A Collaboration Specification Language

Du Li and Richard R. Muntz
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024 USA
{lidu, muntz}@cs.ucla.edu

ABSTRACT

COCA (Collaborative Objects Coordination Architecture) was proposed as a novel means to model and support collaborations over the Internet. Our approach separates coordination policies from user interfaces and the policies are specified in a logic-based language. Over the past year, both the collaboration model and the specification language have been substantially refined and evaluated through our experience in building real-life collaboration systems. This paper presents the design of the specification language and illustrates the main ideas with a few simple design examples. Semantics, implementation, runtime support, and applications are also covered but not as the focus of this paper.

1 INTRODUCTION

Over the last decade, many CSCW(Computer-Supported Cooperative Work) systems have been built. However, in traditional systems, the coordination policies, such as those for access control and concurrency control, are often hard-coded into the system together with the user interfaces. Coordination policies are usually sensitive to the work style and organizational structure of individual groups. Not only do different groups have different needs, but the same group may require different policies in different phases of their collaboration. Therefore, it is important for system support to be flexible and adaptable. But in general, traditional CSCW systems are complex to build and do not easily accommodate changes.

We proposed a novel approach called COCA [16] to model and support collaborations over the Internet. Our approach is different from traditional CSCW systems in the following ways. First, we separate coordination policies from user interfaces in building collaboration systems. Second, we provide an executable specification language to describe coordination policies. Those policies are enforced at runtime by the COCA virtual machine (*cocavm*), a copy of which runs at each participant site. Third, at runtime participants in the same collaboration take on roles(e.g. the professor and student roles in distance learning). Different roles are each controlled by a different set of rules. As a result, the design and implementation of collaboration tools are greatly simplified. Tools are both easy to build from scratch[17] and to adapt from legacy tools[18]. The same set of tools can be reused in many different scenarios with different policies and without changes to the tools. Moreover, the coordination policies are explicitly and collectively specified instead of being scattered in all the involved components of the system. Systems thus built are more tractable to manage and evolve than those developed in traditional ways.

The notions of role and collaboration are well-accepted in object-oriented systems design and analysis, e.g. [34], [33], and [13]. However, their uses of those two concepts are different from ours, as was concisely illucidated by the following excerpt [27]:

> Role models provide excellent separation of concerns due to their focus on one particular collaboration purpose, while traditional class diagrams necessarily interwine all different object collaboration aspects. When composing role models, several aspects of object collaborations can be specified without prematurely committing to a class structure that might turn out to be too rigid later on.

In the CSCW literature, Intermezzo[8], QUILT[14], etc. used the concept of *role*. In Intermezzo and many other systems, roles are used only for access control. We use *role* as a unit to specify a wide range of coordination policies including access control, concurrency control[1], session control, etc. As a result, the language in Intermezzo is much simpler and limited in expressive power, as compared with the language in COCA. QUILT built

[1] It is not unusual that preferences are given to some class of participants over another when a conflict arises.

three roles into the group editor: reader, commentator, and co-author, each with different privileges. In COCA, the users are free to specify as many roles as the application requires.

An important difference between groupware and traditional software is the critical need for an efficient and scalable group communication model. Early collaborative systems were generally built based on grouping of multiple point-to-point communications. This approach incurs tremendous overhead on both the message senders and the communication paths, and latency increases with the size of the receiver population. Deering[6] proposed IP multicast as a better solution to this problem. It has been well-accepted in the networking literature (e.g.[10, 23]) that IP multicast is an efficient model for group communication, both for delivery of time-critical media streams and for non-realtime messages. We adopt IP multicast as the group communication model of COCA.

Major novelties of our specification language include the following. It uses roles as a unit to specify a wide range of policies including access control, concurrency control, session control, etc. It provides logic-based language constructs for efficient and flexible group communication. The simple syntax leads to concise and easily understood specifications. It is potentially possible to develop tools to mechanically verify and validate the specifications, to derive test cases, to detect deadlock situations, to reason about role behavior, etc.

Over the past one year, considerable experience with COCA has been accumulated and the system extended to improve the usefulness of COCA in practice. The runtime support system has been running since February 1998. Since the summer of 1998, we have been exploring application domains and building systems. Specifically, COCA has been proved applicable in the following domains: electronic meeting systems[17], online auction systems[18], and so forth. Today over 40,000 lines of code are running, including the core language runtime and applications. The performance has been good despite the fact that the prototype is done in pure JAVA.

The remainder of this paper is organized as follows. We first briefly overview the collaboration model of COCA in the next section. In section 3, we try to give the readers an initial feel of the language with a simple example. Section 4 describes our specification language. A more complete design example is discussed in section 5. Some important implementation issues are discussed in section 6. A more detailed comparison with related work is left to section 7. We conclude this paper in section 8 with some directions for future research and development.

2 GENERAL MODEL

A collaboration is a group activity (e.g. a course, a meeting, a game, etc.) which involves a group of participants. A participant is usually a human being but can also be a set of software agents. In a collaboration there are usually multiple participants each playing one or more roles. The concept of **role** refers to the set of participants governed by the same set of coordination policies. In a collaboration, one participant may be playing multiple roles and one role could be enacted by multiple participants at the same time. For example, in a project meeting, one participant plays the project *manager* role and the others are in a normal *member* role. The *manager* is able to see and modify the drawings of the normal *member*s, while the latter can see but are not allowed to modify the drawings of other participants.

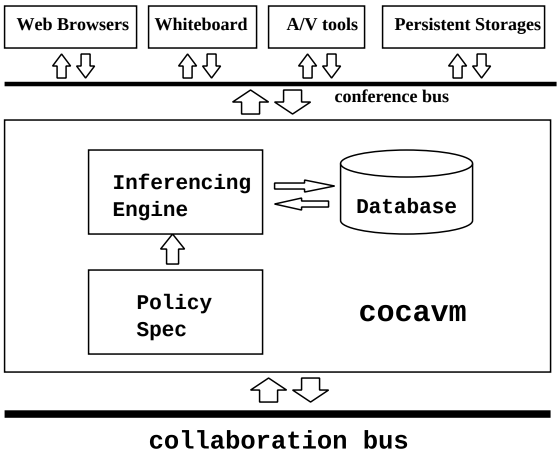

Figure 1: Internal structure of the *cocavm*

In our model, a *cocavm* runs at each participant site to enforce the coordination policies by controlling the interactions between this participant and other collaborators. As shown in Figure 1, a *cocavm* consists of an inferencing engine and an internal database. The inferencing engine monitors messages communicated in the collaboration, firing the active rules unified with the message, and performing actions according to the policy specification. The internal database maintains state information regarding the ongoing collaboration.

Participants in a collaboration use collaboration tools, such as web browsers[18], whiteboard tool[17], floor control tool[16], and the audio/video tools, to collaborate with each other. At each site, the collaboration tools and a *cocavm* are connected by a conference bus. All the *cocavms* in the same collaboration are connected by a collaboration bus. Those buses contain channels which communicate messages between connected entities. A channel can be both unicast(one to one) and multicast(one to many).

csdr, the collaborative session directory tool, is the simple runtime user interface of COCA. The user can use it to create sessions out of a given collaboration specification, browse existing sessions, and join a session by taking roles from it. In particular, the session creator must provide, among other information, actual IP addresses and port numbers for the channels declared in the collaboration bus. Individual participants must provide this information for the conference bus channels. Such bindings were termed relocation of role and collaboration respectively[16].

3 A SIMPLE EXAMPLE

Floor control is often used in CSCW systems to control the mutual exclusive access to shared resources. There are many different floor control policies[7]. In Figure 2 we specify a centralized policy. There are three roles. The floor *moderator* role (line 8-34) controls who can obtain the floor at any time. A floor *aspirant* (line 36-60) is a participant who needs to apply for the floor from the *moderator*. And the floor *holder* (line 62-63) is the participant who currently holds the floor. At one time only one participant can be the *moderator* and only one can be the floor *holder*. There are no constraints on how many participants can take the *aspirant* role concurrently.

As is shown in Figure 3, participants in the floor *moderator* and *aspirant* roles use their corresponding tools (or graphical user interface, GUI) to interact with each other through the *cocavms*. The tools are connected to the *cocavms* by channels named *local-in* and *local-out* respectively. And all the *cocavms* are connected by a channel *remote*. The *moderator* tool expects two commands from the *cocavm*: one to *request* the floor, and the other to notify the user to whom the floor is *granted* when a coordination decision is made. It sends three commands to *cocavm*: one to *grant* the floor to a given participant, one to *deny* a floor request, and the other to *revoke* the floor from the current floor *holder*. The *aspirant* tool sends two commands to the *cocavm*: one to *request* the floor, and the other *releases* the floor. Messages from *cocavm* to the *aspirant* tool are *grant*, to notify the *aspirant* to whom the floor is granted, and *deny*, to report that the floor request is denied.

```
1. collaboration sfc
2. {
3.     collaboration-bus
4.     {
5.         channel(remote).
6.     }
7.
8.     role moderator
9.     {
10.        conference-bus
11.        {
12.            channel(local-in).
13.            channel(local-out).
14.        }
15.
16.        on-arrival(gate(remote),request(Floor,Agent)) :-
17.            local-out !request(Floor, Agent).
18.
19.        on-arrival(gate(remote),release(Floor,Agent)) :-
20.            local-out !grant(Floor, self),
21.            take holder.
22.
23.        on-arrival(gate(local-in),grant(Floor,Agent)) :-
24.            remote !grant(Floor, Agent),
25.            isa(self, holder),
26.            drop holder.
27.
28.        on-arrival(gate(local-in),deny(Floor,Agent)) :-
29.            remote !deny(Floor, Agent).
30.
31. on-arrival(gate(local-in),revoke(Floor,Agent)):-
32.            remote !revoke(Floor, Agent),
33.            take holder.
34.    }
35.
36.    role aspirant
37.    {
38.        conference-bus
39.        {
40.            channel(local-in).
41.            channel(local-out).
42.        }
43.
44.        on-arrival(gate(local-in), request(Floor)) :-
45.            remote !request(Floor, self).
46.
47.        on-arrival(gate(local-in), release(Floor)) :-
48.            remote !release(Floor, self),
49.            drop holder.
50.
51.        on-arrival(gate(remote), grant(Floor, self)) :-
52.            local-out !grant(Floor),
53.            take holder.
54.
55.        on-arrival(gate(remote), revoke(Floor, self)) :-
56.            drop holder.
57.
58.        on-arrival(gate(remote), deny(Floor, self)) :-
59.            local-out !deny(Floor).
60.    }
61.
62.    role holder
63.    {}
64.}
```

Figure 2: A simple floor control policy.

In Figure 2, when an *aspirant* tries to request the floor, the command arrives at *aspirant cocavm*'s *local-in* gate (line 44). It is forwarded to the *moderator* via the "remote" channel (line 45). When this message arrives at the *remote* gate of the *moderator* (line 16), it is forwarded to the *moderator* tool (line 17), leaving the decision to the user.

When the participant in the *moderator* role decides to grant the floor to some participant (line 23), the message is multicast via the *remote* channel (line 24), meanwhile the *moderator* drops the *holder* role if she is currently in that role (line 25-26). When the participant *cocavm* receives the message which grants her the floor (line 51), the message is reported to the tool (line 52) and the floor

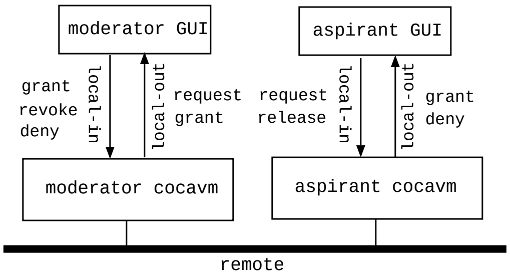

Figure 3: The floor moderator and aspirant roles.

holder role is taken (line 53).

To be concise, when a floor request is denied, the aspirant is notified (line 28-29, 58-59). When the current floor *holder* releases the floor, she drops the *holder* role and meanwhile multicasts the message so that the *moderator* assumes it (line 47-49, 19-21). And when the *moderator* decides to *revoke* the floor, the *moderator* assumes the *holder* role and the current floor holder drops it (line 31-33, 55-56).

The floor *holder* role is fluid. Anyone takes on this role when granted the floor by the *moderator* and drops it as a result of releasing the floor or when the floor is preemptively revoked. At this moment, there is no rule specified for the floor *holder* role. More in-depth discussion of this role resumes in section 5.

4 THE SPECIFICATION LANGUAGE

4.1 The Core Language

Following the conventions of Prolog, variables begin in upper case. Constants, functors and predicates begin in lower case. Anonymous variables are denoted by underscores.

Definition 4.1 *A* **term** *is defined inductively as follows:*

1. *a variable is a term*
2. *a constant is a term*
3. *if f is an n-ary function symbol and $T_1, T_2, ..., T_n$ are terms, then $f(T_1, T_2, ..., T_n)$ is a term (called a compound term), $n \geq 0$. For convenience, we use f/n to denote a functor f with arity n.*

Definition 4.2 *A* **formula** *is defined inductively as follows:*

1. *if p is an n-ary predicate symbol and $T_1, T_2, ..., T_n$ are terms, then $p(T_1, T_2, ..., T_n)$ is a formula (called an atomic formula or more simply, an* **atom***), $n \geq 0$. We use p/n to denote a predicate p with arity n.*
2. *if F is a formula, so is (***not** *F).*

Definition 4.3 *A* **literal** *is an atom or the negation of an atom. A* **positive literal** *is an atom. A* **negative literal** *is the negation of an atom.*

Definition 4.4 *If q is a positive literal, $p_1, p_2, ..., p_n$ are literals, $n \geq 0$, then*

$$q : -p_1, p_2, ..., p_n.$$

is a **rule**. *Atom q is called the head, and $p_1, p_2, ..., p_n$ combined is called the body of this rule.*

4.2 Database Operations

The above-defined core language is no different from Prolog. To avoid unauthorized modification to coordination policies and for more efficiency, however, COCA deliberately separates the rule base and the database. We use a set of database operators rather than **assert** and **retract** in Prolog. Those Prolog predicates do not distinguish predicate definitions and database facts.

Definition 4.5 *We define* **database formulas** *as follows. If T is a compound term, the following atomic formulas are database formulas. Keywords* **query**, **add**, *and* **delete** *are database operators.*

1. **query** *T: evaluates to true if compound term T is unified with* **any** *compound term in the database;*
2. **add** *T: add compound term T to the database, evaluates to true if it succeeds;*
3. **delete** *T: delete a compound term unified with T from the database nondeterministically, evaluates to true if it succeeds.*

Those operators are atomic, synchronized, and nonblocking. In our implementation, database facts are hashed by functor names. Mutual exclusion is enforced so that no two operators can work at the same time on compound terms with the same functor name.

We can extend this set to support blocking operations as well. For example, $\textbf{query}^{sync}$ and $\textbf{delete}^{sync}$ can be used to denote blocking database operations respectively. Although they are only for synchronization within the same *cocavm*, those operators have a close relationship to the tuple space operations in Linda[12]. $\textbf{query}^{sync}$ corresponds to **rd**, $\textbf{delete}^{sync}$ to **in**, **query** to **rdp**, **delete** to **inp**, and **add** to **out**.

The role of the database in COCA is two-fold. First, it provides an ephemeral memory (as opposed to persistent storage systems such as a relational database) for capturing and recording the state information regarding the ongoing collaboration, based on which many a coordination decision is made. Second, it implements a synchronization mechanism based on which concurrenct activities in each *cocavm* are coordinated.

4.3 *cocavm* Identification

A *cocavm* has a unique identification. This id is denoted by keyword **self**. It could include the following information: host name, user login name, command port number on which this *cocavm* receives commands, a time stamp when this *cocavm* is launched, a list of roles this *cocavm* is enacting, etc.

4.4 Communication

4.4.1 Channel and Gate

The following channel declaration defines a channel with name C:

channel(C).

We use the term **gate** to denote a service access point to a communication channel. Gates use the same name as the corresponding channels.

Definition 4.6 *We define the following* **communication formulas** *as complementary to* **Definition 4.2**, *where operators "!" and "?" are called* **offer** *and* **accept** *respectively.*

1. G **!** $(T_1, T_2, ..., T_n)$
2. G **?** $(T_1, T_2, ..., T_n)$

(1) sends out a list of terms through gate G, (2) blocks until a list of terms arriving at gate G unify with T_1, T_2, ..., and T_n as a whole.

Definition 4.7 *We define active rules as having the following form:*

on-arrival(**gate**(G), $T_1, T_2, ..., T_n$) :-
$p_1, p_2, ..., p_m$.

4.4.2 Dynamic Grouping

A role name say R can be used as a channel without being explicitly declared in the collaboration bus to deliver messages to all the participants *currently* in the role. However, since only participants enacting role R can receive those messages, we require that only R can define **on-arrival** rules at the gate it represents.

Predicate **channel**/2 can be used to declare channels dynamically. The following names a channel C which connects a group of participants denoted by $Agent_1$, $Agent_2$, ..., $Agent_n$:

channel(C, [$Agent_1$, $Agent_2$, ..., $Agent_n$]).

$Agent_i$ can be a defined static channel name, a role name, some **self**, or even another defined dynamic channel name. A channel (static or dynamic, excluding that denoted by a role name) can be later redefined. The same channel predicate, if the second argument is a variable, can be used to get the list of agents connected by a given channel.

An offer predicate can send messages through a variable channel like the following.

C !$(T_1, T_2, ..., T_n)$

However, the variable C must have been bound to an agent (some **self**), an agent group, a role name, or a declared channel name.

There are two ways to receive messages sent via a dynamical channel. One is to define active rules at the gate denoted **self**. Another way is to define an active rule in which the gate name is a variable. When a message arrives not unifying with any active rules with constant gate names, the variable gates rules will be tried.

4.5 Events

Events include message arrivals, timer signals and user posted events. Messaging events were discussed in the previous subsection. Here we discuss the latter two.

4.5.1 Event Generator

The following predicate sets a timer.

set-alarm(*TimerName*(*FireTime*)).

After a timer is set up, it fires only once and posts an event with the same name as the timer. If it is to be fired repeatedly, it must be reset each time. A timer event is always asynchronous. The firing time of a timer can be a relative time, i.e, some period from now, or an absolute time set on a particular time or date.

The following predicate posts a user-defined event either synchronously or asynchronously.

post-event$^{sync|async}$ (*EvtName*($T_1, ..., T_n$)).

4.5.2 Event Handler

When an event occurs, the corresponding event handler or active rule is fired. If the event is asynchronous, the event handler will be executed in parallel with the current thread. If it is synchronous, the firing will be equivalent to calling a predicate in the current thread. An event handler is defined as below:

on-event(*EvtName*($T_1, ..., T_n$)) :-
$p_1, ..., p_m$.

4.5.3 **wait-for** Predicates

Predicate **wait-for**/1 can be used by the current thread to wait for some event to occur. In previous releases of COCA, when a message was expected from a certain object, we needed an active rule to wait for it. This new construct can make it simpler and clearer. For example, when a message is sent to an object and an answer is expected from it, we simply specify the following.

p :- ...
g !(SomeRequest),
wait-for(SomeEvent),
...

A second argument can be added to **wait-for** which specifies the timeout period before **wait-for**/2 can be failed. Any nonnegative number is valid or the symbol "∞" can be used to denote infinity. If it is an unbound variable, when the predicate succeeds this variable is bound to the length of time from when the **wait-for** was executed until the event occurred.

4.6 Role

4.6.1 Defintion

At least one role must be defined in a collaboration. Each role definition has the following form.

role <role name>
{
[**conference-bus** { <channel declarations> }]
<rules>
}

A conference bus declaration is optional. We can define roles which do not interact with other local components. The conference bus is used for communication between the *cocavm* and the collaboration tools for each role. So channel names declared in a role are local to that role. Predicate **isa**/2 can be used to test if a given participant (some **self**) is currently in some role.

4.6.2 The Daemon Role

A collaboration typically defines a special role called *daemon*, which controls for example who is allowed to take which roles and how many participants can take a certain role [2]. When a participant attempts to take a role from an ongoing session, the daemon of that session is contacted. Possibly authentication is performed. Only when the participant is qualified by the session control policy does she obtains the rule set specified for that role. A session without a daemon role will be open to all, i.e., any participant can take any role.

The session control may not necessarily be centralized. It is possible for a session to have multiple daemons, e.g. for availability and scalability. The user can specify a daemon role which can be taken by multiple agents.

4.6.3 Constructor and Destructor

Operators **take** and **drop** can be used to take a role and to drop a role respectively. In the following, R stands for the name of some role and $n \geq 0$.

1. **take** $R(T_1, T_2, ..., T_n)$
2. **drop** R

To perform initialization when a role is taken and to clean up when a role is dropped, we define constructor and destructor rules as follows.

1. **on-take**($T_1, ..., T_n$) :- $p_1, ..., p_m$.
2. **on-drop** :- $p_1, ..., p_m$.

Arguments of the constructor rules are optional. For each role we can define one or more constructor rules which are fired at a *cocavm* when a participant takes on this role. Only the constructor whose arguments unify with those of the **take** predicate is chosen to execute upon initialization.

However, we define at most one destructor for each role. The destructor is fired at a *cocavm* when the role is dropped. At this point, it becomes inappropriate for any rules defined for that role to continue their execution. Our strategy is to mark all the arrival message and event queues in the *cocavm* so that no more messages or events will be processed after the mark. When the *cocavm* comes to a quiescent state, the destructor is fired to clean up.

4.7 Collaboration

A **collaboration** is defined in the following form:

collaboration <collaboration name>
{
[**collaboration-bus** { <channel declarations> }]
<role definitions>
}

[2] [27] described four possible role constraints: role-dontcare, role-implied, role-equivalent, and role-prohibited. Actually more possibilities exist. For example, one can not take on the student role in a given course if she was once the teaching assistant of the same class. Sometimes it is up to the concensus (voting) of the admitted participants to decide whether to admit a new participant, etc.

An operator "::" is used for identifying the scope of a name. The normal form is $S :: R :: C$, where S stands for collaboration or session name, R for role name, and C for channel name or predicate name. The prefixes can be omitted if there is no confusion.

A collaboration may consist of sub-collaborations. For example, in activities such as conferences and classrooms, it is not uncommon that participants with similar interest form subgroups. So the language should provide constructs to facilitate establishing new sessions out of given collaboration types and tearing down existing sessions. The following operators are defined for this purpose.

1. **create** *Session*, *Collaboration*
2. **destroy** *Session*

After a session is created, the user can join it by command "**take** *Session* :: *Role*".

4.8 Policy Composition

To better capture and specify coordination policies, we often need to decompose a complex collaboration into smaller pieces, and compose existing pieces to form a larger one. Here we introduce some constructs for policy composition. These constructs provide a basis for defining a library of reusable coordination policies and predicates.

4.8.1 Parameterization

The first construct is for definition of policy templates. For example, in defining the floor control policy in section 3 to control the drawing floor, we discover that it also applies to the audio floor and the video floor in multimedia collaborations[7]. It would be advantageous then to formalize the floor name so that the same specification can be reused.

template<Floor>
collaboration *sfc*
{
 ...
}

Then we can actualize this policy template by replacing the formal "Floor", such as *sfc*<drawing-floor> and *sfc*<audio-floor>, which becomes the name of a collaboration type. Multiple formal names in a policy template are allowed. The occurrences of each will be substituted by corresponding actual names.

4.8.2 Inheritance

The second construct is for policy inheritance. For example, if we have already defined collaborations A and B. Now we want to define a new collaboration C. And it turns out we can reuse the definitions in A and B. So C just needs to extend those two existing collaborations.

collaboration C **extends** A, B
{
 ...
}

The collaboration bus channels and roles of the resulting collaboration C are a union of those defined in A, B, and itself. Wherever there is a name conflict, the above scoping operator "::" and the appropriate prefixes will be applied to resolve it automatically.

Roles can also be **extended** from roles defined in ancestor collaborations or other roles defined in the same collaboration. In the above example, role r_3 in collaboration C can extend both role r_1 in collaboration A and role r_2 in collaboration B as follows.

role C::r_3 **extends** A::r_1, B::r_2
{
 ...
}

In a sequel, conference bus channels and rules of C::r_3 will be a union of those defined in A::r_1 and B::r_2 respectively.

4.8.3 Polymorphism

The ordering of rules is important. The rule set of a derived role is always put before those of its parents. When a predicate is evaluated, the rule set of the role which is on the lowest level of the inheritance hierarchy is consulted first. If several parents exist, inherited rules are put by the the order in which the parents appear. If there are multiple layers of inheritance, the same rule applies recursively. Inheritance of collaborations or roles are acyclic. In the case that a role is inherited more than once in the same layer or different layer, the rule set of that role is included only once.

Constructor and destructor rules are invoked when a role is taken or dropped. Those defined in the lowest hierarchy are considered first. In the above example, in the constructor of C::r_3, if the user wants to execute the initialization code of role A::r_1 as well, the constructor of the latter should be called explicitly as follows.

C::r_3::**on-take**($T_1, ..., T_n$) :- ...
  A::r_1::**on-take**($T_1, ..., T_n$),
  ...

In COCA we use a role name as an implicit communication channel and active rules can be defined at the gate it represents. For example such an active rule can be defined for role $A::r_1$. When a message is sent to gate $A::r_1$, participants in role $C::r_3$ should also receive it. The reason is that the active rule defined at gate $A::r_1$ is inherited by $C::r_3$ and becomes a part of it.

5 A COMPLETE EXAMPLE

Here we first consider a whiteboard meeting, then discuss a concurrency control policy, and show how these two policy modules can be composed into one for project meetings. We assume that all participants join a meeting at about the same time. It is impossible to have one short example to illustrate all the language constructs. Policies to handle late joins and other examples can be found in [15].

5.1 Whiteboard Meeting

We have implemented a whiteboard tool[17] which can be used either in a single-user mode or in a multi-user mode. In the former case, the user can draw, delete, and modify objects such as lines, circles, and free-hand pictures, load images, etc. In a multi-user mode, COCA can be used to enforce specified coordination policies such as those for access control, concurrency control, and session control. For example, a policy can be specified where, when a user attempts to annotate an object, the command is sent to the owner of that object. If the owner personally does not like the annotation, then it will not appear on both parties' whiteboards and will not be propagated to other sites. In the following, however, we only consider a simplest case in which everybody can do anything and what you see is what I see (WYSIWIS). In[17] we showed how the same whiteboard tool can be used under many different coordination policies.

```
collaboration meeting
{
    role drawer
    {
        conference-bus
        {
            channel(local-in).
            channel(local-out).
        }

        on-arrival(gate(local-in), Cmd) :-
            drawer !Cmd.

        on-arrival(gate(drawer), Cmd) :-
            local-out !Cmd.
    }
}
```

Figure 4: A policy for WYSIWIS meeting.

In this specification, each command from the whiteboard is propagated to all participants and any commands from other sites are sent to the local whiteboard, both without examination. However, conflicts can arise. For example several participants may happen to modify the same object on the screen. Some concurrency control policy must be enforced to resolve such conflicts. Here for simplicity, we adopt a floor control scheme in which participants take turns to draw on the whiteboard. Specification of other concurrency control policies (e.g. [9]) can be found in [15].

In a floor control scheme, a particpant must have the floor to draw on the whiteboard. We change the first rule to the following.

```
on-arrival(gate(local-in), Cmd) :-
    isa(self, holder),
    remote !(Cmd).
```

Each time the local participant attempts to draw an object, the command is not propagated to other sites unless the participant currently is the floor *holder*. Note this policy is not complete. It says nothing about how to become a floor *holder*.

5.2 Floor Control

In section 3, we discussed a rather simple floor control specification. However, the *sfc* policy thus defined may not work well in large-scale collaborations over the Internet. Here we extend it to handle some exceptions which are typical in such situations.

In a floor control policy it is important to make sure there is always one and only one participant in the *moderator* role. If the *moderator* is lost, for example due to process failure or network partition, some other participant must be chosen to become the *moderator*. There is a similar issue with regard to the floor *holder* role. If the floor *holder* is detected lost, then the *moderator* must be able to reproduce a floor so that the collaboration can continue. As specified in Figure 5, we use a soft-state protocol for this purpose. The *moderator* and the floor *holder* periodically send out a liveness report, from which the other participants know their liveness and try to repair if there is a loss.

The floor *holder* adds to its database the fact that it is the current *holder* of the floor. This fact is deleted when the role is dropped. Then a timer is set so that it multicasts a liveness report every 10 seconds.

When the *moderator* role is taken, the participant automatically assumes the floor *holder* role. Here two timers are set. The "reporter" sends out the liveness report every 10 seconds. And the "checker" checks the database records every minute. If no liveness report is received from the floor *holder* in one minute, the *moderator* itself

```
collaboration fc extends sfc
{
    role holder extends sfc::holder
    {
        on-take :-
            add holds(self, floor),
            set-alarm(reporter(10000)).

        on-drop :-
            delete holds(self, floor).

        on-event(reporter(Period)) :-
            remote !live(self, holder),
            set-alarm(reporter(Period)).
    }

    role moderator extends sfc::moderator
    {
        on-take :-
            take holder,
            set-alarm(reporter(10000)),
            set-alarm(checker(60000)).

        on-event(reporter(Period)) :-
            remote !live(self, moderator),
            set-alarm(reporter(Period)).

        on-arrival(gate(remote), live(_, holder)) :-
            time(Now),
            add live(holder, Now).

        on-event(checker(Period)) :-
            query live(holder, LastT),
            time(Now),
            Now - LastT >= Period,
            take holder,
            set-alarm(checker(Period)).
    }

    role aspirant extends sfc::aspirant
    {
        ...
    }
}
```

Figure 5: The extended floor control policy.

assumes the floor *holder* role. When a report comes from the *holder*, however, the database is updated accordingly.

The *aspirant* role can be specified similarly. The differences are that an *aspirant* does not need to send the periodic liveness reports and that it must detect the *moderator* loss. When the *moderator* is discovered to be lost, a new one must be chosen from the participants. A number of policies can be applied here depending on the taste of the participants. For example, the decision can be made by voting, by the alphabetic ordering of participants' names, etc. For reasons of space, the *aspirant* specification is listed in [15] instead.

```
collaboration projmeeting
extends meeting, fc
{
    role manager extends moderator, drawer
    {}

    role member extends aspirant, drawer
    {}
}
```

Figure 6: The synthesized project meeting policy.

5.3 Synthesis

Suppose we want to have a project meeting in which participants take turns to draw on the whiteboard. We should somehow put together the above specified two collaboration types. In Figure 6, the collaboration we want, *projmeeting*, extends collaborations *meeting* and *fc*. Only two roles are available here. The project *manager* is composed from the floor *moderator* and the *drawer*. And the normal project *member* is a composition of the *aspirant* and the *drawer*. So the *manager* controls the floor. Participants in both roles are free to draw on the whiteboard once they become the floor *holder*. Roles inherited, e.g. *drawer*, *moderator*, *aspirant*, and the floor *holder* are made invisible to participants of the collaborative sessions created out of this collaboration.

6 DISCUSSION

This section discusses some important implementation issues. We first briefly present a well-accepted algorithm in subsection 6.1 for maintaining temporal relationships between messages. The support of transaction is discussed in subsection 6.2. Subsection 6.3 discusses the semantics of the event constructs introduced in section 4. In subsection 6.4 we show how to specify messaging policies for more flexibility. Subsection 6.5 illustrates how to specify some predicates which were introduced as built-in predicates in section 4 using the language itself. Backtracking is discussed in subsections 6.6.

6.1 Logic Clock

Each *cocavm* has a logical clock. When a message is sent out, the local logic clock is advanced by one. When a message is received from another *cocavm* with logic time t_1, and the logic clock of this *cocavm* reads t_2. If $t_1 \leq t_2$, then we set $t_2 \leftarrow t_1 + 1$. This scheme maintains the causal relationship of messages across *cocavms* and can be easily extended to support a consistent total ordering of all messages[32].

6.2 Transaction

Within the *cocavm*, there could be multiple threads of execution. It is necessary to have some synchronization mechanism between concurrent threads. For this purpose, operators of the internal database are atomic and synchronized. And we further introduce a special symbol "@" as syntactical sugar to define atomic predicates and sequences. An atomic sequence is analyzed before execution. All the database functors (or relations) involved in such a sequence will be locked when the critical section is entered. The lock is released when the sequence succeeds or is failed. A thread blocks if it can not obtain the lock. To prevent deadlocks, all the involved functors must be locked or none.

An atomic sequence corresponds to the concept of transaction. To support transactions in the internal database, we would have to support rollback, i.e., when one predicate in the atomic sequence fails, the effects of the whole transaction must be undone. This is not hard to support though. For example, we can analyze the sequence and make a backup copy of those functors that could be affected. Once an operation fails, we can overwrite the original copy with the backup copy. If the whole transaction succeeds, we simply throw away the backup copy.

6.3 Event Semantics

The general principle is to finish the execution of a rule as fast as possible. When an event is posted (synchronously or asynchronously), the blocking **wait-for** predicates are given higher priority than the **on-event** rules. If there are several threads waiting for an event, then only one of them is picked up nondeterministically to evaluate its **wait-for** predicate. If no such thread is eligible to continue, then the **on-event** rules are considered.

Messages are processed in the same way. Predicate **accept** corresponds to **wait-for** and **on-arrival** to **on-event**.

6.4 Messaging Policy

In our current implementation, a thread is waiting for message arrivals at each gate. By default, messages are processed in an FIFO order. It is sometimes useful to change this order for example according to the role of the sender, the content or the logic time of the message, etc.

We can easily specify the messaging policies. The **on-arrival** rules defined at each gate can insert the arrived messages to a central queue maintained in the database. An (infinite) loop there can activate various event handlers which process messages in the queue. The following is only one way to implement a messaging policy, where implementation of the underlined predicates are left to the user. These two predicates are policy-dependent.

on-take :-
  add queue([]),
  post-event^{async}(execution-loop).
on-arrival(**gate**(G), ...) :-
  @enqueue(G, args(...)).
enqueue(G, Msg) :-
  delete queue(L0),
  <u>insert(L0, msg(G, Msg), L1),</u>
  add queue(L1).
on-event(execution-loop) :-
  <u>dequeue(G, Msg),</u>
  post-event^{sync}(msg(G, Msg)),
  post-event^{sync}(execution-loop).

Timer events and user-posted events can also be processed through the queue. To do this, the built-in **post-event** predicates must be overridden to insert the events into the queue. But care must be taken so that the **post-event** predicates used above still work as intended.

This scheme has significant advantages. First, the need for concurrency control inside the *cocavm* is ameliorated, since at any moment there is only one rule active in a *cocavm*. We can maintain a central event queue, which includes all message arrivals, timer or user-posted events, instead of a pure message queue. Second, it is easier to stop the execution of the current rule set say when the participant wants to switch to a different role or a different policy (rule set) at runtime[19]. When the command to freeze the runtime state is received, each *cocavm* can simply neglect messages with logical time later than that of the freeze command.

This design assumes that predicates in event handlers can finish within a reasonable time. When a blocking predicate is called, it is still feasible to specify a sophisticated threads scheduling policy to suspend the waiting thread and continue to process another event. But if a predicate takes too long or loops forever, however, all the other events must suffer even indefinitely, as such situations are hard to detect mechanically.

6.5 Some Predicates Revisited

A number of database predicates can be easily defined by atomic sequences and atomic predicates, for example, **update**, **collect**, **deleteAll**, etc. While **collect** and **deleteAll** must be implemented assuming a total ordering of facts, **update** does not. Predicate **update** is implemented simply by an atomic sequence as follows.

update(OldT, NewT) :-
  @(
    delete OldT,
    add NewT
).

The blocking database operators introduced above can be implemented with **wait-for**. For example, predicate "**query**^{sync} f(a, b)" can be defined by the following. Operator **delete**^{sync} can be defined similarly.

query^{sync} f(a, X) :-
  query f(a, X).
query^{sync} f(a, X) :-
  wait-for(add-f(a, X)).

However, when a fact say "f(a, b)" is added into the database, an event "add-f(a, b)" must be posted either by the user or by the runtime system.

The **accept** operator can also be elegantly defined in terms of **wait-for** as follows, where g is the name of a gate.

g ? (X, a) :-
    wait-for(g(X, a)).
on-arrival(**gate**(g), X, a) :-
    post-event^{async}(g(X, a)).

Predicate **wait-for** also provides a possible means to implement locking and unlocking of database relations, as follows.

lock(Object) :-
    query locked(Object),
    wait-for(unlocked(Object)),
    add locked(Object).
lock(Object) :-
    add locked(Object).
unlock(Object) :-
    delete locked(Object),
    post-event^{async}(unlocked(Object)).

Once an "unlocked(Object)" event is posted, only one thread waiting for it is woken up nondeterministically to consider its **wait-for** predicate. So this definition guarantees that the lock not be grabbed by several threads at the same time.

Since those predicates can be specified, we do not really have to implement them.

6.6 Backtracking

The essence of don't-know nondeterminism is that failing computations "dont't count" and only successful computations may produce observable results. The don't-care interpretation of nondeterminism, on the other hand, requires that results of failing computations be observable. Hence a don't-care nondeterministic computation may produce partial output.

Don't-care nondeterminism is essential in modeling concurrent interactive systems[31]. In concurrent systems it would be too expensive to backtrack a choice, even if it proves to be wrong. The reason is that a choice, and the actions done afterwards, have already influenced the environment: to maintain consistency the whole system should backtrack, but this is too inefficient. It is rather preferrable to provide mechanisms to control the choices, so to avoid as far as possible that wrong decisions are taken[5].

COCA has constructs for communication, asynchronous event posting, and transactions which produce side effects. We take both the don't-know and don't-care interpretation of nondeterminism. Suppose the following is the execution trace of an active rule where $m, n \geq 0$ and q_1 is the first predicate with side effect.

h :- $p_1, ..., p_m, q_1, ..., q_n$.

This is similar to the guarded Horn clauses notation in concurrent logic programming languages[31, 5]. The execution of an active rule is implicitly divided into two parts. $p_1, ..., p_m$ corresponds to the guard part, and $q_1, ..., q_n$ the body. A failed predicate in the guard part may incur backtrack. But once the body part is entered, the side effects ever generated cannot be undone.

As was introduced in subsection 6.2, failure of a predicate within a transaction causes the whole transaction to rollback. Not to contradict the don't-care semantics, we restrict that the definition of a transaction should not call (directly or indirectly) predicates that communicate or post events.

7 RELATED WORK

7.1 Other Approaches in Collaboration Specification

Trellis[11] and DCWPL[3] also advocated separating coordination from computation so that the former can be specified in a more declarative way and interpreted at runtime.

DCWPL provides a rather ad hoc scripting language with facilities for modeling and controlling access to shared artifacts. It allows for definition of a fixed set of artifact attributes such as "Authorized", "MaxInstance", and trigger-like attributes such as "Preexecution", "Postexecution". Even if we can presume that the set of attributes provided is sufficient to capture every possible artifact in all collaborations so that we do not need to extend the language under any circumstances, the problem is that whenever a policy or function is not directly available in the declarative language DCWPL, the user still needs to program it in a procedural language. In this way the declarativeness of the language is undermined. The language we propose in COCA, however, is self-contained. The users can specify whatever coordination policies they want without constantly resorting to a procedural language, assuming the tools have the required functionality. In this way the declarative feature is reserved.

Trellis used a variant of Petri Nets, CTN or colored timed nets, to specify group interaction protocols. Trellis is a client/server architecture in which a centralized controller processes service requests from clients according to the specification. We argue that a centralized architecture like Trellis may work well for small groups which primarily exchange textual messages. But for large groups with hundreds of participants, especially when multimedia data is widely communicated, it is not clear how it is modeled and handled in Trellis and what level of performance can be expected. It is also not clear, at least in the paper, how CTN models the collaboration of large groups whose participants are highly fluid, and how exceptions, such as floor loss and unexpected loss of the current floor holder, are handled in a Trellis specification.

[26] attempted to use LOTOS[22] to specify a group drawing tool. However, our observation is that LOTOS and other process algebra based formal methods are not convenient for specifying collaborations. In LOTOS, communicating components must be explicitly connected by parallel composition operators. This proved sufficient for some situations where communicating components are fixed and relatively small in number. But in many collaborations, only the participant types (namely the roles) are fixed, while the number of involved participants is large and fluid, the exact number cannot be determined at specification time. LOTOS is not directly convenient for specifying such situations. And even for small and fixed-population collaborative situations, LOTOS is also not always convenient. For example, in order to model situations where a process offers a value and several other processes accept it, LOTOS uses a multi-way synchronization which fails if even one process fails to synchronize at that point. Many collaborations can actually tolerate such exceptions by allowing processes that have received the value to proceed while ignoring or trying to re-transmit the lost data to processes which are temporally unavailable due to communication link delay or failure. We replace the parallel composition operators in LOTOS with the collaboration bus and the conference bus, synchronization among communicating components are achieved by explicit message passing.

7.2 Coordination Languages

Moses[24] was originally intended to make the Linda[12] communication safer by ensuring that the interaction of each process with the shared tuple space be managed by a controller. Each controller enforces a set of rules to capture the following two events: when a message is sent out by the local agent, and when one arrives at this agent, the local control state is checked, transformation is possibly performed, then this message is either forwarded to the tuple space, or delivered to the local agent, or blocked according to the rule definition. Our work is different from Moses in the following ways: (1) we explicitly divide participants in a policy group by roles. Different roles are controlled by different set of rules rather than all participants governed by the same set of rules as in Moses. Messages can be directed to roles (fluid sets of participants) as well as individual participants. We further allow participants to dynamically join and leave a collaboration by taking and dropping roles, and support the dynamic modification of rules at runtime. (2) We borrowed the concept of "gate" from LOTOS to denote the points where our controller, the COCA virtual machine, interacts with its environment, and define active rules upon the arrival of messages at these gates. In this way, the controller can monitor messages arriving at multiple different gates instead of just one to and one from the tuple space as defined in Moses.

Both the tuple space in Linda and the internal database in COCA are associate memory for coordination between concurrency activities. Linda is often criticized for its overheads to maintain a tuple space. Such overheads become more aggravated when situated in large-scale distributed systems. In COCA, there are actually two levels of concurrency. First, activities of all the participant *cocavms* are concurrent. Their coordination and synchronization are achieved by explicit, asynchronous message passing. Second, within each *cocavm*, multiple threads are executed concurrently. Their coordination is primarily through database operations which are correspondent to those in Linda. The cost of maintaining such an internal database is trivia, if not none.

In Manifold[1], ports are unidirectional and are considered as part of the definition of a component or *manifold*. Channels are established between the output port of a manifold to the input port of another manifold. Although this kind of connections are not restricted to be one-to-one, multi-point communication must be fulfilled by connecting one output port to multiple input ports. So it is not the multicast bus architecture as we are using.

7.3 Logic Programming

In concurrent logic programming (CLP) languages[31], each goal atom is viewed as a process. Concurrent processes communicate and synchronize via instantiation of shared logical variables. In COCA, however, an active rule is fired upon the arrival of a message at a gate or more generally the occurrence of an event. Concurrency only occurs among those active rules. We do not deliberately pursue the fine-granule parallelism in CLP languages. Concurrent threads in the same *cocavm* synchronize through operations against the shared internal database and different *cocavms* communicate through explicit message passing.

A number of object-oriented extensions to Prolog have been proposed[4]. As was discussed in section 4.8, the extensions we made in COCA, however, are no more than syntactical sugars. They do not have any impact on the language semantics.

Delta-Prolog[25] augmented Prolog with CSP-like communication primitives. But it is not reactive, since it may backtrack on communication[31]. [30] studied the embedding of Linda in a CLP language. Concurrent constraint programming[28] embodies explicit mechanisms for communication and synchronization consisting of two kinds of actions, *ask* and *tell*. Semantics about concurrency and communication in logic has been well-studied in the literature, e.g. [25], [2], and [29].

8 CONCLUSIONS

This language is not completely new. It has language elements from Prolog, concurrent logic programming languages, process algebras, Linda, and object-oriented programming languages. It also has concepts such as atomic sequence (transaction) and events. To better model collaborations it supports the concept of role and role operators. It would be interesting and challenging to develop a formal semantics of this language so that the many pieces from various sources can be reconciled in one elegant mathematic framework.

The latest version of COCA includes the following extensions. A set of role operators were introduced in [19] to model runtime dynamics in collaborative systems, e.g., operators for switching between different roles, transferring a role between participants, evolution of coordination policies on the fly, in addition to the take role and drop role operators discussed in Section 4.6. Coordination policies specified in COCA can be verified, as is discussed in [20]. A new application was explored in [21] to exercise the "dynamic grouping" feature discussed in Section 4.4.2. Further information on COCA and its applications can be found in [15].

ACKNOWLEDGEMENT

Valuable comments from Dr. Lalita J. Jagadeesan at the Bell Labs and Dr. Stott Parker at UCLA improved the presentation quality of this paper.

REFERENCES

1. F. Arbab, I. Herman and P. Spilling, An overview of Manifold and its implementation. *Concurrency: Pratice and Experience*, 5(1):23-70, February 93

2. Anthony J. Bonner and Michael Kifer, Concurrency and Communication in Transaction Logic, In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, Bonn, Germany, Sept. 1996

3. Mauricio Cortes and Prateek Mishra, DCWPL: A Programming Language for Describing Collaborative Work, *ACM CSCW'96 Proceedings.*

4. Andrew Davison, A Survey of Logic Programming-based Object-Oriented Languages, Technical Report 92/3, Department of Computer Science, University of Melbourne, Australia

5. Frank S. de Boer and C. Palamidessi. From Concurrent Logic Programming to Concurrent Constraint Programming. In G. Levi (editor), *Advances in logic programming theory.* Oxford University Press, 1993.

6. Steven Deering, Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990

7. H.-P. Dommel and J.J. Garcia-Luna-Aceves. Floor Control for Multimedia Conferencing and Collaboration. *ACM Multimedia'97*, 5(1), Jan. 1997

8. W. Keith Edwards, Policies and Roles in Collaborative Applications, *ACM CSCW'96 Proceedings*

9. C.A. Ellis and S.J. Gibbs. Concurrency Control in Groupware Systems. *ACM SIGMOD'89 Proceedings*, Portland, Oregon

10. S. Floyd, V. Jacobson, C. Liu, S. McCanne and L. Zhang, A Reliable Multicast Framework for Lightweight Sessions and Application-Level Framing, *ACM SIGCOMM'95 Proceedings*, Boston, 1995

11. Richard Furuta and David Stotts, Interpreted Collaboration Protocols and Their Use in Groupware Prototyping, *ACM CSCW'94 Proceedings.*

12. David Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, Vol.7, No.1, Jan. 1985

13. G. Gottlob, M. Schrefl, and B. Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3), July 1996

14. Mary D. P. Leland, Robert S. Fish, and Rober E. Kraut, Collaborative document production using quilt, *ACM CSCW'88 Proceedings.*

15. Du Li. Home of COCA: Runtime Support and Example Applications, `http://www.cs.ucla.edu/~lidu/coca/`

16. Du Li and Richard R. Muntz, COCA: Collaborative Objects Coordination Architecture, *Proceedings of ACM CSCW '98*, Nov. 1998, Seattle

17. Du Li, Zhenghao Wang, and Richard R. Muntz, "Got COCA?" A New Perspective in Building Electronic Meeting Systems, in *Proceedings of WACC'99 Conference on Work Activities Coordination and Collaboration*, Feb. 1999, San Francisco

18. Du Li, Zhenghao Wang, and Richard R. Muntz, COCA Web: a Generic Platform for World-Wide Collaboration and Electronic Commerce, invited poster, to appear in *the 8th Intl. World Wide Web Conference*, May 1999, Toronto

19. Du Li and Richard R. Muntz. Runtime Dynamics in Collaborative Systems. To appear in the *Proceedings of ACM Group'99 International Conference on Supporting Group Work*, Phoenix, Arizona, November 1999.

20. Du Li, Lalita J. Jagadeesan, James D. Herbsleb, and Patrice Godefroid. Verification of Coordination Policies in a Collaborative Framework. To be submitted to *the 22nd International Conference on Software Engineering (ICSE'2000)*, Limerick, Ireland, June 2000.

21. Du Li, James D. Herbsleb, Lalita J. Jagadeesan, and Richard R. Muntz. Flexible Awareness Control in Dynamically-Grouped Workspaces. In Preparation.

22. L. Logrippo, M. Faci, and M. Haj-Hussein, An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23, 5, Feb. 1992

23. Steven McCanne, Van Jacobson, *vic*: A Flexible Framework for Packet Video. *ACM Multimedia 1995*

24. Naftaly Minsky and Victoria Ungureanu, Regulated Coordination in Open Distributed Systems, *2nd Intl. Conference on Coordination Languages and Models*, Berlin, Germany, Sept. 1997 Proceedings

25. L. M. Perira and R. Nasr. Delta-Prolog: a distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Sustems*, Tokyo, 1984

26. J. Rekers and I. Sprinkhuizen, A LOTOS Specification of a CSCW tool. *Proc. of Design of Computer Supported Cooperative Work and Groupware Systems*, Schearding, Austria, 1993

27. Dirk Riehle and Thomas Gross, Role Model Based Framework Design and Integration, in *Proceedings of ACM OOPSLA'98*

28. V. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proc. of the 17th ACM Symposium on Principles of Programming Languages*, New York, 1999

29. V. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages*, New York, 1991

30. Ehud Shapiro. Embedding Linda and other joys of concurrent logic programming. Tech. Report, CS-89-07, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel

31. Ehud Shapiro, The family of concurrent logic programming languages, *ACM Computing Surveys*, 21(3), Sept. 1989

32. Abraham Silberschatz and Peter Galvin. Operating System Concepts. Fifth Edition, p.563-566. Addison Wesley Longman, Inc., 1998

33. Michael VanHilst and David Notkin, Using Role Components to Implement Collaboration-Based Designs, in *Proceedings of ACM OOPSLA'96*

34. R.J. Wieringa, W. De Jong, and P. Sprint, Roles and dynamic subclasses: a modal logic approach. In M. Tokoro and R. Pareschi, editors, *the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, Springer, 1994