Rule-based Object Coordination

Jean-Marc Andreoli and Hervé Gallaire and Remo Pareschi

Rank Xerox Research Center
6 Chemin de Maupertuis, 38240 Meylan (France)
Email: {Jean.Marc.Andreoli, Herve.Gallaire, Remo.Pareschi}@xerox.fr

Abstract. We discuss a framework in which the traditional features of objects (encapsulation, communication, etc.) are enhanced with synchronization and coordination facilities, using the declarative power of rules. We propose two interpretations of rules, one *re-active* and the other *pro-active*, corresponding to different kinds of interactions between the rules and the objects. Finally, we consider the problem of capturing domain specific knowledge within a general coordination framework, for which constraints offer a promising direction of research.
keywords: objects, rules, coordination, constraints.

1 Introduction

Object-oriented Programming (OOP) is meant to provide an integrated framework for a technology of software components [20]. In OOP languages (such as Smalltalk and C++), objects have an interface that specifies applicable operations, and have a local state shared by the objects' operations. Variables representing the internal state of the object are called *instance variables* and operations of an object are called *methods*. The implementation of the methods is *encapsulated* (and hence "hidden") in the interface itself. Method invocation is obtained through *communication*, namely through *message passing*.

Objects provide a rich primitive notion of modularity which gives strong language support to the development and maintenance of modular software components. Components can be directly implemented as objects, and their services, specified as methods, can be accessed and used by different clients; this, in principle, guarantees *inter-application* reusability of components. Furthermore, encapsulation of implementation is a fundamental contribution to solving the problem of the so-called legacy systems, in that it enables an incremental, evolutionary process for phasing out outdated technology. For instance, an information system written in, say, COBOL can be initially wrapped up in an object, after which the COBOL implementation can be replaced by one based on SQL, supporting the same functionalities. The clients of this object can continue to access its services without being aware of the change.

The promise of a new programming paradigm with inherent support for reuse has however been only partially fulfilled by OOP. The reason for this is simple: to achieve such a goal, encapsulation of implementation and communication-based access to object services are, by themselves, not enough; there is a need to complement them with mechanisms that explicitly allow the building of applications by composing together independent pieces. These mechanisms are global and group-oriented rather than local to a single objects; in general, their purpose is to prescribe how objects should coordinate with respect to each other in order to achieve collective behavior. To give a simple example, in a project management system, we may want to synchronize the launching of an object encapsulating an editor upon reception of all the different files that will be merged, via the editor, into a final project report; or, in electronic banking, different transactions over objects encapsulating independent accounts may need to be sheltered by a "metatransaction" that validates the whole set of events before notifying the operator with the permission to start a new action. We shall refer to such constructs as "object coordination schemas".

Our purpose here is to show how constructs that come from the tradition of declarative (rule-based) programming languages, such as rules and constraints, fit the requirements for object coordination schemas. One possible view of rules is that they declaratively define conditions for mapping the current state of the world into a new state; thus, in a world of objects, they can be used to specify the coordination steps needed to go from one global state to another. Constraints define restrictions over the domain of interpretation of rules; thus, they can be exploited for capturing restrictions over general coordination schemas determined by specific classes of applications.

Declarative programming facilities for object coordination provide concrete instances of "coordination programming" in the sense of [6]; they also fit within the growing trend of scripting languages (e.g. Visual Basic, AppleScript) but add to it the benefits of a well-defined semantics. Specifically, it is possible to do metareasoning about this kind of coordination programs; this allows us to infer program properties that can then be used for supporting program execution via such techniques as abstract interpretation and partial evaluation. Furthermore, declarative constructs offer the particular flavour of transactional behavior that comes from the tradition of automated reasoning, that is, *backtracking*. The "undoing of choices" that backtracking allows could appropriately be coupled with standard transactions to implement "metatransactions", namely rules that coordinate the execution of transactional events over different objects (see for instance the example above on electronic banking). Thus, the application domain of such metatransactions would be given by groups of (transactional) objects not meant *a priori* to communicate and to work together.

In section 2 we describe two different kinds of coordination, both implemented through rules, namely *re-active* synchronization of events and *pro-active* coordination of objects. In section 3 we outline how constraints can be integrated. Finally, section 4 concludes the paper.

2 Rules for Coordination: the Re-active and the Pro-active Case

A popular distinction for classifying software systems is between *transformational* and *re-active* systems [10]. Briefly, transformational systems are totally defined by their input/output behavior, i.e. they correspond to procedures in the traditional sense, while re-active systems are not based on the notion of input/output but instead continuously interact with the environment. Operating systems are typical examples of re-active systems.

It has been claimed in several places (e.g. [21]) that objects make re-activeness an ubiquitous notion in computing. In other words, although the implementation of an object may define simple transformational behavior (like arithmetic operations, list manipulation facilities etc.), the access to such functionalities through an interface is intrinsically re-active, as it involves an interaction between the object and the external world.

Our use of rules to implement object coordination is based on yet another kind of systems, namely *pro-active* systems, which are characterized by a relationship with the environment which is even more advanced than in the re-active case. Indeed, pro-active systems aim at *influencing* and *modifying* the environment, rather than simply re-acting to external stimuli. Here, it is easy and tempting to draw an analogy with biological evolution: pre-human forms of life behave re-actively (with different degrees of specialization depending on their position in the evolutionary scale), pro-activeness is instead typically "human" [7] (with all the risks for the environment that this involves, as we may also note).

Concrete instances of pro-active systems are given by intelligent software agents [15, 17, 16], namely computer applications which autonomously execute a task for their owners, for example to find the best airline connection and schedule other practical details of the trip. Rules in their pro-active interpretation provide a general facility for coding an important category of such agents, namely coordinators, in a way that can be powerfully combined with traditional OOP.

Nevertheless, for certain simple but useful cases of coordination, namely event synchronization, a re-active interpretation of rules is adequate. This will be illustrated in Sec. 2.1. Full-blown coordination requires instead a pro-active interpretation, as will be illustrated in Sec. 2.2.

2.1 Re-active Rules for the Synchronization of Active Objects

Rules have been applied successfully in Artificial Intelligence, especially in the field of expert system design. In this context, rules act upon tokens of knowledge, e.g. the "facts" of an expert system. Each rule specifies how to infer a set of tokens (the right-hand side of the rule) from a set of already established tokens (the left-hand side of the rule).

In a completely different way, OO computing is also a perfect domain of application for rules, especially for the design of systems based on *active* objects [12]. In this case, the tokens manipulated by the rules are object states and method invocations (messages). The left-hand side of rules synchronizes the execution of events corresponding to the modification of object states and to the triggering of messages; the right-hand side of rules expresses the notification of new events. Thus, in this approach, each rule acts as an autonomous, long-lived thread of activity continuously looking for events to be synchronized. This behavior can either be obtained by polling or by some asynchronous signal mechanism; the declarative nature of rules "hides" this implementation aspect.

The computational model thus obtained is purely *re-active*. It applies quite naturally to the design of event managers and all sorts of "synchronizers"; thus, it provides a reductive but useful implementation of object coordination schemas as object synchronization schemas. As a simple example, take the following specification of a remote procedure call (RPC) protocol on the client side:

$$< in\ \ state > \ \mapsto\ < mid\ \ state > < request >$$
$$< mid\ \ state > < reply > \ \mapsto\ < out\ \ state >$$

The client, controlled by the two rules above, starts from an input state where it emits a request, and evolves into an intermediate state where it waits for a reply; upon effective reception of the reply, the RPC protocol is completed, with a corresponding transition of the client into an output state.

One interesting feature of rules is that they have a purely abstract reading (as in the example above), which makes no assumptions about the nature of the clients, the servers, and the messages (they are all represented by symbolic tokens). To use rules as an effective programming tool, we need however to connect them to concrete software entities. This concretization step requires some attention. One possible approach assumes that the pool of tokens is accessed not only by the rules but also by external processes (launched and connected beforehand) whose behavior is implemented through whatever programming language or software tool is deemed appropriate. For example, the keyboard or mouse events generated by a user on a terminal could be mapped into tokens in the pool, and, conversely some specific tokens in the pool could be mapped into events sent to the screen and other output devices. Similarly, certain tokens could be mapped, say, into queries to a database server (with some appropriate format conversions to accommodate the server's query language) while the answers from the server could be mapped back into tokens (see Fig. 1).

2.2 Pro-active Rules for Coordination

The approach presented in the previous section is well adapted to the case of external processes which act as simple re-active black-boxes, communicating with the outside world through purely asynchronous message streams (Fig. 2), but this is of course a very narrow perspective. Many systems offer communication protocols which are much more refined than pure asynchronous message passing, and these protocols would have to be bypassed when connecting the external process to the pool of tokens. They could then be reimplemented at the rule level (as in the rule-based implementation of an RPC given above), but that would be a waste of time and, most probably, of efficiency, contradicting one of the essential principles of object orientation, namely reuse.

An ideal solution to this problem would enable reuse of any protocol. As a first step in this direction, we describe here a new mechanism for connecting external systems to rules, based on a *pro-active* reading of the rules. The idea

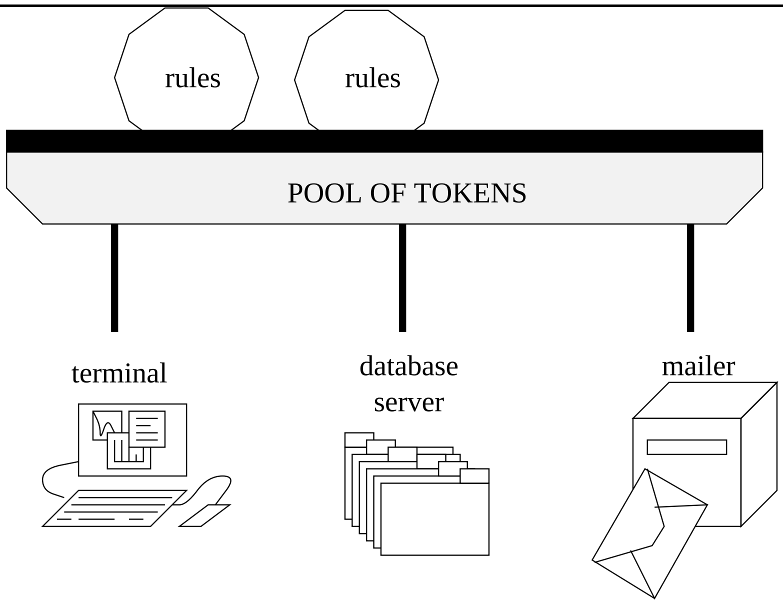

Fig. 1. An example of rule-based synchronization

is to switch from an *extensional* representation of the pool (as a set of explicit tokens) to an *intensional* one. The behavior of the rules is consequently modified: a rule can no longer just wait for the tokens on its left-hand side to appear on the pool, but it must materialize the intensional description of the pool in all possible ways, so as to make it *happen*. This mechanism is described in detail below.

This new, pro-active computational model, subsumes the re-active model. It is adapted to the design of real "coordinators" rather than simple synchronizers, exploiting fully the protocols of communication available at the level of the coordinated systems.

Object Coordination Schemas An object coordination schema involves two kinds of entities: the *coordinator* and the *participants*. We assume here that the participants are active objects, either because they were originally programmed to be such, or because they were wrapped up in active objects (in the case of legacy systems). Rules are used to define the coordinator's behavior. The assumption now is that the pool of tokens on which the rules apply is no longer extensionally available to the coordinator, but is instead handled by the participants themselves (Fig. 3). The tokens on the left-hand side of such rules represent actions on the participants (typically method invocations). Thus, by accessing a token a, the coordinator issues the following request to the participants:

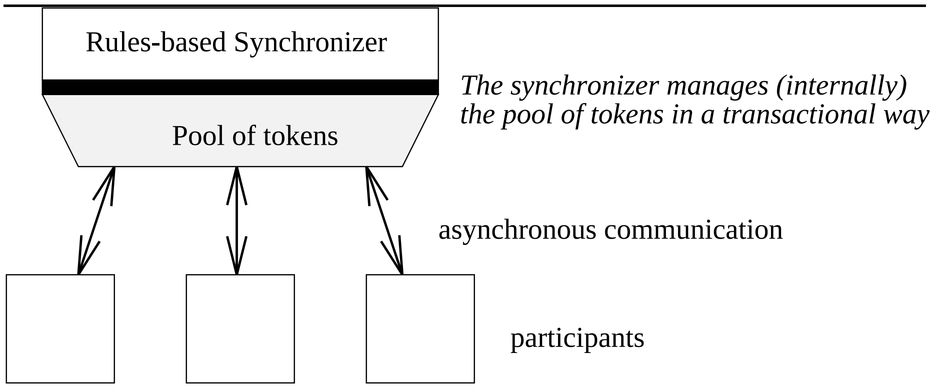

Fig. 2. The re-active interpretation of rules

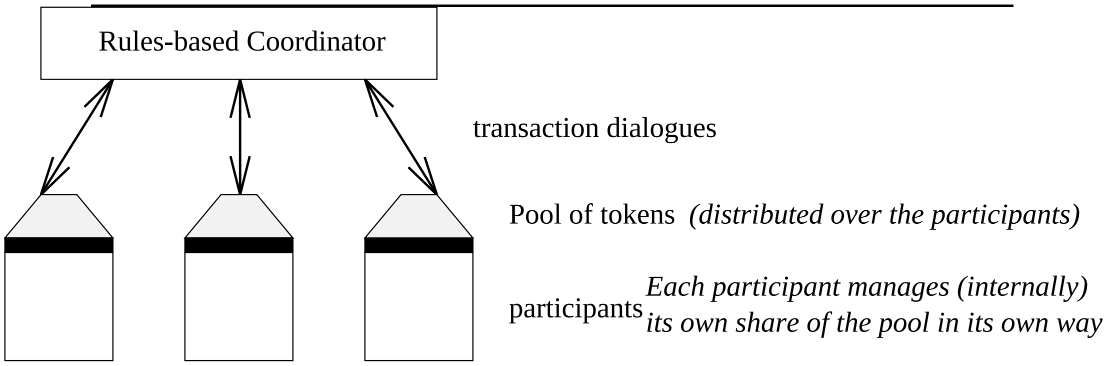

Fig. 3. The pro-active interpretation of rules

$\rightarrow$ **perform an action capable of producing** a

This request may be satisfiable in one or more ways, or may not be satisfiable at all; furthermore, the fact of satisfying it may change the internal state of the concerned participants. But, of course, we want this change to happen only when the rule is certain to apply; that is, when all the tokens in the left-hand side of the rule are available. In other words, we want to enforce a *transactional* reading of the tokens. A token a on the left-hand side of a rule triggers a "transaction dialogue" between the coordinator and one of the participants, consisting of the three phases (Fig. 4):

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

Reservation : the coordinator *reserves* from the participant a specific action from those identified during the Inquiry phase; this action is then said to be *engaged*. From a computational point of view, this may imply grabbing some internal resources of the participant.

Confirmation/Cancellation : the coordinator either confirms or cancels the action engaged during the Reservation phase. If confirmation occurs, then the corresponding resources are modified, otherwise, they are released.

Notice that this pattern for a "transaction dialogue" is meant for contexts where participants are active objects, which can be accessed but are *not* controlled by the coordinator. That is, participants have their own, independent line(s) of activity invisible to the coordinator (e.g. a user dialogue through a GUI). Therefore, the actions declared as enabled during Inquiry may be disabled during Reservation, because of a change in the state of the participant.

Obviously, the interaction between coordinator and participants is far more complex than in the simple re-active case considered before. The Inquiry phase involves a *deferred synchronous* form of communication: the coordinator incrementally receives information about all the possible actions capable of satisfying a given request; but, rather than blocking until this process is complete (which may never be, as there may be infinitely many such actions), it gets hold of a handle from which it can retrieve individual replies later on. The Reservation phase is on the other hand *purely synchronous*: the participant is either enabled or disabled to grab the resources required for the selected action, and this can be checked out instantaneously. Finally, the Confirmation/Cancellation phase is *purely asynchronous*, as it does not involve any reply from the participant. Similarly, the tokens on the right-hand side of a rule are simple notifications of new tokens to the participants, which are performed asynchronously; they do not have the transactional reading of the left-hand side.

Meta-transactions The rules themselves can be viewed as meta-transactions, which impose an overall transactional behavior on participants which are themselves transactional. Thus, a transaction dialogue between the coordinator and one of the participants identifies one thread in the overall process of checking the applicability of a rule; the interleaving of these threads is itself monitored in a transactional way. Deadlocks between different rule applications can easily be detected and eliminated at the level of a single coordinator. If several coordinators are involved, deadlock detection can still be achieved by refining our transactional dialogue, so as to propagate the necessary information across the coordinators.

There is very little homogeneity imposed on the participants, which can offer very different transactional behaviors. Non-transactional participants can be accommodated too, once wrapped into a trivial transactional layer which ignores the complex transactional aspects involved in the Reservation and Confirmation/Cancellation phases, simply forcing the participant into an error state in case of Cancellation.

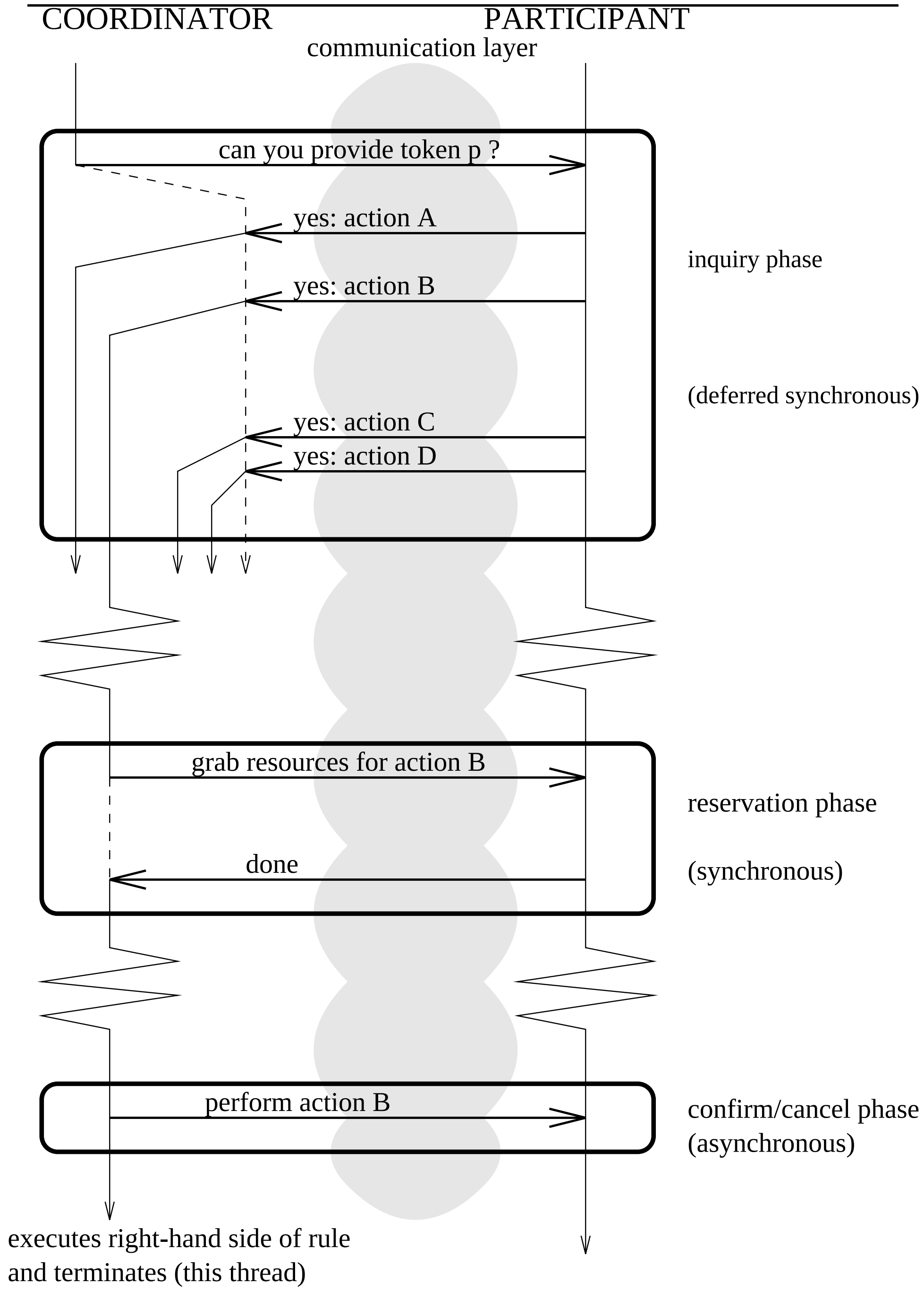

Fig. 4. A pattern for transaction dialogues for rule-based coordination

Notice that transactionality, or, more precisely, atomicity, is inherent to the kind of production rules we use, and is not specific to our system. The re-active rules we considered in section 2.1 already have an atomic behavior, in that a rule either consumes all its left-hand side, if it is present and the rule has been choosen by the engine, or does nothing at all. Even simple Horn clause languages with guards also have a notion of atomicity at the level of rule application: rule instances make local conflicting instantiations of variables and, once the guard is passed by one instance, its own local changes are made global while the changes made by the other instances are cancelled, thus ensuring an all-or-nothing behavior.

However, atomicity is not transactionality. The atomicity property of re-active rules, or guarded Horn clauses, is purely *internal* to the rule engine and can hence be handled without the need of an explicit roll-back mechanism. This becomes necessary with pro-active rules, since, in that case, the rule engine does not have direct control over the implementation of the tokens it manipulates. The transactional nature of pro-active rules is therefore explicit, and must be supported by external software agents with an explicit transactional behavior.

2.3 Applications

Coordinators will be mainly applied to the area of business process support, i.e. in such domains as workgroup and CSCW, workflow management, distributed decision making, which involve complex interactions among human and software agents.

A simulation of remote banking has already been developed. It supports complex dialogues between a bank operator and various bank accounts (considered as the participants in the coordination). The database server holding the accounts is replicated and failure to reply by one replica can be detected (by a time-out mechanism) and handled. In Fig. 5, we illustrate a simplified "transfer rule" which, upon reception of an order from the bank operator, atomically transfers an amount split across two accounts (`Acct1` and `Acct2`) into a third account (`Acct`). In our language, the rule looks like:

```
transfer(Acct1,Amnt1,Acct2,Amnt2,Acct) @
extract(Acct1,Amnt1) @ extract(Acct2,Amnt2) <>-
    insert(Acct,Amnt1+Amnt2).
```

The symbol `<>-` separates the two sides of the rule while the symbol `@` separates the tokens on each side. In fact, several other rules are required to handle failure of one account to provide the amount.

We have already mentioned before that the re-active reading of rules is subsumed by the pro-active one. The example above illustrates this fact. Typically, the orders from the bank operator are events which we want our rule to "react" to. This is indeed, in fact, what happens. Here, the operator is viewed as a bag of orders (events) and the Inquiry phase for the token `transfer(Acct1,Amnt1, Acct2,Amnt2,Acct)` will retrieve each of them successively. The bag may of

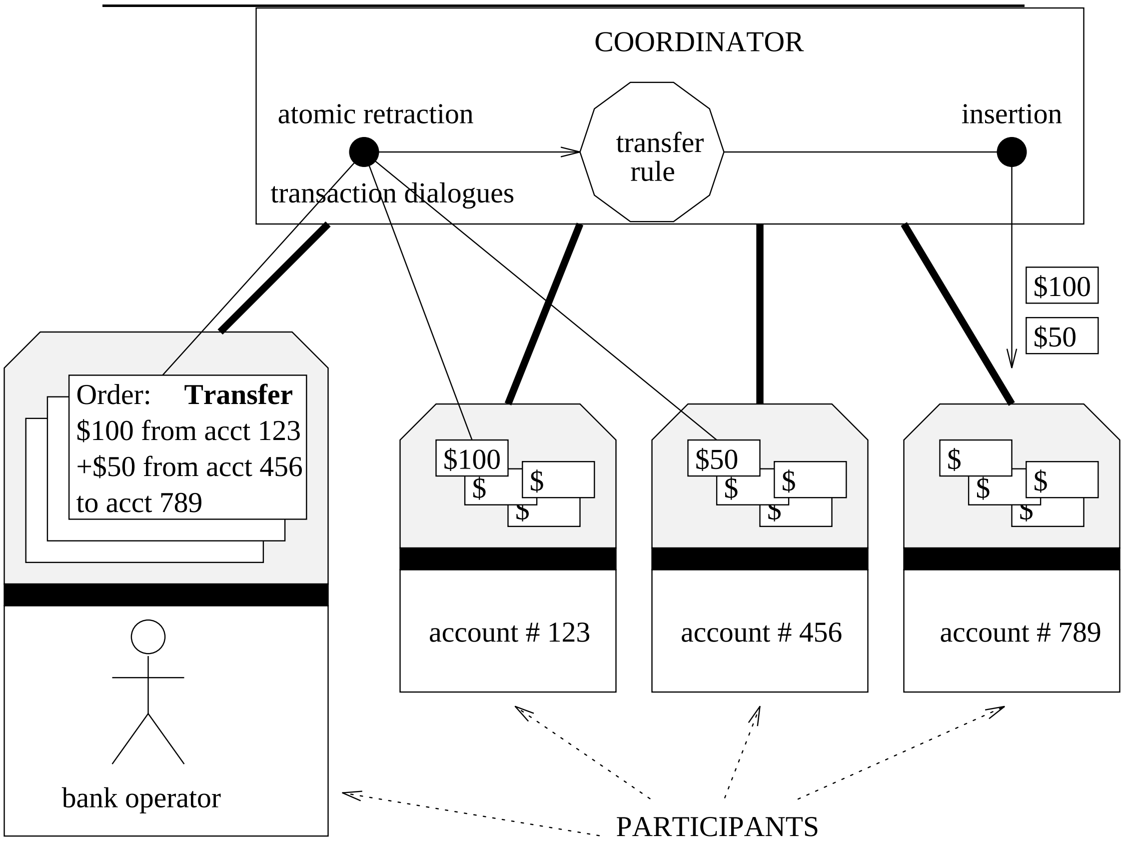

Fig. 5. Rule-based coordination for remote banking

course be asynchronously filled in by the operator's interactions. It may happen that an order returned by the Inquiry is not processed, if the rule cannot grab the other tokens it requires. In that case, the order will stay in the bag, unprocessed. Of course, we would wish to discard such un-processed orders, after a certain time-out period. One way of achieving this is by a rule of the form

```
transfer-date(Date) @ out-of-date(Date) <>-
    timeout-procedure.
```

Here the Inquiry phase for the token `transfer-date(Date)` also retrieves the bank operator's orders (so it competes with the previous rule), but it is only interested by the date at which each order is emitted. The Inquiry phase for the token `out-of-date(Date)`, on the other hand, only returns a (dummy) result when the time elapsed between the date in the argument and the present exceeds a fixed timeout. Finally, the insertion of the token `timeout-procedure` triggers some procedure which informs the operator of a failure of its order (an argument identifying the order would probably be needed).

2.4 Implementation

We have applied both the re-active and the pro-active interpretation of rules to the case of the rule-based model LO [4, 3, 2], an abstract model for describing concurrent processes whose semantics is given by a recent development in Logic, namely Linear Logic [8]. Linear Logic provides a powerful framework for studying different kinds of interactions within and among processes, and effectively subsumes other frameworks like Petri Nets.

The re-active implementation offers limited support to connect the coordinator and the participants, but implements the whole set of connectives and inference mechanisms defined in the LO model. On the other hand, the pro-active implementation under development will offer transparent access to the participants through the use of the CORBA standard for object inter-operability [19], but is currently limited to a subset of the LO model (excluding the cloning operator). The gain in expressive power derived from the pro-active interpretation of rules already makes this subset interesting. One important feature in the implementation of the coordinator is that it will be multi-threaded, with different threads handling different rules and different instances of the same rule. This will ensure fairness in the management of the transaction dialogues.

3 Coping with Domain-specific Knowledge

So far, we have described an approach dedicated to "programming" certain types of behaviors of distributed systems, in a declarative manner.

This discussion addresses only a first step in synchronization and coordination. Indeed, we know from many studies in various disciplines that coordination is a process of managing dependencies among activities. It is very tempting to characterize different kinds of dependencies and to identify the coordination processes that can be used to manage them [11]. This type of analysis can be done rather abstractly; for instance, Malone and Crowston in [11] rely upon an interdisciplinary study from as diverse areas as computer science, organization theory economics, psychology or linguistics. They identify and analyze, from these perspectives, processes managing shared resources, producer/consumer relationships, simultaneity constraints, task/sub task dependencies, group decision making as well as communication. This leads them to various design perspectives, thus alternatives, for classes of processes. Going a step further we are now engaged in the analysis of many applications that require coordination; we will extract from these a set of common coordination patterns that can be (re)used in given classes of applications, at a higher level than the programming constructs we discussed in the previous section. Thus, we will be able to provide a toolkit which can then be given to "programmers" [5]. Examples of such high level primitives have already been identified. An attempt in a more restricted context has led to the development of [9]. This research is still at an early stage, but we are confident that generic primitives will be identified.

Coordination leads us to a different attitude towards programming, as already mentioned: it deals with dependencies. Such dependencies need to be maintained during the life of the system. Our specific view of dependencies is that they are active, even pro-active; i.e. they support a programming paradigm. There is a relatively recent field of computer science that we now need to connect to, namely Constraint Programming. The study of constraints and constraint solving does not belong only to mathematical logic; on the other hand, as a programming tool, constraints have mainly been introduced in the context of logic languages, particularly logic programming of the Prolog tradition [18] but not exclusively, as some versions of constraint programming stem from concurrent logic programming and Linear Logic [14, 13, 1]. In any case, other non-logical frameworks have been and are being used to program with constraints. The significant point for our purpose is that the type of coordination programming we have proposed in the previous sections is not domain specific, as our coordination constructs do not make any assumptions about the application domains to which they can be applied. Yet, we know that we can extract coordination patterns that are specific to given applications. Constraints can offer the higher-level primitives to incrementally introduce domain specific knowledge, in just the same way that the now "classical" constraint programming systems build upon knowledge of constraint solving in specific domains: in finite domains we have many propagation algorithms which prune the domains to establish a solution to a set of constraints, in the real fields we have simplex algorithms, in the Boolean we have an efficient algebra, etc. From this point of view, our work is thus to be seen as another step towards model-based computing, as coined by Bobrow and Saraswat.

4 Conclusion

To recap, object-oriented programming provides some crucial capabilities: encapsulation of behavior, and communication as programming primitives. Yet, OOP has not proved as effective as first hoped, as traditional OO languages suffer from a significant drawback: they do not deal with the coordination of activities in which objects are involved. Our proposal here is to achieve this by superimposing declarative constructs, such as rules and constraints, on objects so as to implement coordinators capable of "reactive" as well as "pro-active" behavior.

Acknowledgement

We are grateful to Steve Freeman for helpful comments on this paper.

References

1. Andreoli, J-M., Borghoff, U., Pareschi, R. (1994). Constraint Based Knowledge Brokers. *Proc. of PASCO'94*, Linz, Austria.
2. Andreoli, J-M., Castagnetti, T., Pareschi, R. (1993). Abstract Interpretation of Linear Logic Proofs. *Proc. of ILPS'93*, Vancouver, Canada.
3. Andreoli, J-M., Ciancarini, P., Pareschi, R. (1993). Interaction Abstract Machines. In Agha, G., Yonezawa, A. and Wegner, P. Eds, *Research Directions in Concurrent Object Oriented Programming.* MIT Press, Cambridge, Ma.
4. Andreoli, J-M., Pareschi, R. (1991). Communication as Fair Distribution of Knowledge. *Proc. of OOPSLA'91*, Phoenix, Az, U.S.A.
5. Borghoff, U. (1994). LoKit, a Toolkit for building Collaborative Applications. Rank Xerox Research Centre, Grenoble, France, Technical Report.
6. Carriero, N., Gelernter, D. (1990). *How to Write Parallel Programs.* MIT Press, Cambridge, Ma, U.S.A.
7. Gehlen, A. (1986). *Urmensch und Spätkultur. Philosophische Ergebnisse und Aussagen.* Aula Verlag GmbH.
8. Girard, J-Y. (1987). Linear Logic. *Theoretical Computer Science* **50**, 1–102
9. Goldberg, Y., Safran, M., Shapiro, E. (1992). Active Mail, a Framework for Implementing Groupware. *Proc. of the ACM Conference on Computer Supported Cooperative Work*, Toronto, Canada.
10. Harel, D., Pnueli, A. (1985). On the Development of Reactive Systems. In Apt, K.R. ed, *Logic and Models of Concurrent Systems*, Springer Verlag, Berlin.
11. Malone, T.W., Crowstone, K. (1994). The Interdisciplinary Study of Coordination. *ACM Computing Surveys* **26**:1, 87–119
12. Nierstrasz, O. (1994). Composing Active Objects. In Agha, G., Yonezawa, A. and Wegner, P. Eds, *Research Directions in Concurrent Object Oriented Programming.* MIT Press, Cambridge, Ma.
13. Saraswat, V.A., Lincoln, P. (1992). Higher Order Linear Concurrent Constraint Programming. Xerox Parc, Palo Alto, Ca, U.S.A., Technical Report.
14. Saraswat, V.A., Rinard, M., Panangaden, P. (1991). Semantic Foundations of Concurrent Constraint Programming. *Proc. of 9th ACM Symposium on Principles of Programming Languages*, Orlando, Fl, U.S.A.
15. Shoam, Y. (1991). Agent-Oriented Programming. Standford University, Robotics Laboratory, Stanford, Ca, U.S.A., Technical Report, To appear in the Journal of Artificial Intelligence.
16. Steels, L. (1994). Beyond Objects. *Proc. of ECOOP'94*, Bologna, Italy.
17. Tokoro, M. (1993). The Society of Objects. *Proc. of OOPSLA'93*, Vancouver, B.C., Canada.
18. Hentenryck, P.van (1989). *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Ma, U.S.A.
19. Vinoski, S. (1993). Distributed Object Computing with CORBA. *C++ report* **5**:6, 33–38
20. Wegner, P. (1989). Conceptual Evolution of Object-Oriented Programming. Brown University, Dept. of Computer Science, Providence, RI, U.S.A., Technical Report.
21. Wegner, P. (1994). The Expressive Power of Interaction. Brown University, Dept. of Computer Science, Providence, RI, U.S.A., Technical Report.