Supporting Cooperation in Project-Enterprises

G. Canals, C. Godart, M. Munier, S. Tata
UMR n°7503 LORIA - Université Henri Poincaré - Université Nancy 2
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex - FRANCE
E-mail: godart@loria.fr

Abstract— **Due to the popularization of Internet, cooperative applications of *project-enterprise* type are expected to become commonplace on the WEB. By *project-enterprise*, we understand short-lived enterprises which are built by aggregating several partners around a project and for the duration of this project. They require new technologies to organize their short duration networks. For this purpose, this paper describes a flexible approach to build cooperation support software by assembling basic and generic cooperation patterns.**

Keywords—**Cooperation patterns, cooperation modes, methodology, project-enterprise**

I. Introduction

Due to the popularization of Internet, cooperative applications of *project-enterprise* type are expected to become commonplace on the WEB. The concept of *project-enterprise* depicts the idea that many applications are the result of the cooperation between several actors, playing different roles, who build a temporary relational system which is structured by a common objective project and for the duration of a project. This corresponds also to the idea of temporarily virtual enterprise [1]. Building trade is a good example of such short-lived enterprises: it implicates a lot of partners (architect, research consultant, control office, building firm, electrician, carpenter, ...) which build an enterprise for the duration of the building work.

We think that the number of project-enterprises is really huge and that project-enterprises are an important market for the internet technology which allows short duration and low-cost connections between partners who had not invested into specialized communication channels. However, cost consideration is not sufficient: internet technologies must modify at the very least the habits of partners and must compensate this modification with an increase of value. We think that the development of flexible cooperation models, which can apply to many different situations and for many different partners, acts in this direction. This is the objective of the work described in this paper. Our approach is to define generic cooperation patterns (or cooperation bricks), which can be combined to build efficient cooperation policies. As in addition project-enterprise partners have not necessary a large computer men staff, this process must be easy to implement. However, we do not cover all the aspects of project-enterprises. We are mainly interested in the development of new technologies and new services to support the concurrent engineering, and especially the co-design activities, inherent to project-enterprises.

This paper describes the approach we develop to reach this objective with some intermediate results to prove its (partial) feasibility. Section II introduces the approach. Section III describes a first taxonomy of cooperation modes on which we are currently reasoning and gives an example based on co-designing a building (III-C). Section IV explains why and how it can work (IV-A considers some formal aspects, IV-B quickly describes an implementation strategy we are actually experimenting). Finally, section V concludes.

II. An approach to build cooperative applications

Our approach to build cooperative applications is based on a design methodology illustrated in figure 1. This methodology is based on the assumption that any project-enterprise can be described as a network of distributed activities. These activities are interconnected through some communication links that allow the exchange and the sharing of various software artifacts and documents.

The methodology consists in two main steps: in a first step, designers build a cooperation table starting from their knowledge about the target application and from a taxonomy of cooperation modes. This cooperation table describes the different kinds of interaction that may occur between the different activities in the project. In a second step, developers implement the cooperative application based on the cooperation table resulting from the first step and on a library of cooperation patterns. This is illustrated in figure 1.

Our goal is to provide tools to support this approach. A first step toward this goal is the definition of a set of cooperation modes to which designers can

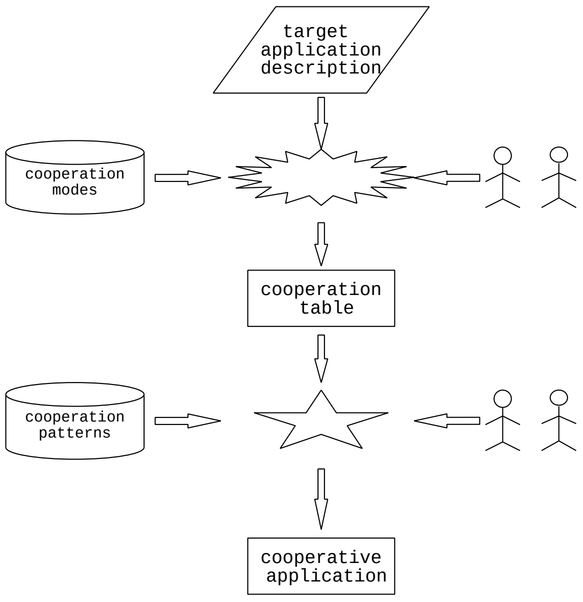

Fig. 1. An Approach to Build Cooperative Applications

refer. We discuss a taxonomy of cooperation modes in the next section. The result of the fisrt step is a cooperation table: we suppose that each row of this table establishes a relation between two activities, an object and a cooperation mode. Based on this table and on a set of cooperation patterns, developers must be able to build the cooperative application. A cooperation pattern is a programming view of a cooperation mode.

III. A taxonomy of cooperation modes

Before to describe a first taxonomy of cooperation modes on which we are currently reasoning, we introduce a general cooperation architecture on which our cooperation modes are based. Especially, we focus on cooperation by object sharing.

A. Cooperation by object sharing

This section quickly introduces our model of cooperation based on object sharing. We consider that a cooperative application is a network of connected and interacting activities. Each activity has its proper subgoal which contributes to the global goal of the global application and stores objects it works with in its own object base (see figure 2).

Human agents work being connected to an activity. They modify objects with their usual tools by checking them out of the activity base and checking them in back to the base when the modification is complete.

Activity bases are multiversionned: every object belonging to an activity base is in fact stored as a version DAG. When a user wants to modify an object, he generally checks out from the activity base the latest version of the object, modifies it and checks in back to the base, creating then a new version for this objects. This is illustrated in figure 2 where a user connected to activity `three` checks out object `A`, modifies it and checks in back to the base.

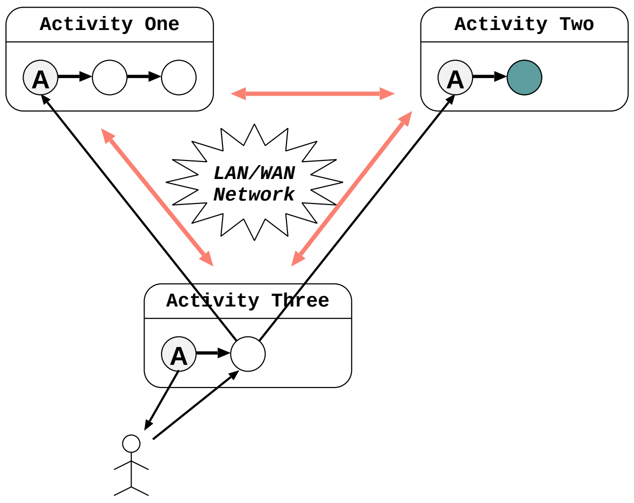

Fig. 2. Object Sharing

In addition, each object may exist in different versions in different activity bases at the same time. In fact, different activities interact by transferring copies of the objects from remote activity bases to their proper base. This can be done just to simply read the value of the object, or to create a new dag of cooperative versions, as introduced in the following section. This is depicted in figure 2: each activity (`one`, `two` and `three`) owns its proper copy of object `A`: activities `one` and `two` transferred the object `A` from the base of the activity `three`. They both derived a new version graph from the version they copied.

In this context, transfers must respect some cooperation policies or rather, based on this architecture, a cooperation policy is defined by a set of constraints on transfers between activity bases. Of course, organization of workspaces and workspace transfers can be largely deepened, but we content us here with the principles requested to understand the cooperation modes introduced below.

B. Cooperation modes

B.1 Taxonomy criteria

Our taxonomy of cooperation modes is based on the following three main criteria.

• **read/write dependencies**. The behavior of the interactions between different activities depends upon the nature of operations these activities apply to objects. As example, when two activities modify the same object, some work risks to be lost. It is not the case when the two activities simply read these objects. We say that, for a given object, it exists a *read/read* dependency between two activities if they read the same object; it exists a *write/read* dependency if one of them reads the object and the other modifies it; it exists a *write/write* dependency if the two activities modify it. Different *read/write* dependencies can apply for the same couple of activities but for different objects. Note that the *write/read* dependency is oriented; the others are not.

• **synchronous/asynchronous view**. We say that, for a given object, two activities have a synchronous view of this object if they observe exactly the same sequence of states of the object. The modifications of the object that leads to a new state transition may be produced by one or the other of the two activities, but also.
When two activities share an object and may observe a different state sequence for that object, they have an asynchronous view of this object. Two activities can have a synchronous view of an object and an asynchronous view of another.

• **final state dependency**. For a given object, two activities are *final state dependent* if they must share the same value of the object at the time they terminate their execution. The two activities are not required to simultaneously terminate : one can terminate producing a final value of the object while the other continue its execution using this value, but without modifying it.
It is easy to understand that, for a lot of activities in a lot of applications, its is necessary to synchronize their view of shared objects before to terminate. This applies especially for activities which have an asynchronous view of some object(s) shared with other activity(ies).

B.2 Cooperation modes

Combining these three criteria, we can distinguish between twelve modes of interaction (see figure 3) between two activities and for a given object. In addition, we have introduce the *concurrent* and the *competitive* behavior to characterize two activities which do not cooperate but must be synchronized.

1. **concurrent.** Two activities are *concurrent* if they access the same object but each one ignores (and must ignore) the existence of the other. It is the general paradigm largely implemented in traditional database management systems.

Dependencies			Cooperation Mode
Read/Write	Asyn./Syn.	Final St.	
Read-Read	A	NFS	neighbouring
Write-Read	A	NFS	developer-inspector
Write-Write	A	NFS	alternatives
Read-Read	S	NFS	mirror read
Write-Read	S	NFS	sync. developer-inspector
Write-Write	S	NFS	groupware editor
Read-Read	A	FS	co-inspectors
Write-Read	A	FS	client-server
Write-Write	A	FS	cooperative write
Read-Read	S	FS	sync. co-inspectors
Write-Read	S	FS	sync. client-server
Write-Write	S	FS	sync. cooperative write
			concurrent
			competitive

Fig. 3. Cooperation Modes

2. **competitive.** Two activities are *competitive* if they must execute in isolation with respect to the application process.

3. **neighbouring**. Two activities are neighbours if they have transferred a version of the same object in their proper workspace to read it. Probably that the shared object is not critical for the application (it can be as example a user's manual of a programming language).

4. **developer/inspector.** Two activities follows a *developer/inspector* scheme if one (the developer) modifies an object and the second (the inspector) reads one or several successive cooperative version produced by the other. As example, a user has the responsibility to edit a part of a document; the other inspect at a given time the part assigned to the first.

5. **alternative.** Two activities are *alternative* if, starting from the same original object, they develop two alternative versions of the same object without interfering. Each alternative version can lead to a dag of cooperative versions. There is no transfer between their workspaces. May be that the different versions will have to be merged and this might not be an easy task, but this is not the role of the two alternatives.

6. **mirror read.** Two activities read the same version of an object. This can be for security purpose or because of the sharing of a common awareness source.

7. **synchronous developer/inspector.** The same behavior than *developer/inspector*, but in addition, the inspector see each last cooperative version produced by the developer as soon as possible.

8. **co-editor.** Two activities are *co-editor* if they modify simultaneously the same object. Each new cooperative version produced by an activity is seen as soon as possible by the other.

9. **co-inspector.** Two users are *co-inspector* if they read, possibly different cooperative versions, of the same object, but they must agree on the last value they have read before to terminate their execution.

10. **client-server.** Two activities follow a *client-server* scheme if one has the responsibility to modify an object and the other reads this object to integrate it in its proper work. Client and server can momentarily see different versions of the object, but they must agree on the value of the shared object before the server stabilizes it.

11. **cooperative write.** Two activities develop two alternative versions of the same object and must merge their modifications before to conclude.

12. **synchronous co-inspector.** Two activities read the same version of an object and must agree on the last version they read before to conclude simultaneously.

13. **synchronous client-server.** A client-server, but in addition, the client must read each new cooperative version produced by the server as soon as possible.

14. **synchronous cooperative write.** A cooperative write, but in addition, both activities see always the same version of the object. In other terms, each new cooperative version produced by an activity is seen as soon as possible by the other.

To conclude this section, we want to underline that this taxonomy consider only unidirectional cooperation modes. However, more complex modes can be built from these basic one. As example, we call *writer/reviewer* the combination of two symetrical *client-server* modes in which one object transferred in one direction is the review of the other transferred in the other direction.

C. An Example Case

To illustrate the approach introduced above, we apply it now to a simple project-enterprise in which four partners (an architect, a structural engineer, a HVAC engineer and a town planer, see figure 4) contribute to the development of a common concept, the plan of a building [2].

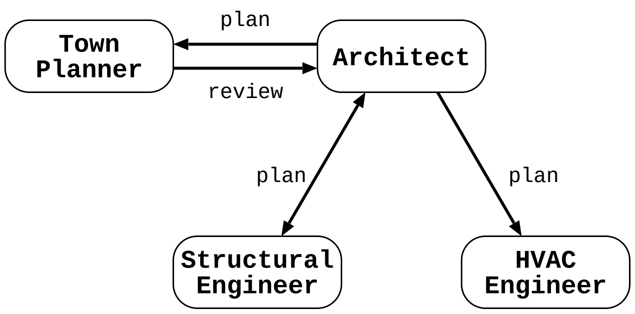

Fig. 4. A Project-Enterprise

The organization of work and especially the cooperation model is defined in the cooperation tables[1] illustrated in figure 5. The goal of the architect is to produce the plan of the building. It works under the control of the town planer which verifies that the external appearance of the building respects the rules defined for the town. They share two objects : the `plan` and the `plan-review`. The architect creates a first version of the plan and sends it to the town planner. The town planner verifies if the plan respects the district rules and returns a plan-review. Then, the architect and the town planner exchange asynchronously several versions of the plan and the corresponding plan-review. Finally, they must agree on a common value of the plan.

With this view of things, a *client-server* mode is well adapted to organize their interactions concerning the plan. To ensure that every review will be used by the architect, a *synchronous client-server* mode is better adapted for plan-reviews. As defined in section III-B.2, this means that the architect will have to take into account all successive versions of the review.

The architect and the structural engineer have chosen to interact very closely; they work on the plan at the same time, in a *cooperative write* mode. The only constraint is that they have to merge their modifications before to conclude.

An agreement between the architect and the HVAC engineer is that the architect provides the HVAC engineer with some preliminary versions of the plan. The objective is to allow him to organize his work in advance. However, the architect wants the HVAC engi-

[1]For some cooperation modes, the two activities do not fulfill the same role. For instance, when we use the client/server mode, one activity is the server while the other is the client. We underlined the role fulfilled by the activity.

neer to see the final version of the building plan: they work following a *client-server* mode.

activity	*remote activity*	*object*	*cooperation mode*
architect	struct. eng.	plan	cooperative write
architect	HVAC eng.	plan	client/server
architect	town planner	plan	client/server
architect	town planner	review	sync. client/server

activity	*remote activity*	*object*	*cooperation mode*
town planner	architect	plan	client/server
town planner	architect	review	sync. client/server

activity	*remote activity*	*object*	*cooperation mode*
struct. eng.	architect	plan	cooperative write

activity	*remote activity*	*object*	*cooperation mode*
HVAC eng.	architect	plan	client/server

Fig. 5. Cooperation Tables for the Example

IV. How and why can it work ?

This section gives some hints about how to put the approach into practice from the point of view of correctness of execution and from the point of view of system implementation.

A. *Correctness criteria to integrate cooperation modes*

Synchronizing interactions as introduced above is not a simple job. Our way to do it is to implement each cooperation mode as a generic cooperation (software) pattern and to develop a glue to assemble patterns.

Currently, we have yet experimented the approach with a subset of cooperation modes, including these introduced in the above example. Each generic pattern is defined as a set of rules which constrains the transfers between workspaces. These patterns are dynamically glued thanks to a protocol which implements a correctness criterion, the COO-serializability [3], [4]. This criterion characterizes the execution of processes which interact in *client-server* and/or *cooperative write* cooperation modes. In addition, the protocol can restrict the set of executions accepted by the criterion to manage *synchronous client-server* and *synchronous cooperative write* modes.

B. *Distributed control for cooperative activities*

Most distributed systems are based on a client/server architecture in which, though single activities may execute at geographically distributed nodes, the knowledge about the processes which execute is kept in a centralized database at the server level. This centralization makes it easier to synchronize and monitor the overall execution as all decisions are taken on this server which has a global view of the whole system. However, this means that activities have to be connected to this server continuously (figure 6-a). This does not correspond to the nature of the applications that we consider in which:

- Each actor already has his own environment and doesn't want to change his habits. Moreover, he has to be free to work as he wants in his environment.
- Due to the expensiveness of network connections, actors are generally disconnected when they work, but this sould not prevent them to work.
- In large projects, nobody possesses the entire knowledge of the system. Therefore, it is extremely difficult and often impossible to determine, in a centralized way, all the possible impacts of a given change.

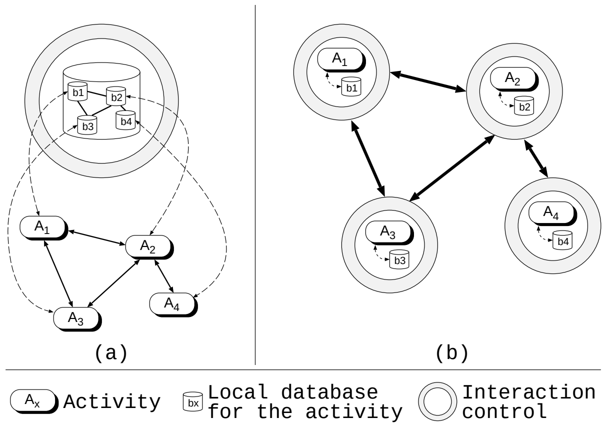

Fig. 6. Centralized Control vs Distributed Control

To support these requirements, our approach is based on a peer-to-peer architecture (figure 6-b). That is, an activity is viewed as a self contained component that cooperates with other components by exchanging, during its execution, some (possibly preliminary) results. By self contained we mean that an activity should manage itself its own data, both for data storage and interaction control. Doing this, each activity is responsible for its data exchanges with others activities. So, unlike configuration management tools or transactional systems, we avoid activity denpendency towards any kind of server, neither for data access nor for interaction control.

The main idea is that when an activity want to communicate with another one, these activities begin by negotiating a cooperation protocol (i.e. a cooperation mode). This negotiation will ensure, at least, that the protocol one activity wants to use is known by the other. Thus each activity will have a cooperation table which purpose will be to show which protocol to use to control the exchanges of a given data with a given activity. Figure 5 shows cooperations tables built for each activity of the example depicted figure 4.

Then, the system will ensure that all exchanges between these two activities will respect the negotiated protocol, while keeping them independent of one another: in case of protocol violation, only the faulty exchange is refused. From the user point of view, the main advantage of our approach is that we define cooperation protocols to coordinate data exchanges between activities and not to control the activities themselves. Therefore, an activity is largely independent of other activities for the task it performs. This means that each actor of the system is free to work like he wants, as long as his data exchanges with other actors are correct.

We go in further details with this architecture and its implementation as CORBA services in [5].

V. Conclusion

The above sections illustrate the fact that the approach is feasible from both a theoretical point of view and from an implementation point of view. From a theoretical point of view, our approach comes from a previous experience in the area of software development environments and was implemented in the COO system [6]. It works because we are able to refer for all considered cooperation modes to a common correctness criterion that acts as an integration mechanism. In addition, the current protocol is generic and application independent: this is a guarantee that it can be used by organizations with a small computer sciences expertise. All that is implemented in the COO system [6]. Our objective is to continue in this way but for a larger and better representative set of cooperation mode. The goal is to develop a framework for specifying and building distriubted cooperative applications with a predictable behavior.

References

[1] M. Hardwick and R. Bolton, "The industrial Virtual Enterprise," *Communications of the ACM*, vol. 40, no. 9, September 1997.

[2] K. Benali, M. Munier, and C. Godart, "Cooperative models in co-design," in *International Conference on Agile Manufacturing (ICAM'98)*, Minneapolis, USA, June 1998.

[3] G. Canals, P. Molli, and C. Godart, "Concurrency control for cooperating software processes," in *Proceedings of the 1996 Workshop on Advanced Transaction Models and Architecture (ATMA'96)*, Goa, India, 1996.

[4] G. Canals, P. Molli, M. Munier, and C. Godart, "A Criterion to Enforce Correctness of Cooperative Executions," *Information Sciences, Elsevier Sciences Inc.*, vol. 110, October 1998.

[5] M. Munier and C. Godart, "Cooperation services for widely distributed applications," in *Tenth International Conference on Software Engineering and Knowledge Engineering*, 1998.

[6] C. Godart, G. Canals, F. Charoy, P. Molli, and H. Skaf, "Designing and Implementing *COO*: Design Process, Architectural Style, Lessons Learned," in *International Conference on Software Engineering (ICSE18)*, 1996, IEEE Press.

About the Authors

Gérôme Canals is an associate professor in the Computer Science Department, University of Nancy 2 Technology Institute, Nancy, and a member of the ECOO team at LORIA. He received a Ph.D. in Computer Science from the University of Nancy I in 1992.

His research interests include distributed persistent workspaces, advanced transaction models, distributed shared memories and group communication protocols.

Claude Godart is presently a full professor at the University Henri Poincare, Nancy, France and leader of the ECOO research team at the Laboratoty of Research in Computer Sciences and its Applications (LORIA), a joint venture between INRIA, CNRS and the universities of Nancy, He received a Ph.D in Computer Sciences from University of Nancy I in 1981.

His research interests include software tools for cooperative applications and software processes, advanced transaction models and distributed persistent workspaces.

Manuel Munier is a Ph.D. candidate at the University of Nancy I and member of the ECOO team at LORIA.

His research interests include distributed cooperative applications, advanced transaction models and distributed concurrency control schemes.

Samir Tata is a Ph.D. candidate at the University of Nancy I and a member of the ECOO team at LORIA. He holds an engineering degree from the university of Tunis.

His research interests include formal tools for interaction specification in distributed cooperative applications, advanced transaction models and non-standard concurrency control schemes.