and pro-active rules.

Re-active rules

Re-active rules act upon pools of tokens of knowledge (the facts). Each rule specifies how to infer a set of tokens (the right hand side of the rule) from a set of already established tokens (the left hand side of the rule). In object oriented the tokens manipulated by the rules correspond to objects and method invocations (messages). The left hand side of rules synchronizes the execution of events corresponding to the modification of object states and to the triggering of messages. The right hand side of the rules expresses the notification of new events.

Each reactive rule acts as an autonomous, long lived thread of activity continuously looking for events to be synchronized. The computational model obtained is purely re-active. It applies quite naturally to the design of event managers and all sort of synchronizers.

Pro-Active rules

Proactive systems aim at influencing and modifying the environment, rather than simply re-acting to external stimulus. The idea behind pro-active rules is to switch from an extensional representation of the pool of tokens to an intentional one. A rule no longer just wait for the tokens on its left hand side to appear on the

pool, but it materialize the intentional description of the pool, so as to make it happen. The pro-active computational model is adapted to design real coordinators rather than simple synchronizers.

Object Coordination Schemes

An object coordination schema involves two kinds of entities: the coordinator and the participants (active objects). Rules are used to define the coordinator's behavior. The pool of tokens on which the rules apply is handled by the participants themselves. The tokens on the left hand side of such rules represent actions on the participants (method invocations). Thus, by accessing a token A, the coordinator issues the following request to the participants: perform an action capable of producing A. This request may be satisfiable in one or more ways, or may not be satisfiable at all. The fact of satisfying a token may change the internal state of the concerned participants. This change happens only when the rule is certain to apply, that is, when all the tokens in the left hand side of the rule are available (transactional reading of the tokens). A token a on the left hand side of a rule triggers a transaction dialogue between the coordinator and one of the participants, consisting of three phases: Inquiry, Reservation and Confirmation/Cancellation

- Inquiry: the coordinator inquiries whether the participant can produce the token A. The participants returns a set of possible actions that could perform to produce A.
- Reservation: the coordinator reserves from the participants a specific action from those identified during the inquiry phase, this action is then said to be engaged.
- Confirmation/Cancellation: the coordinator either confirms or cancels the action engaged during the reservation phase. If confirmation occurs, then the corresponding action is executed and the resources are modified.

```
1.transfer(Acct1, Amnt1, Acct2, Amnt2, Acct)@
1.
2.extract(Acct1, Amnt1)@ extract(Acct2, Amnt2)<>-insert(Acct, Amnt1+Amnt2)
3.
4.transfer-date(Date)@ out-of-date(Date)<>-timeout-procedure
```

Figure A.37 : Remote Banking

In (**Figure A.37**), we show a transfer rule (line 1) in a remote banking simulation introduced in [Andr96b]. The rule upon the reception of an bank order from the bank operator atomically transfers an amount from two accounts (Acct1 and Acct2) into a third account (Acct). The bank operator is viewed as bag of orders (events) and the Inquiry phase for the token tranfer(Acct1,Amnt1,Acct2,Amnt2,Acct) will retrieve each of them successively. It may happen that the order returned by the Inquiry is not processed, if the rule cannot grab the other tokens it requires. To discard such un-processed orders, after a certain time-out period a second rule is included (line 4).

A.23 Synchronizers [Frol93a]

Multi-object coordination patterns are expressed in the form of *multi-object constraints*. A multi-object constraint maintains certain properties such as temporal ordering and atomicity associated with message invocations processed by a group of objects. *Synchronizers* are special objects that specify multi-object constraints. A synchronizer observes and limits the message invocations accepted by a set of objects, whether

or not an object process a message invocation depends on the current status and invocation history of the group of constrained objects.

In (**Figure A.38**),we can see the abstract syntax for synchronizers. The structure of a synchronizer is specified using the {...} constructor. Each synchronizer has a name that allows its instantiation and a list of formal parameters that are bound to actual values when the synchronizer is instantiated (**Figure A.38** line 15). Each synchronizer has also an init part which declares the list of local names that hold the state of the synchronizer (**Figure A.38** line 16).

```
1.binding ::=name := exp |
2.       binding1; binding2
3.
4.pattern ::= object.name|
5.     object.name(name1, ..., nameN) |
6.     pattern1 or pattern2  |
7.     pattern exp
8.
9.relation ::= pattern updates binding |
10.    exp disables pattern  |
11.    atomic(pattern1, ..., patternN)|
12.    pattern stops|
13.    relation1, relation2
14.
15.synchronizer ::= name(name1, ..., nameN)
16.    { [ init binding ]
17.    relation }
```

Figure A.38 : Abstract Syntax for Synchronizers.

The specification of the synchronizers is done using pattern matching (**Figure A.38** line 4). The rules defining pattern matching are:

- The pattern o.n matches all messages invoking method n in object o.
- The pattern o.n(x1,.., xN) matches all messages matched by the pattern o.n and binds the actual values of the arguments of a matching message to the names x1,..., xN.
- The pattern p1 or p2 matches messages that match either p1 or p2.
- A message matches the pattern p where exp if the message matches the pattern p and the boolean expression exp evaluates to true.

The relation part of a synchronizer specifies the different multi-object constraints (**Figure A.38** line 9). There are four possible types of relations:

- A relation of the form `pattern updates binding` changes the state of the enclosing synchronizer according to binding each time an object is invoked by a message that matches pattern. In order to maintain consistency of synchronizers, bindings are established as atomic actions. The updates operator can be used to record the invocation history of the object (which invocations have been processed by the object and in which order).

- A relation of the form `exp disables pattern` prevents the acceptance of messages that match pattern if the expression exp is true in the current state of the synchronizer. A disables operator defines conditions that must be met before the object can be invoked by certain messages. Prevented invocations are delayed at the object if the conditions in the enforced multi-object constraints are not satisfied. Synchronizers that contain both updates and disables relations can enforce temporal ordering when the legality of the invocations is determined by the past invocation history.
- A relation of the form `atomic(pattern1, ..., patternN)` invokes i kinds of messages that match the patterns pattern1,.., patternN respectively. The relation ensures that the acceptance of a message from either kind occurs along with acceptance of messages from the other i-1 kinds without any observable middle states. Either all or none of the patterns are matched and there is no temporal ordering between the matching invocations. The atomic operator gives rise to indivisible scheduling of multiple invocations at multiple objects.
- An instantiated synchronizer remains in effect until observing an invocation that matches the pattern of a stops relation. The relation `pattern stops` implies the acceptance of a message matching the pattern and that terminates the synchronizer. A synchronizer without a stops operator remains in effect permanently.

```
1.VendingMachine (accepter, apples, bananas, apple_price, banana_price)
2.{ init amount := 0.
3.  amount < apple_price disables apples.open,
4.  amount < banana_price disables bananas.open,
5.  accepter.insert(v) updates amount := amount + v,
6.  (accepter.refund or apples.open or bananas.open) updates amount := 0 }
```

Figure A.39 : The Vending Machine

In (**Figure A.39**), we show a synchronizer specification introduced in [Frol93a] to coordinate the different parts of a fruits vending-machine. The vending machine has two slots: one for apples and one for bananas. The name apples refers to the apple slot and the name apple_price to the price of an apple (similar for the bananas). The name accepter refers to an object representing the coin accepter. The apple and the banana slots have an open operation that can be invoked if the accepter contains enough money. The variable amount holds the amount of money contained in coin accepter.

A.24 Wrappers [Ciob05a]

This work introduces a specification language where components are described as objects, coordination is defined as a process and their integration is given by wrappers. The semantic integration of the coordinating process and coordinated entities is based on bisimulation.

Classes and Objects

A class specification consists of specification of attributes and specification of operations. An operations specification includes the signature of the operation and its behavioral specification expressed in the terms of its parameters and attributes values before and after its execution. Objects are autonomous units of exe-

cution which are either executing the sequential code of exactly one method, or passively maintaining their states. An object instance is a pair (R | state), where R is an object reference and state is an ordered sequence of pairs (attribute, value). The result of the execution of a method R.m(d) over a state st consists of a new state st' whose attributes values are computed according to the behavioral specification of m. In other words st' = R.m(d)(st).

Coordination

A coordination process provides a high-level description of the interaction between objects. It syntax is inspired by process algebras as CSS and π-calculus[Miln99a]. Interaction with the environment is given by some global actions and interaction between components is given by a nondeterministic matching between complementary local actions. Coordination processes are described by a set of equations. The process expressions E are defined by guarded processes, non deterministic choice E1 + E2 and parallel composition E1|E2. There is also an empty process 0. In (**Figure A.40**) we can see the syntax grammar for the processes.

```
1.proc <proc_spec_name>
2.{
3.  global actions : <lact_list>;
4.  local actions: <gact_list>;
5.  process: <proc_id_list>;
6.  guards: <guard_id_list>;
7.  equations:
8.    <eqn_list>
9.}
10.
11.where
12.<lact_list> ::= <label_list>
13.<gact_list> ::= <label_list>
14.<label_list> ::= <label> |<label>,<label_list>
15.<label> ::= <identifier> | ~ <identifier>
16.<proc_id_list> ::= <id_list>
17.<guard_id_list> ::= <id_list>
18.<id_list> ::= <identifier> | <identifier>, <id_list>
19.<eqn_list> ::= <eqn> | <eqn>; <eqn_list>
20.<eqn> ::= <proc_id> = <pexpr>;
21.<pexpr> ::= 0 | <label>.<pexpr> | [<guard_id>]<pexpr> |
22.            [not <guard_id>]<pexpr> | <pexpr> + <pexpr> |
23.            <pexpr>|<pexpr>
```

Figure A.40 : Coordination processes syntax grammar.

In (**Figure A.41**) we show the specification of a coordination process introduced in [Ciob05a]. The coordination process corresponds to the specification of an Alternate Bit Protocol (ABP) communication protocol as a coordination between a Sender and a Receiver.

```
1.proc ABP
2.{
3.  global actions: in, out, alterS, alterR;
4.  local actions: ch1, ch2;
5.  processes: A, A', V, B, B', T;
6.  equations:
7.    A = in.A';
8.    A'= ~ch1.ch2.V;
9.    V = [sok] alterS.A + [not sok] A';
10.   B = [rok] B' + [not rok] out.alterR.B;
11.   B'= ~ch2.B;
12.}
```

Figure A.41 ABP Communication protocol as a Coordination process.

Interaction Wrapper

A coordination process can be considered as an abstract interface of the system, an interaction wrapper describes an implementation of this interface by means of a collection of objects. The coordinating process specifies coordination directives and the coordinated object interpret these directives using an interaction wrapper. The interaction wrappers provide the link between the high level coordination processes and the lower level executing objects. In (**Figure A.42**) we show the syntax of an interaction wrapper.

```
1.<wrap_spec> ::= <wrap_name> (<wparam_list>)
2.                implementing <proc_spec_name>
3.                {<amap_list> <gmap_list>}
4.<wparam_list> ::=<wparam> | <wparam_list>; <wparam>
5.<wparam> ::= <class_name> <object_ref>
6.<amap_list> ::= <amap> | <amap_list> <amap>
7.<amap> ::= <action_name> -> <cmd>;
8.<gmap_list> ::= <gmap> | <gmap_list> <gmap>
9.<gmap> ::= <guard_name> -> <bexpr>;
```

Figure A.42 Wrapper Syntax Grammar

In (**Figure A.43**) we show the wrapper's specification for the previous described ABP protocol. The wrapper instructs a Sender S and a Receiver R in order to correctly follow the directives of the protocol. In line 3, we can see how a directive in received from the coordination process is translated into an execution of method read by S. The directives alterS and alterR (lines 4 and 5) are translated into executions of methods chBit and chAck by and R respectively. In line 5 we can see that whenever the directive tau(ch1) is possible in the coordination process, the directive is translated into a synchronization of the methods sendFrame of S and recFrame of R. The synchronization is accompanied by a communication between them. Similar for the directive tau(ch2). Finally the last two lines, corresponds to the comparison of the sending bit and the received acknowledge done at the object level.

```
1.wrapper w (Sender S, Receiver R) implementing ABP
2.{
3.  in -> S.read();
4.  alterS -> S.chBit();
5.  alterR -> S.chAck();
6.  tau(ch1) ->
7.    R.recFrame(S.data, S.bit) ||
8.    S.sendFrame();
9.  tau(ch2)
10.   S.recAck(R.ack()) || R.sendAck();
11.out -> R.write();
12.sok -> S.bit == S.ack;
13.rok -> S.bit =/= R.ack;
14.}
```

Figure A.43 Protocol ABP wrapper specification

Temporal Properties of the Coordinated Objects

Since the semantics of the coordinated objects is given by labeled transition system, this work proposes the use of temporal formulas written in CTL (Computation Tree Login) for describing their properties. Temporal formulas are verified using a model checking algorithm.

APPENDIX B

Petri Nets

Petri Nets are a graphical and mathematical modeling tool used to describe and study systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, Petri Nets can be used as a visual communication aid similar to flow charts, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems.

According to [Bert93a] Petri Nets can be classified into three classes: models type I, II, and III. Models of type I allow qualitative analysis of systems (the logic of the system), models of type II include temporal extensions, and model of type III allow the quantitative analysis (evaluation of performances). In models of type I we have: Place-Transition Petri Nets, Coloured Petri Nets, Predicate-Action Petri Nets, and Numerical Petri Nets. In models of type II we have: Temporised Petri Nets, Temporal Petri Nets, With Temporal Arcs Petri Nets, With Temporal Flux Petri Nets. In models of type III we have: Stochastic Petri Nets, Stochastic Temporised Petri Nets, etc. We will focus exclusively on this chapter in the modeling and the semantics of Petri Nets type I, in particular in Predicate-Action Petri Nets which we will use to formalise and validate CoLaS coordination groups.

B.1 Type I - Modeling and Semantics

B.1.1 Place-Transition Petri Net

A Petri Net is a particular kind of directed graph, together with an *initial marking* M0. The underlying graph of a Petri Net is a directed, weighted, bipartite graph consisting of two kinds of nodes, called *places* and *transitions*, where arcs are either from a place to a transition, or from a transition to a place. In graphical representation places are drawn as circles, transitions as bars or boxes. Arcs are labeled with their weights (positive integers).A *marking* assigns to each place a nonnegative integer k, we say that p is *marked with k tokens*. In a graphical representation we place k black dots (tokens) in place p. A marking is denoted by M, an m-vector where m is the total number of places.The pth component of M, denoted by M(p) corresponds to the number of tokens in place p.

In modeling, using the concepts of conditions and events, places represent conditions and transition represent events. A transition (an event) has a certain number of *input* and *output places* representing the preconditions and the post-conditions of the event, respectively. The present of a token in place is interpreted as holding the truth of the condition associated with the place. In another interpretation, k tokens in place indicate that k data items or resources are available.

A place-transition Petri Net is a five-tuple:

PN = <Pl, Tr, I, O, M0>

- Pl = {p1, p2, ..., pm} — a finite set of places.
- Tr = {t1, t2, ..., tn} — a finite set of transitions.
- I = Pl x Tr -> N — input function. Specifies the number of tokens than should be present at a place to allow the transition to be fired.
- O = Pl x Tr -> N — output function. Specifies the number of tokens generated in a place when a transition is fired.
- M0 = Pl-> N — initial marking. Specifies the number of tokens initially set in each place.

The evolution of M0 in the time is represented as M.

The behavior of many systems can be described in terms of system states and their changes. In order to simulate the dynamic behavior of a system, a state or marking in a Petri Net is changed according to the following transition (firing rule):

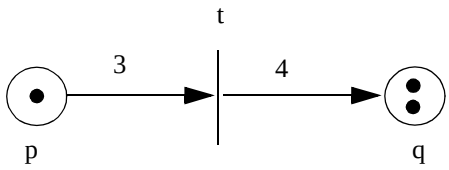
1) a transition is said to be enabled if each input place p of t is marked with at least I(p,t) tokens.

2) an enabled transition may or may not fire (depending on whether or not the event actually takes place)

3) a firing or an enabled transition t removes I(p,t) tokens from each input place p of t, and adds O(p,t) tokens to each output place p of t.

A transition without any input place is called a *source transition*, and one without any output place is called a *sink transition*.

Graphical Representation



P = {p,q} T = {t} M0(p) = 1 M0(q) = 2 I(p,t) = 3 O(q,t) = 4

Figure B.1 Graphical representation of a Petri Net

In (**Figure B.1**) we show the representation of a Petri Net composed of two places p and q, and one transition t. In place p there is initially one token, and in place q two tokens. Transition t can only be fired if and only if the number of tokens is at least three in place p, and the number of tokens generated in place q when the transition t is fired is four.

Semantics

A transition t can be fired, if and only if for all p in Pl:

M(p) >= I(p,t)

If t is fired, a new marking M' is generated in which we will have:

M' = M(p) - I(p,t) + O(q,t)

Modeling

A Petri Net corresponds to the representation of a system at a given time.The marking M corresponds to the number of available resources at that time. The modeling of a system using Petri Nets starts with the modeling of each sub process composing the system using a Petri Net, and then joining all the Petri Nets by their common places two by two in order to obtain a final Petri Net.

It is important to remark that in Place-Transition Petri Nets the tokens do not have and identify and thus they can not be differentiated. When different resources must be modelled, it is necessary to define a place per type of resource, even if those resources are used in a similar way.

B.1.2 Coloured Petri Nets

Coloured Petri Nets introduce the notion of identity assigning colours to the tokens. In this model of Petri Nets, the places in the net contain individualised tokens, each colour associated with a token identifies uniquely a token or a set of tokens. The firing of a transition depends on the identity of the respective tokens. A transformation to a place-transition Petri Net is always possible. A Coloured Petri Net corresponds to a six-tuple:

PN = <Pl, Tr, I, O, M0, C>

- Pl = {p1, p2, ..., pm} — a finite set of places.
- Tr = {t1, t2, ..., tn} — a finite set of transitions.
- I = Pl x Tr x C-> N — input function. Specifies the number of tokens of each colour than should be present at a place to allow the transition to be fired.
- O = Pl x Tr x C-> N — output function. Specifies the number of tokens of each colour generated in a place when a transition is fired.
- M0 = Plx C-> N — initial marking. Specifies the number of tokens per colour initially set in each place.
- C — a finite set of colours.

The evolution of M0 in the time is represented as M.

Graphical Representation

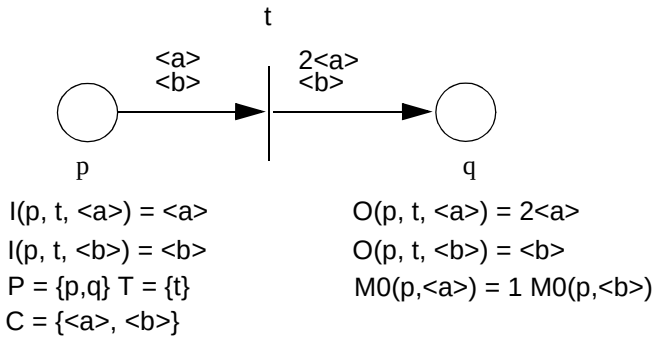


Figure B.2 Graphical Representation of a Coloured Petri Net

Semantics

A transition t can be fired, if and only if for all p in Pl:

M(p,t, c) >= I(p,t,c)

If t is fired, a new marking M' is generated in which we will have:

M' = M(p,c) - I(p,t,c) + O(q,t,c)

B.1.3 Predicate-Action Petri Nets [Kell76a]

In Predicate-Action Petri Nets systems are decomposed into two parts (**Figure B.3**): a data part and a control part. The data is defined in the form of variables, and the control part by an extended Place-Transition Petri Net. For each transition in the Petri Net a label of the form "if predicate(X) do action(X)" is added. The predicate and the action concern the variables of the system (the data).

Graphical Representation

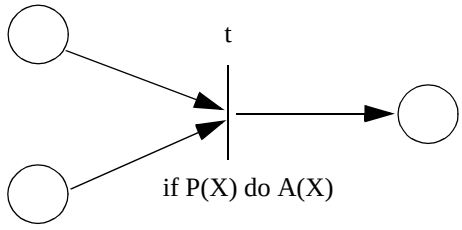


Figure B.3 Graphical representation of a Predicate-Action Petri Net

semantics

A transition t can be fired, if and only if for all p in Pl:

M(p,t, c) >= I(p,t,c) AND (P(X) is true)

If t is fired, a new marking M' is generated in which we will have:

M' = M(p,c) - I(p,t,c) + O(q,t,c)

and the action A(X) associated with t is done.

B.1.4 Numeric Petri-Nets [Symo80a]

Numeric Petri Nets are similar to Predicate-Action Petri Nets, they differentiate in that in Numeric Petri it is possible to specify for each place entering in a transition a sensibilitation condition CS for the firing rule RT. In (**Figure B.4**) we can see that each place entering in the transition t specifies a sensibilisation condition (i.e., CS1 and CS2), the firing rules RT1 and RT2 specify the number of tokens to be removed from the two places when these conditions validate to true.

Graphical Representation

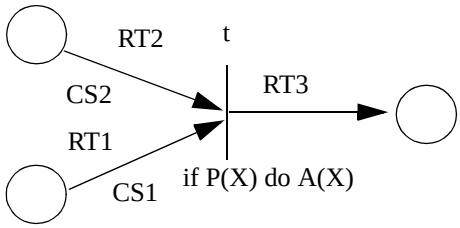


Figure B.4 Graphical representation of a Numeric Petri Net

B.2 Validation [Bram83a]

There are two families of validation methods for Type I Petri Nets.

- Enumeration Analysis: this consists of the construction of an accessibility graph from the initial marking M0. The graph is obtained by firing one by one all the possible transitions starting from the initial marking until no new transition could be fired. Each node of the graph corresponds to a marking of the system, each arch to the transition which allowed to generate the new marking. This is the most common method used for the verification of properties in Petri Nets.

  For Colored and Predicate-Action Petri Nets, the principle used to construct the accessibility graph is the same, only the fire rules change.

  Some techniques of reduction and projection can be used during the enumeration analysis to reduce the size and the complexity of the graph. The reduction and projection techniques allow to obtain simplified views of the system. The reduction technique allow to reduce the graph before the accessibility graph is built. The projection allows one to reduce the accessibility graph in order to obtain an equivalent abstract view. It is up to the person analyzing the system to specify the adequate equivalence relation as well as the transitions of the model that will remain visible (the others will become interns and non visible)
- Structural Analysis: this consists of specifying invariants associated with places. The results obtained are independent of the initial marking. The invariants represent the fact that a predicate joining the marking of a certain number of places remains always valid.

B.2.1 Formal Verification of Petri Nets [Mura89a]

A major strength of Petri Nets is their support for analysis of many properties and problems associated with concurrent systems. Two kinds of properties can be studied with a Petri-net model: those which depend on the initial marking (marking-dependent properties), and those which are independent of the initial marking (structural properties). We will use the technique called enumeration analysis to verify certain properties in the Petri Nets. The enumeration analysis technique consists of the construction of an accessibility graph from the initial marking M0. The graph is obtained by firing all the possible transitions until no new transition could be fired. In the example shown in (**Figure B.5**) we show a Place-Transition Petri Net with its initial marking matrix M0.

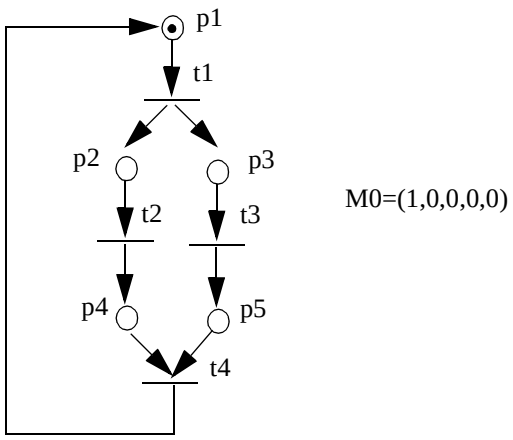


Figure B.5 Place-Transition Petri Net

1) For the initial marking M0, there is only one transition t1 that can be fired. The new marking matrix M1obtained after the transition t1 is fired is: M1 = (0,1,1,0,0,0). This transformation can be denoted as M0 (t1->M1.

2) For the marking matrix M1there are only two transition t2 and t3 that can be fired. The new marking matrixes obtained after the transitions t2 and t3 are fired: M2 = (0,0,1,1,0) and M3 = (0,1,0,0,1). These transformations can be noted as M1 (t2->M2 and M1 (t3->M3 where

3) For the marking M2 only the transition t3 can be fired. The new marking matrix M4 obtained after the transition t3 is fired is: M4 = (0,0,0,1,1). This transformation can be denoted as M2 (t3->M4.

4) For the marking M3 only the transition t2 can be fired. The new marking matrix M4 obtained after the transition t2 is fired is: M4 = (0,0,0,1,1). This transformation can be denoted as M2 (t2->M4.

5) Finally, for the marking M4 only the transition t4 can be fired. The new matrix obtained after the transition t4 is fired is M0. No new marking can be generated, then the process stops.

M0* defines the set of marking generated from M0 = {M0, M1, M2, M3, M4}.

Starting from marking M0 transitions t1 and t2 can be fired. After these transition the new marking is M2. This transformation can be denoted as M0 (t1 t2 -> M2 and S = t1 t2 as a firing occurrence sequence M0 (S->M2

behavioral properties

- Reachability: a marking Mi is said to be *reachable* from an initial marking M0 if there exists a sequence of firings that transform M0 to Mi. It has been proved that the reachability problem is decidable although it takes exponential space (and time) to verify in the general case.
- Boundness: a place Pi is said to be *bounded* for an initial marking M0 if for all marking accessible from M0 the number of tokens in Pi is finite. A Petri Net is said to bounded for an initial marking M0 is all the places are bounded for M0*.
- If for all Mi belonging to M0* if we have Mi(Pi=) <= k where k is an finite number, we is said that Pi is *k-bounded*. If this property is true for all the places of the Petri Net is said that the Petri Net is *k-bounded*.
- Safeness: a Petri Net is said to be *safe* for an initial marking M0 if for all accessible marking every place contains at most one token. A Safe Petri Net is a particular case of 1-bounded Petri Net.
- Liveness: a transition tj is said to be *live* for an initial marking M0 if for all marking accessible from Mi belonging to M0* there exists a firing sequence containing tj from Mi. A Petri Net is said to be *live* for an initial marking M0 if all the transitions are live. In other words there are not transitions in the Petri Net that can not be fired.
- Conform: a Petri Net is said to be *conform* if it is safe and live.
- Quasi-Alive: a transition tj is said to be *quasi-live* for an initial marking M0 if there exists a firing sequence containing tj from M0. A Petri Net is said to be *quasi-live* if all its transitions are quasi-alive.
- Blocking: a blocking is a marking where no transition is enabled. In other words no evolution is possible from a certain marking. A Petri Net is said to be *free from blockings* for an initial marking M0 if no marking Mi belonging to M0* is a blocking.
- Reversibility and Home State: a Petri Net is said to be reversible if from each marking Mi belonging to M0* M0 is reachable from Mi. In other words, in a reversible net one can always get back to the initial state.
- Coverability: a marking Mi in M0* is said to be *coverable* if there exists a marking Mj in M0* such that the number of tokens in Mj(p) is superior or equal to the number of tokens in Mi(p) for each p in the net.
- Fairness: many different notions of fairness have been proposed in the literature of Petri Nets. Two of them are: bounded fairness and unconditional (global) fairness. Two transitions t1 and t2 are said to be in bounded-fair relation if the maximum number of times that either one can fire while the other is not firing is bounded. a Petri Net is said to be *bounded-fair* if every pair of transitions in the net are bounded-fair. A firing sequence S is said to be unconditionally (globally) fair if it is finite or every transition in the net appears infinitely often in S. A Petri Net is said to be *unconditionally fair* net if every firing sequence S from M in M0* is unconditionally fair. Every bounded-fair net is unconditionally-fair net and every bounded unconditionally-fair is a bounded-fair net.

Bibliography

[Abad89a] M. Abadi and Z. Manna, *Temporal Logic Programming*, Journal of Symbolic Computation, 8: 277-295, 1989.

[Ache00a] F.Achermann, S.Kneubuehl, O.Nierstrasz, *Scripting Coordination Styles*, Coordination 2000, Antonio Porto and Gruia-Catalin Roman (Eds), LNCS, Vol. 1906, Springer-Verlag, Limassol, Cyprus, September 2000, pp.19-35.

[Ache01a] F.Achermann, O.Nierstrasz, *Applications = Components + Scripts: A Tour of Piccola*, Software Architectures and Component Technology, Mehmet Aksit (Ed), pp.261-291, Kluwer, 2001.

[Agha86a] G.. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986

[Ahuj86a] S.Ahuja, N.Carriero, and D.Gelernter, *Linda and Friends*, IEEE Computer, Vol. 19, No. 8, 1986, pp. 26-34.

[Andr91a] G..R.Andrews, *Concurrent Programming*, Benjamin/Cummings Publishing, 1991.

[Andr96a] J.M.Andreoli, S.Freeman, and R.Pareschi, *The Coordination Language Facility: Coordination of Distributed Objects*, Theory and Practice of Object Systems (TAPOS), Vol. 2, No. 2, 1996, pp. 635-667.

[Andr96b] J.M.Andreoli, H.Gallaire, and R.Pareschi, *Rule-Based Object Coordination*, in [Cianc96a], pp. 1-13.

[Andr99a] L.F.A.Andrade, J.L.L.Fiadeiro, *Interconnecting Object Via Contracts*, Proceedings of UML'99, Bernhard Rumpe (Ed.), LNCS 1723, pp. 566-583, Springer Verlag.

[Andr00a] G..R.Andrews, *Foundations of Multithread, Parallel, and Distributed Programming*, Addison-Wesley, 2000.

[Aldr03a] J.Aldrich, V.Sazawal, C. Chambers, D. Notkin, *Language Support for Connector Abstractions*, ECOOP 2003, LNCS 2743, pp. 74-102.

[Alle94a] R.Allen, D.Garlan, *Formal Connectors*, Internal Report CMU-CS-94-115, Carnegie-Mellon University, Pittsburg, USA.

[Aksi89a] M. Aksit, *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Department of Computer Science, University of Twente, The Netherlands, 1989

[Aksi92a] M. Aksit and L.Bergmans, *Obstacles in Object-Oriented Software Development*, OOPSLA´92, pp.341-358, Vancouver, Canada.

[Arap91a] C.Arapis, *Specifying Object Interactions*, in D.Tsichritzis, editor, Object Composition, University of Geneva, 1991.

[Arba93a] F.Arbab, I. Herman, and P.Spilling, *An Overview of Manifold and its Implementation*, Concurrency: Practice and Experience 5 (1), 1993, pp. 23-70.

[Arba96a] F.Arbab, *The IWIM Model for Coordination of Concurrent Activities*, in[Cian96a], pp.34-56.

[Arba98a] F.Arbab, P.Ciancarini, C.Hankin, *Coordination Languages for Parallel Programming*, Journal of Parallel Computing, Vol. 24, No. 7,1998, pp. 989.

[Arba98b] F.Arbab, *What Do you Mean, Coordination ?*, Bulletin of the Dutch Association for Theoretical Computer Science (NTVI), March 1998. http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief98.ps.gz.

[Ayac85a] J.M.Ayache, J.P.Courtiat, and M.Diaz, *Utilisation des réseaux des Petri pour la modélisation et la validation des protocoles*, Technique et Science Informatiques, AFCET, 1985.

[Badu02a] L.Baduel, F. Baude and D. Caromel, *Efficient, Flexible, and Typed Groups Communications in Java*, Proceedings of Java Grande 2002, November 3-5, Seattle, Washington, USA, 2002, pp. 28-36.

[Bana96a] J.P.Bânatre and D.Le Métayer, *GAMMA and the Chemical Reaction Model*; *Ten Years Later*, Coordination Programming: mechanisms, models and semantics, pp. 1-39, IC Press, London.

[Barr02a] L.Barroca, J.L.Fiadeiro, *Coordination Contracts as Connectors in Component-Based Development*, Proceedings of Integrated Design and Process Technology IDPT 2002, Pasadena, California, 2002.

[Berg94a] L.Bergmans, *Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*, Ph.D Thesis, University of Twente, The Netherlands, 1994,

[Berr98a] A.Berry, S.Kaplan, *Open, Distributed Coordination with Finesse*, Proceeding of the on Applied Computing SAC98, Atlanta, Georgia, USA, pp.178-184, 1998.

[Bert93a] B.Berthomieu, J.P.Courtiat, M.Diaz, and G.Juanole, *Techniques de description formelle pour la conception des protocoles de communication*, Rapport N. 7869, LAAS, Toulouse, 1993

[Bert03a] B.Berthomieu, P,-O.Ribet, F.Vernadat, *L'outil TINA--Construction d'espaces d'etats abstraits pour les réseaux de Petri et réseaux Temporels*, Modélisation des Systèmes Réactifs, MSR'2003 Hermes.

[Bloo79a] Toby Bloom, *Evaluating Synchronization Mechanisms*, In Seventh International ACM Symposium on Operating System Principles, pp. 24-32, 1979

[Bosc97a] R.Bastide and D.Buchs, *Models, Formalisms and Methods for Object-Oriented Distributed Computing*, ECOOP'97 Workshop Reader, Vol. 1357, Finland, pp. 221-255, June 1997.

[Bram83a] G.W.Brams, *Réseaux des Petri: théorie et practique*, Masson, 1983.

[Brio98a] J-P.Briot, R.Guerraoui, K-P. Löhr, *Concurrency and Distribution in Object-Oriented Programming*, ACM surveys 1998.

[Brio89b] J-P.Briot, *Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment*, Proceedings of ECOOP'89, British Computer Society Workshop Series, Cambridge University Press, pp. 100-129, July, 1989.

[Buff97a] M. Buffo, E. Urland, J. Rolim and D. Buchs, A Coordination Model for Distributed Systems", in [Garl97a], pp. 410-413.

[Bush95a] F.Buschmann, R.Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System Of Patterns*, West Sussex, England: John Wiley & Sons Ltd., 1996.

[Burn93a] A.Burns and G..L.Davies, *Concurrent Programming*, Addisson-Wesley, 1993.

[Carr89a] N.Carriero and D.Gelernter, *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, 1989, pp. 444-458.

[Carr94a] N. Carriero, D.Gelernter and L. Zuck, *Bauhaus Linda*, in [Cian94a], pp. 66-76.

[Chan79a] E.G. Chang and R.Roberts, *An improved algorithm for decentralized extrema-finding in circular configurations of processors*. CACM, Vol. 22, No. 5, pp. 281-283.

[Cian94a] P.Ciancarini, O.Nierstrasz, A.Yonezawa (Eds), *Object-Based Models and Languages for Concurrrent Systems*, ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, 1994, LNCS 924, Springer Verlag.

[Cian96a] P.Ciancarini and C.Hankin (Eds), *First International Conference in Coordination Models, Languages and Applications-Coordination'96*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag.

[Cian99a] P.Ciancarini and A.Wolf (Eds), *Proceedings of the third International Conference in Coordination Models and Applications-Coordination'99*, Amsterdam, The Netherlands, 1999, LNCS 1594, Springer Verlag.

[Cian01a] P.Ciancarini, *Coordination Models and Languages*, Course Material, Lipary, July 2001.

[Cinc94a] Cincom Inc., *VisualWorks™ Distributed Smalltalk Programmer's Reference*, P46-0115-0202, http:/www.cincom.com, 1994.

[Ciob05a] G. Ciobanu, D. Lucanu, *A Specification Language for Coordinated Objects*, Proceedings of the International Workshop on Specification and Verification of Component-Based Systems, Lisbon, 2005.

[Coul94a] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems*, Addison Wesley, 1994.

[Crow91a] K.Crowston, *Towards a Coordination Cookbook: Recipes for Multiagent Actions*, PhD Dissertation, Sloan School of Management, MIT, Cambridge, USA, 1991.

[Crow96a] J. Crowcroft, *Open Distributed Systems*, UCL Press, London, 1996.

[Cruz99a] J.C.Cruz, and S.Ducasse, *CoLaS: a Group Based Approach for Coordinating Active Objects*, in [Cian99a], pp. 355-371.

[Cruz99b] J.C.Cruz and S. Ducasse, *Coordinating Open Distributed Systems*, Future Trends of Distributed Computing Systems, IEEE, pp. 125-130.

[Cruz01a] J.C.Cruz, *CORODS a Coordination Programming System for Open Distributed Systems*, LMO 2001, Le Croisic, France, 2001, pp. 11-26.

[Cruz01a] J.C.Cruz, *Supporting Development of Open Cooperative Object Information Systems with CORODS*, OOIS 2001, Calgary, Canada, 2001, pp.

[Cruz02a] J.C.Cruz, *OpenCoLaS a Coordination Framework for CoLaS Dialects*, in COORDINATION 2002, York, United Kingdom, 2002, pp. 133-140.

[Deck95a] K.S.Decker and V.R.Lesser, *Designing a Family of Coordination Algorithms*, Computer Science Technical Report 94-14. University of Massachusetts, Amherst, USA, 1995.

[Didr99a] K.Dridra, *A Cooperation Service for CORBA Objects*, EuroPar'99, LNCS 1685.

[Dijk68a] E.W. Dijkstra, *Cooperating Sequential Processes*, In F. Genuys (ed) Programming Languages, 43-112. New York, Academic Press.

[Doll92a] J.Dollimore, G.Coulouris, *The relevance of Object Groups and Multicast in Shared Distributed Object Systems*, Fifth ACM SIGOPS European Workshop: Models and paradigms for distributed systems structuring, September 21-23, Le Mont Saint-Michel, France, pp. 1-4.

[Duca97a] S.Duccasse, *Intégration Réflexive des dependances dans un modèle à classes.* Ph.D. Thesis, Université Nice-Sophia Antipolis, 1997.

[Duca98a] S.Ducasse and M.Guenter, *Coordination of Active Object by means of Explicit Connectors*, DEXA worshops, View, Austria, IEEE Press, 1998, pp. 572-577.

[Espa94a] J.Esparza and M. Nielsen, *Decidability Issues for Petri Nets- a survey*, Inform. Process. Cybernet., Vol. 30, pp. 143-160, 1994.

[Ferb99a] J.Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Inteligence*, 1999, Addison-Wesley.

[Film84a] R.E.Filman and D.P.Friedman, *Coordinating Computing: Tools and Techniques for Distributed Software*, 1984, McGraw Hill.

[Fran96a] N. Francez and I.R. Forman, I*nteracting Processes: a Multi-party Approach to Coordinated Distributed Programming*, Addison-Wesley, 1996.

[Frol93a] S.Frolund, *Coordinating Distributed Objects-An Actor Approach to Synchronization*, MIT Press, 1996.

[Gamm95a] E. Gamma, R.Helm. R.Johnson, J.Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Garl97a] D.Garlan and D. Le Métayer (Eds), *Second International Conference in Coordination Models, Languages and Applications-Coordination'97*, Berlin, Germany, Sept, 1997, LNCS 1282, Springer Verlag

[Gele85a] D.Gelernter, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 7, No. 1, 1985, pp. 80-112.

[Gele92a] D.Gelernter, N.Carriero, *Coordination Languages and their significance*, Communica-

tions of the ACM, Vol.5, No.32, 1992, pp. 102-107.

[Guer92a] R.Guerraoui, R.Capobianchi, A.Lanusse, P.Roux, *Nesting Actions Through Asynchronous Message Passing: The ACS Protocol*, ECOOP 1992, The Netherlands, June/July 1992, pp. 170-184.

[Guer92b] R.Guerraoui, R.Capobianchi, A.Lanusse and P.Roux, *KAROS: un langage à objects concurrents destiné à des applications distribuées*, Techical Report CEA, CE Saclay DEIN/ SIR, 1992.

[Guer98a] R. Guerraoui, P. Ferber, B. Garbinato, K. Mazouni, *System Support for Object Group*s, Proceedings of OOPSLA'98, Vancouver, Canada, 1998.

[Helm90a] R.Helm, I.Holland, D.Gangopadhyay, *Contracts: Specifying behavioral Compositions in Object-Oriented Systems*, OOPSLA/ECOOP'90, Vol.25, October 1990, pp.169-180.

[Hern96a] J.Hernandez, M.Papathomas, J.M.Murillo, F. Sanchez, *Coordinating Concurrent Objects: How to deal with the Coordination Aspect?* , In J.Bosch and S. Mitchell (eds), Aspect-Oriented Programming Workshop ECOOP'97, Finland, 1997.

[Hoar85a] C.A.R.Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1995.

[Holz96a] A.A.Holzbacher, *A Software Environment for Concurrent Coordinated Programming*, in [Cian96a], pp. 249-266.

[IONA94a] IONA, *An Introduction to ORBIX+ISIS. IONA Technologies and ISIS Distributed Systems*, 1994.

[Jenn93a] N.R.Jennings, *Commitments and Conventions: The Foundations of Coordination in Multi-Agent Systems*, The Knowledge Engineering Review Journal, Vol. 8, No. 3, 1993, pp. 223-250.

[Jenn96a] N.R.Jennings, *Coordination Techniques for Distributed Artificial Intelligence*, in [O'Har96a], pp. 187-210.

[Kafu96a] D.Kafura, M.Mukherji, *Coordination in Statically-Typed Concurrent Object-Oriented Languages*, 1996.

[Kell76a] R.M.Keller, *Formal Verification of Parallel Programs*, CACM, July, 1976.

[Kicz91a] G.Kiczales, J.des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Kicz97a] G.Kiczales, J.Lamping. C.Maeda, C.Videira Lopes, J-M.Loingtier, J. Irwin, *Aspect-Oriented Programming*, ECOOP 1997, Mehmet Aksit and Satoshi Matsuoka (Eds.), LNCS 1241, Springer Verlag, Finland, 1997, pp. 220-242.

[Kiel96a] T.Kielmann, *Designing a Coordination Model for Open Systems*, in [Cian96a], pp. 267-284.

[Kris93a] B.Kristensen, *Traverse Activities: Abstractions in Object-Oriented Programming*, Object Technologies fr Advanced Software, First JSSST International Symposium, Vol. 742, Springer Verlag, 1993, pp. 279-296.

[Kris97a] B.Kristensen and D.May, *Activities: Abstractions for Collective behavior*, ECOOP'97, Vol. 1098, Linz, Austria, 1997, pp. 472-500.

[Land97a] S. Landis, and S. Maffeis, *Building Reliable Distributed Systems with CORBA*, Theory and Practice Object Systems 3, April 1997.

[Lea99a] D.Lea, *Concurrent Programming in Java-Design Principles and Patterns*, Second Edition, Addison-Wesley 1999.

[Less87a] V.R.Lesser and D.D.Corkill, *Distributed Problem Solving*, in Encyclopedia of AI (ed. S.C.Shapiro), pp. 245-251, John Wiley and Sons, 1987

[Lisk83a] B.Liskov and R.Shifler, *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM TOPLAS, July, 1983

[Lope97a] C.V.Lopez and G..Kiczales, *D: A Language Framework for Distributed Programming*, PARC Technical Report, TR SPL97-010P9710047, Xerox Parc, 1997

[Mage95a] J.Magee, N.Dulay, S.Eisenbach and J.Kramer, *Specifying Distributed Software Architectures*, ESEC'95, Barcelone, Spain.

[Mage99a] J.Magee, J.Kramer, *Concurrency: State Models and Java Programs*, Wiley, 1999.

[Malo93a] T.Malone, K.Crowston, *The Interdisciplinary Study of Coordination*, Technical Report #157, Center for Coordination Science, MIT, Cambridge, USA.

[Mats94a] S. Matsuoka and A. Yonezawa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming*, in Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993, pp. 107-150.

[McHa93a] C.McHale, *Synchronisation in Concurrent Object-Oriented Languages: Expressive Power, Genercity and Inheritance*, Ph.D. Thesis, Deparment of Computer Science, Trinity College, Dublin, 1994.

[Mins94a] N.Minsky, Jerrold Leichter, *Law-Governed Linda as a Coordination Model*, in [Cian94a] pp.125-146.

[Mins97a] N.Minsky and V.Ungureanu, *Regulated Coordination in Open Distributed Systems*, in [Garl97a], pp. 81-97.

[Mint92a] H.Mintzberg, *Structure in Five: Designing Effective Organizations*, 1992, Prentice-Hall.

[Mili04a] G. Milicia and V. Sassone, *The Inheritance Anomaly: Ten Years After*, Proceeding of the ACM Symposium in Applied Computing, SAC 2004, Nicosia, Cyprus.

[Miln99a] R.Milner, *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999.

[Mish89a] S.Mishra, L. Peterson, and R. Schilicting, *Implementing Fault-Tolerant Replicated Objects Using Psync*. IEEE Symposium on Reliable Distributed Systems, 1989.

[Moss81a] J.E.B.Moss, *Nested Actions: an Approach for Reliable Distributed Computing*, Ph.D. Thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, USA, 1981.

[Mukh95a] M.Mukhjeri and D.Kafura, *Specification of Muti-Object Coordination Schemes Using Coordinating Environments*, Draft, Virginia Tech, 1995.

[Mull93a] S.Mullender, *Distributed Systems,* ACM Press, 1993

[Mura89a] T.Murata, *Petri Nets: Properties, Analysis and Applications*, Proceeding of the IEEE, Vol. 77, N. 4, April, 1989.

[[Neli01a] A. Nelisse, T. Kielmann, H.E. Bal, J.Maassen, *Object-Based Collective Communication in Java,* ACM Java Grande, ISCOPE01 , Palo Alto, California, USA, 2001.

[Nier87a] O.Nierstrasz, *Active Objects in Hybrid*, OOPSLA'87, Vol. 22, Orlando, Florida, 1987, pp. 243-253.

[Nier89a] O.Nierstrasz, *A Survey of Object-Oriented Concepts*, Object-Oriented Concepts, Databases and Applications, ed. W. Kim and F. Lochovsky, pp. 3-21, ACM Press and Addison-Wesley, 1989

[Nier93a] O. Nierstrasz, *Composing Active Objects-The Next 700 Concurrent Object-Oriented Languages,* Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner and A. Yonezawa (Eds.), pp. 151-171, MIT Press, 1993.

[Nier00a] O.Nierstrasz, *Concurrent Programming,* course Material, http://www.iam.unibe.ch/cp-w01.pdf

[O'Har96a] G..M.P.O'Hare and N.R.Jennings (eds), *Foundations of Distributed Artificial Intelligence*, 1996, Wiley Press.

[OMG95a] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1995, http://www.omg.org/corba.

[OMG00a] Object Management Group, *Fault Tolerant CORBA Specification,* Document ptc/2000-04-04.

[Orfa95a] R.Orfali, D.Harkey, J.Edwards, *The Essential Distributed Objects Survival Guide,* Jon Wiley & Sons Inc., 1995

[Owic82a] S.Owicki, L.Lamport, *Proving Liveness Properties of Concurrent Programs*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982.

[Papa94a] M.Papathomas, G..S.Blair, and G..Coulson, *A Model for Active Object Coordination and its Use for Distributed Multimedia Applications*, in [Cian94a], pp. 162-175.

[Papa95a] M.Papathomas, *Concurrency in Object-Oriented Programming Languages*, in Object-Oriented Software Composition, Addison Wesley, 1995.

[Papa96a] M.Papathomas, A*TOM: An Active Object Model for Enhancing Reuse in the Development of Concurrent Software*, Research Report RR 963-I-LSR-2, LSR-Imag, Grenoble, France, 1996.

[Papa98a] G.A.Papadopoulos, F.Arbab, *Coordination Model and Languages*, CWI report, SEN-R9834, The Netherlands, 1998.

[Petr62a] C.A. Petri, *Kommunikation mit Automaten*, Ph.D. Thesis, University of Bonn, Bonn West

Germany, 1962.

[Pint95a] X.Pintado, *Gluon: and the Cooperation Between Software Components*, in Object-Oriented Software Composition, Addison Wesley, 1995

[Pric00a] N.Price, *Component Interaction in Distributed Systems*, Ph.D.Thesis, University of London, 2000.

[Puti97a] F.Puntigam, *Coordination Requirements Expressed in Types for Active Objects*, Proceedings of ECOOP 97, LNCS 1241, Finland, 1997.

[Rows97a] A.Rowstron and A.Wood, *Bonita: a Set of Tuple Space Primitives for Distributed Coordination*, 30th Hawaii International Conference on Systems Sciences- HICCS30, Maui, Hawaii, 7-10 Jan, 1997, IEEE Press, Vol. 1, pp. 379-388.

[Schr93a] M.D.Schroeder, *A State of the Art Distributed Systems: Computing with BOB*, 1993, in [Mull93a].

[Sun03a] Sun Microsystems, *JavaSpaces Service Specification, Version 2.0*, 2003, http://www.sun.com/software/jini/specs/js2_0.pdf

[Sun04a] Sun Microsystems, *Concurrency Utilities*, Java 2 Standard Edition 5.0, http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html

[Sutt05a] Herb Sutter, *A Fundamental Turn Toward Concurrency in Software*, Dr Dobb, March 2005.

[Symo80a] F.J.W.Symons, *Representation, analysis and verification of communication protocols*, Research Report 7380, Telecom Australia, 1980.

[Tich97a] S.Tichelaar, *A Coordination Component Framework for Open Distributed Systems*, SCG-Master Thesis, May 1997, University of Bern.

[Vare99a] C.Varela and Gul Agha, *A Hierarchical Model for Coordination of Concurrent Activities*, in [Cian99a], pp. 166-182.

[Vern96a] F.Vernadat, P.Azéma, *Covering Step Graph*, 17th Int. Conf. on Application and Theory of Petri Nets 96, Osaka, Japan, LNCS 1091, Springer, 1996.

[Vern97a] F.Vernadat, F. Michel, *Covering Step Graph Preserving Failure Semantics*, 18th Int. Conf. on Application and Theory of Petri Nets 97, Toulouse, France, LNCS 1248, Springer, 1997.

[Wood93a] M. Wood, *Replicated RPC Using Amoeba Closed Group Communication*, IEEE International Confererence in Distributed Computing Systems, 1993.

[Yell97a] D.Yellin, R. Strom, *Protocol Specifications and Component Adaptors*, ACM TOPLAS, Vol. 19 Issue 2, 1979.

[Yone87a] A. Yonezawa and M.Tokoro, *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, Mass, 1987

[Ziae03a] R.Ziaei, G. Agha, SynchNet: *A Petri Based Coordination Language for Distributed Objects*, GPCE 2003, LNCS 2830, pp. 324-343.

Curriculum Vitae

Personal Information:

Name:	Juan Carlos Cruz Molinares
Nationality:	Colombian
Date of Birth:	22 October 1967
Place of Birth:	Barranquilla, Colombia

Education:

2006	Ph.D in Computer Science in the Software Composition Group (SCG), Prof. Dr. Oscar Nierstrasz and Prof. Dr. Stéphane Ducasse, IAM, University of Bern, Switzerland Subject of the Ph.D. Thesis: "A Group Based Approach for Coordinating Active Objects"
1995	DEA: Diplôme d'études approfondies en informatique (Speciality: Parallelism and Distributed Systems), Institut National Polytechnique de Grenoble-INPG, Ecole Nationale Supérieure en Informatique et Mathématique Appliquée de Grenoble (ENSIMAG), Prof.Dr. Jean-Luc Koning and Prof. Dr. Yves Demazeau, France Subject of the Master Thesis: "Vers un Ingénierie des Protocoles d'Interaction Pour des Systèmes Multiagents"
1990	Engineer in Computer Science, University of the Andes, Bogotá, Colombia

Professional Activities:

2004-	IT Specialist/Security, Union Bank of Switzerland (UBS AG), Zürich, Switzerland
2001-2003	Software Consultant, Daedalos AG, Zürich, Switzerland
1999-2000	Software Consultant, Valtech AG, Zürich (former ObjectShare AG)
2002-2003	Software Consultant, Union Bank of Switzerland (UBS AG), Information and Knowledge Management Division, Responsible: Ing. Dipl. Andreas Baer.

1999-2002	Software Consultant, Union Bank of Switzerland (UBS AG), Golden Retriever System, Capacity Management Division, Responsible: Ing. Dipl. Andreas Baer
1995-1999	Research Assistant t at the Software Composition Group, IAM-University of Bern. National Swiss Foundation project: NFS 2000-46947.96. Group leader Prof.Dr. Oscar Nierstrasz
1994-1995	Research Internship at the Laboratory of Artificial Intelligence LIFIA-Grenoble, France Group leader: Yves Demazeau. Validation of Interaction Protocols in Multiagent Systems
1992-1994	Computer Science-Research Engineer at the Laboratory of Nuclear Physics of Annecy-le-Vieux (LAPP), France. Development of a parallel control system for the new level 2 of the experience L3 at the LEP (CERN). Group Leader: Andre Degre
1991-1992	(Part-time) Development engineer at the DRI institute (Ministry of Agriculture). Development of a system for planning and control of foreign investments.
1990-1992	Development and Support Engineer at INFOTEC S.A. Development of the CAD/CAM tool IXCEL v6.0. for the design and fabrication of cloths. Group Leader: Dr. José Tiberio Hernandez
1989-1990	Research assistant at the CAD/CAM team of the Computer Science Department-University of the Andes. Development of CAD/CAM tools.

SCG Responsibilities:

Co-Supervisor of the following Master Thesis at the University of Bern:

Sander Tichelaar,	1996-1997	A Coordination Framework for Open Distributed Systems
Daniel Kuehni,	1997-1998	APROCO: A Programmable Coordination Medium,
Thomas Hofmann,	1999-2000	OpenSpaces:An O.O. Framework for Reconfigurable Coordination Spaces

Teaching Experience:

1990-1991	Introduction to Programming, Professor, Computer Science, University of the Andes, Bogotá, Colombia.

Publications:

J. C. Cruz	"CIM in the textile area", Memo No. 31, University of the Andes, CIFI, 1992
J.J.Blasing, et-al	"The L3 second level for LEP with the ST 9000 transputer and the STC104 asynchronous packet switch from SCG-Thomson", Proceedings of DAQ96, Osaka 13-15 Nov. 1996.
J.C.Cruz, S. Tichelaar	"A Coordination Component Framework for Open Systems" SCG-Report, 1997.
J.C.Cruz, S. Tichelaar	"Managing evolution of Coordination Aspects in Open Systems", CTIS'98.
J.C.Cruz, S. Ducasse	"CoLaS: A Group Based Approach for Managing Coordination of Active Object", COORDINATION'99
J.C.Cruz, S. Ducasse	"Coordinating Open Distributed Systems', Proceedings of Future Trends in Distributed Computing Systems, FTDCS99
S.Tichelaar, et. al.	"Desing Guidelines for Coordination Components", SAC2000.
J.C.Cruz	"CORODS: A Coordination Programming System for Open Distributed Systems", LMO2001
J.C.Cruz	"Supporting Development of Cooperative Object Information Systems with CoLaS", OOIS2001.
J.C.Cruz	"OpenCoLaS: A Coordination Framework for CoLaS Dialects", COORDINATION'2002