Conceptual and concrete architectures in the design of CSCW applications

Daniela Micucci[1,2], Marcello Sarini[3], Carla Simone[1], Francesco Tisato[1], and Andrea Trentini[1]

[1]Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Via Bicocca degli Arcimboldi, 8 – 20126 MILANO

[2]Dipartimento di Matematica "Federico Enriques"
Università degli Studi di Milano
Via Cesare Saldini, 50 – 20133 MILANO

[3]Dipartimento di Informatica
Università degli Studi di Torino
Corso Svizzera, 185 – 10189 TORINO

e-mail: {micucci, simone, tisato, trentini}@disco.unimib.it, sarini@di.unito.it

***Abstract* — CSCW applications support the development of various types of cooperative applications and their integration. However, little attention has been paid to the gap between the semantic level of articulation work and the semantic level of the implementation platform. This semantic level gap is widely recognized in CSCW as a big source of problems from the application usability point of view. The paper presents a framework for the construction of supports to articulation work which tries to overcome the semantic gap by starting from the requirements of articulation work and by identifying its conceptual model (components, their aggregations and communication language). This leads to a component based view of the resulting system, where components play active and specialized roles and interact via well identified communication patterns. In this view, it was natural to think of a conceptual architecture based on a multi-agent approach, where agents are heterogeneous. Finally, the paper shows how the conceptual architecture can be mapped onto a concrete architecture, whose characteristics are generality, efficiency, robustness and so on.**

BACKGROUND AND MOTIVATIONS

One of the main efforts of Computer Supported Cooperative Work (CSCW) is the development of technologies supporting cooperation of autonomous and interdependent human actors. These technologies are built on the basis of the observation of how the actors define and use mechanisms making their collaboration possible. These observations led the recognition of two levels that jointly constitute cooperative work: *articulation work*, devoted to *coordinate* the necessary activities to reach the goals; and *operational work*, devoted to the actual *execution* of these activities according to the established coordination [1]. In other words, CSCW claims for a separation of concerns at the level of cooperation among actors. The same technique characterises the development of distributed systems where coordination and functional requirements are distinguished [2]. This paper focuses on the components of CSCW applications that support articulation work, disregarding the problems related to the support of operational work.

Actually CSCW applications have been developed without the adoption of a real separation of concerns: some frameworks do not explicit the incorporated coordination model supporting articulation work; in some others ones the coordination model is influenced by the implementation platform. In literature this lack is described as the gap between the semantic level of both articulation work and the implementation of the related support [3]. This *semantic level gap* is widely recognized in CSCW as a big source of problems from the application usability point of view. Usability is strictly related to flexibility: if users can influence the support of articulation work, they understand the logic by which the support is constructed (visibility vs. transparency [4]).

On the other hand, literature proposes several coordination models [2], most of them proposing modelling solutions at the implementation level (i.e., they emphasise generality).

Our goal is the development of a framework to support articulation work bearing in mind the semantic level gap. Thus, we had two possibilities:

- building the coordination model at the semantic level of articulation work on top of existing application oriented coordination models;
- deriving characteristics of the pertinent coordination model from the requirements of articulation work. "Mapping" them to an existing application oriented coordination model.

We consider the second option a safer way to fully respect the requirements in the framework construction.

Our research effort has two main inputs:

- study results concerning the main categories (and their combinations) that users apply to conceive support of articulation work. From the modelling point of view, this approach leads to a component-based view of the resulting system: each component has an active and specialized role in the articulation work and interact with the other components by means of well identified communication patterns. In this view, it was natural to think of a *conceptual* architecture based on a multi-

agent approach, where agents are heterogeneous.
- an application oriented coordination model defining a *concrete* architecture, whose main characteristics are generality, efficiency, and robustness.

The paper proposes the requirements of articulation work and presents an appropriate conceptual model in terms of components, their aggregations, and communication language. Finally, the paper describes the main features of the concrete architecture and concluding with its exploit upon the conceptual one.

THE REQUIREMENTS OF ARTICULATION WORK

CSCW applications were initially developed to satisfy circumscribed needs: the management of business processes, the sharing of data and information, various forms of co-authoring, various forms of communication supports and so on. Once the field reached a more mature stage, the resulting conceptual and technological fragmentation was perceived as one of the main obstacles to the development of the field itself. Despite the specific application domain, cooperative work requires a strong integration of the related functionalities: cooperation is characterized by a fine grained co-presence of structured and ad-hoc forms of collaboration, and by a continuous shift between them [5]. Hence, it was perceived as fruitful to structure cooperative applications in smaller units of functionality, each of them focusing on a specific *coordination mechanism* [3] and serving smaller groups of highly interacting people (roles). The overall application is then a composition of units and their interactions. Therefore, the first requirement of conceptual architecture is the ability in constructing single coordination mechanisms as *aggregations of heterogeneous coordination mechanisms*.

Separation between cooperative applications and support of the *organizational context* in which they operate is another aspect of the fragmentation. The *organizational context* is a resources aggregation that has to be cooperatively managed through its specialised coordination mechanisms. Moreover, its evolution influences evolution of the cooperative applications and vice-versa. Hence, the second requirement of the architecture is to allow strong interactions between applications and their organizational contexts.

In the previous section, we mentioned the flexibility requirement. This can be further specified in terms of *incremental design* (each component can be active although not fully defined and be completed at run time); *modifiability* (component structure and its interactions can be modified once the component itself is activated); and *dynamicity* (new components and component aggregations can be added at any time).

Cooperative applications are essentially interactive, that is, their evolution is fully governed by user actions through a suitable user interface. These actions are related to creation, definition, enactment, activation, and modification of coordination mechanisms. We have already mentioned that a suitable conceptual architecture should be expressed in terms of a heterogeneous multi-agent system: this guarantees modularity as well as communicative behaviour of each component. The natural choice is to consider *reactive agents* to allow the conceptual architecture both reacting to user events, and elaborating them by triggering the cooperative behaviour of pertinent components [6].

THE CONCEPTUAL ARCHITECTURE

Each cooperative application can be decomposed into aggregations of independent agents (coordination mechanisms) that collaborate and exchange information using an expressive coordination language. This is the basic intuition underlying the ABACO (Agent-Based Architecture for COoperation) architecture [7].

This section describes the agents constituting the architecture, though the next one describes their internal structure and coordination language.

Agents are classified in:
- *Application agents*, used to construct coordination mechanisms as aggregations of agents (*constellations*) corresponding to basic building blocks representing different categories of articulation work;
- *System agents*, which provide communication services to Application agents, and manage users interaction.

System agents

System agents constitute an infrastructure used by Application agents, independently of their specific definition. System agents are distinguished in *Interface agents* and *User agents*.

Each *Interface agent* is a communication mediator connecting different agents according to the structure of the constellation in which agents themselves reside: typically, groups of agents in a constellation, and groups of constellations into compound constellations. These agents realise a trade-off between the efficiency of communication and the support to flexibility in the dynamic construction of the applications.

Once related aggregation is defined, Interface agents may be created as *Local Interface agents* or *External Interface agents* to allow communication between agents in the same constellation or compound constellation respectively.

External Interface agent serves all the constellations of a specific compound constellation. Communications crossing different constellations in the same compound constellation pass through the related External Interface agent.

Each *User agent* (UA) plays as communication mediator between human actors and the agents of the application. In fact, a User agent allows human actors to interact with the application through any graphical user interface (GUI) satisfying their needs and preferences. Each GUI is integrated in a *Wrapper agent* that communicates, back and forth, with the User agent. Conversely, User agent dispatches received communications to the appropriated Interface or Application agents.

Application agents

In order to guarantee the greatest degree of dynamicity, Application agents are organized to respond to every kind of stimuli an external user can ‘send to the architecture’. There are three kind of roles a user can take: *Programmer*, who implements the building blocks that can be selected to construct the coordination mechanisms constituting the cooperative application; *CM-definer*, who selects the building blocks of each coordination mechanism and defines

their aggregations; and *End-user*, who puts at work instances of the defined coordination mechanisms supporting actual cooperative activities. There are three classes of Application agents (see Figure 1 and 2), which are described here below.

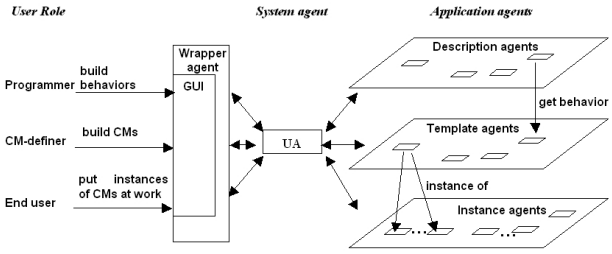

Figure 1 - Application Agents

Description agents. They contain the basic building blocks behaviours that are realised in terms of the Agent Communication Language (ACL) associated with the architecture. Their role is reminiscent of the 'description actors' proposed in the Actor Model [8]. The Programmer creates them (through the User agent). Description agents elaborate two types of event: *definition* and *modification* (when a Programmer constructs them), and corresponding templates *creation* (when a CM-definer builds a coordination mechanism, see below).

CM-templates. They are the agents constituting the coordination mechanism. That is, they are an aggregation of building blocks (agents). Specifically, one or more aggregations may describe in a fully homogeneous way both the *organizational context* and the CMs governing cooperative activities. CM-templates elaborate events concerning their *definition* and *modification*, and the *creation* of their instances.

CM-instances. They are agents constituting a specific instantiation of a defined coordination mechanism. Obviously, several instantiations of the same coordination mechanism can coexist. An instantiated coordination mechanism can be *enacted*, that is, appropriate resources are assigned to it, and *activated*. *Enactment* and *activation* are the events generated by the End-user. Moreover, CM-instances are able to elaborate events concerning their *definition* and *modification*.

These three classes of Application agents concurrently populate the architecture. They cooperate, together with System agents, to provide users with the desired functionality. To that aim, agent behaviour contributes to the definition of communication protocols that are triggered by the various events users generate. Application agents show behaviours that are partly uniform among them: typically, every agent, after its creation, is able to receive and elaborate triggers concerning its definition/modification. This responds to the requirement of incremental definition. Moreover, all of them will be enacted and activated, sooner or later. When agents are created they are provided with the basic capabilities to be incrementally defined (*default behaviour*). On the other hand, agent's behaviour is also specialised according to the specific agent type. The definition/modification capabilities are put at work to construct the desired behaviour of each Template agent. This is realised through an appropriate communication protocol that also involves the User agent and the pertinent Description agents. Instance agents follow a similar protocol as far as their definition/modification are concerned.

Next section defines the coordination model that we propose to construct the above architecture. The model specifies the agent internal structure and the coordination language to represent the communication protocols. Section V provides examples of some basic protocols.

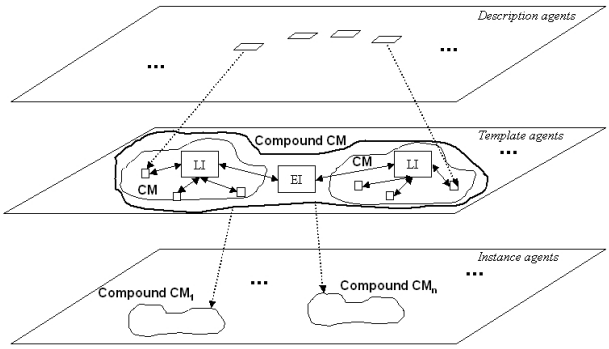

Figure 2 - Different agent aggregations

THE COORDINATION MODEL

The agent internal structure is composed by *local data* and *behavior*. The former contains its local data and the latter its communicative behaviour (see Figure 3). Agent internal state is expressed in terms of values from an attributes list characterising the agent. Moreover, each agent contains a local working memory used to store information needed for its behaviour: typically information gathered through communication or necessary for local computations.

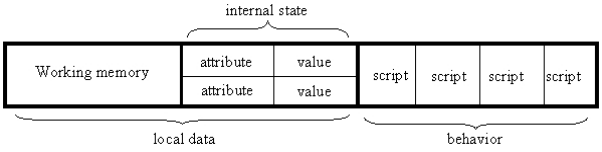

Figure 3 - Agent's internal structure

The behaviour is expressed in terms of communication patterns that follow the ACL syntax described in Table 1. Agent behaviour is the result of processing incoming messages. These messages are characterised by the type of 'task' they refer to, and by additional information for the handler to perform its task.

<behavior>::=	<pattern>$^+$
<pattern>::=	<message> → <handler>
<handler>::=	[<local computation>;] [<reaction>]
<reaction>::=	[IF <cond>] <out-comm> | [IF <cond>] <reaction> // <reaction>
<out-comm>::=	send to <agent> : <message>
<agent>::=	all (<list>) | anyof (<list>)
<list>::=	<agent-type>$^+$ | <agent-name>$^+$
<message>::=	tell (<data> [{<attribute: value>*}]) | ask (<info> [{<attribute: value>*}]) | create (<attribute: value>* [{<attribute: value>*}]) | define (<attribute: value>* [{<attribute: value>*}])
< data >::=	<attribute: value>* | new-behavior: <behavior>
< info >::=	<attribute>* | behavior
<value>::=	<any-data-type>* ; <agent-type>*

Table 1 - The ACL syntax

A message is characterised by the following primitives:

- *tell* - agent is sending *data* possibly together with a context *{...}* where to interpret them;
- *ask* - agent is sending a request of *info* and the context *{...}* where to interpret it.
  Both *data* and *info* may concern *attributes* or *behavior*. For sake of conciseness, the syntax does not explicitly specify that *attribute*, *value*, and *behavior* may be variables or constants. Moreover, *attributes* and their *value*s may be of *any data-type* and/or *agent type*. Attributes and their values are application dependent.
- *create* - agent is requested to create an agent according to the specified *data*, where usually the required attributes are *CM* (coordination mechanism), *category* (expressing the kind of building block), and *name* (name of the new agent being created);
- *define* - agent is requested to define its behaviour according to the specified *data*.

A *handler* is a composition of local computations and a (possibly) guarded and concurrent message sending.

The ACL syntax expresses communication capabilities in a similar way to the ones characterising the Actor Model [8]. The way to express semantic is similar, but it is out of the scope of this paper. We can only say that the main difference concerns the possibility in expressing the reflection capabilities required to cope with the flexibility requirement in terms of dynamic definition and modification of agent behaviours.

Next section illustrates through some examples how the coordination mechanisms supporting articulation work are specified in the conceptual architecture.

EXAMPLES OF AGENT BEHAVIORS

Assume that a programmer has already created a set of Description agents and has defined the general behaviours that regulate their possible interactions. Hereafter we exemplify how Template agents can be created and defined by a CM-definer role by exploiting behaviours specified by Description agents. The CM-definer role also defines the aggregation of agents expressing coordination mechanisms.

Creation of Template agents

A stimulus from the CM-definer triggers a chain of interactions among agents. The result is the creation of a Template agent with the definition of its default behaviour (see Box A). The creation message and its attributes is sent to the User agent through the CM-definer user interface. The User agent redirects the message to the pertinent Description agent, which maintains the behaviours related to the category of the agent being created. The Description agent performs a local computation to create a Template agent according to the received parameters (the aggregation of the created agent, its category and its name). Moreover, the Description agent overwrites the created Template agent with its default behaviour. Box A exemplifies the creation of Role and Task Template agents. In particular, the scripts assigned to *Role(DA)* and *Task(DA)* in Box A describe the default behaviours associated to the created Template agents. The former allows the Template agent to ask the appropriate Description agent about the behaviours associated with the definition of an attribute, while the latter one allows the Template agent to receive from its Description agent the asked behaviour and to incorporate it.

Definition of Template agents

The definition of a Template agent concerns the designation of the attributes characterising the category it belongs to with the related behaviours. Some attributes are simply pairs of <attribute, value> that become parts of the agent internal state; others express mutual references between agents: they have a semantics described in terms of behaviours that define the interactions between the involved agents. When a CM-definer chooses (through its interface) to define attribute pairs of a Template agent of the first kind, they are simply written as pairs in the local data of the agent. In the second case, this stimulus triggers a chain of interactions among agents that produces the definition of the agent behaviours corresponding to the semantics of the defined attributes (see Box B). More precisely, the message defining an attribute of a Template agent is sent through the User agent to the pertinent Template agent under definition

At this point the involved Template agent, according to its default behaviour, is able to ask to the related Description agent (that is the Description agent belonging to the same category) the specific behaviours associated to the attribute under definition. Then, the Template agent receives from the involved Description agent the requested behaviours instantiated with the values currently associated to the definition. As an example we consider here the definition of two specific attributes: *responsible for Task*, and the symmetric *under responsibility of*, which define the appropriate behaviours describing an interaction between Role and Task Template agents previously created. The attribute *responsible for Task* represents Role duties in activating the associated Task (see Box B, (3) in Role(DA)) and in waiting for the results from its computation (see Box B, (4) in Role(DA)). *Under responsibility of* instead, means that the Task, once activated, performs some local computation and then sends back to the activating Role the related results (see Box B, (3) in Task(DA)). Box C describes the pattern of UA's behaviour related to definition of attributes *responsible for Task* of Roles and *under responsibility of* of Tasks. Notice that the matching between the incoming message and the left hand side of behaviours patterns, considers both the name of the primitive, the related parameters, and the context in which the message is sent. In this way, the variables describing the general behaviour are "unified" with the values associated to the received message. This leads to instantiate the selected behaviour according to the current values expressed by the CM-definer in the related attribute definition.

```
Role(DA)
create(CM, Role, Role_name) →
    create_agent(CM, Role, Role_name);
    send to Role_name: tell(new-behavior:
      (1) define(<attribute: value>{<CM:value> }) →
          send to Role(DA): ask(behavior
          {<attribute:value>,<reply: Role_name >})

      (2) tell(new-behavior: behavior) →
          incorporate_behavior
)

Task(DA)
create(CM, Task, Task_name) →
    create_agent(CM, Task, Task_name);
    send to Task_name: tell(new-behavior:
      (1) define(<attribute: value>{CM }) →
          send to Task(DA): ask(behavior
          {<attribute: value>,<reply: Task_name>})

      (2) tell(new-behavior: behavior) →
          incorporate_behavior
)
```

Box A - Creation of a Template agent

```
Role(DA)
ask(behavior {<responsible for Task:Task_name>,
  <reply: Role_name>}) →
    send to Role_name: tell(new-behavior:
      (3) tell(<activate: Task_name>) →
        send to Task_name:
        tell(<event: activate><activator:Role_name)>)

      (4) tell(<result:value>) →
        update_local_data(result)
)

Task(DA)
ask(behavior {<under-responsibility-of:Role_name>,
  <reply: Task_name>}) →
    send to Task_name: tell(new-behavior:
      (3) tell(<event: activate>,
        <activator: Role_name>) →
          update_local_data;
          send to Role_name: tell(<result:value>)
)
```

Box B - Fragment of behaviors related to the definition of a Template Agent

```
UA
define(<responsible for Task:Task_name, Role_name>
  {<CM:value>}) →
    send to Role_name:
      define(<responsible for Task: Task_name>
      {<CM:value>}) ||
    send to Task_name:
      define(<under responsibility of: Role_name>
      {<CM:value>})
```

Box C - Fragment of User agent behaviour

THE CONCRETE ARCHITECTURE

RTP is a general-purpose architecture for the development of strongly modular and configurable distributed systems. The architecture reifies the definition of a formal model (*abstract machine*) supporting the operational semantics of complex systems. RTP provides also a framework implementing the architecture and supporting the implementation of domain related systems.

Most of the underlying ideas of the architecture borrow from previous research activities and projects, in particular Kaleidoscope [9], RAID[1] [10] and [11], RTO[2], Architectural Reflection, HyperReal, and MAIN-E.

The architecture is mainly based on some core considerations:

- a system is made up by connected entities performing some kind of activity;
- an entity exchanges information with other entities;
- every activity is controlled by "supervisor" entities, which can be also distributed;
- only "supervisor(s)" may start performing activities without being triggered;

The considerations above has lead to the definition of three primary components inside the architecture: *Performer*, an entity executing a specific task; *Projector*, a special kind of Performer whose task is to allow information exchange between Performers; and *Strategist*, a domain-related active entity that controls the system behaviour.

Conceptually, a Performer can be modelled as a classical state machine: it has a local behaviour defining what to do when receiving a command but it is not aware of command sequences. In the RTP architecture we have a hierarchy of Performers: a Performer representing a generic command acceptor and a specialised *ProgrammablePerformer* representing a concrete state machine. Figure 4 shows the UML[3] [12] Class Diagram of ProgrammablePerformer with its set of states and ECA[4]-transitions.

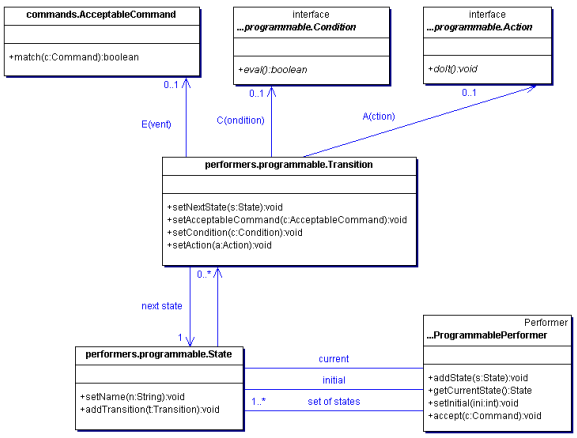

Figure 4 - Performer's State Machine Class Diagram

The Performer environment is made up of variables belonging to two different sets: one containing *local variables*, i.e. only visible inside the performer; the other containing *visible variables*, i.e. the variables that the Performer decides to export. The latter, in turn, has two subsets: *exported visibles* (writeable from the inside and readable from the outside), and *imported visibles* (vice versa). We consider an advantage confining a Performer visibility in its environment only: in this way a Performer can be exploited in different concrete topologies.

Projections are the entities that define mappings from visible exported variables of a Performer into imported

[1] Rilevamento dati Ambientali con Interfaccia DECT
[2] Real-Time Objects
[3] Unified Modeling Language
[4] Event-Condition-Action

visible variables of another Performer. In other terms, a Performer can observe the exported variables of another Performer not directly but only if they are projected (i.e. *copied*) into its own imported variables. A set of projections defines the *topology* of a system.

In terms of architecture, *Projectors* exploit projections. They are special kind of Performers defining alignment mechanisms without dealing with alignment timings. The system topology is realised by a special kind of Performer, hereafter *Topologist,* whose role consists in relating Performers. The Topologist first creates Projectors and then binds the related exported and imported visibles of Performers through the pertinent Projectors.

Figure 5 shows the projection (reified through a *ConcreteProjector*, specialising a general Projector) class diagram. A Projector links (passes values of Visibles from) one source to many targets.

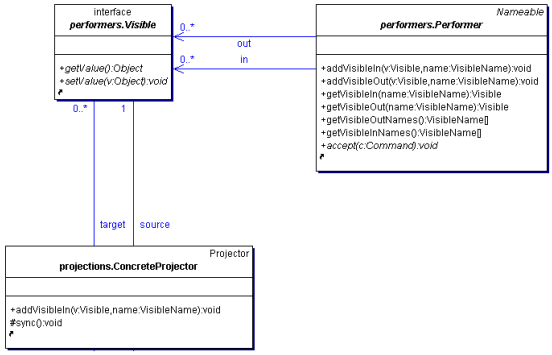

Figure 5 - Projection Class Diagram

Issues about *when* and *why* elaboration and information alignment have to be done are encapsulated inside *traces*. Every trace defines a part of the system behaviour in terms of partially ordered set of *requests*. In turn, a request is a pair (*command*, *recipient*), where recipient denotes either a Performer or a Projector. Finally, a *Strategist* is the entity that is able to create and dynamically manage traces. In doing so, it controls the part of the system related to the trace. A strategy encapsulates any domain-related issues and so, it is the only entity that must be re-designed (better say, developed) when the application domain changes.

ABOUT MAPPING CONCEPTUAL AND CONCRETE ARCHITECTURE

To define a mapping between the conceptual and the concrete (RTP) architecture we have to sketch a bridge for each level of the conceptual architecture. The three conceptual levels must be described in terms of RTP concepts (see Figure 6).

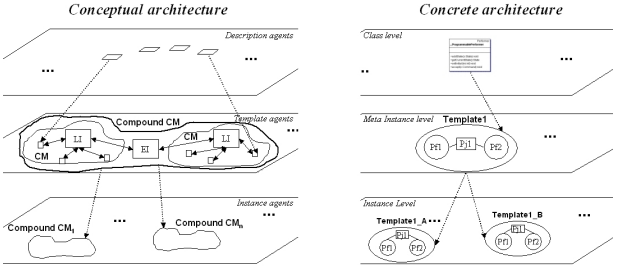

Figure 6 - Conceptual and concrete architecture mapping

Since we use UML (and general Object-Oriented concepts) to define RTP architecture, we have two levels of abstraction that are to be mapped onto three: the *class level* (representing classes and relationships between them) and the *instance level* (representing concrete objects created from classes). We chose to map the "Description Agents" level with RTP class level and the other two remaining ones with the RTP instance level in which we defined a "template" abstraction mechanism to distinguish between the two levels in the conceptual architecture. In particular, we chose to map the "CM-template" level with the instance level (instead of the class level) to fulfil the flexibility requirement of the conceptual architecture. Conversely, using the class level for the "CM-template" would have prevented the incremental and dynamic definition of the agent aggregations realising the Template level. More specifically, the mapping that is leading the ongoing implementation of the conceptual architecture is as follows:

- the "Description Agents" level is mapped to the class level in RTP: a high-level language programmer personifies the "programmer role". In this view Performers represent (Description) Agents,
- the "CM-templates" level is mapped by a runtime phase in which only particular kind of instances appears. We have to introduce a "component" concept to reify a meta-level representing building blocks aggregates. At this level (called *Meta Instance level*) we can use Projector entities to map the Interface agents and again Performers to map (Template) Agents. In fact, Projectors allow information exchange among Performers and this is similar to the functionality of Interface agents which are communicator mediators connecting different agents. Since Projectors have only one imported visibles, they can manage one kind of information exchange at once. This means that an Interface agent may be mapped onto several Projectors.
  The User agent can be mapped onto a Topologist. In fact, the former receives from CM-definers (through their user interfaces) information representing the interaction between agents of the Template level in terms of attribute definition. In this way, it is reasonable to consider as an equivalent in the concrete architecture a Topologist. The role of the "CM-definer" in the concrete architecture is to aggregate Performers and Projectors to build a *TemplateComponent* (see Figure 7). The latter is equivalent to a coordination mechanism in the conceptual architecture,
- the "CM-instances" level is mapped by a successive runtime phase with no restrictions on the kind of instances allowed. The transition from the above level to the *Instance level* (see Figure 6) is realized using the *generate()* method on instances of TemplateComponent. The generate() method spawns a concrete *Component* reflecting the TemplateComponent definition.

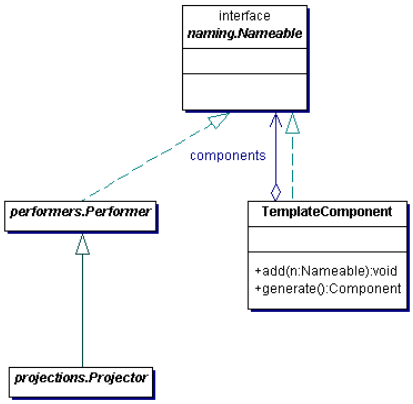

Figure 7 - TemplateComponent Class Diagram

In the following we sketch the definition protocol for Template agents in terms of concrete architecture. We did not consider in details the ACL scripts expressing agent behaviours. They may be reported in terms of states and transitions of the state machine expressing Performer behaviour according to the semantics of the considered script. Rather, we focus on the interactions among agents involved in the definition protocol. In particular we will consider the definition of Role and Task agent for what concerns the attributes *responsible for Task* and *under responsibility of* relating them according to what described in Section "Definition of Template agents".

The Meta Instance level in concrete architecture is the level where agent behaviours are defined. In the example proposed, this level contains the Performers involved in this operation representing the *UA*, the previously created *Role1*, and *Task1* agents, respectively (see Figure 8).

Remark that in concrete architecture, all possible behaviours associated to *Role1* are defined in the class level through the Role class definition. At Meta Instance level *Role1* is an instantiation of its class definition, and so, it includes all the behaviours.

This is slightly different from what described in the creation protocol of the conceptual architecture. There, the specific attribute definition for the created Template agent was expressed through an ask-tell interaction among the pertinent Template agents and its associated Description agent in order to get the appropriate behaviour (see Section "Creation of Template agents"). Instead, in concrete architecture this protocol is implicit in the instantiation of the Performer representing the Template agent from the associated class Performer belonging to the class level. In fact, in the concrete architecture there is not direct interaction between Performers representing agents in the class and Meta Instance level because the former one is a compile time level and the second one is a run time level.

For what concerns definition protocol, this means that in the Meta Instance level is necessary to actualise the behaviour incorporated in the Performer with the information provided by the user through her user interface. In our example, we have to actualise the behaviour satisfying the definition of the *responsible for Task* attribute.

To complete the transformation of the definition protocol in the concrete architecture, the UA Performer plays the role of *Topologist* responsible for the projections creation that bind the Performers involved in the attribute definition: respectively *Role1* and *Task1* Performers. In this way, two projectors are defined by UA: one allowing *Role1* Performer to send information about activation to *Task1* Performer, the other allowing *Task1* Performer to transmit the results achieved from its computation (see Figure 9).

This means that Performer *Role1* makes imported visible the variable associated to the definition of the information about activation, and exported visible the variable associated to the definition of the computation result (and vice versa for the Performer *Task 1*).

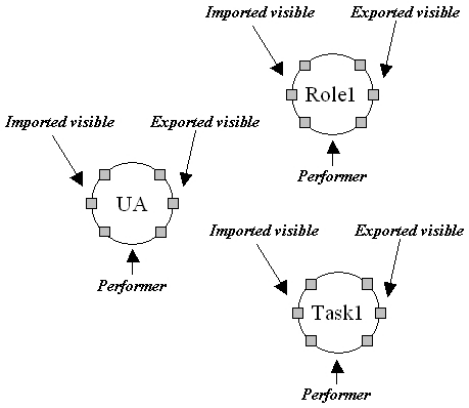

Figure 8 - The Performers involved in the definition of the *responsible for Task* attribute

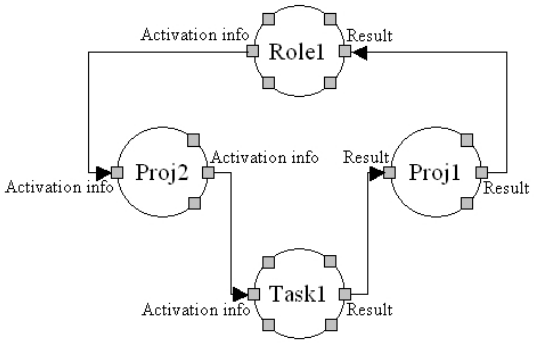

Figure 9 - The Projectors and their bindings in the definition of the *responsible for Task* attribute

CONCLUSIONS AND FUTURE WORKS

In this paper we described ABACO, a conceptual multi-agent architecture aiming at the support of articulation work among actors involved in the execution of interdependent and distributed activities. This architecture is based on a set of heterogeneous reactive agents that communicate through a common language. In this way, the architecture allows the dynamic and incremental definition of agent aggregations that can express collaboration among actors both at local organization levels and at increasing levels in the organization. This kind of architecture was designed bearing in mind the requirements of flexibility imposed by the distributed nature of articulation work. In our effort, we divide the conceptual level from the concrete level combining advantages of a top down approach (from requirements to conceptual architecture capturing them) with the ones of a bottom up approach (from a general purpose and robust concrete architecture to applications

built on the basis of the conceptual one). According to this view, we focused our research efforts in looking for concrete architectures whose characteristics fitted with the requirements of flexibility raised by the conceptual architecture. Hence, the required architecture needs to provide the designers with modularity, support for dynamic interaction among distributed agents and architectural reflection. We choose RTP as concrete architecture because it is a general-purpose architecture for the development of strongly modular and configurable distributed systems that encompasses the mentioned features. Following our choice, we proposed then a conceptual mapping between conceptual and concrete architectures for two reasons: to express the operational semantics of ABACO in terms of the RTP constructs and to implement the ABACO architecture in order to experiment its usability in real work environments.

The definition of a simple mapping is essential to create a straight path from the conceptual architecture to a tentative implementation. The path must be very easy to understand and straightforward to apply, without any freedom of choice[5]. In choosing our mapping we tried to comply with well-known guidelines, such as the ones described for example in sections 4.3, 4.4, 6.4, and 7.5 of [13]. Moreover, our concrete architecture fits into the categorization proposed in [14] with respect to the elemental components of an agent-oriented application framework:

- Communication and Coordination (Projectors and Requests)
- Planning (Traces, creation and modification of)
- Scheduling (Traces, execution of)
- Execution Monitoring (Engines)

Our proposal is a research effort toward the definition of an approach to the design of agent-based CSCW applications. Recall that the systematisation of the different experiences in the use of agents is a novel research effort inside the CSCW area [15]. We dare to say that our approach (a conceptual model to support articulation work incorporated in a conceptual architecture, a concrete architecture and a suitable mapping among them), may be used as a guideline to design CSCW agent-based applications. According to [16] a "design methodology" should take into account the macro (societal) level and the micro (agent) level aspects. Our concrete architecture formalises the former with Projections/Projectors and the latter with the state model (Event Condition Action – Finite State Automaton-like). Of course we do not know yet if the proposed mapping will be the last (and the best) one, and our future research work will be headed towards simplification, refining and verification. Alternative mappings to other concrete architectures, such as for example [17], [18], and [19], should also be investigated.

In addition, an alternative mapping could consider the definition of three levels of Instances in the concrete architecture to be associated to the three levels of the conceptual architecture, instead of considering a Class level and two Instance levels. In this way, the current Class level, will be substituted by a Meta-Meta Instance level. This means that the Description Agents of the ABACO architecture are represented by instances of Projectors whose behaviours can be dynamically defined by Programmers at run time. Following this idea, the incremental design claimed in the ABACO architecture can be implemented in terms of information exchange among objects of the Meta-Meta Instance level (representing Description agents in ABACO) and objects of the Meta-Instance level (representing Template agents in ABACO). In fact, the former objects store the agent behaviours just defined while the latter ones receive only the behaviours defined by the CM-definer role. Instead, in the current mapping, an agent of the Meta-Instance level (a Template agent in ABACO), is created with all the behaviours pertinent to the category it belongs to even if these behaviours have not been still defined by the CM-definer.

[5] Sounds bad, but it's not: any generative technique should be as "mechanical" as possible.

REFERENCES

[1] K. Schmidt and L. Bannon, "Taking CSCW Seriously: supporting Articulation Work," Computer Supported Cooperative Work, The Journal of Collaborative Computing, vol. 1, pp. 7-40, 1992.

[2] G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages," Advances in Computers, vol. 46, 1998.

[3] K. Schmidt and C. Simone, "Coordination Mechanisms: Towards a conceptual foundation for CSCW systems design," Computer Supported Cooperative Work, The Journal of Collaborative Computing, vol. 5, pp. 155-200, 1996.

[4] P. Dourish, "Computational Reflection and CSCW Design," Rank Xerox EuroPARC, Cambridge, UK, Technical Report EPC-92-102 1992.

[5] C. Simone and K. Schmidt, "Mind the gap! Towards a unified view of CSCW," presented at Fourth International Conference on Design of Cooperative Systems (COOP2000), Sophia-Antipolis (Fr), 2000.

[6] M. Divitini, M. Sarini, and C. Simone, "Reactive Agents for a systemic approach to the construction of Coordination Mechanisms," in Agents supported Cooperative work, Y. Yiming, Ed.: Kluwer Academic Press, to appear.

[7] Divitini, M., Simone, C., and Schmidt, K. "ABACO: Coordination Mechanisms in a Multi-agent Perspective," Second International Conference on the Design of Cooperative Systems, INRIA Press, Juan-les-Pins, France, 1996, pp. 103-122.

[8] G. A. Agha, ACTORS: a model of concurrent computation in distributed systems. Cambridge: The MIT Press, 1988.

[9] A. Savigni and F. Tisato, "Kaleidoscope. A reference Architecture for Monitoring and Control Systems," presented at First Working IFIP Conference on Software Architecture (WICSA), San Antonio, TX, USA, 1999.

[10] D. Micucci, "Exploiting the Kaleidoscope architecture in an industrial environmental monitoring system with heterogeneous devices and a knowledge-based supervisor," presented at SEKE 2002 Conference, Ischia, Italy, 2002.

[11] D. Micucci, A. Savigni, and F. Tisato, "Environmental Monitoring Systems: the Kaleidoscope architecture and the RAID case," Como, Italy, 2001.

[12] M. Fowler and K. Scott, UML Distilled. Second Edition: Addison-Wesley, 2000.

[13] Wooldridge, M., and Jennings, N.R. "Pitfalls of Agent-Oriented Development," Autonomous Agents 98, Minneapolis MN USA, 1998.

[14] Brugali, D., Sycara, K. "Towards Agent Oriented Application Frameworks," ACM Computing Surveys (CSUR), 2000.

[15] Agents supported Cooperative work, E. Churchill and Y. Yiming, Eds. Kluwer Academic Press, to appear

[16] Wooldridge, M., Jennings, N.R., and Kinny, D.. "A Methodology for Agent-Oriented Analysis and Design," Autonomous Agents 99, Seattle WA USA, 1999.

[17] Brazier, F.M.T., Jonker, C.M., Treur, J., and Wijngaards, N.J.E.. "An Agent Architecture for Dynamic Re-design of Agents," Proceedings of the Third International Conference on Multi-Agent Systems, ICMAS'98. IEEE Computer Society Press, 1998, pp. 401-402.

[18] Busetta, P., Rönnquist, R.,Hodgson, A., and Lucas, A.. "JACK Intelligent Agents Components for Intelligent Agents in Java," AgentLink Newsletter, 1999.

[19] Nwana, H.S., Ndumu, D.T., Lee, L.C. and Collis, J.C.. "ZEUS: a toolkit and approach for building distributed multi-agent systems," PAAM'98, London, 1998.