International Journal of Cooperative Information Systems
© World Scientific Publishing Company

A SOFTWARE ENGINEERING APPROACH FOR COORDINATION POLICIES ELICITATION

HENRY MUCCINI*
Dipartimento di Informatica, Università dell'Aquila, Via Vetoio 10
L'Aquila, 67010, ITALY

FABIO MANCINELLI†
Dipartimento di Informatica, Università dell'Aquila, Via Vetoio 10
L'Aquila, 67010, ITALY

Received (to be inserted
Revised by Publisher)

In a software system, software processes can be concurrently run by system actors (e.g., agents, components), by making use of shared or dedicated software resources. Coordination is a research area oriented to understand, explicitly model and analyze how activities can depend on each other, in a concurrent system. Many coordination models and languages have been presented in the last few years in order to make coordination a first class entity in the development of a software system. However, what usually happens, is that coordination is described late in the software development and it is not integrated in a typical software process. What we claim, with our research, is that if coordination becomes explicit and formalized as soon as possible in the software life cycle, then it is possible to create coordination-aware software systems, which are easily maintained and developed. What we present, in the following, is a software engineering view of coordination. Diagrams produced during a UML-based software modeling, are analyzed in order to help a software engineer in the difficult task to elicit coordination requirements from informal ones. We show how UML models can drive an elicitation process in order to identify coordination requirements, and can graphically describe which elements need coordination.

Keywords: Coordination Policies Elicitation, UML diagrams, model-based coordination

1. Introduction

In current software engineering developments, it is frequent to deal with software systems which have to manage highly complex concurrency relationships between different processes, and which are developed and deployed in a distributed way. Multi-tier software systems, concurrently accessed by millions of users, geographically distributed all over the world, represent a well know example of such systems.

*email:muccini@di.univaq.it
†email:mancinel@di.univaq.it

The dimension and the complexity of highly concurrent systems make fundamental to study communication and coordination issues. In general terms, coordination is an interdisciplinary research field with the goal of *"managing dependencies between activities [...] performed by actors"* [34].

Activities and actors may be treated from (at least) two different perspectives: *a cooperative system view* and a *software system view.*

An activity can be a development or project activity, which is concurrently handled by different groups (i.e., the actors). Coordination, from this perspective, has to analyze how groups of people have to communicate/cooperate and how their development activities have to be coordinated. Cooperative information systems (CISs) support this form of coordination through a "combination of technology, people and organizations that facilitates the communication and coordination necessary for a group to effectively work together", as quoted by Ramage [41]. The technological aspect of a CIS has to manage access to information, computing services and cooperative database systems, and to provide a workflow and groupware support.

From a software system view, an activity can be considered a computational process, performed by a component or an agent (i.e., actors). A process can depend from other processes and thus interaction, communication and synchronization mechanisms are required. This is a software system perspective, where coordination has "the goal of finding solutions to the problem of managing the interaction among concurrent programs" [3].

In this paper we deal with coordination from a software system perspective.

In our research, we want to stress out the need of *an explicit description of software systems coordination, as soon as possible in the software life cycle.* We handle this problem from a software engineering perspective, by proposing a model-based approach where actors, resources and processes are the entities to be coordinated. We propose a model-based analysis of system requirements in order to identify and make explicit such coordination elements; a UML-based modeling notation is used to describe how such elements can be related and the policies dictating how they communicate and synchronize (in general, how they need to be coordinated). The identification of coordination policies is performed by means of an analysis of the system requirements and high-level specifications.

This research is motivated by the fact that coordination is relevant in design, development, debugging and maintenance of all concurrent systems [3]. However, despite the research done in this area, it seems that coordination reasoning and software development usually do not converge in order to produce real, coordination-aware systems; coordination is usually described explicitly in a *late development stage.* Models and languages have been proposed in order to make coordination explicit [38]. They offer facilities for controlling synchronization, communication, creation and termination of computational activities [38]. However, while new development paradigms are proposed to design and implement even more distributed soft-

ware systems (e.g., component-based programming, software architecture, COTS), coordination languages and models maintain their own specific range of application; they are not usually integrated in the software production process and usually they produce results that are not well integrated in the software product.

Fortunately, some cases are outside this discussion.

IWIM [4] (*Idealized Worker, Idealized Manager*), for example, is a coordination model concretely implemented by Manifold [5], a strongly-typed, event driven coordination language. Manifold code can be translated into the C language, by producing a real implementation of the IWIM model.

SODA [45] (Societies in Open and Distributed Agent spaces) is a methodology which focusses on agent society issues providing explicit coordination through a semi-formal description of the intended overall application structure and organization. It spans from coordination elicitation to analysis and design.

MASE [35](Multiagent Systems Engineering) is a methodology to engineer multiagent systems. MASE makes use of two different UML-like notations, namely the Agent Modeling Language (AgML) and the Agent Definition Language(AgDL), in order to model agent classes and interactions, the communication hierarchy, the communication Class and Deployment diagram, and to formally describe the internal behavior of each individual agent.

Both IWIM, SODA and MASE will be extensively analyzed in Section 7.

Other examples of mapping conceptual coordination models into real implementations are provided by Jada [14], JavaSpaces [30] and PageSpace [12].

Our approach goes in the direction of the previously outlined research work. We claim that coordination issues have to emerge as soon as possible, since they often exist from the very beginning and can influence architectural and design choices. The research we are doing in this area goes towards this direction. If coordination policies become explicit early, it becomes possible to verify the adequacy of a software architecture model (or, of the code itself) with respect to expected coordination constraints [37,33], as well as to refine or disambiguate coordination requirements themselves.

In this paper, *we propose a way to elicit coordination policies from informal requirements by using UML [42] diagrams*. We want to provide the software engineer a way to extend UML models with coordination issues. In some sense, we are proposing a sort of *model based coordination framework* to help developers eliciting, modeling and analyzing coordination policies.

Here a brief summary of the paper: UML Use Case diagrams are analyzed in order to identify the software system elements which require to be coordinated. Such elements are tasks, resources and actors (as explained in Section 3) and are called *"coordination entities"*. A UML-compliant Entity diagram is modeled in order to specify such entities. The next step identifies a *"coordination structure"* between such elements (i.e., a coordination pattern). A UML-compliant Coordinator diagram is used to model this information. The last step helps a software engineer

to understand how to implement the system coordination (i.e., the *"coordination policies"*). These policies should help to elicit coordination information embedded into software requirements and specifications. Figure 1 graphically summarizes the three coordination aspects analyzed in the following of this paper.

This approach is part of a general framework which aims to encapsulate the coordination in a software development process, enabling elicitation, description, analysis and validation of system coordination properties [26].

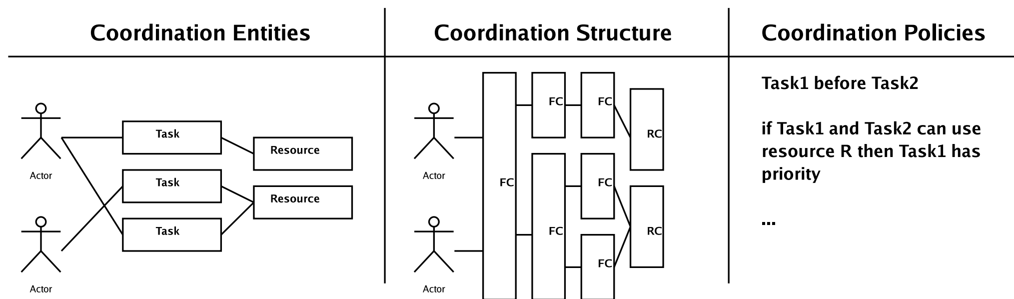

Fig. 1. Coordination entities, structure and policies

The paper is structured as follows. In Section 2 an overview of the framework is presented. Section 3 briefly introduces the UML diagrams we use in our approach. Section 4 is devoted to describe the Whiteboard case study while Section 5 focuses on the approach, by describing how we carry on elicitation of requirement constraints from UML diagrams. Section 6 presents some related work while Section 7 summarizes the contribution of this paper and proposes further research directions.

2. The Framework

This section outlines the general coordination framework we are working on. It proposes a UML-based software development process that, starting from initial software artifacts *i)* allows to identify the elements to be coordinated, to elicit and to specify coordination requirements in order to define coordination policies; *ii)* uses these elements to drive the Software Architecture (SA) description; *iii)* checks the conformance of the architectural description with respect to the coordination requirements and finally *iv)* uses the SA to drive the construction of the lower level coordination model. In this section, we summarize the four-steps framework, graphically presented in Figure 2, as a way to contextualize the coordination requirements elicitation step described in depth in the following of this paper.

1. *Coordination requirements are elicited and formalized:*

as it will be extensively described in Section 3, UML diagrams are used to help the elicitation of the two complementary aspects in system coordination, that is, *structural* and *behavioral* coordination. Structural coordination identifies coordination entities and the coordination structure. Behavioral coordination identifies coordination policies. This analysis is developed in terms of actors, tasks and resources dependencies.

2. *SA is built upon Requirements:* ([23,26])

   if coordination entities and policies are known, we can use this information to drive the SA specification. While describing the SA of a system, the coordination structure is used to identify architectural *components* and *connectors*, while the coordination policies help to understand how architectural components have to interact through the connectors, in order to realize the coordination policy.

3. *SA is analyzed with respect to requirements:* ([25,24])

   Coordination models and policies specify the expected coordination, while the architectural model its implementation. What may happen is that the SA model does not conform to the initially discovered coordination laws.

   The Charmy framework (presented elsewhere [25,24]) is a model-consistency checking approach (based on the SPIN [22] model checker) which verifies if an architectural model correctly implements coordination requirements. Charmy is supported by an automated, easy to use, tool which takes in input the architectural models and the coordination requirements, both expressed through UML diagrams, and automatically checks the consistency between such specifications. More information, examples and the tool supporting the approach are described in [25,24].

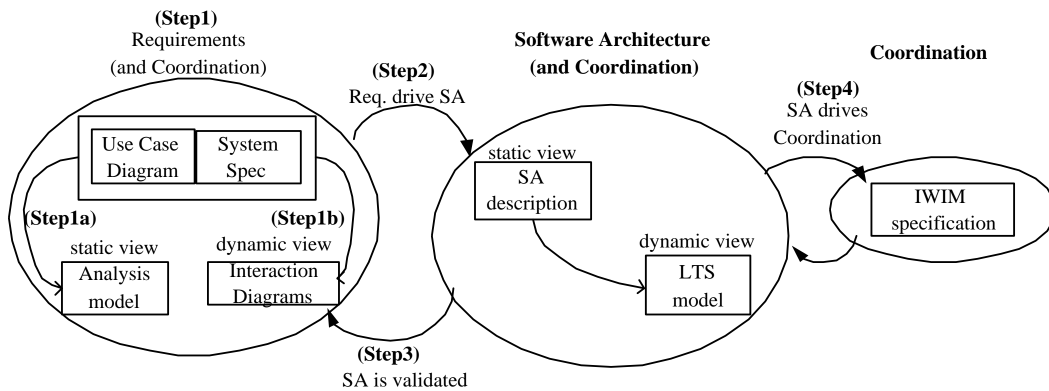

Fig. 2. Summarizing the Process

4. *Mapping SA to low-level coordination models:* ([23,26])

   the SA model can be used to drive the construction of a coordination model. The idea is to create a mapping among SA elements and coordination language entities. In this way, we can propagate coordination requirements into the following development stages.

   In [26] a sort of structural mapping among SA components, connectors, ports and IWIM [4] processes, ports and channels is sketched. In Section 7 we will shortly present an ongoing work on mapping ACME [1] architectural specifications into the coordination language Manifold [5]. In both cases, information elicited at the requirement level is propagated into design and implementation.

3. UML diagrams and stereotypes

In our approach to elicit coordination properties, we make use of two different UML diagrams: the Use Case diagram and the Class diagram.

A Use Case captures functional requirements. Use Case diagrams show how Use Cases are associated to actors. *Use Cases* represent a possible way of using the system (i.e., high-level functionalities) while *Actors* are who or what (humans or subsystems) carry out Use Cases. Actors and Use Cases identify roles [42] (i.e., types) that can be instantiated to identify the system users or functions, respectively. Multiple instances of them may interact in the system. For example, the actor type User may be instantiated by Al, Jack and John who can run multiple times the same Use Case.

Following a generic UML-based development process, Use Case diagrams may be analyzed in order to generate a UML Class diagram. In a Object-Oriented perspective, UML Class diagrams show the classes of the system, with their operations and attributes, and the inter-relationships with other classes. In different contexts, a Class diagram can be used to explore domain concepts in the form of a domain model, or to analyze requirements in the form of a conceptual model. A conceptual Class diagram allows to model “abstract” cooperating classes, and associations between them. A conceptual class is nothing more than a class, with a different semantics, expressed by using stereotypes [42]. A stereotype is part of the UML extension mechanisms and defines a new model element type. For example, the Analysis model (used in the Rational Unified Process (RUP) [32]), is a stereotyped Class diagram where classes may assume three different meanings. They can describe window abstractions or abstract communication interfaces (in this case, they are called “boundary” classes), the logical data structure (through “entity” classes), and control of other objects, using “control” classes. From a diagrammatic point of view, a stereotyped class is drawn by placing the stereotype name around angle brackets (or by using icons); a stereotyped Class diagram is a collection of inter-related stereotyped classes.

In order to model coordination information using UML, we will make use of stereotypes in order to introduce new modeling elements, which enable coordination modeling.

These stereotypes are the following:

- <<resource>>
  Identifies a system entity which represents a limited and shared (hardware or software) resource, which is used by some of the system functionalities, in order to be executed.
- <<task>>
  Identifies system entities which are associated to system functionalities, previously defined in the Use Case diagrams.
- <<run>>
  Identifies the relationships between system entities which execute tasks and the tasks they run.
- <<access>>
  Identifies the relationships between the system tasks and the resources they need to access in order to be executed.
- <<resource coordinator>>
  Identifies a system entity which regulates the access to a particular resource.
- <<flow coordinator>>
  Identities a system entity, which is associated with a particular task. It coordinates single or multiple executions of this task with respect to other tasks.

4. The Whiteboard case study

In this section we describe a simple videoconferencing system based on the whiteboard metaphor.

A whiteboard is a shared graphical space that the users can use to draw sketches while conferencing together, just as if they were on the same room. The system should support an arbitrary number of conferences in parallel, and must enable remote users to communicate using the whiteboard metaphor.

Each conference has a special user called *chair* which is the supervisor of the conference and moderates it. A conference is opened when its *chair* is connected to the system, otherwise it is closed.

Each opened conference may be joined by an arbitrary number of users. There are two different users: *standard user*, which attends a conference only as an auditor, and *participant*, which attends a conference and actively participate to it. Users join a conference by establishing connection to the system from remote locations, using the network. Users can leave a conference whenever they want.

Each opened conference has a single *whiteboard* and a single *broadcast audio channel* that can be used only by one *participant* at a time.

When a user joins a conference, by default, it is assigned to the *standard user* class. Only the conference *chair* can grant the access to the conference and can raise the privilege level of a user to the *participant* class. He can also regulate the use of both the whiteboard and the broadcast audio channel by granting their use to the conference *participants*.

Each user of a conference can talk to another user by using a *private audio channel*.

Each user of a conference can choose to listen to the broadcast audio channel or to the private audio channel. Anyway, for each user there could be only one audio channel (private or broadcast) active at any time.

In Section 3 we will present the approach and apply it to this case study.

4.1. Use Case diagram for the Whiteboard system

From the informal description of the whiteboard system requirements, we extract a set of Use Cases (denoted by UC(i)) for our system (Figure 3), whose actors are the following:

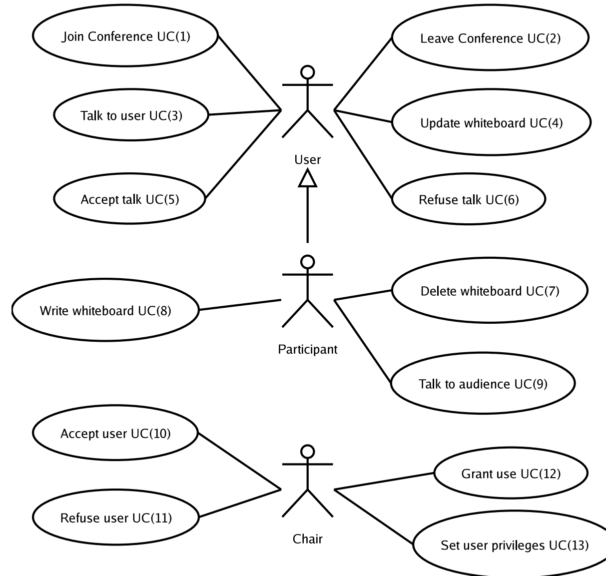

Fig. 3. Use Case diagram for the whiteboard system

- *User*: A *standard user* which wants to attend the conference but has low privileges.
- *Participant*: A user which participates to the conference as a speaker, and

therefore may interact with the audience by using both the whiteboard and the broadcast audio channel.

- *Chair*: a chair which manages the conference by granting access to the conference itself, and regulates the use of the whiteboard and the broadcast audio channel.

The Use Cases associated with the *user* actor are the following:

- *Join conference* UC(1): A user may join one of the available conferences. Only one conference at a time, may be active for a given user.
- *Leave conference* UC(2): A user may leave a conference whenever he wants. By leaving a conference, the user is allowed to join other conferences.
- *Talk to user* UC(3): A user may talk with another user, which is attending the same conference, through a private audio channel.
- *Update whiteboard* UC(4): A user may request an update to the whiteboard displayed on its client, in order to be synchronized with the system whiteboard, which resides on the server. The update may be done automatically by the client, or explicitly upon user request.
- *Accept, refuse talk* UC(5), UC(6): A user which receives a talk request may accept of refuse it. The conversation is initiated only upon acceptance of the talk request. Talking to another user, prevents the user to listen to the conference broadcast channel.

The Use Cases associated with the *participant* actor are all those associated with the user actor (since a participant is a particular kind of user) plus the following:

- *Write, Delete whiteboard* UC(7), UC(8): A participant, which has access to the whiteboard, may write sketches on it or delete some sketches previously written. Access to the whiteboard is exclusive, i.e. only one participant at a time may use the whiteboard.
- *Talk to audience* UC(9): A participant may use the broadcast audio channel to talk to all the users which are currently attending the conference. Access to the broadcast audio channel is exclusive.

The Use Cases associated with the *chair* actor are the following:

- *Accept, Refuse user* UC(10), UC(11): When a user requests to join a conference, the chair may accept or refuse his request.
- *Grant use* UC(12): The chair must grant the use of the whiteboard or the broadcast audio channel to the participant which is asking to use them.

- *Set user privileges* UC(13): The chair may change the privileges of the users which are attending a conference by raising them to the *participant* level or lowering them to the standard *user* level.

5. Step1: Coordination requirements elicitation and formalization

In Requirement Engineering (RE) [37] one of the most important activities is requirements elicitation, which allows to analyze the system in order to discover as many as possible requirements [43]. When software requirements are explicit, it becomes much easier to reason about them. In particular, if we do not have an explicit understanding on how system elements coordinate, it becomes difficult to explicitly reason about them.

Usually, in traditional software processes, coordination requirements are not explicitly stated. They are embedded into system requirements and specifications. In cooperative software development processes, some attention has been paid on identifying cooperative, collaborative and coordination requirements [8]. These work analyze how collaborative, distributed groups need to cooperate and communicate in order to produce a cooperative system. Our perspective is different: given a concurrent system to be realized, we want to understand how software processes can be run by the system users, and how they make use of available resources.

In the following, we describe a way to elicit coordination information implicitly stated by software informal requirements. We take a software engineering, model-based, perspective in which UML is used as the standard modeling notation. We do not argue that UML diagrams may be used to automatically elicit coordination requirements; they are not so informative. What we claim is that they may drive the analysis of requirement specifications in a way that enables the coordination policies elicitation.

In brief, this is what we are proposing:

- given a software system, modeled using UML, we analyze the diagrams in order to identify which model elements need to be coordinated (i.e., the *coordination entities*). This aspect is analyzed and applied to the Whiteboard example in Section 3;
- in order to identify how entities can depend on each other, a hierarchy of coordinators is identified and called *coordination structure*. Section 4 analyzes this step and applies it to the case study;
- *the coordination policies*, which defines the coordination laws, that is, how the coordination entities are coordinated, are finally identified and embedded in explicit coordination modules (Section 6).

The coordination structure assumes the presence of a coordination medium, which enables the communication between the coordination entities. The definition

of which type of medium will become clear when the architecture of the system will be defined in detail, later in the software lifecycle.

Coordination entities and structure identify what we previously called the *"structural coordination"*. During the structural coordination analysis, we focus on identifying the coordination topology of our system. This information is modeled by using two different diagrams: an Entity diagram and a Coordinator diagram. Both of them are stereotyped Class diagrams and are drawn with the goal of capturing the coordination entities and the coordination structure, respectively.

Coordination policies, instead, identify laws the coordination entities have to implement. This aspect is called *"behavioral coordination"*.

In both structural and behavioral coordination, we make an explicit distinction between "flow coordination" and "resource coordination", as discussed in Section 4.

5.1. Coordination Entities

Coordination has the goal to manage dependencies and interactions between different software programs or tasks, run by different actors, in a concurrent system [34,3]. Moreover, Malone and Crowston [34] recognize that "whenever multiple activities share some limited resource, a resource allocation process is needed to manage the interdependencies among activities". This means that, since coordination analyzes dependencies between activities, it has to analyze dependencies due to the use of the same (limited) resources. The need to consider actors, in a coordination process, is highlighted in [34]: since activities (to be coordinated) are performed by actors, actors are active elements in the coordination process.

So far, software coordination needs to take into account three different entities: *tasks*, *resources* and *actors*. We call such entities coordination entities, and we show how UML diagrams can be used, in conjunction with system specifications, to capture information on coordination entities.

Use Case diagrams describe how system functionalities are assigned to system actors, by modeling both actors and their Use Cases. In particular, the actor coordination entity is directly modeled by this diagram.

In order to identify and model the other coordination entities, we can "implement" the Use Case diagram by using a conceptual Class diagram. We introduce an Entity conceptual diagram, where three types of stereotyped classes are shown: actors (already a standard stereotype in UML), tasks (used to identify conceptual components used to implement Use Cases) and resources (used to identify hardware and software resources, accessed by tasks). These classes are the main ingredients to produce an Entity diagram. By means of an Entity diagram, resources can be assigned to tasks and actors can run tasks. By using other UML extension mechanisms (tagged values and constraints), the following communication pattern can also be imposed: an actor can be related with a task component with a stereotyped association <<run>>. A task component can <<access>> a class resource.

An actor cannot directly talk with a resource class. A formal description of these stereotypes is outside the scope of this paper.

After this initial structural analysis, we can conclude that:

- a Use Case diagram is drawn in order to identify actors;
- an Entity diagram identifies resources and their assignment to different tasks.

A deeper look into Use Case diagrams, let us remember that actors and Use Cases represent "roles", not system elements. For each role, many instances can be active on a given time. Thus, many actors can simultaneously run multiple instances of the available tasks. A coordination elicitation process may want to consider this finer-grained concept of coordination entities. Multiplicity, assigned to Entity classes, can model this further information.

5.1.1. Coordination Entities of the Whiteboard example

From the Use Case diagram of the Whiteboard example in Figure 3, we derive the Entity diagram which enables us to identify the coordination entities which are present in our system. Such a model is shown in Figure 4 and can be read in the following way: the previously identified Use Cases are realized by using a set of stereotyped <<task>> classes, which give, in some sense, a first picture of the system architecture and design. A set of resources is also identified. These resources will be directly accessed by the tasks, in order to carry on the functionalities they implement. The stereotyped <<access>> relationships, define which task needs to access a particular resource. As defined in the Use Cases, only some actors are associated with a particular Use Case. In this view, this information is preserved by associating, in a consistent way, the right actors to the tasks implementing the functionalities actors are associated with. For example, the *User* actor, may run the *"talk to user"* task in order to establish a conversation with another user attending the same conference. This is a realization of the *"talk to user"* Use Case, and makes use of one of the audio channel available in the system. Such an association, is not explicited at the Use Case level, and it is required in order to define the interdependencies at the coordination level, among the entities in the system.

The model shown in Figure 4 does not contemplate all the functionalities of the previously described Use Cases, but focuses on the most important and relevant

ones.

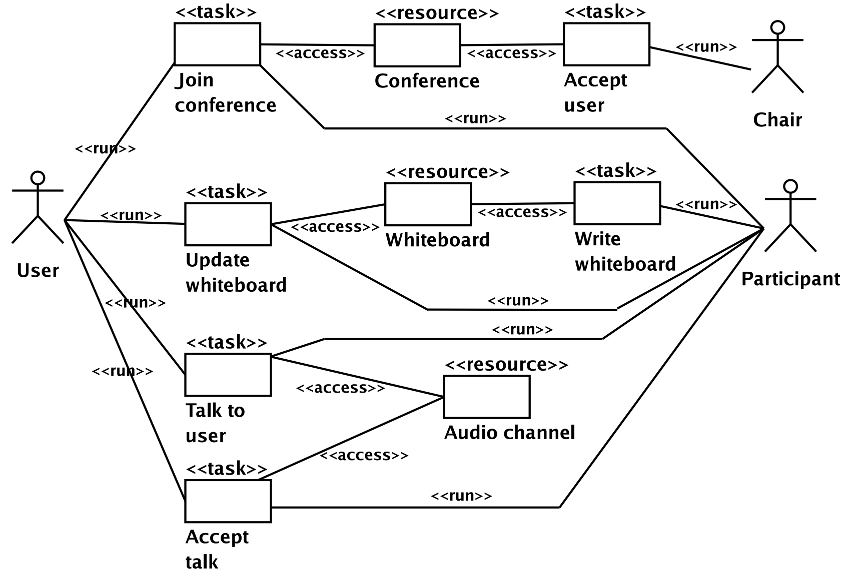

Fig. 4. The Entity model for the whiteboard system

5.2. Coordination Structure

So far, we know which entities have to be coordinated. This second step identifies coordinator components which will be responsible to coordinate actors, tasks and resources.

A first idea could be to introduce a global, monolithic, coordinator which takes as inputs the actors requests, coordinates the various activities and assigns different resources to different tasks.

It is easy to show that this approach has several limitations: i) given a complex system, it may become very difficult to apply this big-bang approach. The coordination entities may be many and the coordination policies to implement may become very complex. ii) This solution is not modular: requirement changes could oblige to completely modify the coordinator behavior. iii) If processes and resources want to be coordinated, *flow* coordination and *resource* coordination need to be addressed. With the term "flow coordination" we mean all those coordination policies which identify how different processes, run by different agents, can compete or cooperate to accomplish the assigned task. "Resource coordination", instead, regulates the access to shared resources. In the currently proposed coordination model, those two concepts are embedded in the same component, while we would like to take them distinct.

In order to fix the discussed limitations, we propose a divide et conquer ap-

proach where many coordinator components are provided. The idea is to introduce a set of specialized coordinators, each one responsible for a different coordination aspect. With the term coordinator, we identify system entities, that will be eventually reflected in the system architecture, which mediate the communication between the other architectural components. A coordinator follows an identified set of rules which regulate the flow of system functionalities executions and the access to resources needed to execute a given task.

For each resource, identified in the Entity diagram, a resource coordinator (RC) is provided. This component regulates the access, of different tasks and users, to a specific resource. For each Entity diagram task, a flow coordinator (FC) is provided (i.e., a coordinator which regulates the interaction of this task with other tasks). Instead of introducing a single FC for each task, we prefer to introduce a set of (more specific) coordinators specialized in identifying i) how a process can select a resource, ii) dependencies between tasks run by the same actor and iii) dependencies between tasks run by different actors. This hierarchical coordination structure assigns, to each coordinator, a specific job. When the job is completed, lower-level coordinators become responsible to coordinate other aspects.

This structure subsumes some properties: i) if requirements change, we should be able to modify only a limited subset of the presented coordinators. The existence of many coordinators, in fact, allows for a separation of coordination concerns and helps in managing changes. ii) The coordination structure implements the concept of exogenous model of coordination (i.e., computation and coordination are handled by different components).

This coordination structure can be considered “maximal” with respect to a generic software system. This means that not necessarily all the coordinators need to implement a coordination policy. Some of them can be removed, when not needed. Moreover, the distinction between resource coordinators and flow coordinators becomes explicit.

It is worth noticing that the coordination structure introduced can be easily extracted from the Entity diagram (by associating RCs and FCs to resources and tasks, respectively) and may be modeled by using a new stereotyped UML Class diagram (called *Coordinator diagram*), where:

- the stereotyped classes can be actors, flow coordinators or resource coordinators. This structure hides the computational entities, which will be reintroduced at the software architecture level;

- actors can only communicate with higher-level coordinators. The relationship between actors and RCs is stereotyped with the name of the task the actor wants to run;

- coordinators can communicate each other, conforming to the hierarchy outlined.

5.2.1. Coordination structure hierarchy

The previously described analysis yields a coordination structure which comprises four different coordination levels, in which both flow and resource coordination aspects are analyzed and implemented inside the FCs and RCs components.

Each level elicits a different aspect of coordination, as follows:

Level 0: handles the access to shared resources, which are used by different tasks in order to be executed.

Level 1: handles two coordination aspects. It i) identifies the association between tasks and resources, and ii) coordinates multiple executions of the same task UC(i):

- for each Use Case *i* (UC(i)) in the Use Case diagram, the first aspect analyzes which resource UC(i) needs to use in order to accomplish its task. If multiple instances of the same resource are available, this step is responsible to allocate the resources with respect to the active tasks. The policies have to be extracted from the informal requirements.
- if the same actor can execute multiple instances of the same task, we need to identify how those executions have to be coordinated. We expect to have purely sequential executions, when the tasks are synchronously executed. We can define different policies, otherwise. Since the same resource may be accessed by different processes, a RC identifies how a resource may be accessed by multiple instances of the same Use Case. For example, a first-come first-served or priority-based RC policy can be used.

Figure 5.a) graphically summarizes the activities of a level 1 flow coordinator called FC1.x, where "x" is one of the Use Cases to be coordinated. FC1.x receives, from the upper level coordinators, many instances of the *x* task to be coordinated. The FC1.x component decides how the instances have to be coordinated and how resources have to be assigned to such tasks.

Level 2: coordinates different tasks run by the same actor and different instances of the same actor running many tasks. Again, the description of this level can be decomposed in two sub-levels:

- since the same actor may run different tasks, we need to analyze dependencies between such tasks. Assuming that processes are completely independent, we can accept a parallel execution flow of both of them. In this case, we can assume the FC to be empty. Anyway, different tasks can use the same shared resource. A RC needs to identify how different tasks may access the same resource;
- when many instances of the same actor class are executed, it becomes important to analyze how they interact. For example, three actors can simultaneously run their tasks. What may happen, is that an operation from Actor1 is allowed

only when something else has been previously executed by another actor. A RC analyzes how the same resource is accessed by the actor processes.

Figure 5.b) graphically summarizes the activities of a level 2 flow coordinator called FC2.u1, where "u1" is one of the Actors. FC2.u1 receives, from the upper level coordinators, many different tasks to be coordinated. Such tasks are run by many instances of the same actor class. The FC2.u1 component decides how the tasks run by the same family of users would be coordinated. The output of FC2.u1 will be directed to the appropriate FC1.x coordinator.

Level 3: points out how different actors are coordinated. A FC dictates how different actor tasks are dependent. For example, when a process is run, some other task needs to follow immediately. A RC examines how the same resource is shared among different actors.
Figure 5.c) graphically summarizes the activities of a level 3 flow coordinator, called FC3. FC3 receives the Use Case execution requests from different users, decides how tasks run by different users coordinate and demands level 2 coordinators for the other coordination activities.

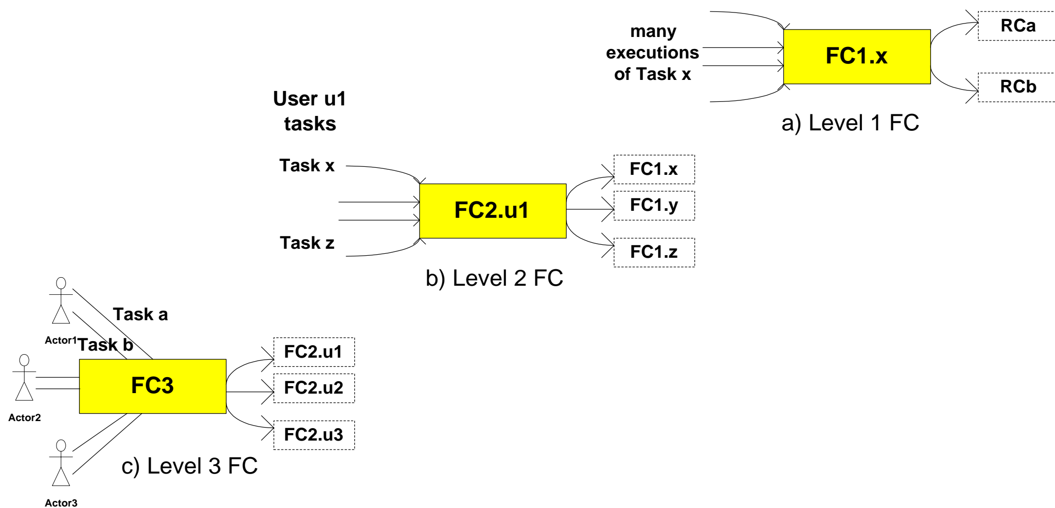

Fig. 5. Level 1, Level 2 and Level 3 Flow Coordinators

The information captured during these steps are implemented inside FCs and RCs. Flow coordination in level 1 is implemented inside FC1, level 2 decisions are realized by FC2 while level 3 policies are embedded into FC3. Resource coordination is orthogonal to each step and implemented by the RC previously defined. UML state diagrams and UML sequence diagrams could be used to model the behavior of FC and RC coordinator components.

In the next step we will analyze how to identify the coordination policies, that is, how to identify the coordination laws dictating how FC and RC coordinators work.

5.2.2. Coordination Structure of the Whiteboard example

The Use Case realizations identified in the previous stereotyped Class diagram yield a coordination structure which is layered, as previously described, in four levels. In each level there are a variable number of coordinators, depending on how the model has been constructed. In our case, each identified resource is in the first level and is associated with a resource coordinator RC_i (Figure 6.a). An RC coordinates the access of many tasks to the same, shared, resource. For example, tasks like "Join Conference", "Accept User" and "Grant access" need to use the same shared resource "Conference". RC1, which handles the resource Conference coordination, decides how such tasks can use the resource (e.g., concurrent access, exclusive access, task priority on the resource, round robin access, etc.). In the second level, a coordinator for each task of each user is introduced (Figure 6.b). FC1.1, for example, is the User flow coordinator for the "Join Conference" Use Case (UC(1)). It handles the coordination among multiple executions of the Joint Conference Use Cases, run by a system User. Moreover, it selects the resources this task requires in order to be properly executed. FC1.1 to FC1.4 are coordinators dedicated to the User tasks. FC1.5 to FC1.9 are flow coordinators dedicated to the Participant actor tasks. FC1.10 and FC1.11 coordinates the Chair tasks. The third level (Figure 6.c) contains three coordinators, each one associated with each actor previously identified. FC2.1 is responsible for the coordination among multiple executions of different tasks for the "User" actor. Whenever different tasks are coordinated, the execution of multiple instances of the same tasks is demanded to FC1.x coordinators. FC2.2 and FC2.3 work as FC2.1, but for the "Participant" and "Chair" actors, respectively. Finally, a top level coordinator (Figure 6.d) is introduced in order to handle coordination between tasks executed by different actors. The FC3.1 coordinator receives all the execution requests from the three actors, coordinates the execution of tasks run by different actors and demands lower

level coordinators for the following coordination steps.

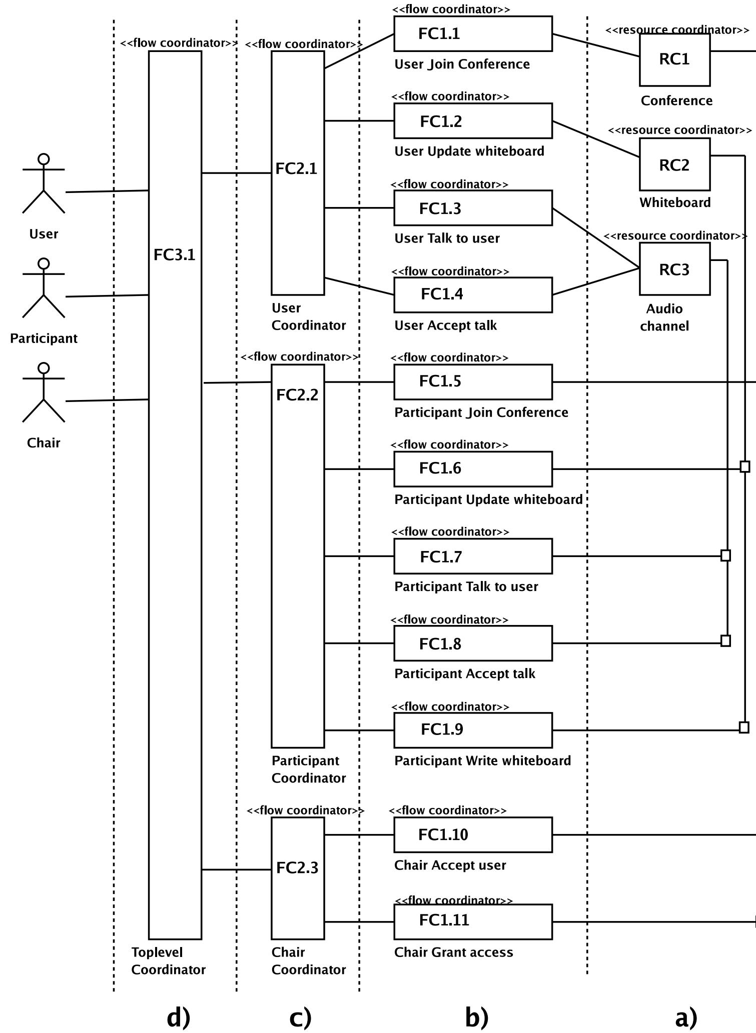

Fig. 6. The Coordinator diagram of the Whiteboard system

5.3. Coordination policies

Once we have identified the coordination entities and structure, we still need to analyze which policies regulate the coordinators behavior.

In Section 4 we said that our coordination structure is intended to be maximal. In fact, when processes are dependent and they share a common resource, both FC and RC is required. On the contrary, when processes are completely independent and they use different resources, we can assume they may run concurrently, without

the need of an explicit coordinator. This result is shown in Figure 7.

	Shared Resources	Unshared Resource
Dependent Process	F.C. + R.C.	F.C.
Independent Process	R.C.	-

Fig. 7. Relationship between Resource and Process coordination

Summarizing, in order to implement a flow coordination, we need to identify coordinator components which implement the coordination rules dictating how:

1. multiple executions of a task can choose in between available resources (FC1);
2. multiple executions of different tasks, run by different instances of the same actor, can be coordinated (FC2);
3. different Users can be coordinated when running different tasks (FC3).

In order to implement a resource coordination model, we need to understand how:

- the same resource is assigned to multiple instances of the same Use Case;
- the same resource is shared among different Use Cases;
- the same resource is accessed by different instances of the same actor;
- the same resource is shared among different actors.

In this case, a unique resource coordinator (RC), for each resource, is incrementally refined in order to encompass all these coordination aspects.

At this point, we identified which components are responsible for flow and resource coordination, in the different steps, and how they are integrated together. Another important issue to be analyzed is how the coordination policies may be identified and implemented inside the coordinator components. In Linda, a fixed set of coordination primitives are defined while in Gamma, a condition-action relationship is identified. In our general case, the coordinators, to correctly reflect the expected behavior, need to conform to the system specifications: if we assume the system specifications are informal (as usually they are), the best way to identify the coordination policies is reasoning about them and query the commissioner in order to gain a valid understanding on the system coordination.

Based on this consideration, what we propose is a questionnaire in which each question is intended to investigate relationships between resources, processes and actors to identify the policies implemented by the FCis and RCis. All the questions need to be instantiated to the specific actors/Use Cases/resources in order to write the actual coordination policy. Use Case diagrams, Analysis models and software specifications can be used for this purpose. In the following, we present a minimal set of questions applicable to different software systems. Context or system dependent questions can be naturally introduced to refine this initial set of questions.

Level 1:
- Flow Coordinator FC1:
Q: Which resources are used by the Use Case under analysis?
Q: How many instances of the same resource are available?
Q: How do the Use Cases access the different resources?
Q: How may the actor access the different resources?
Q: How many instances of the same Use Case, an actor can call?
Q: In which order these functionalities will be executed?
- Resource Coordinator RC:
Q: If a resource R is used by UC(i) and multiple instances of UC(i) may be run by the same actor, how is R accessed by the different instances?

Level 2:
- Flow Coordinator FC2:
Q: What Use Cases can be executed by the same actor?
Q: How do Use Cases interact?
Q: How many instances of the same actor can populate the system?
- Resource Coordinator RC:
Q: If different Use Cases UC(i), UC(j) can access the same resource R, how their access have to be regulated?
Q: If actors of the same type can access the same resource in order to realize some Use Cases, what access policy can be applied?

Level 3:
- Flow Coordinator FC3:
Q: How many Uses Cases and resources are shared among different actors?
Q: Are there any privileges on some of the actors?
- Resource Coordinator RC:
Q: If different actors can make use of the same resource R, what is the policy regulating the access to R?

5.3.1. Coordination policies of the Whiteboard example

In this section, we provide a set of policies which regulates the behavior of each coordinator identified in the coordination structure. *Level1 Flow Coordinator (FC1)*

Level1 flow coordinators handle two aspects of the system functionalities execution. The first one deals with how they are coordinated with respect to the multiple instances of the resources they use. The second one, instead, deals with how multiple instances of the Use Case execution are coordinated between them.

- Flow coordinator: *FC1.1*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.2*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.3*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.4*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.5*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.6*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.7*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.8*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.9*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.10*
  Policy: Process all the requests in parallel.
- Flow coordinator: *FC1.11*
  Policy: Process all the requests in parallel.

Level2 Flow Coordinator (FC2)

Level2 Flow coordinators handle how different Use Cases, carried on by the same

actors are coordinated, and how different instances of the same actor type are coordinated together. For the first aspect, partial orderings on the Use Cases are typically elicited. The second aspect, instead, takes into account how such flows are handled (for example, parallel or sequential executions).

- Flow coordinator: *FC2.1*
  Policy: Whiteboard updating UC(4) and talking requests UC(3) must be performed after joining a conference UC(1). A user can talk to another user UC(3) after he has accepted that request UC(5). Other requests are handled in parallel.
- Flow coordinator: *FC2.2*
  Policy: Whiteboard updating UC(4) and talking requests UC(3) must be performed after joining a conference UC(1). A user can talk to another user UC(3) after he has accepted that request UC(5). The other requests are handled using a First Come First Served policy.
- Flow coordinator: *FC2.3*
  Policy: Process all the requests using a First Come First Served policy.

Level3 Flow Coordinator (FC3)

Level3 Flow coordinators handle how different actors are coordinated. In this case it coordinates the functionalities initiated by the different actors.

- *Use Cases: Join Conference UC(1), Accept User UC(10)*
  Policy: A user may join a conference only after the chair has accepted him to join.
- *Use Cases: Write whiteboard UC(8), Grant use UC(12)*
  Policy: A participant may use the whiteboard only after the chair has granted him the use.

Resource Coordinators

Resource coordinators regulate the access to shared resources by imposing some policies on their use. Such policies have been incrementally identified throughout all the Flow Coordinators policy elicitation process. In the following we present only the final result:

- Resource Coordinator: *RC1*
  Policy: Concurrent access.

- Resource Coordinator: *RC2*
  Policy: Exclusive access. Only one user at a time may use the whiteboard.
- Resource Coordinator: *RC3*
  Policy: Exclusive access. Only one user at a time may use the private audio channel.

6. Related Work

In this section we focus on those work which can be directly related to our research. We introduce SODA [45] and MASE [35], two methodologies which share with our paper the idea of eliciting, modeling and analyzing coordination starting from the initial artifacts and during the all software development process. We then describe IWIM [4], as the coordination model (representative of a class of models called process-oriented [38]) which can be more closely related to the one we use.

SODA is an agent-oriented methodology for the analysis of Internet-Based applications as multi-agent systems. SODA can be considered to be the first methodological approach which elicits coordination issues as first class entities and propagates this information to the analysis and design steps. It defines a conceptual framework and a software engineering procedure for the analysis and design of societies and environments of cooperating agents. Even though SODA is more focused on an agent perspective, we share with this methodology some important aspects: resources are explicitly modelled in our approach as well and our interaction model, given by coordination policies, is also explicited in SODA by interaction rules and interaction protocols. We are not so detailed in the description of the interaction model, since we put ourselves in the first stage of the software lifecycle, when the system architecture is not yet defined. Finally, we do not have a role model as SODA: our tasks are system functionalities which will be detailed in a system design as it will become available. The result of SODA analysis is comparable to ours: a set of high level abstractions and their mutual relationships, providing the designers with a semi-formal description of the intended overall application structure and organization.

MASE is a methodology to engineer “multiagent” systems. An agent is an autonomous, cooperative, reactive, and pro-active entity [44], which requires to be coordinated when interacting with other agents. A multiagent system is a collection of agents working together in order to achieve specific goals. MASE identifies agent types, describes the interactions among them, defines coordination protocols for each interaction and maps agent interactions into internal components. MASE can be strictly related to our work since UML-like diagrams are used in order to describe the structure of the system. The main difference is that we model actors, tasks, resources and coordinators while in the MASE methodology, the focus is more oriented on agents. Independently from this difference, in both cases a stereotyped Class diagram defines which elements can communicate in a hierarchical fashion.

Moreover, in both cases a behavioral language is used to describe the dynamic interaction between design elements. By the way, this methodology provides a way to model coordination, and not a way to elicit it from system requirements (as we do).

IWIM is an exogenous coordination model which supports *separation of concerns* and *anonymous communication*. Its coordination model is described through processes, ports, channels and events and it makes a strict separation between coordination and computation (as we do). "Worker" components are responsible for computation while "manager" components coordinate workers and other managers, by assuming that workers can communicate exclusively through managers. In our approach, we use the Coordinator diagram in order to provide a model-based specification of flow and resource coordinators. Differently from IWIM, at this stage, we do not define concepts like channels, events and ports. In fact, we leave the identification of such elements to the architecture level of the software description. Moreover, what we propose is a methodological approach to elicit, model and analyze coordination requirements while IWIM is a coordination model which does not provide any feature enabling software engineers to elicit coordination requirements. In fact, processes are coordinated assuming a-priori knowledge on how they synchronize/interact.

Our choice of analyzing coordination entities, structure and policies recalls the distinction between "organizational rules", "organizational structure" and "organizational patterns" proposed by Zambonelli et al. in [46]. They use three different organizational models to describe global constraints (what a system of agents can/cannot do), the topology of the interaction and communication patterns, respectively. Their organizational structure can be somehow compared with our concept of coordination structure.

Our choice of describing both structural and behavioral coordination resembles the one done by Linda [2] and Gamma [9] since they all make a distinction between *coordinable entities* and *coordination laws*.

We also mention the FIPA Agent-based Unified Modeling Language (AUML) [7] project which extend OO tools and notations in order to capture unique features of MAS.

In Aspect Oriented Programming (AOP) [18], special concerns are identified and concentrated into different, separate units of decomposition, called "aspects". AOP handles coordination [40] by introducing a coordination aspect, which is implemented by modules that are finally composed with the source code. Generally, AOP and coordination are handled during low-level design. This seems to be one of the most important difference with our approach. By the way, further investigations are required to better investigate such issue.

Research themes that can be related to our entire framework are those on integrating Coordination and Software Architectures [27,20,13] and consistency checking. Research on the former theme, try to push coordination issues inside a software

architecture description and to show how coordination languages may be effectively used to design and analyze SAs. Differently from what we do, there is no effort spent to analyze how coordination-aware architectures can be used to produce a coordination-aware implementation. The latter theme is part of the general theme of system (in)completeness and (in)consistency which has received a lot of attention in the last few years, see e.g. [29,28]. This theme is strongly related to what we propose since the third step of our framework is used to check the conformance between coordination requirements and software architectures (i.e., to check if the coordination requirements are correctly implemented by the software architecture specification).

At the best of our knowledge, our approach is the first attempt to produce a UML-based coordination-aware software development, relating coordination, software architectures and low-level design. UML profiles for distributed systems coordinated via Tuple Spaces [6], and for the development of multimedia and collaborative applications in open distributed environments [36] have been proposed. Both of them strongly differ from our approach.

7. Conclusions and Future work

The general framework, outlined in Section 1, may be summarized as follow: in the first step, the coordination structure and the policies are extracted and used, in step2, to generate an architectural model. The SA behavior is successively checked with respect to the coordination requirements previously identified (step3). Finally, the SA model is used to derive a lower-level coordination model in the fourth step.

An important and new part in the framework is represented by its first step: how to elicit coordination policies from system specifications. This paper focussed on this step, showing, through a case study, how resources and processes may be coordinated in an incremental way. The approach we proposed, initially analyzes how to identify the elements to be coordinated (i.e., the coordination entities), then groups them together defining a sort of communication pattern (i.e., the coordination structure) and finally extracts the coordination laws (i.e., the coordination policies) the coordinators have to implement. Two different UML-like Class diagrams are produced, in order to provide a model-based specification of coordination. We decided to use two different diagrams, in order to be conform to the well accepted multi-view modeling approach [3142] which claims that different system perspectives should be modeled using different diagrams. We adopted a stereotyped description of UML diagrams in order to made our description fully compliant to UML.

The approach makes a clear separation between coordination and computation, adopting the so called “exogenous” model of coordination [3] in which, coordination modules are created separately and independently from the computation modules they are supposed to coordinate. We choose this coordination model since it is easily mappable into a UML-based process, it is closer to concepts used by the software architecture community (i.e., components and connectors), and it has been used to

implement coordination-aware software systems [10].

Our approach is inspired by what we believe are important concepts in coordination:

- Separation between *static and dynamic coordination*: an elicitation process for coordination requirements has to identify the system entities to be coordinated and the coordination laws dictating how they synchronize, communicate and depend on each other. We proposed different UML-based diagrams and informal questionnaire to identify both aspects.
- Identification of *flow and resource coordination*: as advocated by many papers, coordination needs to take into account how processes depend on each other and how resources are shared. We defined a Coordination diagram with the goal of identifying and separating such concepts.
- Separation between *coordination and computation*: we follow this coordination model since we believe it is easily implementable by software architectures. In fact, software connectors are usually seen as architecture-level coordinators.
- *Hierarchical coordination structure*: a unique, global, coordinator disables separation of concerns, modularity and reuse.

With this approach, our aim was to provide the software engineer with a methodology which explicitly takes care of coordination aspects, in order to drive the design and the realization of the entire system. Since those aspects are often left for the latest stages of software development, we believe that reasoning on such aspects, even from the requirements phase, can put in evidence problems which could not be easily fixed later. The use of diagrams for identifying system entities and the coordination structure, is useful for driving the design phase where the architecture of the system is identified. Moreover having a hierarchical structure, gives a high degree of flexibility when the software engineer must face changing requirements.

In the following, we describe ongoing and future work. Section 7 describes how the proposed approach can be extended. Section 7 shows our working directions related to the general framework.

7.1. Future work on the approach

Ongoing and future work, on the proposed approach, go in the following directions:

- to find a *formal or semi-formal notation to specify the coordination policies.* In previous work, we used Sequence diagrams to specify the choices made by the coordinators (as shown in Figure 2, Step1b). Unfortunately, UML sequence diagrams seem to be not sufficient. They describe a possible scenario of coordination, not the general rule governing how actors, Use Cases and resource are coordinated. Some

extensions can be thought and will be analyzed in future work. We could use different UML diagrams (like sequence, state, activity, deployment), each one expressing one particular "view" [21] of the coordination. This idea is still to be evaluated in future research. A possible alternative is to use a process algebra to model the flow coordination and some specific language for the resource coordinators;

- to understand the flexibility of this approach with respect to requirement changes. Since the approach is incremental, we guess that by modifying some requirements, only a specific portion of the coordination structure needs to be modified;

- to extend this approach to different application domains. In CIS, for example, actors and groups, processes and shared resources are again core elements, which require to be elicited and coordinated;

- to *implement a tool* that, starting from Use Case and Entity diagrams (expressed in the XML format),

1. automatically extracts the questions of interest for each step. By analyzing the diagrams, the tool can easily identify the actor types, the number of their instances, the resources and processes. Using this information, the general questions proposed in Section 6 may be instantiated to the system;
2. visualizes the questions;
3. provides the user an environment to edit the specifications.

7.2. Future work on the general framework

Ongoing work on the framework will analyze how the coordination information captured by this approach, can be used to identify an architectural description, refining the framework second step (architectural definition). The idea we have in mind is to map the coordination structure and communication pattern we identified, into architectural elements. Architectural components need to be identified following traditional SA decision processes and reusing information captured by Entity diagrams. Architectural connectors have to implement flow and resource coordinators. SA components and connectors need to be combined together following the coordination pattern identified in the Coordinator diagram.

A future work idea is to use the produced coordination-aware software architecture in order to develop real code. Papadopoulos et al. [39] have proposed and implemented an approach to map ACME [1] architectural concepts into the coordination language Manifold [5]. Architectural concepts, formalized in ACME, are

syntactically compiled into Manifold code and this information is enriched with coordination concepts. A tool has been proposed in order to automatically realize this mapping.

Our medium term goal is to integrate our research with theirs, so that, coordination-aware software architecture produced with our approach can be directly implemented in Manifold.

Acknowledgments

The authors would like to acknowledge the Italian M.I.U.R. national project SAHARA that partly supported this work and the anonymous reviewers for their constructive help to improve the quality of this paper.

References

1. The ACME Framework. On-line at: http://www.cs.cmu.edu/∼acme/.
2. S. Ahuja, N. Carriero and D. Gelernter. Linda and Friends. *IEEE Computer*, vol. 19, no. 8, August 1986, pp. 26-34.
3. F. Arbab. What Do You Mean, Coordination? In the March '98 *Issue of the Bulletin of the Dutch Association for Theoretical Computer Science* (NVTI).
4. F. Arbab. Coordination of massively concurrent activities. *CWI Report* CS-R9565 (1995).
5. F. Arbab, I. Herman and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, Vol. 5, No. 1, February 1993, pp. 23-70.
6. E. Astesiano and G. Reggio. UML-SPACES: A UML Profile for Distributed Systems Coordinated Via Tuple Spaces. In IEEE Proc. Fifth Int. Symp. on Autonomous Decentralized Systems, March 2001, Texas.
7. The FIPA AUML Projects. On-line at: http://www.auml.org/.
8. D. Ballarini, M. Cadoli, M. Gaeta, T. Mancini, M. Mecella, P. Ritrovato and G. Santucci. Modeling Real Requirements for Cooperative Software Development: A Case Study. In Proc. 2nd Work. on Cooperative Supports for Distributed Software Engineering Processes, March 2003, Benevento, Italy.
9. J. -P. Banâtre and D. Le Métayer. GAMMA and the Chemical Reaction Model: Ten Years After. In J-M. Andreoli, C. Hankin and D. Le M?tayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*. World Scientific, 1996, pp. 1-39.
10. L. Barroca, J. L. Fiadeiro. Coordination Contracts as Connectors in Component-Based Development. in Proc. Integrated Design and Process Technology, IDPT-2002, June 2002.
11. B. Bauer, J. P. Muller, J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In Int. *Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 3, pp. 1-24, 2001.
12. P. Ciancarini, A. Knoche, R. Tolksdorf and F. Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. *Computer Networks and ISDN Systems*, 28(7-11):941-952, 1996.
13. P. Ciancarini and C. Mascolo. Model Checking a Software Architecture. In Proc. Int. Workshop on the Role of Software Architecture in Testing and Analysis, *Rosatea98*, Italy.
14. P. Ciancarini and D. Rossi. Jada: Coordination and Communication for Java agents.

In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, Vol. 1222 of *LNCS*, 1997.
15. Coordination 2000. Proc. Fourth *Int. Conference on Coordination Languages and Models*, Coordination 2000, Cyprus, 2000. LNCS 1906.
16. K. Crowston. Towards a Coordination Cookbook: Recipes for Multi-Agent Action. Ph.D. Dissertation, MIT Sloan School of Management, Cambridge, MA.
17. S. Ducasse, T. Hofmann and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In Proc. [15].
18. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. Communications of the ACM, vol. 44 n. 10, Oct. 2001.
19. G. Ferrari, C. Montangero, L. Semini and S. Semprini. Mobile Agents Coordination in Mob_{adtl}. In Proc. [15].
20. D. Hirsch, P. Inverardi and U. Montanari. Reconfiguration of Software Architecture Styles with Name Mobility. In [15].
21. C. Hofmeister, R. Nord and D. Soni. *Applied Software Architecture.* Addison-Wesley, 1998.
22. G. Holzmann. The SPIN web site. On line at: http://cm.bell-labs.com/cm/cs/what/spin/index.html.
23. P. Inverardi and H. Muccini. Coordination models and Software Architectures in a Unified Software Development Process. In Proc. [15].
24. P. Inverardi, H. Muccini and P. Pelliccione. Automated Check of Architectural Models Consistency using SPIN. In ACM *Proc. Automated Software Engineering Conference* (ASE 2001), November 2001.
25. P. Inverardi, H. Muccini and P. Pelliccione. Checking consistency between architectural models using SPIN. On the First ICSE'01 Int. Workshop *From Software Requirements to Architectures* (STRAW'01), May 2001.
26. P. Inverardi and H. Muccini. Software Architectures and Coordination models. In the Journal of Supercomputing (Kluwer), Vol. 24, No. 2, Feb. 2003.
27. P. Inverardi, A.L. Wolf and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. Proc. Second Int. Conference on Coordination Languages and Models, Coordination 97, Germany, 1997. Lecture Notes in Computer Sience, 1282.
28. IWLI: Second International Workshop on Living with Inconsistency. ICSE'01 workshop, May 13, 2001 Toronto, Canada.
29. IWSSD-10: Tenth International Workshop on Software Specification and Design. Track on (In)completeness, (in)consistency and (sufficient) correctness. November 2000, San Diego, California.
30. JavaSpaces. On-line at: http://java.sun.com/products/javaspaces/.
31. P. Kruchten. Architectural Blueprints - The "4+1" View Model of Software Architecture. IEEE Software, 12(6) November 1995, pp. 42-50.
32. P. Kruchten. The Rational Unified Process: An Introduction. Addison-Wesley Pub. Co. 2nd edition, March 2000.
33. A. van Lamsweerde. Requirement Engineering in the Year 00: A Research Perspective. In Proc. of *The Future of Software Engineering*, FOSE2000, part of ICSE2000, June 2000.
34. T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. ACM Computing Surveys, 1994 (March), 26 (1), 87-119.
35. Multiagent System Engineering (MASE). Research Project, Kansas State University. Project web site: http://www.cis.ksu.edu/~sdeloach/ai/mase.htm.
36. The MultiTEL project. On-line at: http://polaris.lcc.uma.es/~lff/MultiTEL/.

37. B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In Proc. of "The Future of Software Engineering", FOSE2000, part of ICSE2000, June 2000.
38. G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, Vol. 46 of Advances in Computers. Academic Press, 1998.
39. G. A. Papadopoulos, O. Papapetrou and A. Stavrou. An Implementation Framework for Software Architectures Based on the Coordination Paradigm. Internal Report.
40. M. Pinto, L. Fuentes, M. E. Fayad, and J. M. Troya. Separation of Coordination in a Dynamic Aspect Oriented Framework. In the 1st International Conference on Aspect-Oriented Software Development, April 2002.
41. M. Ramage. The Learning Way: Evaluating Co-operative Systems Ph.D. Thesis, Department of Computing, Lancaster University, April 1999.
42. IBM Rational Corporation. Uml Resource Center. On-line at: http://www.rational.com/uml/.
43. I. Sommerville. Software Engineering. Addison-Wesley Pub. Company. 6th edition (August 11, 2000).
44. M. Wooldridge and N. Jennings. Intelligent Agents: Theory and Practice. Knowledge Engineering Review, 10(2): 115-152, 1995.
45. A. Omicini SODA: Societies and Infrastructures in the Analysis and Design of Agent-based systems AOSE 2000, year 2000.
46. F. Zambonelli, N. Jennings and M. Wooldridge. Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems. International Journal of Software Engineering and Knowledge Engineering, Vol. 11, Number 3, June 2001.