Eliciting Coordination Policies from Requirements [*]

Henry Muccini[†]
Dipartimento di Informatica
Universitá degli Studi Dell'Aquila
67100 L'Aquila, Italia
muccini@di.univaq.it

Fabio Mancinelli
Dipartimento di Informatica
Universitá degli Studi Dell'Aquila
67100, L'Aquila, Italia
mancinel@di.univaq.it

ABSTRACT

Software coordination models and languages describe how agents, resources and processes work together to implement a software system. One of their limitations is that they are used late in the software development and they are not integrated in a typical software development process.

What we claim, with our research, is that if coordination becomes explicit and formalized as soon as possible in the life cycle, then it is possible to create coordinated-aware software systems. Moreover, it is possible to verify the adequacy of a Software Architecture (SA) model (or of the code itself) with respect to these dynamic constraints as well as refine or disambiguate coordination requirements themselves.

In previous work, we presented a UML-based development process to elicit, describe, analyze and validate system coordination properties that might be then specified with a suitable coordination language. In this general picture, the aim of this paper is to implement the first step, i.e., to elicit and formalize coordination policies. We propose a five steps approach that incrementally identifies the elements to be coordinated (i.e., static coordination) and how these entities may be coordinated (i.e., dynamic coordination).

Keywords

Coordination Policies Elicitation, Coordination and Requirements, Software Process

1. INTRODUCTION

Coordination, in software systems, has been recognized to be an important area in software engineering "with the goal of finding solutions to the problem of managing the interaction among concurrent programs" [4]. Many coordination-specific languages and models have been proposed in the last years (and surveyed in [31]) in order to describe parallel and distributed systems from the interaction point of view, abstracting away the computation and focusing on how constraints and conditions have to be selected to ensure correct interactions.

Despite the research done in this area and the many models and languages proposed by this community, it seems that *i*) coordination reasoning and software development do not converge in order to produce real, coordination-aware systems; while new development paradigms are proposed to design and implement even more distributed software systems (e.g., component-based programming, software architecture, COTS), coordination languages and models maintain their own specific range of application, are not usually integrated in the software production process and usually[1] produces results that do not flow in the software product. Moreover, *ii*) coordination is explicitly described in a *late development stage*, i.e., during the coding stages of a system development. Many important design choices related to the way components (i.e., agents) interact, are *already encapsulated* in the system Requirements and in the high-level, architectural specification but described only at the code level. What we claim is that coordination issues have to emerge as soon as possible since they often exist from the very beginning and can influence, in a hidden way, architectural and design choices.

What we proposed in previous work is a framework[2] to encapsulate the coordination inside a software development process and enabling elicitation, description, analysis and validation of system coordination properties.

The framework proposed a UML-based software development process that starting from initial software artifacts, e.g. Use Case diagrams, allows to *i)* identify the elements to be coordinated, elicit and specify coordination requirements in order to define coordination policies; *ii)* use these elements to drive the Software Architectural (SA) description; *iii)* check the conformance of the architectural description with respect to the coordination requirements and finally *iv)* use the SA to drive the construction of the lower level coordination model.

The latter three steps have already been analyzed in other

[*] An extended version of this paper is available in [29]

[†] Currently on leave at University of California, Irvine

[1] fortunately, some cases are outside this discussion, like Linda [2] and Manifold [5]. Linda has been integrated in Java (to obtain Jada [11]) and its tuple space coordination model has been implemented in JavaSpaces [27] and in WWW servers, like PageSpace [10]. Manifold code has been parsed into C language.

[2] in the following, the term "framework" will be used to denote the complete picture while the term "approach" identifies each single step in the framework.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC 2003, Melbourne, Florida, USA
Copyright 2003 ACM 1-58113-624-2/03/03 ...$5.00.

papers [20, 22, 23]. Those papers are motivated by the need to clearly distinguish the characteristics and the roles that Software Architecture and Coordination have in the development process.

However, the intent of this paper is to focus on the first one, i.e., *how to make coordination policies explicit by means of an analysis of the system requirements and high level specifications.* If coordination policies become explicit and formalized it becomes possible to verify the adequacy of a SA model or of the code itself with respect to these dynamic constraints [30, 28] as well as refine or disambiguate coordination requirements themselves.

As in Linda [2] or Gamma [6], *coordinable entities* and *coordination laws* are identified so we identify the elements to be coordinated (called static coordination) and how they coordinate (dynamic coordination). The approach is then applied to the Whiteboard system (described in the following).

The paper is structured as follows. In Section 2 an overview of the framework is presented. Section 3 is devoted to describe the Whiteboard specifications while Section 4 focuses on the approach, describing how we carry on elicitation and formalization of requirement constraints from Use Cases. In Section 5, the approach is applied to the Whiteboard system. Section 6 presents some related work while Section 7 summarizes the contribution of the present paper and proposes further research directions.

2. THE FRAMEWORK

This section summarizes the general framework/process which elicits coordination requirements from system specifications and Use Case diagrams, models coordination information using UML diagrams and drives the other steps in the life-cycle. We opted for UML models in order to use a notation that is familiar to software engineers and already in use in software development processes.

The framework is structured in four steps and used to motivate/contextualize the first step, described in depth in the following of this paper.

1. *Coordination requirements are elicited and formalized:* (Section 4)

   what we claim is that software specifications contain information on entities, processes and resources and on how they coordinate, but do not make them explicit. What we propose as the starting point in our coordination process is a way to identify *what* is needed to be coordinated (i.e., actors, processes and resources) and *how* these objects coordinate. This separation puts in evidence two complementary aspects in system coordination: *static* and *dynamic* coordination.

   UML Use Case diagrams will be used as the way to profitably analyze system specifications in order to elicit static and dynamic coordination information. Coordination policies are then described, putting in evidence how actors, processes and resources coordinate (i.e., dynamic coordination);

2. *SA is built upon Requirements:* ([20, 23])

   the formalization we did in the first step is then used to drive the SA specification. To describe the SA of a software system, the static coordination model is used to identify architectural *components* and *connectors* (i.e., the static description); the dynamic coordination model helps to understand and how architectural components interact through the connectors (i.e., the dynamics).

3. *SA is analyzed with respect to requirements:* ([22, 21])

   the SA model is checked with respect to coordination requirements to discover modelization problems. Thus, the coordination specification is assumed to be the expected behavior while the architectural model one of its possible implementation. The Charmy framework (presented elsewhere [22, 21]) has been used to verify if an architectural model correctly implements coordination requirements.

   The approach makes use of the model checker SPIN [19], its language Promela [19] and Linear time Temporal Logic (LTL) [19] formulae. More information, examples and the tool supporting the approach are described in [22, 21].

4. *Mapping SA to low-level coordination models:* ([20, 23])

   the SA model can be used to drive the construction of a coordination model. The idea was to create a mapping among SA elements and Coordination languages entities.

   In [23] a sort of structural mapping among SA components, connectors and ports and IWIM [3] processes, ports and channels are sketched.

The framework uses an "exogenous" [4] model of coordination in which coordination modules are created separately and independently from the computation modules they are supposed to coordinate. We choose this model since it is closer to concepts used by the software architecture community (i.e., components and connectors) and it is easily mappable into the UML-based process. Figure 1 summarizes the framework.

In the following section we will introduce the Whiteboard case study that will be used in Section 5 to put in practice the approach we are going to describe.

3. THE WHITEBOARD EXAMPLE

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. We want to develop a system in which remote users may 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 open 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 location using the network. Users

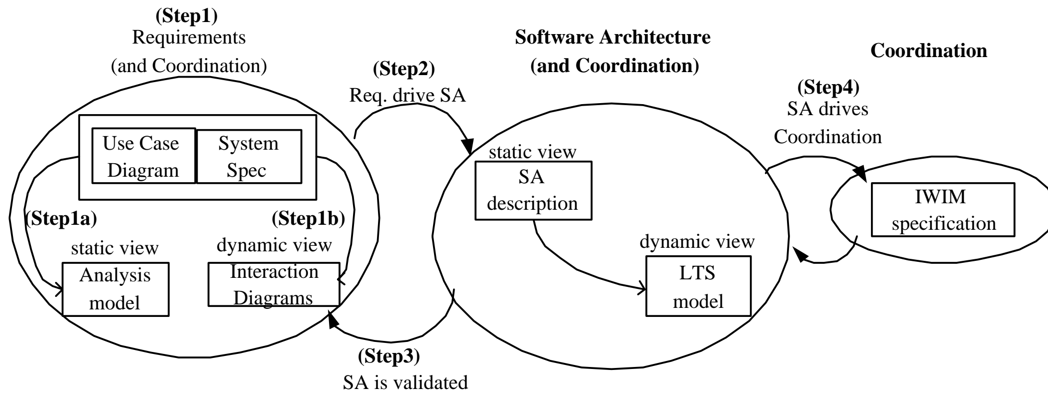

Figure 1: Summarizing the Process

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 giving priorities 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 one of the private audio channel. Anyway, for each user there could be only one private audio channel active at any time.

The system *administrator* (which administrates the whole system, creating the available conferences and assigning the chairs to those conferences) does not have access to the conferences but can only create, remove conferences and change conferences attribute. It resembles mostly the root user of a standard UNIX system. Operation on a conference can be carried on only when a conference is closed.

4. STEP1: COORDINATION REQUIREMENTS ELICITATION AND FORMALIZATION

In the Unified Software Development Process (UP) [26], Use Cases drive the evolution of each development step. They capture functional requirements while 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 [32] (i.e., types) that can be instanciated 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 instanciated by Frank, John and Greg who can run multiple times the same Use Case.

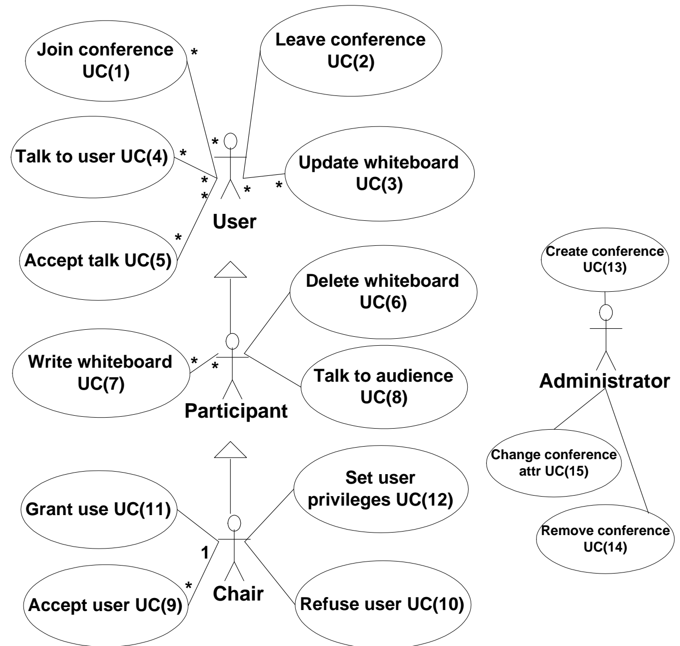

Figure 2: Use Case diagram of the Whiteboard system

Figure 2 shows the Use Case diagram for the Whiteboard example: *Standard user*, *Participant*, *Chair* and *Administrator* are the actor types; the Chair is a “generalization” [32] of the Participant (i.e., an instance of Chair can communicate with the same kinds of Use Case instances as an instance of Participant). The same relationship exists between Participant and Standard user. The Use Cases model the Whiteboard functionalities described in the Whiteboard informal specification in the previous section. The notation UC(i) will be used in the following to identify different Use Cases. The (*,*) multiplicity among an actor and its Use Cases states that each actor may run several instances of a Use Case and each use case instance may be initiated by multiple actors. The (*,1) multiplicity among an actor and its Use Cases declares that only one instance of the Use Case may be run by each instance of actor.

In the following subsection we will show how Use Case

diagrams may be used to elicit coordination requirements. We do not argue that this simple diagram may elicit requirements; it is not so informative. What we claim is that they may drive the analysis of requirements specification in a way that enables the coordination policies elicitation.

We structure the description of this step into two different subsections in which coordination entities are identified and formalized (static coordination, Section 4.1) and then coordinated (dynamic coordination, Section 4.2).

4.1 Static coordination

Our claim is that to coordinate software systems at least three (classes of) entities must be taken into consideration:

- *functional* entities, that identify the processes to be coordinated;
- *resource* entities, that will be accessed by functions;
- *agent* entities, that are able to run functions.

As in Linda active tuples or in Gamma chemical reactions are the coordinable entities, in software systems processes, resource and agents are the coordinable entities.

In the following we describe how Use Case diagrams and the UP may be used to identify these entities.

Following the UP, Use Case diagrams may be analyzed in order to generate an "Analysis model" [26]. An Analysis model is a stereotyped[3] UML class diagram describing *how* a specific Use Case is realized in terms of "abstract" cooperating classes and associations between them. Classes, in this model, always fit one of three basic stereotypes: *boundary*, *control* or *entity* [26]. Boundary classes represent abstractions of windows, forms, communication interfaces; entity classes reflect logical data structures; control classes represent coordination, sequencing, transactions and control of other objects and are often used to encapsulate control related to a specific Use Case.

To create this model a set of Use Cases is selected and the Analysis classes that participate in their realization are identified. For example, Figure 3 illustrates the Analysis model for Use Cases UC(1), UC(3), UC(4), UC(7), UC(9) in Figure 2). UCC(i) denotes the control class used to coordinate UC(i) and will be referred as Use Case Coordinator in the following. Boundary classes have been not drawn, since they are not of interest for our approach.

Taking advantage of this model, we can discover the functional, resource and agent entities (i.e., the elements to be coordinated):

- *actors*, in the analysis class model, represent the *agents* to be coordinated,

- the *entity classes* represent the *resources* the system may use for its execution (shared among agents),

- Use Cases identify high-level functions, initiated by Actors and making use of resources. *Control classes* are responsible for the Use Cases' coordination.

Notice that we are not introducing new stereotypes. Instead, we are reusing existing stereotypes in order to describe coordination oriented information.

So far, we know what entities the system has to coordinate. What remains to analyze is how they coordinate, i.e.,

[3]stereotyped [32] means that a new semantics is associated to the elements in the diagram. Notice that this notation is still part of UML.

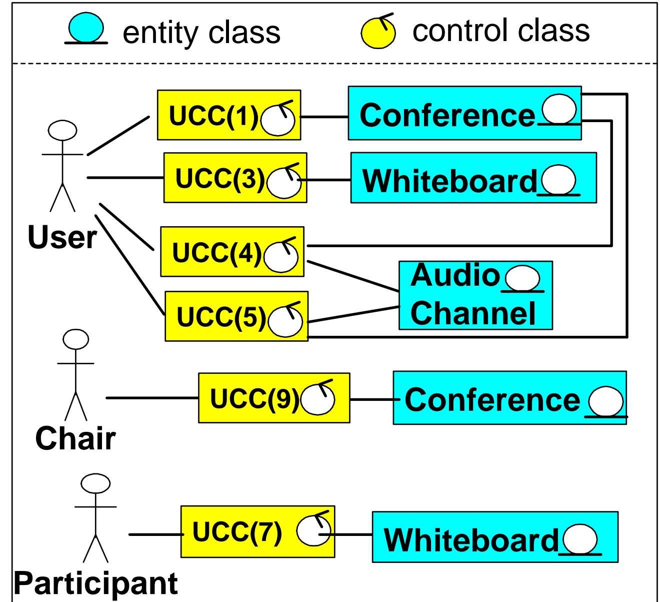

Figure 3: Analysis model for UC(1), UC(3), UC(4), UC(7), UC(9)

how control classes, in the Analysis model, act in order to coordinate those entities.

What we propose in the following is an approach that, given the Use Cases Analysis model, Use Case diagram and system requirement specifications, combines them in order to put in evidence the coordination among agents, resources and processes. We do not use the UP methodology for combining analysis classes together since it does not keep in consideration concepts relevant for coordination.

4.2 Dynamic coordination

In terms of Actors, resources and Use Cases, coordination may be defined as *the rules dictating how multiple Use Cases, initiated by Actors, can collaborate, utilizing available resources.* In these terms, to coordinate a real system implies *to identify how multiple agents may run multiple instances of each Use Case making access to available resources*, that is, different actor types and different actor instances $(\mathrm{Act(k)}_l)$ need to be coordinated when they run different Use Case types and different Use Case instances $\mathrm{UC(i)}_j$ competing to access a set of available resources R(m).

From this definition, two conclusions may be drawn: *i)* coordination encompasses two main factors: *flow* coordination and *resource* coordination. With the term "flow coordination" we mean all those coordination policies that 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 accessed by many instances of several Use Cases. *ii)* Analyzing all these different parameters in a big-bang approach could become very complex.

What we propose in this paper is a divide et conquer approach that, in five steps, refines the initial coordination model and analyzes both flow and resource coordination issues. At each step, a coordinable entity is analyzed (i.e., resource, multiple instances of the same Use Case, multiple

Use Cases, multiple instances of the same actor, multiple actors) and specialized components FCi and RCi are introduced to coordinate flows and resources, respectively. In this way, we realize an "exogenous" model of coordination. The five steps, graphically presented in Figure 4, follow:

Step1 given the system Actors and the Use Cases, this step identifies the association Uses Cases - resources. For each Use Case i (UC(i)) this step analyzes what resource (R) UC(i) needs to accomplish its task. If multiple instances of the same resource are available (e.g., multiple printers) this step is responsible to distribute these resources to the Use Cases. For each UC(i), a $FC1_i$ coordinator is introduced to implement the policy UC(i) uses to access the resources;

Step2 coordinates multiple instances of the same Use Case UC(i). If the same actor can execute multiple instances of the same Use Case, we need to identify how those executions have to be coordinated. FC2 coordinators are used to embed the coordination policy. For example, in the Whiteboard example, the same User can make a second talk ($UC(4)_2$) only if previous talks have been closed. The RC2 coordinators identify how the same resource may be accessed by multiple instances of the same Use Case. For example, the AudioChannel resource can be used twice by the same User only if it is not already used. Figure 4, Step2, graphically describes this step;

Step3 coordinates different Use Cases run by the same actor. Since the same actor may run different Use Cases, we need to analyze how these Use Cases need to be coordinated (e.g., concurrent or sequential execution). Component FC3 is responsible to realize the coordination between different Use Cases. For example, in the Whiteboard example, the User may run UC(4) only if UC(1) has been run and UC(2) did not apply after UC(1). RC3 coordinates the access from different Use Cases to the same resource. For example, RC3 embeds the policy dictating how the Conference resource is accessed by UC(1) and UC(4);

Step4 analyzes how multiple instances of the same actor can be coordinated. Antonio and Peppe (i.e., $actor(k)_1$ and $actor(k)_2$ in Figure 4, Step4), instances of User, run their own Use Cases. Their executions need to be coordinated. FC4 contains information on how their processes can be coordinated (e.g., First come first served) while RC4 analyzes how the same resource is accessed by their processes (e.g., how Antonio and Peppe can talk to other users (UC4) competing on the same resource AudioChannel);

Step5 points out how different actors are coordinated. Finally, we consider the last piece in this (incremental) puzzle. FC5 implements the coordination relationships between different actors (e.g., Participant1 and User2 want to join the conference) and RC5 examines how the same resource is shared among different actors (e.g., how the Whiteboard resource is shared among a Chair and a User).

Figure 4 shows the various steps and how they are integrated in a general picture encompassing all the different aspects; notice that for each Use Case/actor pair, we have a corresponding FC1 coordinator in step1 and for each shared resource m, we have a $RC2_m$ component in step2. Flow coordinators at level $i+1$ (FC$i+1$) coordinate flow coordinators at level i while resource coordinators (for each m) are refined step-by-step (i.e., $RC5_m$ is a refinement of RCi_m, with $1 < i < 5$).

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

FC1 *a Use Case can make decisions on what resources it can use* (e.g., how UC(4) can access AudioChannel and Conference resource in Figure 3);

FC2 *different instances of a Use Case would be coordinated* (e.g., how the same User can run multiple times UC(4) in Figure 3);

FC3 *different Use Cases interact* (e.g., how User can coordinate the use of UC(3) and UC(4) in Figure 3);

FC4 *different instances of Users run different Use Cases* (e.g., how N Users compete to update the whiteboard in Figure 3);

FC5 *different Users run different Use Cases* (e.g., how Participant and User can be coordinated in order to join the conference in Figure 3).

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

RC2 *the same resource is assigned to multiple instances of the same Use Case* (e.g., how the AudioChannel resource is accessed by multiple requests of UC(4) by the same User in Figure 3);

RC3 *the same resource is shared among different Use Cases* (e.g., how the Conference resource is accessed by UCC(1) and UCC(4) in Figure 3);

RC4 *the same resource is accessed by different instances of the same actor* (e.g., how User1 and User2, instances of User, can access the same resource);

RC5 *the same resource is shared among different actors* (e.g., how the Whiteboard resource is shared among Chairs and Users in Figure 3).

Moreover, this definition makes clear that a coordination function should encompass five different parameters and can be schematized as follow: $Act_k(l)$, UC(i,j), R(m), where:

- "Act" identifies the system actors and has two parameters: "k" that identifies the actor type (e.g., User, Participant and Chair in Figure 2) and "l" to describe how many instances of each actor are involved;

- "UC" identifies the Use Cases: parameter "i" enumerates the different Use Cases implemented in the system (i=4 in the example) and "j" denotes how many times the same UC may be called by the same user (j=1 for the Registration UC, j=* for the other UCs);

- "R" identifies the resources used to realize the UCs: "m" enumerates the different resources.

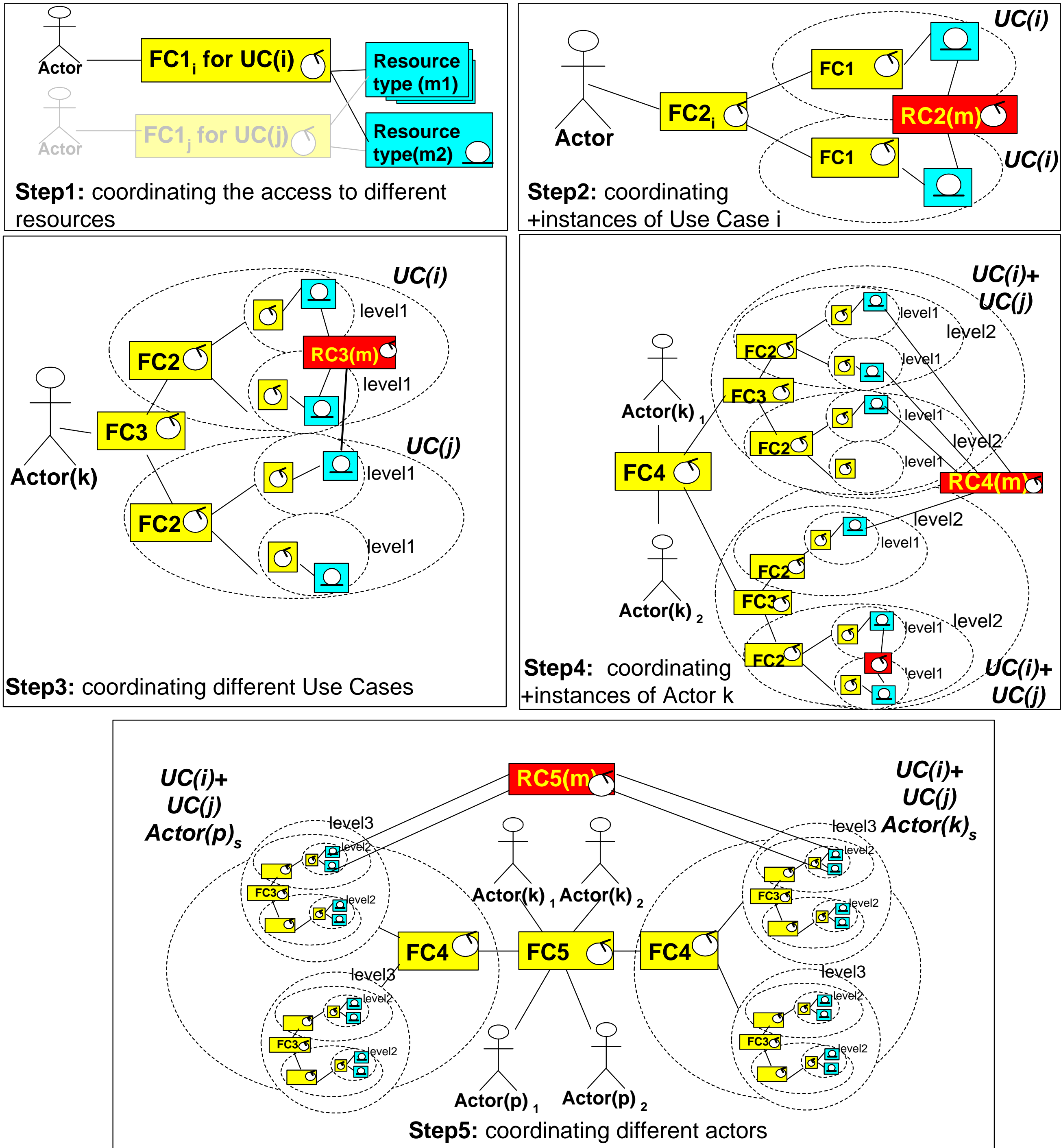

Figure 4: The five steps and their integration

At this point, we identified what 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.

Step1: *Resources assignment.*
- Flow Coordinator FC1:
Q: What 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?

Step2: *Coordination of multiple instances of the same Use Case.*
- Flow Coordinator FC2:
Q: How many instances of the same Use Case, an actor can call?
Q: In which order these functionalities will be executed?
- Resource Coordinator RC2:
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?

Step3: *Coordination of different Use Cases.*
- Flow Coordinator FC3:
Q: What Use Cases can be executed by the same actor?
Q: How do Use Cases interact?
- Resource Coordinator RC3:
Q: If different Use Cases UC(i), UC(j) can access the same resource R, how their access have to be regulated?

Step4: *Coordination of multiple instances of the same actor.*
- Flow Coordinator FC4:
Q: How many instances of the same actor can populate the system?
- Resource Coordinator RC4:
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?

Step5: *Coordination of different actors.*
- Flow Coordinator FC5:
Q: How many Uses Cases and resources are shared among different actors?
Q: Are there any privileges on some of the actors?
- Resource Coordinator RC5:
Q: If different actors can make use of the same resource R, what is the policy regulating the access to R?

5. APPLYING THE APPROACH TO THE WHITEBOARD EXAMPLE

In this section we will apply the approach using the Whiteboard system specification (in Section 3), the Use Cases pictured in Figure 2 and the Analysis model in Figure 3. We will concentrate only on a small subset of the system functionalities which are sufficient to put in evidence the main aspects of the approach. These functionalities are identified by the following Use Cases:

- *Join conference* (1): Each user must join a conference before doing anything else on the system.
- *Update whiteboard* (3): Each user may receive the actual content of the whiteboard.
- *Talk to user* (4): A user can talk to another user using a private channel.
- *Accept talk* (5): A user who receives a talk request from another user must accept it.
- *Write whiteboard* (7): A participant can write sketches to the whiteboard.
- *Accept user* (9): The chair must authorize a user to join a conference.

5.1 Step1

Given the analysis model in Figure 2 we identify three types of resources: *Conference* which is used to store all the information related to a given conference, *Whiteboard* which is accessed by the users to draw their sketches and *Audio channel* which is used by the users to talk together. The following table summarizes the *Step1* that shows which resources are used by which Use Case and what happen when multiple instances of the same resource is available.

Actor	**UC**	**Resource**	**FC1**
User	1	Conference	Input Only one active
User	3	Whiteboard	Input Only one active
User	4	Conference Audio channel	Input Only one active Input Only one active
User	5	Conference Audio channel	Input Only one active Input Only one active
Participant	7	Whiteboard	Input Only one active
Chair	9	Conference	Only one

The previous table states (in rows 1, 3, 5) that, if more

Conferences are available, a User can choose the conference he wants to join, talk in or write in. The "Only one active" option implies that if a User chooses Conference C1, he can choose another Conference C2, only if C1 has been released. The Chair can supervise and accept users (UC(9)) only in the Conference he is in charge as a chair.

If different conferences are available, multiple Whiteboard and Audio channel resources are available and the actors can decide what to use for their communication.

5.2 Step2

The following table summarizes how multiple instances of the Use Case are coordinated. A multiplicity (m, n) means that an actor may execute n instances of the Use Case, while a Use Case may be executed by m instances of actors.

Actor	UC	Mult	FC2	RC2
User	1	(*, *)	FCFS	FCFS
User	3	(*, *)	FCFS	FCFS
User	4	(*, *)	FCFS	FCFS
User	5	(*, *)	FCFS	FCFS
Participant	7	(*, *)	FCFS	FCFS
Chair	9	(1, *)	FCFS	FCFS

The first four columns in the table state that every actors can execute multiple instances of their Use Cases (expressed in the "Mult" column) and these different instances are coordinated following a First Come First Served (FCFS) policy (expressed in column "FC2"). Notice how the information captured in step 2 differs from that captured in step 1 and how they can be combined: for example, combining the first rows in both tables, we have the policy "a User can join (UC(1)) different conferences, following a FCFS policy, but only one conference at a time". The last column in the table, analyzes how the resources are accessed by multiple instances of the same Use Case: the RC2 in the first row says that if the same Conference resource is required by multiple "join conference" operations (UC(1)), then it can be accessed using a FCFS policy. The same policy applies in the other rows. The application of the approach to the Whiteboard example, turned in FC2 and RC2 coordinators applying the same policy. Notice that this is not a general rule. For example, we can have that the write (UC(7)) processes are scheduled in their FCFS order while the Whiteboard resource is assigned based on the text dimension (or other parameters).

5.3 Step3

The following table summarizes, for each actor, how different Use Cases are coordinated between them. Each row in the table refers to an actor and contains a coordination policy between the different Use Cases associated to that actor.

Actor	UCs	FC3
User	1, 3, 4, 5	If 1 then (3 OR 4 OR 5)
Participant	7	-
Chair	9	-

The flow coordinator FC3 in the first row establishes a coordination policy between the Use Cases UC(1), UC(3), UC(4), UC(5). Basically, it states that the conference must be join (UC(1)) before updating (UC(3)), talk (UC(4)) or accept talk (UC(5)) can be performed. UC(7) and UC(9) do not have any coordination requirements.

Resource	UCs	RC3
Conference	1, 4, 5	Prioritizes UC(4)
Audio channel	4, 5	if 5 then 4
Whiteboard	9	-

The resource coordinator RC3, instead, analyzes how different Use Cases can access the resources. In the Whiteboard system, the Conference resource, when accessed by UC(1), UC(4) and UC(5) prioritizes operation UC(4). The meaning of "prioritize" can be naturally refined. UC(5) needs to access the Audio channel resource before UC(4) in order to implement a correct resource coordination.

5.4 Step4

The following table summarizes how multiple instances of the same actor are coordinated considering the Use Cases they are executing:

Actor	Mult	UC_1	UC_2	FC4
User	*	4	5	If 5 then 4
Participant	*	7_1	$7_{(2)}$	If Participant_1 has higher priority then 7_1 else $7_{(2)}$
Chair	*	9_1	$9_{(2)}$	Exclusive access

In this table the Use Cases are analyzed in pairs. The first row, for example, states that whenever two actors are executing the Use Cases regarding the "talk functionality", UC(4) must follow UC(5) (i.e. a user must accept a talk before actually talking). The Use Cases in this table must be considered using the information provided by the table of the previous step. UC(4) and UC(5), infact, are already coordinated from a single actor perspective. This table extends that coordination as described in the previous section.

The resource coordinators RC4 for Conference, Audio Channel and Whiteboard implement a FCFS policy with an exclusive access.

5.5 Step5

Finally, this step captures the inter-relationships between the coordinated Use Cases executed by different actors. Those relationships are reported in the following table:

Actor1	Actor2	UC act1	UC act2	FC5
User	Chair	1	9	if 9 then 1

UC act$i$ stays for Actor$i$ Use Case. The table puts in evidence that a User can join the conference (UC(1)) only after his acceptance (UC(9)) from the conference chair.

Resource	Actor1	Actor2	FC5
Conference	User	Chair	FCFS
Audio Channel	User	Participant	UC(7) prioritized
Whitebox	-	-	-

From the last table we have that if the Conference is accessed by a User and the Chair to realize UC(1) and UC(9), respectively, a FCFS policy is applied while the Audio Channel gives priority to UC(7) called by the Participant.

Let's summarize the flow coordination in this experiment, going backward from step 5 to step 1: *v)* each User can join the conference only if the chair gives the authorization; *iv)* if multiple Users receive the authorization, they can communicate together only if there is a sequence of Accept.Talk (UC(5) then UC(4)) from different Users; *iii)* each User, in isolation, can perform UC(3), UC(4) and UC(5) only if UC(1) has been previously executed; *ii)* If UC(1) then multiple instances of UC(i) may be run, using a FCFS policy; *i)* finally, when a User has joined a conference Ci, he cannot join other conferences Cj unless Ci has been previously released.

6. RELATED WORK

The coordination research area includes many different topics. Since the framework integrates SAs and coordination in the same process and makes possible analysis and consistency checking, in this paper we relate only work on architecture- and coordination-based software processes and (coordination) properties analysis and checking.

Software Development Process:

The framework is implemented introducing coordination issues in the Unified Software Development Process (UP) [26]. The UP is Use Case driven, Architecture-Centric, Iterative and Incremental. Use Cases drive the evolution of each development step, capturing functional requirements, while Use Case diagrams show how Use Cases (i.e., high-level functionalities) are associated to Actors. Each diagram in the UML implicitly describes some coordination information. What we proposed in this paper is a way to make explicit this coordination information.

Some other development processes have been defined to explicitly deal either with Software Architecture or Coordination:

In [8] Bass and Kazman propose a six-step Architecture-Based Development Process which *i)* elicit the architectural requirements (expressing quality-based architectural requirements through quality-specific scenarios) *ii)* design the architecture (using views), *iii)* document the architecture (considered as a crucial factor in the SA success), *iv)* analyze the architecture (with respect to quality requirements), *v)* realize and finally *vi)* maintain the architecture.

[34] gives the authors view on the SA phase in the development process. Choosing a SA description, it has an impact on the non-functional requirements. The authors evaluate the effects, of the proposed architectures, on timing, scalability, fault-tolerance and extendibility quality properties.

In [33] Rosa et al. analyze why non-functional requirements (NFRs) are rarely taken into account in development processes and introduce a framework, named *Parmenides*, in order to address the problem of explicitly dealing with them.

The development processes summarized share in common the ability to describe quality properties or non-functional requirements, but do not explicitly describe how, starting from the requirement specifications, coordination policies may be captured and formalized for further analysis.

Consistency checking:

The general themes of system (in)completeness and (in) consistency have received a lot of attention in the last few years, showing the growing interest on this area, see e.g. [25, 24]. As outlined in [15] consistency checking among multi-perspective or multi-language specifications is not a young field of research. There are different papers proposing different approaches to handle consistency. In [16] the authors propose an approach for multiple views consistency checking.

The third step in our framework (implemented through the Charmy tool) may be related to both of these papers since what we finally do is to check the conformance between requirements and software architectures. The main difference is that those approaches are not used to check coordination properties but more general model consistency.

Other work:

Some recent work partially covers the issues presented in the framework: [20, 1, 17] show how SA and coordination models may be related, [13] describes how coordination may be described at the design level and [14] proposes a refinement process to describe policies.

At the best of our knowledge, the work we propose is the first trying to elicit coordination requirements from UML [32] diagrams and relates coordination, software architectures and low-level design. Other approaches have been proposed to extend UML with coordination information, like "Agent UML" [9] and various UML extensions. They will be carefully analyzed in future work.

7. CONCLUSIONS AND FUTURE WORK

The general framework, outlined in Section 2 and shown in Figure 1, may be summarized as follow: in the first step, static and dynamic coordination information is 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 (step4), the SA model is be used to derive a lower-level coordination model.

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 iterative way. The approach we proposed initially analyzes how to identify the elements to be coordinated (i.e., the coordinable entities) and then extracts the coordination policies (i.e., the coordination laws) these elements must implement. The approach makes a clear separation between coordination and computation, following an exogenous model of coordination. If a different coordination model is required, the identification of the coordination entities (Section 4.1) needs to be revised while Section 4.2 can still be used to elicit the coordination policies from requirements.

Something important to notice is that even if the approach has been presented to coordinate software applications, it can also be used to coordinate organizational structures (i.e., to define how people can interact to reach certain goals) or software architecture (i.e., how architectural components and connectors can dynamically interact).

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

- to find a larger, industrial, case study to justify the soundness of the approach and to see measure how it performs in real systems;

- to find a *formal or semi-formal notation* to specify the

coordination policies. Initially, we used Sequence diagrams to specify the choices made by the coordinators (as shown in Figure 1, 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. Moreover, they cannot model concurrency and synchronization concepts. We could use different UML diagrams (like sequence, state, activity, deployment), each one expressing one particular "view" [18] 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;

- to be able to *generate a global coordinator*: the coordination model we proposed in Figure 4 specifies two types of coordinator components for each step (FCi and RCi), spreading the system global coordination through different components. Combining together the coordinators (or their specifications) we could concentrate all these controls on a single, global, coordinator;

- to identify a *library of coordination components* that may be composed to implement coordination policies;

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

1. automatically extracts the questions of interest for each step. 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 4.2 may be instantiated to the system;
2. visualizes the questions;
3. provides the user an environment to edit the specifications;
4. automatically generates the system coordinator, combining the lower-level coordination specification.

Future work on the framework will analyze how the coordination information captured by this approach, can be used to identify an architectural description, refining the existing Step2. A tool could be realized in order to automatically convert coordinators into architectural components or connectors.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge the Italian M.I.U.R. national project SAHARA that partly supported this work and the (many) anonymous reviewers for useful suggestions on how to improve this paper.

9. REFERENCES

[1] F. Achermann, S. Kneubuehl and O. Nierstrasz. Scripting Coordination Styles. In Proc. [12].

[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. Coordination of Massively Concurrent Activities. *CWI Report*, CS-R9565, 1995.

[4] F. Arbab. What Do You Mean, Coordination? In the March '98 *Issue of the Bulletin of the Dutch Association for Theoretical Computer Science* (NVTI).

[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] 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 Mtayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*. World Scientific, 1996, pp. 1-39.

[7] L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice*. SEI Series, Addison-Wesley 1998.

[8] L. Bass and R.Kazman. Architecture-Based Development. *CMU/SEI*-99-TR-007, ESC-TR-99-007.

[9] 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.

[10] 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.

[11] 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.

[12] Coordination 2000. Proc. Fourth *Int. Conference on Coordination Languages and Models*, Coordination 2000, Cyprus, 2000. LNCS 1906.

[13] S. Ducasse, T. Hofmann and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In Proc. [12].

[14] G. Ferrari, C. Montangero, L. Semini and S. Semprini. Mobile Agents Coordination in Mob_{adtl}. In Proc. [12].

[15] A. Filkenstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. *IEEE Trans. on Software Engineering*, Vol. 20, No. 8, pp. 569-578, August 1994.

[16] P. Fradet, D. Le Metayer, M. Perin. Consistency Checking for Multiple View Software Architectures. In *Proc. European Software Engineering Conference* (ESEC/FSE'99), pages 410-428, Toulouse, 1999.

[17] D. Hirsch, P. Inverardi and U. Montanari. Reconfiguration of Software Architecture Styles with Name Mobility. In [12].

[18] C. Hofmeister, R. Nord and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1998.

[19] G. Holzmann. SPIN web site. On line at: http://cm.bell-labs.com/cm/cs/what/spin/index.html.

[20] P. Inverardi and H. Muccini. Coordination models and Software Architectures in a Unified Software Development Process. In Proc. [12].

[21] 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.

[22] 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.

[23] P. Inverardi and H. Muccini. Software Architectures and Coordination models. To appear in the Journal of Supercomputing (Kluwer), Vol. 24, No. 2, Feb. 2003.

[24] IWLI: Second International Workshop on Living with Inconsistency. ICSE'01 workshop, May 13, 2001 Toronto, Canada,

[25] IWSSD-10: Tenth International Workshop on Software Specification and Design. Track on (In)completeness, (in)consistency and (sufficient) correctness. November 2000, San Diego, California.

[26] I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, Object Technology Series (1999).

[27] JavaSpaces. On-line at: http://java.sun.com/products/ javaspaces/

[28] 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.

[29] H. Muccini and F. Mancinelli. Eliciting Coordination Policies from Requirements. Technical Report, University of L'Aquila, year 2002. On-line at: http://www.henrymuccini.com/publications.htm

[30] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In Proc. of "The Future of Software Engineering", FOSE2000, part of ICSE2000, June 2000.

[31] 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.

[32] Rational Corporation. Uml Resource Center. On-line at: http://www.rational.com/uml/index.jtmpl

[33] N. S. Rosa, G. R. R. Justo and P. R. F. Cunha. A Framework for Building Non-Functional Software Architectures. In Proc. *ACM. SAC 2001*, Special Track on Coordination Models, Languages and Applications.

[34] S. Stuurman and J. Van Katwijk. Evaluation of Software Architectures for a Control System: A Case Study. In *Proc. Coordination 97*, LNCS 1282.

10. AUTHORS BIOGRAPHY

Henry Muccini received is Ph.D. degree in Computer Science from the Universitá degli Studi dell'Aquila (Italy) in 2002. Currently he has a post-doctoral researcher position at the University of California, Irvine and he just get an assistant professor position at the Computer Science department, Universitá degli Studi dell'Aquila. Henry's research interests are in software architecture-based analysis techniques; specifically, in testing of software subsystems integration against a software architecture specification, in integrating coordination policies inside the architectural description and validating software architectures with respect to semi-formal specifications.

Fabio Mancinelli received the master degree in Computer Science from the Universitá degli Studi dell'Aquila (Italy) in 2001. He is currently a Ph.D. student at the Universitá degli Studi dell'Aquila. His research area is software engineering and architecture applied, above all, to mobile computing. Other research interests range from distributed systems to networks and operating systems.