The Coordination Language Facility: coordination of distributed objects*

Jean-Marc Andreoli, Steve Freeman, Remo Pareschi
Rank Xerox Research Centre, Grenoble lab

Abstract:

The development of communication infrastructures and the rapid growth of networking facilities in information technologies increase information mobility and the decentralisation of work processes in industry and services. This evolution leads to increasing demands on the coordination of information systems. However, information technologies available today are capable of supporting only interoperability of information systems from the point of view of communication infrastructures. This makes possible an easy exchange of information but provides no support for coordination. To fill this gap, we propose the Coordination Language Facility (CLF) as a coordination layer on top of distributed systems infrastructures such as CORBA-compliant Object Request Brokers. The CLF provides support for the coordination of heterogeneous, possibly distributed, active objects within larger units implementing (work) processes. On one hand, coordinator objects are declaratively implemented as rules. On the other hand, the objects participating in a coordination (participants) must instantiate a minimal interface which specifies the negotiation dialogue invoked, at run-time, by coordinators. The coordination activity is split between the implementation of the interface on the participants'side and the execution of the rules on the coordinators' side, thus offering a clear separation of concerns between local and global activities. The interface is specified using the CORBA standard for distributed objects, removing issues of heterogeneity and allowing each component to be implemented in the most appropriate language and environment.

Keywords:

coordination, distribution, CORBA, work processes, rules

1. Introduction

Object-oriented programming (OOP) provides powerful support for building applications from independent components. Certain important classes of applications require, however, extensions of standard OOP with constructs to explicitly coordinate the behavior of multiple objects. Particularly relevant are applications such as *workflow management* [28].

Work processes, central to workflow management applications, describes how resources may be passed between the components of a software system to complete a package of work. For example, a workflow management system might wait for a set of files to arrive and then launch a text editor to merge those files into a report that, in turn, is stored with a document manager. The workflow management system is not concerned with the implementation of the editor or the document manager but is concerned with how their public interfaces communicate and pass data.

There is a lot of interest in workflow software, rooted both in business and in technology. In fact, business executives have realized that organizations are best viewed and analyzed in terms of their processes, and not just in terms of simple business activities [14]. At the same time, the coming age of networked enterprises provides the natural infrastructure to automate work processes by coordinating multiple distributed components. Software constructs that give full flexibility to the development of workflow management systems are still needed, however, as are complementary applications that provide direct support for the notion of process. In this paper we introduce the Coordination Language Facility (CLF), a process-oriented extension of OOP intended to fill this gap.

The CLF deals with entities of two kinds: *coordinators* and *participants*. In its basic form, a coordinator defines a minimal process unit for a course of action to be negotiatied among, and enacted by, multiple

* To appear in the Journal of *Theory and Practice of Object Systems* (1996), special issue on Distributed Object Management.

participants. Each participant is an autonomous software component responsible for its own part of the plan. The coordinator requests performance of actions to participants, and ensures that there is no conflict between the ways of enacting the actions chosen by the different participants. Once everything is agreed upon, the overall plan can be executed. Reaching this stage may involve several rounds of negotiation. For instance, two different flight databases may be requested to produce, respectively, a flight that departs from an airport in Northern Italy and reaches an airport on the US East Coast, and a flight that departs from an airport on the US East Coast and reaches an airport on the US West Coast; they are also requested to agree that East Coast arrival airport and East Coast departure airport are the same. There is a clear relationship between agreement of this kind and the notion of consistency provided by transaction processing systems [16], and indeed it will be shown that the negotiation protocol between coordinator and participants incorporates a standard transactional two-phase commit protocol. There are however some relevant differences too, mainly the fact that participants can choose which answer to return to a given request, and that a coordinator may reject answers that are conflicting and inappropriate. Thus, rather than superimposing transactional consistency over passive objects, coordinators perform *long-lived inquiries* leading to final agreement among autonomous components.

To implement these capabilities, the CLF assumes two complementary programming paradigms. Participants correspond to *active* objects [31] and are distributed over an infrastructure such as CORBA [32]. Coordinators are declaratively implemented as *rules*, and exploit features from declarative programming [20] such as non-determinism, variables and constraints to select among answers returned to specific requests. Furthermore, they exploit the *compositionality* of declarative programs both to model complex coordination behavior and to achieve de-centralized and dynamically reconfigurable coordination. Thus, coordinators can be composed, either to chain given process units into larger units, or to merge different negotiation policies over the same participants; conversely, complex coordinators can be split into smaller coordinators that are distributed over different physical locations. Moreover, coordinators can be dynamically added and removed during execution.

The next section describes our coordination schema in more detail, and illustrates it with some simple examples. Section 3 describes in detail the CLF and its implementation. We then discuss how CLF coordinators are integrated with and communicate with external active objects and legacy systems, and we give more complex examples. Finally, we discuss the relation between the CLF and other scripting languages and coordination models, and draw some conclusions.

2. Rules for coordination

2.1. Introduction

A common distinction for classifying software systems is between *transformational* and *reactive* systems [24]. Briefly, the output of transformational systems is entirely predictable from their input—they correspond to traditional procedures—whereas reactive systems, such as operating systems, continuously interact with their environment. Many scripting languages, such as AppleScript [8] or Visual Basic [29], incorporate typical reactive constructs of the form "on <*event*> do <*action*>". Rules are also a natural programming construct to define reactive systems, and allow various limited, but nonetheless useful, forms of coordination, such as event synchronisation, as described in Section 2.2.

Our use of rules to coordinate objects is based on a third class of system, *proactive*, in which the system actively seeks to influence and modify its environment, rather than just responding to external stimuli. The best known examples of proactive systems are intelligent software agents, such as [37, 38, 40]. They are computer systems that autonomously execute a task for their owners—such as finding an airline connection for a given route. As we will show in the following sections, proactive rules can be combined with traditional object-oriented programming to provide a powerful mechanism for the implementation of *coordinators*, an important subset of such agents. First, however, we clarify the difference between reactive and proactive rules.

2.2. Reactive rules for the synchronisation of active objects

The most widespread application of rules has been in Artificial Intelligence, particularly expert systems, where new tokens of knowledge, or *facts*, are inferred by recursively applying rules to a set of established facts. Object-oriented systems are also a suitable domain for rule-based programming [35], particularly with *active* objects [31]. In this case, the tokens manipulated by the rules are object states and method invocations or *messages*. The left-hand side of a rule defines a combination of object state and messages that signifies that a complex event has taken place, and the right-hand side of a rule defines the new state or messages that arise

from the event. When a rule finds the tokens specified in its left-hand side, it considers the event to have occurred and generates the tokens specified in its right-hand side, which may trigger various actions on the object (state transitions, method invocation, etc.). Thus, each rule acts as an autonomous, long-lived thread of activity continuously looking for events to be synchronised.

This computational model is purely *reactive*. It is a natural design for object synchronisers, such as event managers, and can be thought of as a useful, but restricted, object coordination model. As a simple example, the client side of a remote procedure call could be specified by reactive rules as:

```
<input state> → <intermediate state> <request>
<intermediate state> <reply> → <output state>
```

A client controlled by these two rules starts from an input state, from which it emits a request and changes to an intermediate state where it waits for a reply. When the reply arrives, the RPC protocol is completed and the client changes to its output state.

An interesting feature of rules is that they can be entirely abstract, as in this example, making no assumptions about the clients, the servers, or the messages—all of which are represented by symbolic tokens. In practice, of course, these tokens must be bound to concrete software entities; the details of this step depend on the system being implemented. One approach is that the pool of tokens is accessed not only by the rules engine, but also by external applications, perhaps written in conventional programming languages. For example, a user interface system could map user input events into tokens in the pool and, conversely, map some tokens from the pool into display events, as in the VUE Dialog Manager [7]. Similarly, tokens could be mapped to queries to a database server, with the answers from the server mapped back into tokens (Figure 1).

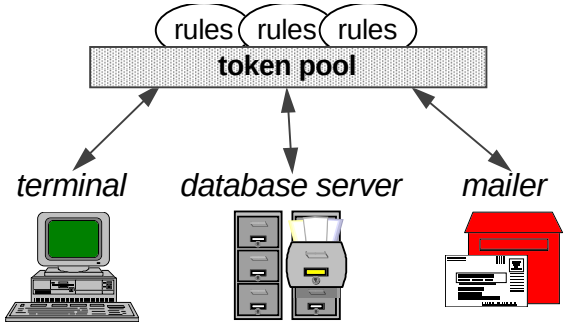

Figure 1. An example of rule-based synchronisation of heterogeneous active objects. Rules and active objects communicate via the tokens in the token pool.

2.3. Proactive rules for coordination

The reactive approach requires relatively little processing from the rule engine (synchroniser). It proceeds in two separate phases: at initialisation, the synchroniser requests the external systems to provide tokens. It then loops, continuously waiting for external systems to generate tokens which it attempts to match with unfulfilled rule pre-conditions, and triggering any rules for which all the pre-conditions have been met. A number of algorithms, such as Rete [18] and its variants, provide efficient implementations of such a scheme. A major limitation of this approach, however, is the limited communication between the synchroniser and the external systems; after the initialisation phase, the synchroniser is passive and must wait for external systems to generate the events it needs to make the rules proceed. In particular, there is no general mechanism for the synchroniser to inform external systems of the specific events it is interested in at a given point of the execution loop. There are two solutions to this problem, but both are unsatisfactory:

- the external systems may be set up to send all the possible event types that the synchroniser might need during a run. This can lead to a lot of unnecessary communication and processing. Furthermore, the synchroniser cannot assume that a token still represents the current state of an external system by the time that token is received, so the programmer must be aware of the potential pitfalls that may result.

- a programmer may write internal rules to manage the communication between the application rules and the external systems. These internal rules look for unfulfilled requirements amongst the application rules and, when triggered, make specific requests to the external systems. Unfortunately, this technique limits

the declarativity and readability of a set of rules and is one reason why rule-based approaches usually do not scale well.

We propose a new *proactive* approach in which each rule in a coordinator proceeds by continuously asking the external systems, or *participants*, to produce the tokens it needs to progress. This means, firstly, that participants are now aware of the needs of the coordinators and need send them only the events which they are currently interested in. Secondly, and more significantly, this changes the nature of the participants. Coordinator requests need no longer be just passive *filters* that specify which of a participant's events a coordinator is interested in, they may also be active *triggers* that request a participant to generate particular events. For a coordinator, these requests describe which potential participant states could contribute the triggering of a rule; a request is withdrawn if the rule state from which it originates becomes invalid. For a participant, these requests describe which of its potential states are of interest to the coordination system. Consequently, a participant must manage not only its internal state, but also a “wish list” of requests from coordinators to change that state.

The participants, however, still decide which requests to satisfy first, and how. In principle, for each token on the left-hand side of a rule, the participant that implements it sends the requesting coordinator a set of identifiers for all the possible participant actions that could produce the token. For example, the token `book_seat("Freeman", "Flight21")` returns a set of identifiers that represent each of the possible actions that could be taken to reserve a seat for Freeman on flight number 21. The seats on the flight might be divided between several travel companies, each of which has taken a block booking, in which case each identifier means that a seat is currently available with a given company. Note that, at this point, we have not reserved any system resources, we are only gathering information about what may be possible.

A coordinator acquires a set of action identifiers for each token in the left-hand side of a rule and, again in principle, produces a set of *instances* of the rule, one for each of the possible combinations of the actions. The coordinator then attempts to get the participants to commit to one of these rule instances; this commitment phase is non-deterministic, *any* solution to a rule is potentially valid. A participant may refuse to commit to an identifier it has already published, perhaps because the underlying resource has been modified or removed. If, however, a participant does commit to an identifier, then it is expected to be able to perform the associated action on request, although the details of the action are private to the participant.

A coordinator is a software *agency*, rather than a software agent; it mediates between its participants, trying to find agreements between them. Unlike reactive systems, the tokens in a rule now represent *potential* system states, and a coordinator attempts to make those systems states *actual* by matching offers and bids from the components it manages. It passes values between the components, leaving it to the components to decide whether to accept or reject those values; this is described in more detail in Section 2.4. For example, should we wish to fly between two destinations with only one connection, we could specify the rule:

```
journey_request(Name,From,To) @ flight_from(From, TransferAt, FlightNum1) @
flight_to(TransferAt, To, FlightNum2) @ book_seat(Name, FlightNum1) @
book_seat(Name, FlightNum2) <>- print_ticket(Name, FlightNum1, FlightNum2)
```

Briefly (see the following sections for a more detailed description), this rules involves three participants: one which holds the tokens `journey_request` and `print_ticket`, and models a user terminal; one which holds the tokens `flight_from` and `flight_to`, and models flight databases; and one which holds the tokens `book_seat`, and models a travel agency. For each user request held by an instance of the token `journey_request`, the rule finds all flights out of the first destination (variable `From`) with the associated flight numbers, finds all flights into the second destination (variable `To`) with the associated flight numbers, and finds for which of these flights we can book a seat. The coordinator attempts to match the `book_seat` tokens in parallel for each combination of the proposed values. If, eventually, one of these threads succeeds, that instance of the rule will commit, the other instances of the rule will terminate (because the instance of the request token `journey_request` is unique and consumed by the successful instance of the rule), and the token on the right-hand side, `print_ticket`, will print our ticket.

The key innovation of our approach is that, whereas the reactive interpretation of rules synchronises the *communication* between active objects, the proactive interpretation coordinates their *behaviours*. In other words, a reactive synchroniser waits for events to occur and generates new ones depending on the state of its rules engine. Participants may also wait for and generate events, so the system as a whole progresses when a required set of events coincide. In the proactive approach, however, the system as a whole progresses when the participants agree to commit to a set of values. The coordinators broker such agreements by passing the participants’ requirements and responses to each other and by ensuring their mutual consistency.

The reactive approach has proved successful with object systems because it works with any kind of active objects capable of waiting for and producing events. The objects need not even be aware that they are involved in a synchronisation and the synchronisation views them as the metaphorical *black boxes*. In the proactive approach, on the other hand, objects must be aware that they are being coordinated and must support a specific protocol to arrive at the agreements brokered by the coordinators. This protocol is captured by a standard interface, described in detail in the next section, which the objects must offer to join in a coordination. In a proactive coordination, objects can be thought of as *resource managers*, rather than simple state machines.

2.4. The coordination schema

There are two kinds of objects in the Coordination Language Facility (CLF): the *coordinators* and the *participants*. Participants, the external components of the system, are active objects; that is they have a state, one or more processing threads, and communication ports. Where necessary, we can integrate legacy software by encapsulating it in a wrapper object, as described in Section 3.4. Coordinators are also active objects with their behaviour controlled by rules.

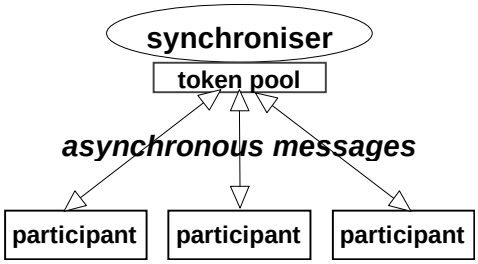

Figure 2. Reactive interpretation of rules. The synchroniser manages the pool of tokens internally.

In a reactive system, a synchroniser maintains a pool of tokens, changing the pool in response to asynchronous interactions with the participants (Figure 2). In a proactive system, however, the pool of tokens is distributed amongst and maintained by the participants themselves (Figure 3) so multiple coordinators, or other applications, can access the same participants independently. As the token pool is now distributed, a proactive system must use a negotiation protocol to maintain consistency, and any conflicts between coordinator requests are resolved within each participant.

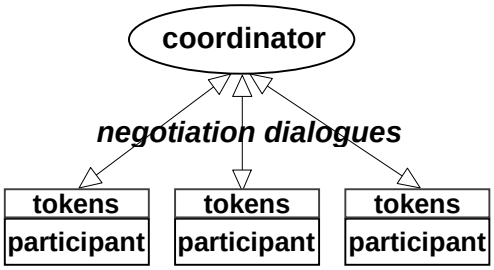

Figure 3. Proactive interpretation of rules. Each participant manages a share of the common token pool; the implementation of tokens is internal to the participants.

A rule is divided into left- and right-hand sides. The tokens on the left-hand side are intended to be removed from the participants, while the tokens on the right-hand side are intended to be inserted into the participants, *after* those of the left-hand side have been successfully removed.

Thus, the occurrence of a token *tok* on the left-hand side of a rule leads the coordinator to issue a request to some participant:

```
perform an action capable of removing tok
```

A request may be satisfiable in one or more ways, or not satisfiable at all, and satisfying the request may change the internal state of the participants concerned. This means that we need to ensure that such changes take place only when the rule is certain to apply, that is when all the tokens in the left-hand side of a rule are available. Thus, we must treat the left-hand side of a rule as a *transaction*. To support this, a token *tok* on the

left-hand side of a rule generates a CLF negotiation dialogue between the coordinator and any of its participants essentially as follows:

Inquiry: the coordinator *inquires* whether the participant holds (or can produce) the token *tok*. The participant returns a, possibly empty, set of actions that it could perform to remove the token *tok*.

Reservation: the coordinator asks the participant to *reserve* one of the actions returned during the Inquiry phase; only one coordinator at a time can reserve an action in a participant. If an action is successfully reserved, the participant commits to perform it on request.

Confirmation/Cancellation: the coordinator either *confirms* or *cancels* the action it has just reserved. If the reservation is confirmed, the action must be executed, leading to the deletion of the corresponding token; if cancelled, the action may become available again to other reservation inquiries.

The occurrence of a token *tok* on the right-hand side of a rule leads the coordinator to request some participant to insert the token *tok*. This is straightforwardly achieved by the following operation of the negotiation dialogue:

Insert: the coordinator asks the participant to insert the token *tok* in its share of the token pool.

An occurrence of a token on the right-hand side of a rule can be viewed as a compensating action to any action attached to an occurrence of the same token on the left-hand side of a rule. Notice that, whereas an action on the left may fail in the Reservation phase, it is assumed that a compensating action on the right eventually succeeds.

It is important to note that this protocol respects the autonomy of each participant. The CLF negotiation dialogue is the only interaction a coordinator has with a participant, which may have other threads of activity. For example, a token may represent the result of an interaction in a graphical user interface; the participant that implements this may have one thread to manage the negotiation dialogue and another to maintain the user interface. Furthermore, the coordinator sees only identifiers for possible participant actions; what a participant does when one of those actions is reserved or confirmed is internal to the participant. A reservation may fail when an action identifier returned during the Inquiry phase is no longer available during the Reservation phase (the action may have been removed by another coordinator or by some activity in another thread of the participant). The Insertion phase is also under the participant responsibility. For example, the participant holding the token `print_ticket` in our airline example might only queue a print request without changing any internal database and might not support inquiries from coordinators about tickets that have been printed. The right-hand side of a rule may also consist of a bottom symbol when there are no insertions, or a termination symbol that will cleanly stop the coordinator.

Clearly, the interaction between coordinators and participants is more complex than for reactive systems, which rely only on asynchronous message-passing. The Inquiry phase uses *deferred synchrony*: the coordinator needs to retrieve a list of all the possible actions capable of satisfying a given request but, rather than blocking until this list is complete (which may never happen if there are infinitely many such actions), the coordinator receives an iterator object from which it retrieves individual replies as required. The Reservation phase is *synchronous*. To avoid race conditions and to reduce coordinator locking, each participant responds immediately to reservation requests. Basically, the reply is either success or failure, but a third alternative is introduced in the next section to account for competing requests. Finally, the Confirmation/Cancellation phase and the Insertion phase are purely *asynchronous* as the coordinator does not expect any reply from the participants.

2.5. Process negotiations and transactions

Our proactive approach differs most from the reactive approach when the tokens on the left-hand side of a rule share argument variables and, thus, are inter-dependent. In this case, the Inquiry phase cannot be a separate initialisation that just connects rules to potential sources of events, since an inquiry on a token may need results (variable instantiations) provided by inquiries on dependent tokens, and therefore may have to be launched each time a solution to the dependent inquiries is obtained (i.e. any time during the coordination, since inquiries return potentially infinite streams of solutions). A variable shared by two tokens can be used to pass values from one participant to another: the source participant must then provide (at least partial) values together with the action identifiers it returns on inquiries, so that the receiving participants can decide which actions (and other instantiations) to return to its own inquiries, depending on the values passed. This leads to a form of long-lived negotiation between the participants to instantiate the variables of each rule, and Inquiry operations are synchronized by the continuous flow of variable instantiations during a coordination. This use

of rules is inspired by the request/subrequest inference strategies used, for example, in the Earley algorithm [17].

When an instance of a rule has collected variable instantiations and actions for all the tokens on its left-hand side during the Inquiry phase, it must commit all or none of them to avoid inconsistencies in the larger system. To achieve that, we use a simple two-phase commit. All the actions are reserved in the Reservation phase, but if one reservation fails, we cancel all those which have already succeeded for the current rule instance and give up on that instance; if they all succeed, they are all confirmed. Thus, we can describe a rule as a *process transaction*: it imposes a higher-level transactional behaviour across an arbitrary set of transactional participants. In the flight example above, the two `book_seat` tokens may refer to two different airline systems, but neither booking is useful without the other. The higher-level transaction in the coordinator prevents our ending up with a worthless single booking. In addition, there can be multiple threads within a coordinator searching and trying to match tokens for different rules and different instances of a rule.

We ensure consistency and avoid deadlock within a coordinator by performing all the reservations for a rule instance within a coordinator-wide critical section. This solution is rather crude, since it means that serialisability of the transactions within a coordinator is achieved by actually serialising them! We could of course adopt a more subtle scheme, but this is not the point we want to stress here and it only allows us to simplify the rest of our description, by identifying a coordinator with the single transaction it runs at any time.

Deadlocks across coordinators deserve more attention, since they cannot be solved by a single transaction scheduler; each participant is supposed to have its own independent scheduler, and we do not even assume interoperability between schedulers. The solution we propose consists in imposing an (arbitrary) order across all coordinators and requiring participants to arbitrate between competing coordinators. Briefly, if a coordinator *a* attempts to reserve an action already successfully reserved by a coordinator *b*, the participant returns a *busy* value if $a < b$; this means that coordinator *a* is invited to roll back any other reservations in its current transaction and try again. If $a > b$, then coordinator *a* blocks until coordinator *b* has confirmed or cancelled its reservation. In the first case (*b* confirms), the reservation for *a* fails, as expected; in the second case (*b* cancels), a coordinator *c*, blocked by *b*, is selected according to some strategy (not detailed here); if $a = c$, then the reservation for *a* succeeds, if $a < c$, then the reservation for *a* returns the *busy* value, and if $a > c$, then the reservation for *a* continues to block. This ensures that, at any time, a coordinator can only be blocked by a lower coordinator and thus, no deadlock can occur. Similarly, livelocks are avoided by assuming fairness of the strategy which selects the pending reservation that replaces a cancelled sucessful one. On the other hand, this scheme may cause a transaction to be rolled back and retried, because one of its reservations returns the *busy* value, although it is not involved in a deadlock. This is the price to pay for the assumption that the transaction schedulers at the level of individual participants are independent.

To support process transactions, the CLF places two demands on its participants: that they respond to the negotiation dialogue and that they arbitrate between conflicting reservation requests. If the actions a participant exports may lead to a change in its underlying state, then the participant must implement some kind of internal transaction mechanism as it may be simultaneously engaged in multiple CLF dialogues with multiple coordinators. Some actions, such as a time query, will not change the state of the participant that implements it. In this case we can either annotate the CLF program so that the coordinator makes inquiries without reservations, or provide an empty transaction layer in the participant to return dummy responses.

We should emphasise here that, although it defines process transactions, the CLF is not a transaction processing system, particularly in its current implementation. First, we rely on the underlying systems to provide reliability; such issues are orthogonal to the CLF model and should be treated separately. Second, and more interesting, the CLF supports long-lived *inquiries* rather than long-lived transactions. A coordinator attempts to find agreement between participants, rather than imposing new state on the world, so a failure to reserve resources for an instance of a rule is not a system failure but part of the process of coordination. Furthermore, this means that the coordinator is locked only during the reservation stage of the two-phase commit, minimising (although not eliminating) the need to worry about locking the larger system.

We now describe the coordination language we have developed and its current implementation.

3. The Coordination Language Facility and its implementation

3.1. Introduction

The theoretical background for the CLF is the LO (Linear Objects) model of coordination [4, 5], which is an abstract rule-based multi-agent computation model. A previous instance of the LO model (ForumTalk [3]) relied on the reactive interpretation of rules and made use of the whole syntax of LO. The CLF, on the other hand, uses only a fragment of the syntax of LO, but compensates by adopting the richer, proactive reading of rules; the extension of the CLF to the whole syntax of LO is under investigation. The LO model is based on the paradigm of "computation as proof search" applied to Linear Logic [23], more specifically to a complete subset of proofs in that logic called "focusing proofs". Linear Logic provides a logical framework in which to model and study the evolution of resource conscious systems, in contrast to Classical Logic which focuses only on truth values. Proof search is essentially non-deterministic and thus suitable for modelling non-deterministic coordination as in the CLF. The restriction to focusing proofs eliminates irrelevant forms of non-determinism deriving from syntactic limitations of sequent systems, but still preserves the full expressive power of proofs, since focusing proofs are a complete subset of proofs.

One crucial property derived from our theoretical model is *compositionality*, as expressed for example by the phase semantics of Linear Logic [23,5]. This means that complex coordination patterns can be obtained by assembling simpler ones, and that the behaviour of the complex system can be computed from that of its components (see section 3.4). Thus, the behaviour of a CLF coordinator specified by the union of two sets of rules is the same as the behaviour of two CLF coordinators running concurrently, each of which specified by one of the component sets of rules; although we should make two further comments:

- given that the behaviour of CLF coordinators is non-deterministic, one cannot state properties directly in terms of behaviours but rather in terms of sets of possible behaviours (modulo non-determinism). To be precise, compositionality states that the sets of possible behaviours are the same for a single coordinator running the union of two sets of rules and for two coordinators running each separate set of rules.

- the behaviours which are compared by the compositionality statement must be understood in terms of overall token consumption and production, not in term of the specific method invocations of the CLF protocol on the participants. We make several assumptions here, which are consistent with the CLF view of participants as resource managers: Inquiry on a participant must be implemented as an idempotent operation, and the participants must arbitrate fairly between competing coordinators in the Reservation phase.

The compositionality property has a direct consequence for the distribution of our architecture: coordination can be distributed simply by starting several concurrent coordinators running related sets of rules and accessing shared participants. This coarse-grain distribution is the only one currently implemented, but it could be complemented with a finer-grain approach in which the execution of individual rules is itself distributed. In this case, rules could be implemented as mobile agents moving from participant to participant, progressing by performing local Inquiry operations at each stage. We plan to investigate this possibility in the future.

The current implementation of the CLF is based on CORBA objects. Each CLF participant must instantiate and publish a set of distributed objects that implement the operations of the CLF protocol; these are described below. A CLF coordinator then imports these objects and binds them to tokens in its rules, thus becoming a client of its CLF participants. This means that a participant can serve any number of coordinators, including none. A coordinator may also have a local token store or blackboard, i.e. a participant that holds internal values.

The requirements that the CLF makes of its participants are both minimal and extensive. They are minimal in that all a participant is required to do is provide a set of objects that implement the CLF CORBA interface. They can be extensive in that, particularly when interfacing to legacy systems, providing a mapping between the CLF protocol and the participant's behaviour may require some interpretation by the coordinators designers. The appropriate result to return when a coordinator requests the next value from a participant may depend on the identity of the coordinator making the request. For example, a participant that is a shared spreadsheet may have to respond with different sets of cells to requests from different coordinators. Whether it is better to implement this as a single, parameterised CLF participant or as multiple virtual CLF participants depends on the intended use of the system.

The rest of this section describes the CLF syntax and the objects in a CLF system.

3.2. The CLF syntax

A CLF program consists of implementations, signatures, interfaces and rules sections. The implementations section links resource bank names to external objects and is implementation-specific, so we will not go into more detail here. The signatures section partitions the parameters of the rule tokens into input and output lists; each token name must have at least one signature. During the Inquiry phase, token inquiries must provide values for the input parameters and yield values for the output parameters. Where a token has more than one signature, the coordinator treats each one as a separate token and generates multiple rule instances. The formal syntax for a signature section is:

SignatureSection	=	**`signatures:`** **newline** *Signatures*
Signatures	=	*Signature*
	|	*Signature* **newline** *Signatures*
Signature	=	*TokenName* **`(`** *ParameterList* **`) :`** *ParameterListOrNil* **`->`** *ParameterListOrNil*
ParameterListOrNil	=	*ParameterList*
	|	ε
ParameterList	=	*ParameterName*
	|	*ParameterName*`,` *ParameterList*

For example, the signatures section for the example in Section 2.3 is:

```
signatures:
   journey_request(Name,From,To) : -> Name, From, To
   flight_from(From, To, FlightID): From -> To, FlightID
   flight_to(From, To, FlightID): From, To -> FlightID
   book_seat(Client, FlightID): Client, FlightID ->
```

which says that an inquiry for the token `flight_from` returns pairs of values `(To,FlightID)` when given a value for `From`, whereas `flight_to` returns single values for `FlightID` when given a pair `(From, To)`, and `book_seat` requires a pair `(ClientID, FlightID)` and returns no values while `journey_request` returns values for `(Name, From, To)` without being given any value. The signature for the token `print_ticket` is omitted here, since, in this example, it is never invoked in an Inquiry operation.

The syntax of the interfaces section is:

InterfaceSection	=	**`interfaces:`** **newline** *Interfaces*
Interfaces	=	*Interface*
	|	*Interface* **newline** *Interfaces*
Interface	=	*TokenName* **`=`** *BankName*

For example, we might have an interfaces section:

```
interfaces:
   journey_request      = user_terminal
   flight_from          = all_flight_info
   flight_to            = all_flight_info
   book_seat            = travel_agent
```

This technique allows CLF programmers to use multiple signatures with a resource bank. In this case, `flight_from` and `flight_to` both refer to the same underlying resource bank `all_flight_info`, but to retrieve different information, whereas `book_seat` refers to `travel_agent`. There must be exactly one interface specified for every token name in a CLF program.

The rules section of a CLF program specifies the behaviour of the coordinator and consists of a set of simple production rules. A rule has left- and right-hand sides, separated by the symbol **`<>-`** (read *becomes*), each of which is a simple multiset of tokens separated by the symbol `@` (read *par*). A right-hand side may also be the symbol **`#b`** (read *bottom*), which means that there is nothing to insert, or the symbol **`#t`** (read *top*) which terminates gracefully the entire coordinator. The formal syntax for the rule section is:

```
RulesSection  =  rules: newline Rules
Rules         =  Rule
              |  Rule newline Rules
Rule          =  TokenList <>- TokenList
              |  TokenList <>- #b
              |  TokenList <>- #t
TokenList     =  Token
              |  Token @ TokenList
Token         =  TokenName ( ParameterList )
              |  TokenName
```

Figure 4 shows a possible visual syntax for a CLF rule. It attempts to integrate also the information coming from the interfaces and signatures sections, and could be greatly enhanced by using colors.

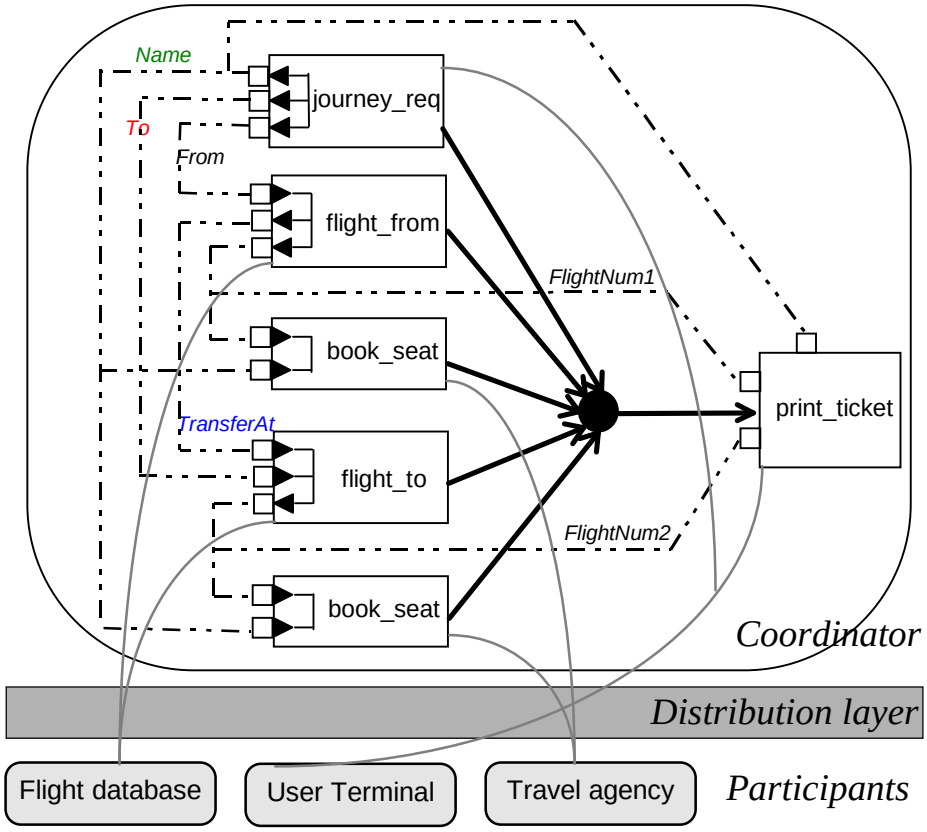

Figure 4. A visual syntax for the CLF rule in Section 2.3. Within the coordinator, tokens (with their signatures for left-hand side tokens) are represented by rectangles, variables by dotted lines, and the transaction point by a black disk. Connections with external participants (interfaces) are shown by grey lines.

3.3. CLF object types

Each token name in a CLF program represents a *ResourceBank* that holds all the available instances of that token name, the *Resources*. Each Resource holds values for its parameters, if any, and a unique identifier—so that Resources with the same values or no parameters can be distinguished. We chose the metaphor of the bank as a place to store resources because, as in the real world, banks can both store existing currency (such as a ResourceBank that manages a set of external files) and create new credit (such as a ResourceBank that generates public keys on demand). The CORBA IDL definition of a ResourceBank is thus:

```
interface ResourceBank {
    readonly attribute CLF::ResourceName   name;
    readonly attribute CLF::ParameterIndex arity;

    exception BadParameter {};
    CLF::Inquirer   inquire(in CLF::ParameterList params) raises (BadParameter);

    enum ReserveResult {ReserveGranted, ReserveFailed, ReserveBusy};
    ReserveResult   reserve(in CLF::Action act, in CLF::Coordinator coord);
    void            confirm(in CLF::Action act);
    void            cancel(in CLF::Action act);

    void            insert (in CLF::ParameterList params) raises (BadParameter);

    exception NoInstance {};
    CLF::ParameterList   values(in CLF::Action act) raises (NoInstance);
};
```

Note that a Resource is represented in a coordinator by a ResourceBank/Action pair, that is: a coordinator can manipulate a Resource by using an Action record to identify it to its ResourceBank. An Action record includes either the values of the Resource (arguments of the token instance) for simple data types, or an object reference for complex data types. We took this approach, rather than using the equivalent of defining an extra Resource interface, for several reasons: to reduce network traffic, as participants can extract simple parameter values from the Action string itself rather than always querying a Resource network object; to reduce distributed state, so we avoid many small network objects with associated problems such as distributed garbage collection; and, because some ResourceBanks may synthesise their Resources on demand and this approach allows lazy instanciation, or no instanciation at all, of Resource objects.

To find out which Resources are available, the coordinator submits an inquiry to the ResourceBank, including in the request a list of input parameter values that should be matched, as defined in the token signature. The request returns an *Inquirer*, an iterator or cursor object that maintains the state of the client's search; reifying the state of the search in an object which is then passed to the client makes it possible to ensure that multiple clients of a ResourceBank each receive an orderly series of candidate values, even when the contents of the underlying bank is changing. The IDL definition of an Inquirer is:

```
interface Inquirer {
    exception NoMoreValues {};
    CLF::Action   next() raises (NoMoreValues);
};
```

A thread can find out all the possible values for a Resource by repeatedly asking the Inquirer for the next value until a request blocks or raises a `NoMoreValues` exception. To simplify, we chose to return a single value at each invocation of the `next` operation, although returning a (non empty) set of values might be more efficient.

The IDL is open to extensions of the CLF protocol; in particular, for simplicity, we have omitted here the Kill operation of the protocol, which allows a coordinator to let a participant know that it is no more interested in a given Inquiry, so that resources possibly devoted to that Inquiry can be collected.

3.4. Assembling larger systems

A key feature of the CLF is that it is possible to connect mutiple coordinators and multiple participants—the virtual token pool that a coordinator manages is the union of the individual token pools of its participants. This is unlike reactive systems in which the token pool, even if it is distributed or partitioned, belongs to the synchroniser and exists independently of the participants. This gives two main benefits: system designers can break up large applications into collections of sub-applications, thus avoiding some problems of scale, and coordinations can be added and removed dynamically, allowing application behaviour to be changed as required.

Large-scale applications can be broken up into smaller coordinations that communicate either through common participants or by a coordinator itself acting as a participant for a higher-level coordinator (Figure 5): a coordinator can also act as a participant by exporting one or more of its internal resource banks. We rely on the CLF's transaction protocol to ensure consistency between coordinators. A CLF programmer can construct trees, or more complex structures, of coordination in which high-level coordinators manipulate only those

participants which are directly relevant to that level or which summarise lower-level coordinations. Each coordinator need only manage relatively few participants, reducing the complexity of each CLF program and the requirements for efficiency and reliability of individual coordinators; see Section 4.2 for an example.

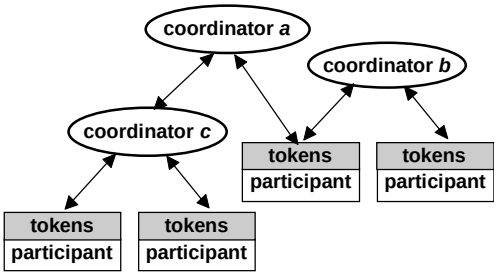

Figure 5. A tree of coordination. Coordinators *a* and *b* communicate via a shared participant, while coordinator *c* is a direct participant for coordinator *a*.

The CLF approach also encourages the dynamic assembly of applications, as coordinators can be added and removed without stopping the system as a whole. In Figure 6, coordinator *a* manages the coordination between participants *1* and *2*. Coordinator *b* is added later to link the behaviour of participants *1* and *2* with participant *3*. For example, a system administrator might need cleanly to shut down a system which is administered by coordinator *a* and so must acquire the resources implemented by participants *1* and *2*. The administrator starts coordinator *b* which waits for a trigger, implemented in participant *3*, and then reserves and removes the resources as they become available; the trigger might be a user interface event such as a button press. The fairness policy defined for participants means that we can assume that coordinator *b* eventually will be able to remove all the required resources, after which the administrator can stop the system.

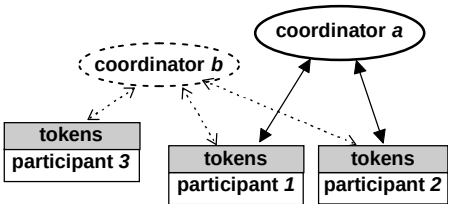

Figure 6. Dynamic coordination. Coordinator *b* adds new behaviour to the participants managed by coordinator *a*.

3.5. Talking to the world

A coordinator, of course, is of little use without participants to coordinate. We can divide participants into three main categories (Figure 7): generic participants, custom participants and legacy wrappers. *Generic participants* are useful across a range of applications. For example, we have written a text participant that implements resources of any arity where the values are all strings; we use this participant to support a coordinator's local resources (equivalent to local variables), to implement event triggers (ie. tokens with no parameters) and to prototype simple participants. Another generic participant is a dispatcher which acts as a surrogate for other participants determined at runtime, e.g. from an object identifier passed in an input argument of the tokens. Also useful are macro participants which combine a fixed set of participants in various ways, e.g. using join operations among tokens. Thus, in the airline reservation example of Section 2.3, a macro participant could be defined which combines (joins) several flights into a journey, so as to allow more than one transfer point. Other examples of generic participants include timers and serialisers (that return ordered sequences of unique identifiers).

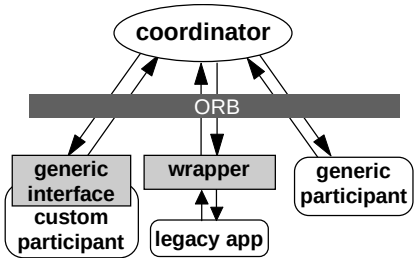

Figure 7. Three types of coordination participants: custom, wrapper, generic

Custom participants are resources written to support a particular application. For example, in our airline reservation case, the system authors might have to write the resource bank that holds the `print_ticket` token to provide a print service for a specialised ticket printer. Similarly, for the `journey_request` token, a programmer might implement a resource bank that asks a user to enter some data—perhaps by popping up a dialogue box on the user's screen or by sending an email message and waiting for a reply. To support such development, we have written a library of classes to implement the behaviour of a CLF resource at various levels of abstraction. Resource bank authors can override the default behaviours at whatever level is appropriate. We have, of course, used these classes to implement our generic resources.

The third category of resource represents perhaps the most interesting and important case—that of interfacing to legacy applications. We can coordinate the activity of such systems by providing a *wrapper*—an active object that mediates between the CLF and the target system. Although the borderline may not always be clear, we can distinguish wrappers and custom participants by the systems they manipulate: a wrapper interfaces to a domain-specific application (such as a flight reservation system), whereas a custom participant interfaces to a generic platform (such as an operating system). This distinction is important in practice because these are the levels that are relevant to developers and end-users, and because the two types of participants may require different tools and implementation strategies. In particular, the authors of a legacy application wrapper will require understanding of both the CLF protocol and the application domain.

4. Examples

To clarify its features, we now present two longer examples of applications of the CLF. The first shows how different types of participant may be assembled to construct an application, and the second shows how a large application may be broken down into a set of smaller coordinations.

4.1. Coordination across institutions

Consider the following example, again from the travel industry:

```
date("next Monday", Date) @ room("small conference", H, Date) @ near(H, H1) @
near(H, H2) @ room("double", H1, Date) @ room("single", H2, Date) @ pool(H1) @
book(H, H1, H2, Date) <>- notify_admin(H, H1, H2)
```

The users wish to arrange a meeting on the following Monday and require a small conference room, a single room and a double room, preferably in the same hotel but at least in hotels which are close to each other; the occupants of the double room also want a hotel with a swimming pool. Working through the tokens in the rule:

- the first token `date` is generic, it translates a text description into a numeric date. The coordinator either performs the inquiry without the reservation dialogue, or the participant provides dummy responses.
- the first `room` token in the rule returns all the hotels with small conference rooms available on the required date. This may involve searching across multiple hotel databases so the inquiry phase, at least, may be implemented by an intermediate service that passes the requests on to the hotels. The reserve and commit phases, however, may be passed through the intermediate service or may connect directly to the hotel system. Such a token is likely to be a custom participant, although it may form part of a domain framework that implements hotel-specific components such as wrappers for standard hotel reservation systems. The later `room` tokens are similar, except that the hotel is now specified by the coordinator, rather than returned by the participant, and so perform simple matches during the inquiry phase.

- the `near` token returns a list of hotels close to the hotel specified in the parameter `H`, starting with the original hotel itself; there are two instances of the token so that the coordinator can return bedrooms in different hotels if necessary. This could be a legacy wrapper for a travellers' database, or another interface to the intermediate service that implements the `room` token. Like the `date` token, `near` searches constant data and so need not fully implement the reservation part of the CLF protocol. The `pool` token is similar and returns a solution if the given hotel has a pool, otherwise it raises `NoMoreValues`.

- the `notify_admin` token sends a message to the users' administrative staff to tell them that the meeting and hotels have been booked. Again, this could be implemented as a wrapper around a standard workflow system or as a custom resource that generates an email message. In fact, given the specific nature of this participant, it might be written in a conventional scripting language, providing a quick implementation of a small piece of behaviour.

It is clear that any useful CLF implementation requires a large set of generic participants and tools to assist in the construction of custom participants—such as the hooks for writing participants in conventional scripting languages. We also believe that we will need to develop packages of coordination rules, custom participants and legacy wrappers when installing the CLF in a new application domain.

4.2. Managing reports within an organisation

A common activity in organisations is for people at each level of a reporting structure periodically to submit a description of recent activity to their managers. The managers collate and comment the descriptions they receive and pass the result up to the next level; the process is then repeated up the management hierarchy. Examples of such descriptions include timesheets, project and activity reports.

A naive approach to automating such a process is relatively easy to implement, but a little thought about the process reveals two main problems that arise in practice. First, a manager may not accept a section of the report and send it back to the staff member to be revised; this loop may be repeated an arbitrary number of times. Second, if staff are unavailable, not all the components of a report may have arrived when its deadline expires, so the best thing may be to submit the incomplete report as it stands. The following rules show how such an activity can be implemented with the CLF. The rules assume that each individual has his or her own CLF coordinator; this assumption can be relaxed by adding more parameters to the tokens to identify users and their relationships.

The first pair of rules apply to individuals at any level below the top. The first requests the individual to write a report and the second handles reports rejected by the individual's manager.

```
waitUntilNext @ reported(Report) <>- submit(Report)
returned(Report, Comment) @ revised(Report, Comment, NewReport) <>- submit(NewReport)
```

The `waitUntilNext` token is read-only and returns a value at the start of each reporting period; it acts as a trigger to start the search for an instance of the rule. The `reported` token waits for the user to write a report and enter it. To notify the user that a report is expected, the Inquire phase could have a side effect of creating a prompt—perhaps by popping up a window or by adding to a task list. If this approach to specifying notification is not suitable, we can add a few simple rules to express it explicitly. The `submit` token takes the report returned by the `reported` token and sends it to the user's manager. An instance of the `returned` token means that the report has been returned by the manager with a comment and the participant has been notified. The report and comment are passed to the `revised` token which waits until the user writes a new version of the report; this new version is then resubmitted to the manager.

The following rules apply to each manager in the enterprise. The first accepts a submitted section and merges it with the previous version of the report; this means that the manager always has access to the current version of the report. The second rule allows a manager to submit the report, with comments, to the next level up the reporting structure. Any version of the report can be approved, so the manager is not forced to wait for missing sections. The final rule allows a manager to reject a subsection of the report and return it, with comments, to the author; the subsection is also removed from the current version of the report.

```
section(Section) @ current(Report) @ merge(Report, Section, NewReport)
  <>- current(NewReport)
current(Report) @ approve(Report, Comments) @ merge(Report, Comments, NewReport)
  <>- submit(NewReport)
current(Report) @ reject(Report, Section, Comments)
  @ remove(Report, Section, PrevReport)
  <>- return(Section, Comments) @ current(PrevReport)
```

The `section` token is linked to the `submit` tokens of the manager's staff members: when they insert a `submit` token with their section of the report, it appears as a new instance of the `section` token (this is done by the banks which handle these tokens). The `current` token maintains the current state of the report; we can rely on the transactionality of the CLF to ensure that there is only one version at a time. The underlying participant creates an empty template report at the start of each reporting period to bootstrap the whole process. The `merge` token takes the current version of the report and a new section and merges them to produce a new version of the report, which may then be passed to the `current` token.

The `approve` and `reject` tokens have similar behaviour, they both receive a report and wait for a response from the manager, possibly putting up a prompt during the Inquiry phase. In effect, these two tokens are competing for the report returned from the `current` token, only one rule can commit that value at a time, and they will generally be implemented by the same external process, which will instanciate one token or the other depending on the manager's input. These tokens also return any comments from the manager; a `reject` token also distinguishes which section of the report is unsatisfactory. The `submit` token, as before, sends the report to the next level up. The `remove` token removes the given section from the report and returns its previous version, while the `return` token sends a section, with comments, back to its author.

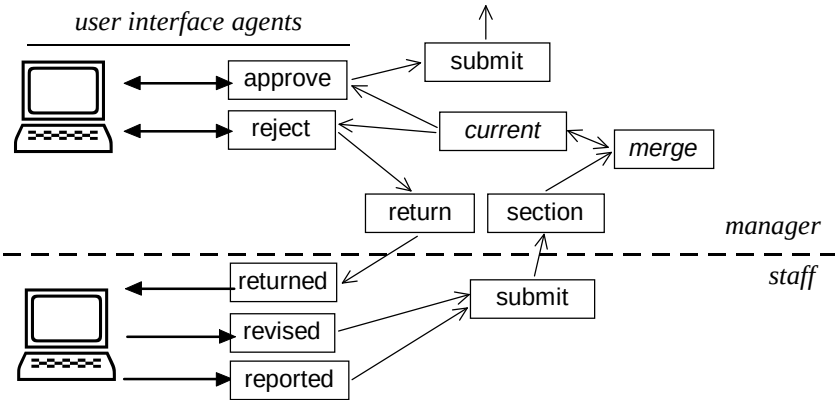

Figure 8. Relationships between tokens in the reporting example. The top half represents the manager's rules, the lower half the staff member's. The `waitUntilNext` and `remove` tokens have been left out for clarity.

Figure 8 shows how the resources in the example relate to each other and the outside world. The tokens on the left of the figure are implemented by participants that communicate directly with the user, perhaps by electronic mail or through a workflow engine. The (`submit`, `section`) and (`return`, `returned`) tokens represent communication between manager and staff and each pair is likely to be implemented by a common process. Finally, `current` and `merge` are private to a manager's environment; `merge` simply processes its parameters and `current` might be thought of as a local variable. Intermediate-level managers, with reporting relationships in both directions, will have both halves of the diagram.

This example shows how the CLF separates coordination from other activity in a distributed application. The rules can be applied to any such reporting process; the details of any individual report, such a the structure of a timesheet and the required user input, are encapsulated within the resources. This application has been fully implemented in the case of the timesheet.

5. Discussion

The CLF is a scripting language *for coordination*; it provides a rapid and flexible mechanism for binding together collections of active objects into a larger system. It has three key features: rules, process transactions, and an open implementation. First, proactive coordination involves the manipulation of a non-deterministic system—we do not control the internal execution of active participants—so rule-based scripting is particularly suitable. Programmers do not have to write large numbers of handlers to deal with all the possible combinations of events (or, alternatively, write a complex run-time system), but specify the high-level behaviour of the system. Second, proactive coordination involves establishing a series of agreements between independent active components, so it is essential that the components agree to the same thing. To ensure consistency, the CLF implements process transactions across its participants, reserving all the values in a rule instance before committing to them. At present this is implemented by centralised coordinators, but we could also use a distributed protocol such as ISIS [11]. Finally, we have invested in an open implementation, based on the CORBA distributed object standard, so we can coordinate any applications that implement the CLF CORBA interface. Furthermore, a coordinator does not interpret the values it receives from its participants but simply passes them on to other participants, so it does not have to be specialised to manage any particular group of components.

This section distinguishes how the CLF, as a scripting language for coordination, differs from more general scripting languages and from other coordination models.

5.1. Why the CLF is not a general scripting language.

Both the CLF and general scripting languages are intended to allow users to write high-level programs to bind together components, but the motivation and approach are very different. Scripting languages are general-purpose languages that can easily be extended to allow interaction with external components, whereas the CLF abstracts out, and is solely concerned with, the coordination of active objects in a distributed system. Thus, while implementers of scripting languages take pains to support extensibility—either by allowing new code to be linked in to extend the language, as with Tcl [34] or Python [41], or by providing general purpose message-passing, as with AppleScript [8] or Visual Basic [29]—the CLF is limited to a minimal set of messages for coordinating multiple independent components.

There are five features which distinguish the CLF from conventional scripting languages: no computation, non-determinancy, transactions, distribution and a minimal protocol. To take each of these features in turn:

CLF has no internal support for computation. All computation in a CLF system is performed in the resources. For example, to add two integers a programmer would supply an `add(In1, In2, Out)` resource, which would wait for two input values and assert their sum when they arrived. Clearly this approach is not efficient for such small computations so, as an optimisation, we allow side-effect free resources, written in a conventional scripting language, to be interpreted within the coordinator.

CLF is non-deterministic. Both the participants it manages and the coordinator itself operate in parallel; the implementation of a CLF coordinator is threaded to allow multiple simultaneous searches for matches. A CLF program says nothing about the order in which these threads run, and so any one of multiple rule instances that have acquired the necessary values may be successful—invalidating the other instances. A CLF program specifies the minimum necessary detail to coordinate a set of resources, so any set of values which triggers the firing of a rule is valid—a programmer who requires more determinism must specify more detail, or use a sequential language.

CLF rules are transactional. CLF's non-determinism requires that a successful instance of a rule must acquire all its resource values atomically, as there may be other rule instances simultaneously competing for the same values. The transactional behaviour, described in Section 2.5, is an essential feature of the CLF that would have to be explicitly programmed in a conventional scripting language.

CLF is distributed: With some exceptions, such as Obliq [13], most scripting languages are for local computation—although they may have extensions for distribution. The CLF, on the other hand, is designed for the coordination of distributed active objects. The state of a CLF coordination is partitioned across its parts rather than being held in a single logical component, such as the coordinator or a tuple space, and only passed between participants as needed.

CLF uses a minimal protocol. Unlike most scripting languages, the CLF implementation is not designed to be extensible. The only communication between coordinators and participants is via the CLF protocol, which allows arbitrary CLF resources to be combined. This separation of concerns supports better modularity and analysis of distributed applications, and simplifies the inclusion of existing software.

5.2. Other coordination models

As mentioned in Section 3, the LO model of coordination provides the theoretical foundation for the CLF. There are, however, other models of coordination which we now discuss in relation to the CLF.

5.2.1. Petri-Nets

The relation between Petri-nets [36] and Linear Logic has already been explored [12], and it is not surprising that the LO model, based on Linear Logic, has strong links with the Petri-net model. The relation is even closer when one considers the specific fragment of LO which is used in the CLF. In fact, the rules in the CLF could be viewed as Petri-nets "transitions" in which the CLF tokens play the role of the "places" in the net. The CLF makes use of first-order rules (allowing variables) and this has equivalents in various extensions of the basic Petri-net model, in particular "coloured" or "predicate" Petri-nets.

The Petri-net model is well established, especially for the specification and verification of low-level communication protocols, but is now the subject of renewed interest from the workflow community [26]. Workflow tools are a typical application of coordination in which the participants may be document management systems or heterogeneous end-user applications (text editors, spreadsheets, etc.). Most current workflow products rely on Petri-net-like representations of the flow of tasks, often with heavy restrictions on the form of the allowed transitions. To maintain the state of the workflow net, i.e. the distribution of the tokens, the workflow engine executes the enabled transitions by invoking external task handlers. This is clearly similar to the behaviour of a CLF coordinator with its participants, but the CLF has two main features which distinguish it from traditional workflow engines:

- A coordination can easily be distributed across multiple coordinators. The CLF does not make the assumption that the whole state of the coordination is centralised inside a single unit, like a workflow engine. A CLF participant may receive parallel task requests from multiple clients, both coordinators and other applications. For example, to improve reliability, it is possible to launch several CLF coordinators running exactly the same program; these will have the same overall behaviour as a single coordinator running that program. A program can also be broken into modules, each of which is run in a separate coordinator, to achieve a global coordination. Finally, where performance is important, a CLF coordinator may be replaced by a custom-written application.
- The tasks in the CLF are defined by the goal they intend to achieve, leaving considerable autonomy to the task handlers as to how they perform the task. More precisely, a participant may propose several solutions to an Inquiry request from which the coordinator must select the best, taking into account the solutions proposed by other participants for related tasks.

5.2.2. Linda

The term *coordination language* was introduced by Linda [21]. The Linda approach has some similarities with ours, but also many important differences. In particular, Linda does not propose a *language* for coordination. A coordination in Linda is specified by a set of software agents, or *coordinating actors*, one for each entity to be coordinated, or *participant*. A coordinating actor, which may be written in any language, communicates both with the participant it handles and with other coordinating actors via a shared associative communication facility, the *tuple space*. The coordinating actors in Linda can submit three kinds of requests to the tuple space:

- **in**: extracts *any* tuple that matches the given pattern; there is no provision for roll-back or to obtain other tuples that match the pattern.
- **rd**: tests for the existence of any tuple matching a given pattern. There is no way of reserving a tuple thus obtained and it may even have disappeared by the time the reply reaches the requester.
- **out, eval**: asynchronously adds a tuple to the tuple space, which may then wake pending **rd** or **in** requests.

In CLF terms, a Linda tuple-space realizes —partially— the functionality of a participant offering a rudimentary *passive* storage service for simple text tuples.

Linda does not distinguish between the two main problems which occur in coordination systems: dealing with heterogeneous participants that require specific adapters, and specifying a global coordinated behaviour between these participants. The underlying assumption is that the implementation language of each coordinating actor, whatever it may be, is expressive enough to coordinate its interactions both with the participant it handles and, via the tuple space, with the other coordinating actors.

In the CLF, however, we make a clear distinction between the two problems. The work of the coordinating actors is split between the "wrappers," which provide the necessary adapters to the participants, and the "coordinator", which captures only the global coordinated behaviour and is not attached to any individual participant. We believe this provides more flexibility and better modularity. Furthermore, it seems quite natural to separate the specification of the global coordinated behaviour from the individual participants and to concentrate it inside a single central unit of specification, namely, the CLF program. Notice that this centralization concerns only the specification level and does not preclude a distributed implementation of the coordinator.

The basic Linda model has been extended in various ways, of which the most relevant to the CLF is the addition of transactions [9]. A transactional tuple space is even closer to the CLF view of a participant, since it can roll back the effect of **in** requests—equivalent to cancelling reservations in the CLF—but it still lacks two properties of CLF participants. First, it has no activity of its own, whereas a CLF participant can create, delete and modify tuples dynamically, either autonomously or when processing the requests from coordinators. Second, it does not support the kind of search through the tuple set, with multiple replies, allowed by the Inquiry operation of the CLF. In practice, the main application for transactionality in Linda has been to support fault tolerance. Furthermore, transactional Linda also requires its coordinating actors to be aware of and explicitly to make use of transactionality, which is treated as an additional tool—to be integrated with those already available in the implementation language. In contrast, process transactions are at the core of the CLF coordination engine and so need not be explicitly programmed as they are implicit in the rules. Programmers of participants, on the other hand, must implement any required transactionality in a CLF wrapper but can use conventional transaction tools to do this. Another extension of the Linda model is to split the tuple space [22], which has been introduced mainly to improve efficiency. This aspect of the problem does not occur in the CLF where any number of participants of any kind is allowed.

There are also related commercial products, close in their design principles to the Linda model, such as ToolTalk [39] or MQ-Series [25], that include enhancements such as sophisticated pattern matching, scope mechanisms for the messages and message logging.

5.2.3. Actors

Actors [1] were initially conceived as completely autonomous entities capable of communication only through their interfaces. It soon became clear, however, that total autonomy was not always effective and that some form of external control, consistent with their encapsulation principle, was needed. The first such enhancement of the basic model was *reflection*, in which each actor is controlled by a meta-actor that has privileged access to specific internal components. A meta-actor may have access to an actor's message queue, thus allowing it to perform local re-ordering and filtering of the actor's messages. The meta-actor may also have access to some part of the internal state of the actor, and modify it before or after invoking the method of the actor upon reception of a message. This approach can be used to implement techniques that represent aspects, but not the full power, of coordination. For example, a pool of actors could share a common meta-actor that computes performance statistics for load balancing.

A later enhancement of the actor model, much closer to full coordination, was introduced with synchronizers and filters [2], and with software adaptors [43]. In this framework, actors behave in the usual way, but the message delivery mechanism is controlled by a set of rules that act on the space of messages. These rules may filter messages according to a certain pattern, or block a message until another message occurs satisfying conditions dependent on the first message, and so forth. This latter form of synchronization is close to the use of rules in CLF coordinators but, again, Actors synchronizers are concerned exclusively with communications and react to the messages occuring in the system they control, whereas CLF coordinators try to act on their participants' behaviours, not just their communications.

5.3. CLF and transaction systems

The transactional aspects of a CLF program are implicit, so their implementation can be changed to match the requirements of particular applications, which has two main benefits. First, an implementation can reuse any existing transaction infrastructure that participants support. The basic CLF protocol between coordinators and participants includes messages for a conventional two-phase commit but could, in principle, be adapted to use more refined transaction schemes, such as the Object Management Group (OMG) transaction service [33]. In this case, a CLF reservation is equivalent to performing the given action within a transaction block that is then committed or aborted depending on whether the action is, respectively, confirmed or cancelled. Of course, such top level transactions could trigger nested transactions inside other external objects, which then become indirect participants. Furthermore, if all the participants and coordinators share a common transaction service, then the CLF run-time can be modified to make direct use of that service. In particular, the requirements that Reservations occur only in critical sections in the coordinators and that participants explicitly arbitrate between conflicting coordinators (Section 2.5) become redundant, as these are the kind of problems that such services address.

Second, the implementation can provide as much or as little support for transactions as necessary. Not all CLF components require the support of a full transaction system; features such as fault-tolerance, security, and nested transaction may be too expensive or too slow for a particular application. The high-level definition of a CLF program means that individual participants need only support two-phase commit. Furthermore, for read-only participants, such as a query service for a static database, the reservation phase can be omitted or faked. Any additional requirements, such as a reliable message service, can be added to the implementation without changing the structure of the application or the rules that describe it.

There are other systems that propose high-level specification languages for coordination based on transactions. In the ConTract model [42], for example, the specification of a coordination is split into several modules. The *Control Flow* script describes the sequentialization of different *Steps* (specified separately) and the detailed aspects of the expected transactional behaviour of the steps is described in other modules. These modules specify which steps are to be performed transactionally, which steps can be compensated and how, and how to deal with resource conflicts. Thus, ConTract allows fine tuning of the transactional behaviour. CLF process transactions are less flexible but are integrated into a more general coordination system based, in particular, on the possibility of long lived Inquiries with multiple replies implicit in the rules.

Rule based coordination has been investigated for database management systems [10,15,27]. In VIP-MDBS [27], for example, *Local Database Systems* (similar to CLF participants) are queried through *Multi-Database Interfaces* (similar to the CLF negotiation protocol) by clients (similar to CLF coordinators). Client requests in VIP-MDBS are also specified in a rule-based language called VPL, which has important differences from CLF rules. VPL uses traditional Prolog rules while the CLF rules are closer to production rules, and VPL relies on a modified unification mechanism to deal with “communication variables”, which inhibit the declarative reading of the rules, whereas the CLF uses neither unification nor pattern matching. VPL has clearly been designed with transaction control in mind while process transactions are only one aspect of CLF rules.

Another rule based coordination system, the Event-Condition-Action (ECA) rules proposed in [15], allows the triggers of the rules to contain “conditions” that are actual operations on the databases in addition to simple “event” tests; this can be considered as a form of proactivity similar to the CLF. In the CLF, simple events tests are realised by Inquiry operations on tokens with no input parameters in the signature, while conditions consist of both Inquiry operations on tokens with input parameters and the transaction operations of the protocol. Furthermore, in ECA, all the events are attached to database transactions which can be coupled with the other transactions of the rules (especially the conditions) with a variety of user controlled mechanisms. This assumes a certain degree of interoperability between the transaction systems, which the CLF does not assume.

6. Conclusions

The Coordination Language Facility introduces a new, proactive interpretation of production rules and applies it to the coordination of heterogeneous active objects. It splits the activity of coordination into a generic part, specified with rules and implemented in coordinators, and a local part, specific to and implemented as part of each participant. This separation ensures better analysis of coordination systems by distinguishing between local computations in the participants and the global core of a coordination; in particular, the coordination rules are amenable to the kind of static analysis developed for the LO model [6]. It also enhances modularity and flexibility as it allows us to combine participants and sets of rules (i.e., coordination programs) as needed.

The CLF coordinates the behaviour of its participants, not just their communications. Being proactive, a CLF coordinator actively seeks change in the state of the larger system, rather than waiting for the participants to publish general-purpose events which may or may not be interesting. It achieves this by constantly querying its participants and attempting to find agreements between them, acting as an agency rather than an agent. To ensure valid agreements, the CLF imposes process transactions across its participants when firing a rule so that the state of the larger system is always consistent. This allows us to combine heterogeneous active objects, and ask them to change their state, even when the objects manage disparate legacy applications.

Our approach provides a true integration of rules and active objects. Each abstract token used in the rules has a concrete meaning at the object level, representing part of the state of an active object. We make minimal assumptions about the objects we coordinate, requiring only that they support multiple Inquirer objects and, if it is possible to change their state, that they support the CLF transaction protocol. While we provide some support for the implementation of CLF participants, this is only for convenience as the internals of any individual participant are private.

We have built a distributed implementation of the CLF protocol, based on the CORBA standard for distributed objects, that provides a flexible mechanism for binding the tokens in a coordinator's rules to services exported by CLF participants. The use of CORBA removes issues of component heterogeneity and allows us to implement each participant with the most appropriate language and infrastructure.

At present, we are investigating several aspects of the CLF:

- static analysis of CLF programs to optimise their execution and avoid unnecessary search and communication;
- replacing the signature mechanism, which allows uni-directional propagation of information in the Inquiry phase, by one based on constraint negotiation [30] which allows bi-directional exchanges;
- enhancing the implementation of coordinators and participants to support more of the LO model;
- writing tools and libraries to support the development and administration of coordination systems;
- adapting the CLF protocol itself for specific kinds of application domains.

Finally, we believe that the bulk of the work in the long-term development of the CLF will be to understand and build the interfaces to external applications and systems, and we intend to develop domain-specific packages of utilities and libraries to support CLF's use in end-user environments.

7. Acknowledgement

We wish to thanks the members of the Coordination Technologies group at the Rank Xerox Research Centre (`http://www.xerox.fr/grenoble/ct/home.html`) , and Hervé Gallaire, director of the Centre, for fruitful discussions on the topic of this paper.

8. References

[1] G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*. MIT Press, Cambridge, Ma, U.S.A., 1987.

[2] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveille, editors, *Object Based Distributed Processing*. Springer Verlag, 1994.

[3] J-M. Andreoli. Coordination in LO. In J-M. Andreoli, C. Hankin, and D. LeMetayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, Imperial College Press, London, U.K., to appear in 1996.

[4] J-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 257-280. MIT Press, Cambridge, Ma, U.S.A., 1993.

[5] J-M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. *In Proc. of OOPSLA'91*, Phoenix, Az, U.S.A., 1991.

[6] J-M. Andreoli, T. Castagnetti and R. Pareschi. Static Analysis of Linear Logic Programming. Submitted to New Generation Computing (follow-on to a paper presented at the ILPS'93 conference, Vancouver, Canada).

[7] P.A. Appino, J.B. Lewis, L. Koved, D.T. Ling, D.A. Rebenhorst and C.F. Codella. An Architecture for Virtual Worlds. *Presence*, 1(1):1-17, 1992.

[8] Apple Computer Inc. *AppleScript manual*, 1993.

[9] D. Bakken, and R. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. Technical report, University of Arizona, Tucson, U.S.A., 1993

[10] E. Bertino, G. Guerrini and D. Montesi. Towards Deductive Object Databases. *Tapos*, 1(1):19-39, 1995.

[11] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communication of the ACM*, 36(12):36-53, 1993.

[12] C. Brown. Relating Petri-Nets to Formulae of Linear Logic. Technical report, University of Edinburgh, Edinburgh, U.K., 1989.

[13] L. Cardelli. Obliq: a Language with Distributed Scope. Technical report 122, Systems Research Center, Digital Equipment Corporation, 1994.

[14] T. Davenport. *Process Innovation*. Harvard Business School Press, Boston, Ma, U.S.A., 1993.

[15] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-running Activities with Triggers and Transactions. In M. Stonebraker, editor, *Readings in Database Systems*, 2nd edition, 324-334, Morgan Kaufmann Publishers, San Francisco, 1995.

[16] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, San-Mateo, Ca, U.S.A., 1993.

[17] J. Earley. An Efficient Context Free Parsing Algorithm. *Communications of the ACM*, 13(2), 1970.

[18] C. Forgy. Rete: A Fast Algorithm for the Many Pattern - Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1), 1982.

[19] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *Proc. of ECOOP'93*, Kaiserslautern, Germany, 1993.

[20] H. Gallaire. Logic Programming - Past or Future?. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*. Elsevier, 1995.

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

[22] D. Gelernter. Multiple Tuple Spaces in Linda, In *Proc. of PARLE'89*, 1989.

[23] J-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1-102, 1987.

[24] D. Harel and A. Pnueli. On the Development of Reactive Systems. In K.R. Apt, editor, *Logic and Models of Concurrent Systems*. Springer Verlag, Berlin, 1985.

[25] IBM Corp. MQ Series, planning guide, 1994

[26] S. Joosten. Trigger Modelling for Workflow Analysis. Technical report, Technical University of Twente, 1994.

[27] E. Kühn, F. Puntigam, and A. Emagarmid. Multidatabase 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.

[28] J. McCarthy and W. Bluestein. *The Computing Strategy Report: Workflow's progress*. Forrester Research Inc, Cambridge, Ma, U.S.A., 1991.

[29] Microsoft Corporation. *Visual Basic Programmer's Guide*, 1993.

[30] U. Montanari and F. Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In Proc. of Coordination96, Cesena, Italy, 1996. To appear.

[31] O. Nierstrasz. Composing Active Objects. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 151-171. MIT Press, Cambridge, Ma, U.S.A., 1993.

[32] Object Managenent Group, *The Common Object Request Broker: Architecture and Specification*, 1991.

[33] Object Management Group, Object Transaction Service, OMG document 94.8.4, 1994.

[34] J.K.Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[35] F. Pachet. On the Embeddability of Production Rules in Object Oriented Languages. *Journal of Object-Oriented Programming*, 8(4):19-24, 1995

[36] J-L. Peterson. *Petri-Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, U.S.A., 1981.

[37] Y. Shoam. Agent-oriented Programming. *Journal of Artificial Intelligence*, 60(1):51-92, 1990.

[38] L. Steels. Beyond Objects. In *Proc. of ECOOP'94*, Bologna, Italy, 1994.

[39] Sun Microsystems Inc., ToolkTalk Programmer's Guide, 1992.

[40] M. Tokoro. The Society of Objects. In *Proc. of OOPSLA'93*, Vancouver, B.C., Canada, 1993.

[41] G. van Rossum, http://www.python.org/

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

[43] 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.