Appears in *Freitag, Jones, Lengauer and Schek (Eds): "Object Orientation with Parallelism and Persistence", Kluwer Academic Publishers (1996)*

Coordination as Negotiated Transactions

Jean-Marc Andreoli
Rank Xerox Research Center, Grenoble, France

Abstract

Usually, task coordination systems distinguish several aspects [11] of task execution: the *functional* aspect is concerned by what tasks are to be performed by the system. The *behavioral* aspect concerns when particular tasks are to be executed. The *technological* aspect concerns how tasks are actually performed. Coordination systems generally do not provide any tools to support the technological aspect; instead, they are open systems which can (or have the ambition to) interoperate with various other systems and platforms which implement the actual actions. Abstracting the "how" part of activities from the "what" and "when" make coordination particularly amenable to the declarative kind of specifications offered by rule-based frameworks.

Working out which tasks have to be done (the "what" part) often requires a negotiation between the coordinator and the external systems which are supposed to then perform the tasks; as for sequentializing the tasks in an appropriate order (the "when" part), it requires some form of synchronization. We claim that these two aspects, negotiation and synchronization, can be elegantly supported in a rule-based framework by two mechanisms which, in the past, have been successfully integrated with rules: respectively, constraint propagation (and solving) and transactions. The situation can be summarized in the following table:

Aspect	Mechanism involved	In a Rule-based framework
what	negotiation	constraint propagation and solving
when	synchronization	transaction

We propose a rule based coordination framework where negotiation and synchronization aspects are specified in terms of, respectively, constraint solving, and transactions.

1 Motivations

Coordination is an issue which occurs in many different situations. At the very low level, the various partners in a communication protocol must coordinate their activities to achieve a given communication pattern, for example, reliable communication on top of an unreliable network. At the other end of the spectrum, at a very high level, participants in a workgroup must coordinate their activities to achieve the expected output of the group. In spite of several similarities, there are also important differences between these two extreme cases. In the former case, the coordination concerns activities which are homogeneous at the level at which they are coordinated: they are "black boxes" capable of sending and receiving messages, and they must coordinate these basic actions in order to achieve the requested communication behavior. But, in the latter case, the coordination system must take into account on the one hand a wide diversity of existing computing devices and systems which the members of the group already rely on and use, and, on the other hand, a diversity in the behaviors of the participants. The participants in that case are fully autonomous software entities which support, through heterogeneous systems, human decision making processes.

1.1 A Scenario

Acme Corp produces a device which includes an electrical component and a mechanical component. Specifications of each of the components have been provided respectively to the electrical engineer and to the mechanical engineer (or team of engineers). Each of them has produced a design. The designs must be checked and approved by a technical manager and a financial manager, on the basis of reports which have been produced by the engineers. This global, coordinated, behavior may in fact translate into more or less complex negotiations between the various participants.

For example, reading the report from the electrical engineer, the technical manager realizes that there is a mistake in the specification. He immediately informs the engineer, who then produces a new design and report, if the required modification is possible (within the deadline). Otherwise, some discussion may occur to explore alternate solutions.

A more complex case occurs when, reading both reports from the electrical engineer and the mechanical engineer, the technical manager realizes that, although the specifications and the designs of the two components are correct, there will be problems in assembling them. For example, a heat sensitive element on the mechanical component may be too close to a heat generating element in the electrical component. The technical manager then requires modifications on both designs, but they must be performed simultaneously, or not at all ("atomically"). In case one of the modifications is impossible, the other modification request must be cancelled, partially or totally, and alternate solutions must be explored. This leads to a form of tripartite (more generally multipartite) negotiation between fully autonomous partners.

Clearly, when several negotiations, possibly leading to concrete modifications of the world, are performed concurrently, there is a risk that inconsistencies may appear. For example, the financial controller may require a modification of design while a negotiation occurs on the technical side. If this modification is finally executed, the ins and outs of the ongoing technical negotiation may be completely disrupted. That does not mean that the negotiation should be restarted from scratch, but it should be aware of the modifications, and take appropriate steps. A certain degree of isolation should therefore be maintained between the different negotiations, so that modifications of the designs do not overlap in an uncontrolled way.

From this scenario, it appears that some form of transactional behavior is interleaved with the negotiation processes, although none of the traditional characteristic properties of transactions, Atomicity, Consistency, Isolation and Duration (also known as ACID) need to be enforced strictly.

1.2 Our Approach

Our goal is to design a coordination system capable of supporting the kind of negotiated transactions illustrated in the previous example. Transactions are long-lived and occur between autonomous software agents, most naturally represented by active objects, inherently distributed and concurrent. In our approach, we keep the basic two-phase commit mechanism of traditional database transactions, as a way of maintaining some form of consistency between distributed objects, but we incorporate it within a larger negotiation mechanism (as, e.g., in [16]). Thus, the purely transactional operations are invoked only at the end of the (successful) negotiations, to enact the agreements achieved by them. Although such transactions may require to be short-lived (because of well-known limitations of the two-phase commit otherwise), the overall negotiations within which they occur may be long-lived, and, in fact, may even be permanent processes.

In our approach, a coordination process involves, on the one hand, a set of "participants", which may be any autonomous software agents (or human agents through graphical user interfaces), and on the other hand, one or several "coordinator(s)", which attempt to ensure a global coordinated behavior of the participants. The participants may be very diverse in their interface to the outside world, and hence cannot be directly connected to the coordinators. Consequently, we assume that each participant is wrapped inside a specific piece of software, precisely called a wrapper, capable on one side to talk to the participant through its appropriate specific channels, and offering on the other side a homogeneous interface which the coordinators can connect to.

Conceptually, at the most general level, a participant is viewed as an autonomous agent managing a set of resources. The resource set need not be explicitly implemented in the participant. This is an essential difference with database systems, which are generally designed to handle only explicit sets of resources encoded as records. In the sample scenario of Section 1.1, the engineer is a participant who handles resources encoding, say, elements of (computer assisted) designs. This set of resources consists not only of the ongoing or past designs already realized by the engineer, but also the designs he can potentially produce in the future. The description of past designs may be explicit, while, obviously, potential future designs can only be represented implicitly. The activity of the engineer precisely consists of transformations from implicit to explicit descriptions. It occurs on the participant's own initiative, either as an answer to a request from the outside world, or in anticipation to such requests, or for any other reason.

The dialogue between the coordinators and the participants, for which we propose a protocol, assumes a client-server architecture in which the coordinators, on the client side, attempt, through the protocol, to manipulate the resources held by the participants, on the server side, in order to achieve a coordination goal expressed in terms of an expected global behavior. Basically, the protocol distinguishes three phases in the operations a coordinator may execute on the participants: the Preparation phase determines which actions the coordinator may take to achieve its goal; the Performance phase schedules, executes and monitors the actions determined in the first phase; finally, the Notification phase generates events corresponding to specified stages in the achievement of the coordinator's goal. Each thread of activity of a coordinator is composed of these three phases in sequence, but the different threads can be interleaved so that, overall, the three phases are deeply interwoven.

2 Rule-based Coordination

This section describes the principles, and the protocol, of a rule-based coordination system, called the CLF [2] (Coordination Language Facility). It provides the basis for a negotiation framework, described in the next section.

2.1 Manipulating Resources with Rules

Production rules offer a very convenient formalism to express the kind of manipulations a coordinator may wish to perform on the resources of a number of participants. Basically, given that each participant is viewed as a bag, or multiset, of resources, production rules can express multiset rewriting transformations. The left-hand side of a rule consists of a set of tokens which represent the resources which are to be removed from the participants while the right-hand side specifies a set of tokens representing resources to be inserted. Both insertions and removals may concern different participants, and rules are read as transactions across the participants. In particular, if the left-hand side of a rule specifies the removal of a resource a from a participant A, together with the removal of a resource b from a participant B, these two removals must occur atomically, meaning that the application of the rule cannot end up in a stable state where, for example, a is removed but not b. The situation is different on the right-hand side, since we assume that inserting a resource in a participant can never fail. This is in fact the only difference between the two operations of insertion and removal, since we attach no particular semantics to these operations, which are implemented on the participants side. In our example of section 1.1, we could have a rule

```
    modif-req, new-e-design, new-m-design -> report-changes
```

The token `modif-req` represents the specification of a modification request of both the electrical and the mechanical designs, and belongs to the technical manager. The tokens `new-e-design` and `new-m-design` represent actual modifications of the designs, proposed by and belonging to, respectively, the electrical and the mechanical engineer. Finally the token `report-changes` represent a report about the performed changes, which belongs, for example, to the production unit which has to implement the change. Note that the rule above is given here in simplified syntax, and is not the rule as it is written by the programmer of the coordinator, since it does not show the dependencies between the different tokens (e.g. they all concern the same modification request).

It would not be realistic to assume that the tokens which appear on both sides of rules represent explicit resources. Indeed, even the participants may not have an explicit representation of their own resources, some of them being only potentially present. In fact, tokens represent resource properties instead of resources and a token p on the left-hand (resp. right-hand) side of a rule means “remove (resp. insert) a resource satisfying property p”.

2.2 A Basic Protocol

We explain here how the three phases of the CLF protocol (Preparation, Performance, Notification) are invoked by the rules of a coordinator. Fig. 1 gives an overview of this mechanism.

2.2.1 The Preparation Phase

Given that rule tokens represent resource properties and not directly resources, the Preparation phase in the application of a rule must select a resource for each token in its left-hand side. The resource selected for a token must satisfy the property expressed by that token. The coordinator cannot do such a selection on its own and must delegate that task to the appropriate participants. This is achieved by invoking the *Inquire* operation of the CLF protocol on the participants. We do not consider here how the different Inquire operations are ordered nor related. This will be explained in section 3.

For purpose of simplicity, we assume here that, for each token, the coordinator has the knowledge of which participants are capable of processing the inquiry for that token, and furthermore that there is only one such participant per token. For example, the coordinator could be initialized with a table mapping each token occurring in the set of rules (assuming it is static) to the appropriate participant. This means that we ignore the problems of information search (each coordinator knows where to fetch the information it requires) and of information mobility (this knowledge is static).

At a given point of time, there may be several resources in a participant which satisfy the property expressed by a token, and this set may evolve with time, due to changes in the internal state of the participant. Therefore, the output of an Inquire operation may be spread on a long, possibly infinite, period of time. The Inquire operation is hence of the “deferred synchronous” kind (each request generates a potentially evergrowing stream of answers). Furthermore, given that resources may either be explicitly or implicitly implemented in participants, some resources satisfying a given token may be explicitly present while others are only potentially present. Such potential candidate resources for a token cannot be converted into explicit resources during the inquiry, since this transformation may be irreversible and therefore does not belong to the Preparation phase. To overcome this problem, we assume that

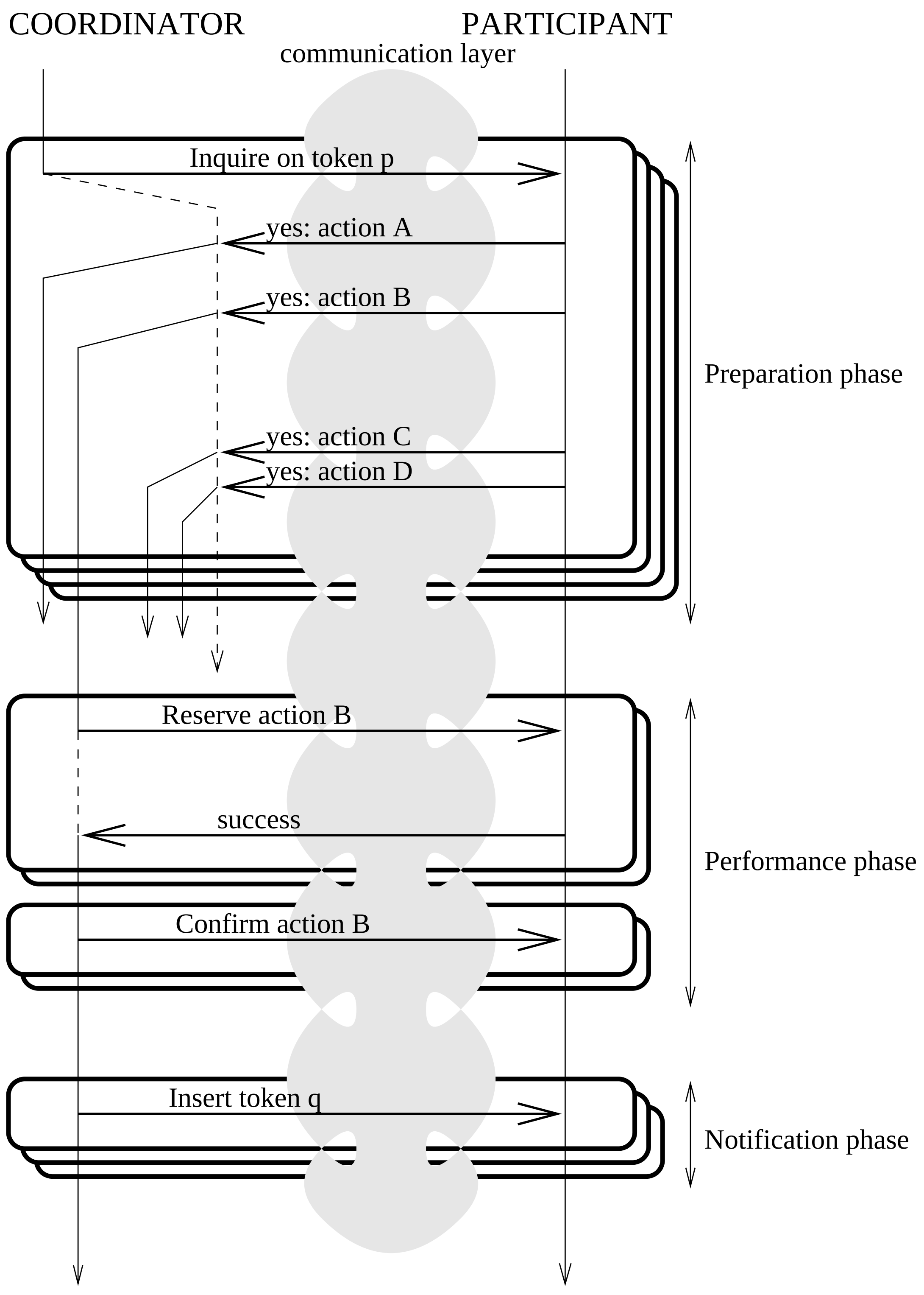

Figure 1: The CLF protocol in a coordinating rule

the Inquire operation does not return explicit resource identifiers, but rather, returns identifiers of actions, each of which is capable, if requested in the Performance phase, of making explicit and then removing a resource satisfying the property expressed by the token passed as argument of the inquiry. The Preparation phase for each rule can therefore be summarized by:

> **Preparation** phase = assignment of an action to each token in the left-hand side of the rule.

In the sample scenario of Section 1.1, an inquiry may consist of a request for the modification of a design, expressed with more or less details, e.g. "replace element `#123` in the design with an element of model `4`". This request may require a rewiring of the connections to the element, and, possibly, a reorganization of the surrounding elements, which may be achieved in several ways. The engineer will explore these different alternatives, and propose them as answers to the inquiry without implementing any of them explicitly (this is done only in the Performance phase).

2.2.2 The Performance Phase

Once an action has been assigned to each token on the left-hand side of a rule during the Preparation phase, this rule is applicable, and if the coordinator selects it for application, it enters the Performance phase. Basically, this phase consists of remotely executing the actions on the corresponding participants in a single transaction.

> **Performance** phase = execution of the actions assigned to the left-hand side of the rule within a single (short-lived) transaction.

The CLF protocol provides a rudimentary mechanism to achieve the transactional behavior, based on a two-phase commit protocol. On the client side, the coordinator can invoke the three basic operations of this protocol: *Reserve, Confirm, Cancel.* Each operation takes as argument a transaction identifier and an action identifier provided as a result of an Inquire operation. No information is attached to the transaction identifiers, and hence need not be maintained in a separate process.

The Reserve operation enacts a contract between the coordinator and the participant which handles the action, in which the participant commits itself to successfully perform the action upon request by the coordinator. This means that the participant must anticipate the set of resources which the action is going to manipulate, and freeze their state, or at least some part of it, and prevent other manipulations to be performed on the frozen part. How this is achieved is the responsibility of the participant. Of course, the participant may refuse the contract. An obvious cause of refusal occurs when, during the Preparation phase, the Inquire operation returns an action which relies on the existence or the state of a resource which has disappeared or has been irreversibly modified when the reservation is requested. Hence, the Reserve operation returns either success or failure (another case is introduced below). It is a purely synchronous operation.

The Confirm and Cancel operations rely on the contract enacted in the Reserve operation. A reserved action can either be confirmed, in which case the action is actually executed, or cancelled, in which case the contract is ended and the participant is freed from its commitment. If the action is confirmed, the contract ensures that the action will be executed successfully. Cancellation always succeeds. In both cases, no result is returned by the operation, which is therefore purely asynchronous.

The Reserve operation may lead, if no precautions are taken, to classical situations of deadlock where two transactions wait for each other to free the same resource. We assume that it is the participant's responsibility to lift such deadlocks, involving competing transactions from different coordinators. To achieve that, we assume that the transaction identifiers are totally ordered by some arbitrary priority order. The result of a Reserve operation can then either be success, failure or the special value "busy". When a reservation conflicts with an already successful one, it should block only if the latter belongs to a transaction of lower priority and should otherwise return the special value "busy". The client is thus informed of the conflict and can then take appropriate actions (see section 2.3).

2.2.3 The Notification Phase

When the Performance phase of a rule has been successfully passed, i.e. the resources selected for each of the tokens on the left-hand side have been removed, then the rule enters the Notification phase, where new resources, attached to the tokens of the right-hand side, are inserted inside the participants.

> **Notification** phase = asynchronous insertion of resources satisfying the tokens of the right-hand side of the rule.

The *Insert* operation of the CLF protocol takes a token as argument and inserts any resource satisfying this token. In fact, an insertion may trigger any transformation in the state of the participant which implements it, but returns no result. From the coordinators point of view it always succeeds; if potential failure has to be accounted for at the level of a coordinator, the operation should occur in a transaction of the Performance phase.

Inserting a resource in a participant may of course wake up suspended inquiries in that participant, by changing its state. However, if a token occurs both on the left and on the right-hand side of a rule, a usual policy is to attach the same resource to both occurrences, so that it is not needed to actually remove the resource to insert it again immediately afterwards. Such tokens only ensure that a resource is present when the rule is applied. They are reserved in the usual way, but when the transaction is successful, the corresponding reservation is simply cancelled instead of being confirmed, and the insertion is not done.

2.3 The Inference Engine

The inference mechanism of the coordinator continuously attempts to apply the rules using the CLF protocol to access the participants. Basically, for each token in the left-hand side of a rule, an inquiry is invoked and the coordinator computes all possible combinations of actions returned by the inquiries, each combination consisting of one action per token in the left-hand side. The process generating combinations is very similar in its principle to the Rete algorithm for production systems [7].

When a combination is complete, it is executed in a transaction. Each of its actions is reserved. If all the reservations succeed, they are all confirmed. If one of the reservations fails, all the others are cancelled. If one of the reservations yields "busy", all the others are cancelled and the whole combination is tried again. In other words, in case of conflict, the lowest priority transactions are pre-empted. This method to lift deadlocks in a two-phase commit mechanism is rather crude and pessimistic, since it forces cancellation and re-reservation of any transactions which conflict, even when the conflict does not lead to a deadlock. Transaction monitors implement much more sophisticated scheduling techniques, allowing maximal concurrency among the transactions, on the basis of a detailed analysis of conflicting operations. However, we argue here that our minimalist approach simplifies the design of wrappers for legacy participants, which may not have been built to work in a transactional environment. For example, it would be absurd to wrap a participant which acts as a simple event generator for a graphical user interface inside a complex transaction monitor.

If we assume that the set of ground rules (as opposed to rule patterns introduced in section 3) is finite, the Preparation phase, which invokes the Inquire operations, can itself be divided into two distinct successive phase: first, an Inquire operation is invoked for each token occurring on the left-hand side of each rule (there are, by hypothesis, only a finite number of such tokens); then the answers are collected and combined, and passed to the Performance phase, in charge of the transactions, and the Notification phase. If the set of rules is infinite it becomes impossible to invoke all the Inquire operations at once and collect the answers afterwards. The two phases must be interleaved in some way. This is where *negotiation* plays a role.

3 Negotiation-based Coordination

In this section, we introduce a negotiation mechanism in the Preparation phase, so as to turn it into a long lived process (possibly ever-living) which, from time to time, launches short lived transactions in the Performance phase. The order in which the Inquiry operations of a rule are invoked in the Preparation phase, and the interactions between these opeartions, are explicited.

To achieve that, we assume that tokens, i.e. resource properties, have a very simple structure of the form $p(a_1, \ldots, a_n)$ where p is called the predicate of the token, taken from a finite set of predicate symbols, and $a_1, \ldots, a_n$ are values taken from any domain. Furthermore, we assume that the rule set is not given in extension, but is represented by a finite set of rule patterns, each rule pattern representing itself a set of actual rules. A rule pattern is like a rule, except that instead of using tokens, it uses token patterns of the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variable names local to the rule. A rule pattern stands for the set of its ground instances obtained by consistently replacing in its token patterns the variables by arbitrary values of the domain. Assuming a infinite domain, it is clear that a rule pattern represents infinitely many ground rules, and, a fortiori, so does a finite set of rule patterns. We therefore cannot use the simplistic strategy mentionned at the end of section 2.3 for the Preparation phase, which works only with finite sets of ground rules.

3.1 Incomplete Requests and Information Thresholds

The method to handle the kind of infinite sets of ground rules proposed here relies on the assumption that Inquire operations in the Preparation phase can deal not only with tokens, but also with token patterns, i.e. tokens which are not fully instantiated. In this way, infinite sets of Inquire operations corresponding to different ground instances of the same pattern can be invoked by a single operation with a non-ground instance of that pattern. In other words, the Preparation phase has to deal with incomplete requests.

However, inquiries, although non-destructive, have a cost, and it would not be realistic to invoke an Inquire operation with a token pattern which is not sufficiently informed. In the sample scenario of Section 1.1, that would mean requesting, say, an engineer to produce all the possible designs he could, without specification, which does not make much sense. Therefore, we assume that, although an Inquire operation can be invoked by a coordinator with incompletely instantiated token patterns, the degree of instantiation of the pattern must reach a certain *threshold* for the operation to be correctly processed by the participant. There are various ways to introduce this notion of threshold in the Preparation phase. We discuss two of them below:

- The static approach assumes that the coordinator is initialized with enough knowledge about the capabilities of the participants to decide when a threshold is reached.
- The dynamic approach does not assume this knowledge but relies on further extensions of the protocol in order for coordinators to dynamically acquire that knowledge.

3.2 Static Thresholds: Signatures

The solution presented in this section assumes that the knowledge of the thresholds can be statically attached to each predicates in the program of the coordinator. Therefore, the coordinator is capable to decide on its own whether an Inquire operation is sufficiently instantiated to be invoked correctly, using information provided to it at initialization time.

3.2.1 Mode Declarations

This initialization information appears as a signature, i.e. a table mapping each predicate into one or several mode declarations. Each mode declaration partitions the arguments of the predicate into input parameters, which must be (fully) instantiated in any inquiry using that mode, and output parameters, which are returned by any inquiry using that mode, and therefore must not be instantiated. Thus, the thresholds here depend only on which arguments of the pattern are (fully) instantiated. For example, a mode declaration

$$p(X, Y, Z) : X, Y \mapsto Z$$

specifies that, in that mode, an Inquire operation invoked with a token pattern of the form $p(X, Y, Z)$ must provide values for its first two arguments X and Y, and leave uninstantiated the last argument Z. Each action identifier, normally returned with each answer to such an inquiry, is coupled with a possible instantiation of the output parameters (in the example, only Z).

The rule patterns defining the coordinator can be checked, at compile time, against its signature. Checking a rule pattern against a signature consists of selecting, for each token pattern in the left-hand side of the rule pattern, a mode attached in the signature to the predicate of the token pattern. Such a mode assignment is considered "allowed" if and only if (i) each variable occurring at an input position of a token pattern also occurs at an output position elsewhere in the left-hand side of the rule pattern and (ii) no variable can occur twice at output positions in the left-hand side of the rule pattern. The first condition requires that each value which is consumed somewhere is also produced somewhere else, while the second condition precludes that a value be produced in two different places (whether it is consumed or not). A rule pattern is correct w.r.t. a signature if there exists one and only one allowed mode assignment. It is ambiguous (resp. ill-formed) if several (resp. no) allowed mode assignments can be found.

If a rule-pattern is correct w.r.t. the signature of the coordinator, then the interleaving of the different inquiries during the Preparation phase follows the mode assignment of each rule: in a rule, the Inquire operation for a token pattern is invoked only when the variables at input positions in the mode assigned to that token pattern are instantiated. The possibility of multiple reply to an Inquire operation leads to the exploration of a search tree (see Section 3.2.2 below).

3.2.2 The Search Loop

The goal of the search loop, in the Preparation phase, is to generate possible instantiations of rules. There may be zero, one or more possible instantiation, and they are all passed to the Performance phase. Given a combination ν of token patterns, a (partial) assignment of variables τ, initially empty, and a set of actions α, initially empty, the algorithm of Fig. 2 instantiates progressively all the variables in the token patterns. The definition of allowed mode assignments precludes the possibility of a variable being instantiated by two different Inquire operations, which would require identity tests between values the coordinator knows nothing of.

```
Procedure SEARCH($\nu, \tau, \alpha$)
$\nu$: combination of token patterns, $\tau$: assignment of variables, $\alpha$: set of actions
    If $\nu$ is non empty Then
        Let $p(\vec{x})$ be a pattern in $\nu$ whose input variables are instantiated in $\tau$;
        Let $\nu'$ be $\nu \ominus \{p(\vec{x})\}$;
        Let $s$ be the stream of replies returned by Inquire($\tau.p(\vec{x})$);
        Foreach reply in $s$ Spawn
            Let $\tau'$ be the output variables assignment attached to the reply;
            Let $a$ be the action attached to the reply;
            Call SEARCH($\nu', \tau \oplus \tau', \alpha \oplus \{a\}$);
    Else ($\nu$ is empty)
        Pass $\alpha$ to the Performance phase
```

Figure 2: The search algorithm

The sequentiality of the algorithm of Fig. 2 is not intrinsic: independent tokens can be searched in parallel. For example, if the signature is

$$p(X,Y) :\mapsto X,Y \qquad q(X,Y) : X \mapsto Y \qquad r(X,Y,Z) : X,Y \mapsto Z$$

and the left-hand side of a rule pattern consists of

$$p(X,Y), q(X,Z), q(Y,T), r(Z,T,U)$$

then, the Preparation phase for that rule pattern will consist of: (i) an Inquire operation for $p(X,Y)$ (no argument is instantiated, as required by the mode of p); then, (ii) for each answer x, y to the previous request, two simultaneous Inquire operations for, respectively, $q(x,Z)$ and $q(y,T)$ (the first argument is instantiated in each case, as required by the mode of q); finally, (iii) for each pair of answers z, t to the previous requests, an Inquire operation for $r(z,t,U)$ (the first two arguments are instantiated as required by the mode of r).

3.3 Dynamic Thresholds: Constraints

The use of signatures to specify thresholds enable the coordinator to statically determine, immediately after it has been initialized, the thresholds of instantiation which enable/disable inquiries. On the other hand, the solution presented in this section allows a dynamic construction of the thresholds, achieved by negotiations between the participants at runtime.

3.3.1 Negotiators

The underlying idea is that each inquiry invoked in the Preparation phase is invoked with a fully unspecified token pattern, but, instead of returning directly a stream of instantiations for the pattern, together with corresponding actions, it returns a stream of negotiators, which, in turn, are in charge of determining possible instantiations for the variables in the pattern together with corresponding actions. Thus, a negotiator for a token pattern $p(X,Y)$ is in charge of producing instantiations x, y for X and Y together with actions capable of extracting from the participant it lives in resources satisfying the token $p(x,y)$.

This level of indirection enables full flexibility in the determination of the instantiations: indeed, the variables in the left-hand side of a rule may occur in several token patterns, and their corresponding negotiators must agree on the instantiations of the shared variables. Such an agreement is reached via a negotiation (hence the term "negotiator"), which is in fact very similar in nature with constraint propagation and resolution, each negotiator broadcasting constraints on its variables and listening to the constraints sent by the other negotiators with which it shares a variable. The rule-based approach we have taken in the CLF is fully consistent with this approach to negotiation; it can be integrated to the CLF protocol in the same way as the notion of constraint has traditionally been integrated in rule-based frameworks. More precisely, our protocol will be inspired by one of the most popular form of constraint propagation, namely range propagation, which has been successfully implemented in many rule-based systems. Of course, we keep our ontology of not specifying how the range constraints are implemented, so that the coordinator may know nothing of the domains it manipulates.

We first introduce the following definitions:

Definition 1 *The scope of a negotiator is the set of pairs consisting of an action and a token, where the action is capable of extracting a resource satisfying the token.*

The scope of a negotiator, like the set of resources in a participant, need not be implemented in extension. We assume that an inquiry on a token pattern $p(X, Y)$ (fully un-instantiated) returns only negotiators whose scope contains only tokens of the form $p(x, y)$ where x and y are values of the domain. During the negotiation, the ranges of x and y will be reduced (until they are eventually reduced to singletons).

Definition 2 *A token range is of the form $p(D_1, \ldots, D_n)$ where p is a predicate and $D_1, \ldots, D_n$ are subsets of the domain. The token range of a negotiator is the smallest token range which includes all the tokens in its scope.*

We now enrich the CLF protocol with the following operation on negotiators:

> *Propagate*: applies to a pair of negotiators N_1, N_2 and returns a stream of new negotiators. Each negotiator N in the returned stream satisfies the following properties:
>
> **Specialization** : The token attached to N is the same as that of N_1. The scope of N is a subset of that of N_1;
>
> **Communication** : Let $p(X_1, \ldots, X_n)$ and $q(Y_1, \ldots, Y_m)$ be the tokens attached to N and N_2. Let $p(D_1, \ldots, D_n)$ and $q(E_1, \ldots, E_m)$ be the token ranges of N and N_2. Whenever the variables X_i and Y_j are identical, then $D_i \subseteq E_j$.

In other words, the Propagate operation forwards as much information as possible from N_2 into N_1 through the shared variables of their attached tokens. Notice that propagation is void in the following cases:

- if the tokens attached to N_2 and N_1 share no variable at all;
- if, whenever X_i and Y_j are identical for some i, j, it is already true that $B_i \subseteq E_j$, where $p(B_1, \ldots, B_n)$ is the token range of N_1.

In these cases, one may assume that the Propagate operation returns the special value "void" (to be distinguished from the empty stream, which characterizes inconsistent constraints).

Consider for example negotiators N_1, N_2 whose scopes consist of tokens of the form $p(x)$ where x denotes entries in a multi-database of bibliographical references. Let $p(R_1), p(R_2)$ be the token range of N_1, N_2, where R_1, R_2 denote the ranges of entries relative to documents published, respectively, between 1980 and 1990 and between 1984 and 1995. The Propagate operation applied to N_1, N_2 is expected to return a stream of negotiators whose scope cover all the entries concerning work published between 1984 and 1990. There are many ways to build such a stream. One obvious way is to return a single negotiator with the reduced scope 1984-1990. Another way may be to return two negotiators in charge, respectively, of the periods 1984 alone and 1985-1990. Such a split may be justified if, for example, the underlying multidatabase is itself partitioned between "before" and "after" 1985. If the knowledge of the publication year is considered a threshold authorizing the actual retrieval of the corresponding entries, then the first negotiator may itself be replaced by a stream of individual negotiators, one for each entry of a document published in 1984, while the second negotiator may still wait for more information, provided by other invocation of the Propagate operation, to proceed. The negotiators for individual entries, which are much more refined than the initial negotiator N_1, are then capable of triggering other propagations.

The interleaving of the different operations of the Preparation phase implements range constraint propagation. Basically, for each token pattern in the left-hand side of a rule pattern, an inquiry is invoked and the coordinator computes all possible combinations of negotiators returned by the inquiries, each combination consisting of one negotiator per token pattern in the left-hand side. When a combination is complete, it enters a negotiation loop (see Section 3.3.2 below), returning a stream of combinations of negotiators which are ready to agree. Each agreeing combination in the stream is finalized, by attempting a Reserve operation on each of its negotiator. If a Reserve operation is attempted on a negotiator whose scope is reduced to a singleton, the token attached to the single element of the scope is returned and the corresponding action is actually reserved; otherwise the operation fails. The rest of the protocol is as usual (Performance, then Notification phases).

3.3.2 The Negotiation Loop

The goal of the negotiation loop, in the Preparation phase, is to generate agreements between negotiators. There may be zero, one or more possible agreements, and they are all passed to the Performance phase. Given a combination ν_o of negotiators, one for each token pattern in the left-hand side of a rule pattern, the negotiation algorithm of Fig. 3 recursively attempts to get closer to an agreement by propagating information across negotiators, yielding more refined combinations of negotiators. An agreement is reached when all possible propagations across negotiators in a combination are void. Formally:

```
Procedure NEGOTIATE($\nu$: combination of negotiators)
    Foreach $N_1$ in $\nu$ Do
        Let $\nu'$ be $\nu \ominus \{N_1\}$
        Foreach $N_2$ in $\nu'$ Do
            Let $s$ be the stream of negotiators returned by Propagate($N_1, N_2$);
            If $s$ is not void Then
                Foreach $N$ in $s$ Spawn
                    Call NEGOTIATE($\nu' \oplus \{N\}$);
                Exit;
    Pass $\nu$ (ready to agree) to the Performance phase
```

Figure 3: The negotiation algorithm

Definition 3 *An agreement for ν_o is a combination ν such that:*

- *Each negotiator N in ν corresponds to one and only one negotiator N_o in ν_o (and hence to one token pattern in the left-hand side of the rule), and the scope of N is included in the scope of N_o;*
- *Any propagation between two distinct negotiators in ν is void.*

For example, consider a rule whose left-hand side consists of

$$p(X), q(X, Y), r(Y)$$

The Inquire operation for p may return two negotiators p_1 and p_2, while the Inquire operations for q and r return a single negotiator each, respectively, q_1 and r_1. Two combinations, p_1, q_1, r_1 and p_2, q_1, r_1, are therefore passed to the negotiation algorithm. The algorithm may first propagate p_1 (resp. p_2) via X into q_1, yielding in one case (p_1) a stream of two new negotiators q_2 and q_3 and in the other case (p_2) a stream of a single negotiator q_4. We therefore have now three combinations p_1, q_2, r_1, and p_1, q_3, r_1, and p_2, q_4, r_1. Propagation goes on in each combinations till all the Propagate operation are void. At that point, the combination is ready for the Performance phase.

Notice that it is important that each propagation may return streams of results instead of single results. A single result would force the scope of the obtained negotiator to cover all the possible tokens satisfying the propagated constraint so that the range of each of its argument is not specific enough to be of any use in further propagation. On the other hand, each negotiator in a stream of results may cover only a subset of the possible tokens satisfying the propagated constraint such that in this subset, new information can be derived on the individual arguments of the token. It is the responsibility of the negotiator to split its scope in such a way and return the stream of individual pieces. This decision is taken on the basis of information thresholds, as in the bibliographic database example above.

When a combination has reached the end of a negotiation loop, and enters the Performance phase, it may happen that the scope of some negotiators are not reduced to singleton sets. In that case, it means the negotiation has failed and the Reserve operation on such negotiators fails. When a reservation fails in such conditions, the participant which holds the culprit negotiator may learn something about the negotiation and revise its strategy for splitting its negotiators scopes (i.e. revise its information thresholds).

4 Related Work

The work presented in this paper relates to many different areas:

Objects, Active objects, Actors : Research on objects initially focused on how to design individual objects for re-usability. The assumption was that it would then be easy to build complex applications by assembling objects and having them interact, using standard mechanisms (mainly, synchronous method invocation for passive objects or asynchronous message passing for active objects). However, it quickly appeared that patterns of interaction between the objects themselves were often quite complex, and required specific programming mechanisms. This lead to focus on distributed object coordination [26, 10], with which this paper is primarily concerned, which is also the concern of script based programming [3] (AppleScript, Tcl, VisualBasic), meta-control of actors [1, 8], shared message services such as Linda [9] etc...

Multi-agent systems : Distributed AI has also focused on providing sophisticated models for interaction between intelligent agents for problem solving (see, e.g. the Contract Net protocol [19]), based in particular on speech acts [25]. From that point of view, the verbs of our protocol (Inquire, Propagate, Insert, Reserve...) can be seen as speech acts.

(Relaxed) Transaction Models and Workflow : The strict (ACID) model of database transactions has been extended and relaxed in many different ways, in particular for coordination of tasks in workflow [17, 23, 24] or transactions in federated databases [5]. Rules, from that point of view, have often been used in such extensions (ECA rules [4], logic rules *a la* Prolog [13]).

The main difference with our approach is that here, we do not try to relax or extend the strict ACID transaction models, but we integrate it within a larger framework, including, in particular the Preparation phase, which plans which actions are to be performed in the (transactional) Performance phase. The focus is therefore on what comes before the transactions themselves rather than on the model of their execution.

Constraints : Finally, one of the most important influence for our work comes from the research on Constraints [12, 22, 15]. The negotiation algorithm presented here is inspired by general constraint satisfaction algorithms (see [14] for an overview). Constraints have been integrated very naturally to deductive rule based systems, in particular Prolog (e.g. the CHIP system [6] among many others), but also, less successfully, to production rules [21].

The main difference with our approach is that most of this work views constraints as a way to improve efficiency of search algorithms, and not as a computation model (here, for interaction between autonomous agents). In that sense, we are closer to the use of constraints in CCP [18, 20]. One characteristic of our algorithm is that it integrates a choice mechanism (the Propagate operation may return a stream of answers) inside the constraint satisfaction algorithm, which improves its expressiveness but, of course, has a cost in terms of efficiency.

5 Conclusion

In this paper, we have shown how the notions of constraint and transaction, mixed in a traditional way to rules, can be used in a natural way to specify both negotiation and synchronization aspects in a rule-based coordination system, i.e. to deal with two of the main aspects of coordination, informally captured by the questions "what?" and "when?". We also propose a protocol between coordinator and participants which delegates to the participants most of the actual operations required during the execution of the coordinator (the "how?" part). This allows to abstract away the real nature of participants, which can be any active objects implementing the operations of the protocol, either directly or through wrappers.

Acknowledgement

I wish to thank the people in the Coordination Technologies team at the Rank Xerox Research Centre, Grenoble lab[1], as well as John Lamping from the Palo Alto Research Centre, for helpful discussions on the topic of this paper. Thanks also to the anonymous referee whose comments helped a lot in improving the paper.

References

[1] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. In R. Gerraoui, O. Nierstrasz, and M. Riveille, editors, *Object Based Distributed Processing*. Springer Verlag, Berlin, Germany, 1994.

[2] J-M. Andreoli, S. Freeman, and R. Pareschi. The coordination language facility: Coordination of distributed objects. *Theory and Practice of Object Systems*, 2(2):77–94, 1996.

[3] W. Cook and W. Harris. The open scripting architecture: Automating, integrating, and customizing applications, 1993. http://http://www.luma.com/william/papers/AppleScript/AppleScript_ToC.html.

[4] U. Dayal, M. Hsu, and R. Ladin. Organizing long running activities with triggers and transactions. In *Proc. of intl. Conf. on Management of Data*, Atlantic City, NJ, U.S.A., 1990.

[1]WWW page at `http://www.xrce.xerox.com/research/ct/home.html`

[5] A. Deacon, H-J. Schek, and G. Weikum. Semantics-based multilevel transaction management in federated systems. In *Proc. of ICDE'94*, Houston, Tx, U.S.A., 1994.

[6] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Proc. of FGCS'88*, 1988.

[7] C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[8] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. of ECOOP'93*, Kaiserslautern, Germany, 1993.

[9] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[10] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object oriented systems. In *Proc. of OOPSLA/ECOOP'90*, Ottawa, Ont., Canada, 1990.

[11] s. Jablonski. Transaction support for activity management. In *Proc. of International Workshop on High Performance Transaction Systems*, Asilomar, Ca., U.S.A., 1993.

[12] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proc. of the 14th ACM POPL Symposium*, Munich, Germany, 1987.

[13] E. Kühn, F. Puntigam, and A. Elmagarmid. Multi-database transaction and query processing in logic. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, Ca, U.S.A., 1993.

[14] V. Kumar. Algorithms for constraint-satisfaction problems. *AI Magazine*, Spring:32–44, 1992.

[15] T. Le Provost and M. Wallace. Generalized constraint propagation over the clp scheme. *Journal of Logic Programming*, 16(3&4):319–359, 1993.

[16] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. In *Proc. of CSCW'92*, 1992.

[17] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems*. Addison-Wesley, Reading, Ma, U.S.A., 1995.

[18] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Pittsburg, Pa, U.S.A., 1989.

[19] R.G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computing*, 29(12):1104–1113, 1980.

[20] G. Smolka. Oz — a programming language for multi-agent systems. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*. MIT Press, Cambridge, Ma, U.S.A., 1993.

[21] M. Tambe and P. Rosenbloom. Investigating production system representation for non-combinatorial match. *Artificial Intelligence*, 68:155–199, 1994.

[22] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Ma, U.S.A., 1989.

[23] H. Wächter and A. Reuter. The contracts model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, Ca, U.S.A., 1993.

[24] G. Weikum and H-J. Scheck. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, Ca, U.S.A., 1993.

[25] T. Winograd. A language/action perspective on the design of cooperative work. *Human Computer Interaction*, 3(1):3–30, 1988.

[26] D. Yellin and E. Strom. Interfaces, protocols and the semi-automatic construction of software adaptors. In *Proc. of OOPSLA'94*, Portland, Or, U.S.A., 1994.