COOPERATION MODELS IN CO-DESIGN

Khalid Benali, Manuel Munier, Claude Godart
UMR n°7503 LORIA – Université Nancy 2 – Université Henri Poincaré
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex - FRANCE
Tel: +33 (0)383.59.30.98 Fax: +33 (0)383.41.30.79 E-mail: benali@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 concurrent engineering enterprises which are created 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 introduces a flexible approach to build cooperation support software by assembling basic generic cooperation bricks. It shows how such a cooperation policy is implemented in a secure way thanks to a flexible transaction model. Our approach is illustrated through an example taken in the AEC domain.

Key words: cooperation, cooperation mode, project-enterprise, virtual enterprise, AEC application

1 INTRODUCTION

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 concurrent engineering enterprises which are created by aggregating several partners around a project and for the duration of this project. AEC is a good representative of such short-lived enterprises: it implies a lot of partners (architect, research consultant, control office, building firm, electrician, carpenter,...) which build an enterprise for the duration of the building work. Project-enterprises require new technologies to organize their short duration networks. Two main requirements characterize the design of this technology:

- as partners can be small enterprises and have a small computer infrastructure, technologies must be easy to understand and to use, and must not disrupt the habits of users,
- for the same reasons, the technology must allow to build safe cooperative applications: the quality of the service is a central point.

With these requirements in mind, we have developed an approach which allows to develop cooperative applications by assembling basic cooperation bricks. To assume acceptance of assembling, protocols of cooperation can be negotiated between partners. To assume safety of assembling and execution, each concurrent engineering activity executes as a transaction, i.e. in a frame which allows to assert safety properties on executions.

This paper illustrates our approach. In a first time, it begins with the presentation of some relative works. In a second time, it describes, through an example taken in the AEC domain, how

our environment allows to build a cooperative application. Then, it gives the principle of our implementation. Finally, it concludes by generalizing the principles developed previously to better cover the characteristics of a lot of concurrent engineering applications.

2 COOPERATIVE APPLICATIONS CONTEXT

Due In order to coordinate concurrent updates performed by activities on a distributed system, we could use either *configuration management* tools like Continuus [1], ClearCase [2,3], Adèle [4], or database systems using a transactional approach [5,6,7,8] to allow concurrent access to shared data by several activities.

Beside configuration management tools and transactional systems which focus on the control of resource updates, CSCW (Computer Supported Cooperative Work) tools are concerned by social aspects of the cooperation. Their main issues, which we will not discuss in this paper, are group management, group awareness or technologies for communication like net-meeting.

Most distributed systems are based on a client/server architecture in which, though single activities may be executed at geographically distributed nodes, the knowledge about the processes being executed 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, the main drawback is that clients have to be connected to this server at all times. Obviously, in the case of project-enterprises this is unacceptable because of:

- **distribution**: most configuration management tools are based on a centralized architecture. Some of them, like Continuus or ClearCase (with the MultiSite extension), can manage several repositories.
- **autonomy**: a workspace is the activity view of the repository presented as files and directories. This allow legacy applications to be used on data managed by the system. Nevertheless, workspaces are generally stored on the server, so activities have to be connected at all times. Using Continuus, an activity can run remotely (either connected or disconnected) by creating a physical workspace. However, to be able to create a new version of data it modified, this activity will have to reconnect to the server. So activities are not autonomous in regards to the repository.
- **cooperation**: configuration management tools are mainly focused on concurrent accesses to shared data (i.e. version and configuration management). Most of them do not control exchanges between activities. When provided, this control is delegated to a *software process management* layer. However, these software process tools (like workflow for instance) generally try to design the whole system with all its activities, data and interactions. That becomes rapidly a very complex task.

This paper introduces our approach to build cooperative applications. Our objective is to provide a framework to coordinate data exchanges between these activities while ensuring they are distributed and autonomous. So, unlike configuration management tools or transactional systems, we want to avoid activity dependency towards any kind of server, neither for data storage nor for interaction control.

3 AN APPROACH TO BUILD COOPERATIVE APPLICATIONS

3.1 A Project-Enterprise Example in the Domain of AEC

Due To illustrate our approach, we reuse the example developed in [9] (which is derived from the example presented in [10]). It consists in designing a one-storey apartment containing a living room with a glass wall. Four kinds of designers cooperate to achieve this work:

- the **architect**: his activity is to design and represent the apartment spatial organization with its walls, windows,... To construct his plan, the architect only takes care of volumes, spaces and luminosity of the apartment.
- the **structural engineer**: his activity consists in specifying the structural elements of the apartment. Such elements (cross walls, beams,...) will be chosen to respect, as far as possible, the choices made by the architect and the overall harmony of the building. Such an activity leads to modify the plan provided by the architect.
- the **HVAC engineer**: he will intervene to change the glass wall according to the climate and the apartment exposure.
- the **town planner**: he controls the architect's and gives advice in return. The architect has to consider this advice and possibly to modify his plan according to it. The town planner has to validate the final plan. The town planner only takes care of town planning aspects of the apartment.

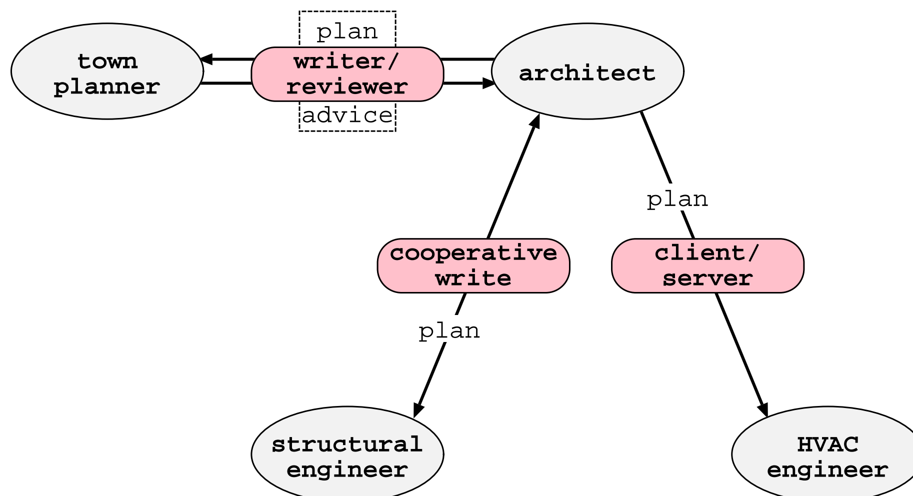

Figure 1: Document Exchanges and Protocols

These activities operate on various documents they will share: the plan and the advice of the town planner. All these exchanges between activities will not be driven by the same set of rules. If in some cases we will encourage activities to cooperate, in some others we will put some constraints. For example, the architect and the town planner share both the plan and the advice, but in a read-only mode. It is obvious that the architect will not be allowed to modify the advice delivered by the town planner, and vice-versa. Protocols used to control exchanges between the various activities are presented below and illustrated in figure 1:

- **client/server**: the architect (the server) provides several versions of the plan to the HVAC engineer (the client). Thus exchanges occur from the architect to the HVAC engineer.

- **writer/reviewer**: the town planner reads (but does not modify) the plan provided by the architect. Then he transmits his advice to the architect that reads (but does not modify) this document and modifies his plan according to it.
- **cooperative write**: both the architect and the structural engineer work on the same plan. During the design activity, they will possibly modify it at the same time, so they will have to merge changes made by the other in order to produce only one version of the plan on which they will agree at the end of the activity.

The first step to build this distributed application is to define the various activities of the system. In our case we use an activity to represent each actor. As our main objective is to make these activities as autonomous as possible, each of them will have its own repository to store documents is uses. Therefore cooperation between activities will occur by exchanging documents between their repositories. But, as we already said, these exchanges have to be controlled by a protocol. So when an activity want to communicate with another one, these activities begin by negotiating a cooperation protocol. This negotiation will ensure, at least, that the protocol one activity wants to use is known by the other. Then, all exchanges between these two will be controlled by this protocol. Thus each activity will have a cooperation table which purpose will be to show which protocol to use to communicate with a given activity. 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. So the cooperation table of an activity should not only store the protocol to be used to cooperate with others activities, but the role fulfilled by this activity too. Such a cooperation table for the architect activity is depicted in figure 2.

Activity	Remote Activity	Document	Protocol	Role
Architect	Structural Engineer	Plan	Cooperative Write	
Architect	HVAC Engineer	Plan	Client/Server	Client
Architect	Town Planner	Plan	Writer/Reviewer	Writer
Architect	Town Planner	Advice	Writer/Reviewer	Reviewer

Figure 2: Cooperation Table for Architect Activity

3.2 Interactions Between Activities

The coordination of the exchanges between two activities is performed by a cooperation protocol. We already defined three kinds of protocols:

- the **client/server** paradigm: one actor (the client) can work on preliminary versions of a document produced by another actor (the server). The only compulsory rules are: the server must produce the final version of each document it has produced in a preliminary version, and the client must take into account the last version produced by the server for each document it has read in a preliminary version. This paradigm permits some cooperative work which enhances productivity. For example, it allows to start the HVAC engineer activity before the end of the architect activity and ensures the HVAC engineer will read the final version of the plan.

  As the HVAC engineer has a copy of the plan in his own repository, he is free to modify this copy if he wants to. But the client/server protocol ensures that he will never be able to communicate these modifications to the architect.

- the **cooperative write** paradigm: two actors can modify at the « same time » the same document. Actually, each of them modifies the copy of this document he owns in its repository. They have to follow some rules: to be aware of each other work (by exchanging preliminary versions of this document) and to converge towards a same view of this document (i.e. they have to agree on the same final version of this document).

  In our example, the structural engineer can start his activity as soon as possible and then provide the architect a more accurate vision of the actual volumes of the plan final version. To do that, the two actors must be allowed to write at the "same time" a common plan.

- the **writer/reviewer** paradigm: this third form of cooperation corresponds to the case in which an actor produces a document under the control of another. As an example, the architect is controlled by the town planner who gives advice and validates the final plan. In this mode of cooperation, the architect (the writer) produces different successive versions of the plan it has to produce, especially to take into account the review of the town planner. This review has for objective to enforce the respect of some rules by the architect. As an example, he can react on the first version of the plan and ask for the surface of the wall glass to be reduced or the type of the woodwork to be changed. On a following version, he can react on the global look of the building. We say that the interactions between the architect and the town planner follow the writer/reviewer paradigm. The writer is the architect, and the reviewer is the town planner.

  This paradigm ensures that the final plan produced by the architect will be reviewed by the town planner, and that the architect will read the last document produced by the town planner. Thus the architect and the town planner have to agree to the same final version of the plan.

From the user point of view, the main advantage of our approach is that we define cooperation protocols to control document exchanges between activities and not to control activities themselves. Therefore, an activity is fully independent of other activities for the task it performs. This means that each actor is free to work like he wants, as long as his document exchanges with other actors are correct.

3.3 Interactions Between Protocols

When two activities need to cooperate, they negotiate a protocol to control their document exchanges between their respective repositories. So, an activity can communicate with many other activities using a different protocol with each of them. Consequently, a same document can be shared in many ways (i.e. using various protocols). For example, the architect share the plan with the structural engineer using the cooperative write paradigm, with the HVAC engineer using the client/server paradigm, and with the town planner using the writer/reviewer paradigm. So, it is at the document level that protocols will interact (and possibly conflict).

For the architect's activity, the three protocols will interact to exchange different versions of the plan without conflicts. Actually, only the activities of the architect and structural-engineer can modify the plan. Nevertheless they negotiated a cooperative write protocol, purpose of which is precisely to coordinate their concurrent writes. The HVAC engineer and the town planner only read the architect's and thus can not conflict with the architect.

Now, let's suppose the town planner asks a fireman for fire security aspects of the apartment (figure 3). As they work together on the plan provided by the architect, they negotiate a cooperative

write protocol. For example, to allow quicker access, the fireman puts an emergency exit on the main street, but for visual aspect the town planner decides to put this exit on the back side of the building. When they agree, the town planner writes his advice the architect, showing modifications that have to de done on the plan.

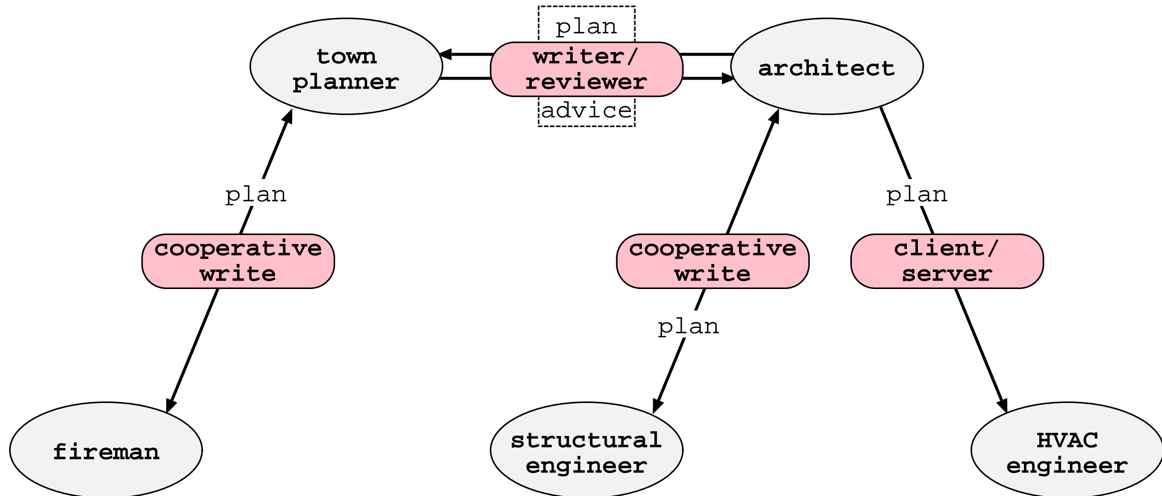

Figure 3: Protocol Interactions

Thus, the town planner shares the plan with both the architect as client in the client/server paradigm, and the fireman in a cooperative write mode. Conflicts between these two protocols may occur at the level of this document. What happens if the town planner decides to modify the plan? Even if the cooperative write protocol negotiated with the fireman allow this modification, the town planner is client in his client/server relationship with the architect and this means that he is not supposed to modify the plan. Beyond the semantics of this conflict, our concern is to define when it should be detected and how it could be resolved.

As our main objective is to make activities as autonomous as possible, each of them has to be responsible for conflicts occurring in its own repository. To detect conflicts between protocols, an activity can proceed in two way: during the negotiation (the activity could test if this new protocol is « compatible » with all protocols already in its cooperation table) or at run time:

- during the **negotiation**: the activity has to be able to test if the specifications of two protocols are « compatible ». This kind of problem can not only be complex, but it can restrict the cooperation between activities. For example, if the town planner uses this solution, it will not be able to choose a protocol to cooperate with the fireman because it can not modify the plan provided by the architect.
- at **run time**: during the negotiation, activities do not care for compatibility between protocols. Conflicts will be detected when they occur effectively, i.e. when an activity try to exchange some documents with another activity. Let an activity A share the same document with various activities B_1..Bn using the protocols $P_1..P_n$. It can proceed in two way:
  - each exchange between A and B_i is controlled by all the protocols $P_1..P_n$. A conflict will be immediately detected and the faulty exchange will be refused. In such a way we always ensure data integrity. But if such an exchange initiated by B_i is refused, this means the activity B_i could be disturbed by a protocol different from the one negotiated with A.

- an exchange between A and B_i is controlled only by the protocol P_i. So exchanges between A and B_i strictly respect the protocol negotiated between A and B_i. If such an exchange creates a conflict for a protocol P_j ($j \neq i$), this conflict will be detected when the activity A will try a new exchange with the activity B_j. As we can see, the activity A is the only one to be aware of this conflict.

Whatever solution we use to detect conflicts between protocols, only the activity in which a conflict occurs is responsible for its resolution (the town planner in our example). The main difference between these three methods is the disruption of such a conflict on the other activities.

Two activities can cooperate by exchanging some documents at run time. In order to coordinate their communications, these activities must begin by negotiating a cooperation protocol. Then, the system will ensure that all exchanges between these two activities will respect this protocol, while keeping activities independent of one another: in case of protocol violation, only the faulty exchange is refused. There is no constraint on the tasks these activities perform. However, an activity cooperating with several other activities, possibly using various protocols, could end in a situation in which all the exchanges it will try will break at least one protocol and so will be refused. Thus we should provide not only tools to define protocols and to detect conflicts between these protocols, but mechanisms for conflict management too:

- **conflict notification**: first, the user (i.e. the actor driving the activity) should be noticed when a conflict occurs. However, the system should not only notify that an exchange operation was refused, but should also detail the reasons it was refused.

  As an example, when the town planner tries to modify the plan for the first time, he should be noticed that he will not be able to communicate his modifications to the architect (as defined by the writer/reviewer relationship, he can only « read », i.e. import in its repository, the architect's plan).

- **protocol renegotiation**: such a deadlock situation could be due to the protocol negotiated between two activities and defined too strictly. So activities should be able to renegotiate this protocol.

  In our example, if the town planner modifies the architect's plan, then he will not be able to communicate his advice to the architect. Actually, the writer/reviewer protocol considers the plan and the advice as a single logical document because they are interdependent (the final advice must be produced with the final plan version, and the final plan must take into account the final advice). So, when the town planner wants to send his advice to the architect, the protocol sends in fact the logical document in order to preserve document consistency. But the town planner modified the plan. As it is not allowed by the writer/reviewer protocol, this exchange will be refused. A solution will be that the town planner and the architect can be able to renegotiate this protocol and decide to cooperate using the cooperative write paradigm for instance.

- **version management**: another solution is to be able to « undo » some work. As for configuration management tools, this means that an activity can develop, in its repository, its own version branch for documents it uses. At the end of this activity, it should be able to produce its results according to any versions of these documents. For instance, the activity could come back to the document version it imported. This means that the modifications this activity did on these documents will not be visible to other activities.

As an example, let's suppose that the renegotiation presented above fails. When the town planner's advice will be written, the only solution is to « undo » all modifications done on the

architect's plan. Then, the town planner can release his advice according to the plan version it imported from the architect.

4 FRAMEWORK OF OUR COOPERATION SUPPORT

Our main objective is to make activities as autonomous as possible, both towards the network (reachability, bandwidth,...) and the other activities (availability, confidence,...). A first step was to assign a local repository to each activity. Thus an activity can store locally a copy of documents it accesses to. The second step is the distribution of the data interchange control. The main idea is that each activity has to be responsible for data it owns in its repository. Moreover, to ensure the consistency of its data an activity should not be dependent of data owned by another activity. This means that an activity should be able to manage its data using only local information. That's the reason why, if two activities want to exchange a document, both of them verify that this exchange respects the protocol they negotiated according to their own data. If at least one of them detects a conflict, then this exchange is aborted.

4.1 Design

Now, we present the framework we designed to *coordinate*, using various *protocols*, *data* exchanges between widely distributed activities. We must define how an activity stores its data locally, what's a cooperation protocol, and how activities control data exchanges using these protocols.

Remember that we don't want to disrupt the habits of actors. From the user point of view, this means that we should be able to work on shared data using legacy tools, which generally can access to files and directories only. So we split the local repository of an activity in two parts: a public one, called *cooperation space*, in which the activity will store versions and configurations of data it uses, and a private which is the *workspace* itself (i.e. where legacy tools will store their files). So, when a user imports a document from another activity, this document is stored in its cooperation space. To be able to access it through its legacy tools, he must extract this document to its workspace to obtain a file his applications can read and write. When he thinks he has reached a consistent state, he publishes the result of his work (i.e. the file). This operation updates the document in its cooperation space, i.e. creates a new version of this document which is now available to other activities. As depicted in figure 4, an activity is made of four main parts:

- a **Cooperation Space** which is the local repository of the activity. It is in charge of creating/deleting resources, checking in/out documents between the cooperation space and the workspace, and managing versions and configurations for local resources.
- a **Workspace** which shows resources as files and repositories, so actors can use their legacy applications.
- a **Protocol** which is based on the cooperation table of the activity. It ensures that a sequence of exchanges between two activities is correct (according to the cooperation paradigm they negotiated).
- a **Coordinator** which is the activity interface for other activities. Thus all exchanges between activities have to be done through their coordinators. This prevents activities to perform, directly between their cooperation spaces, data exchanges which are not controlled by any cooperation mode. Moreover, it allows the cooperation space and the protocol to be defined independently.

This framework can be considered to be a flexible transaction model, i.e. cooperating activities can be seen as being concurrent transactions. But unlike classical ACID (Atomic, Consistent, Isolated, Durable) transactions, these activities (eventually distributed over a Wide Area Network) can exchange intermediate results. So we break down the isolation property. To coordinate these data exchanges between activities, we define various protocols. Using a transactional approach, we base these protocols on correctness criteria which ensure consistency properties in a cooperative context. With regard to the three cooperation paradigms we described above (client/server, writer/reviewer, cooperative write), they respect our correctness criterion called COO-serializability.

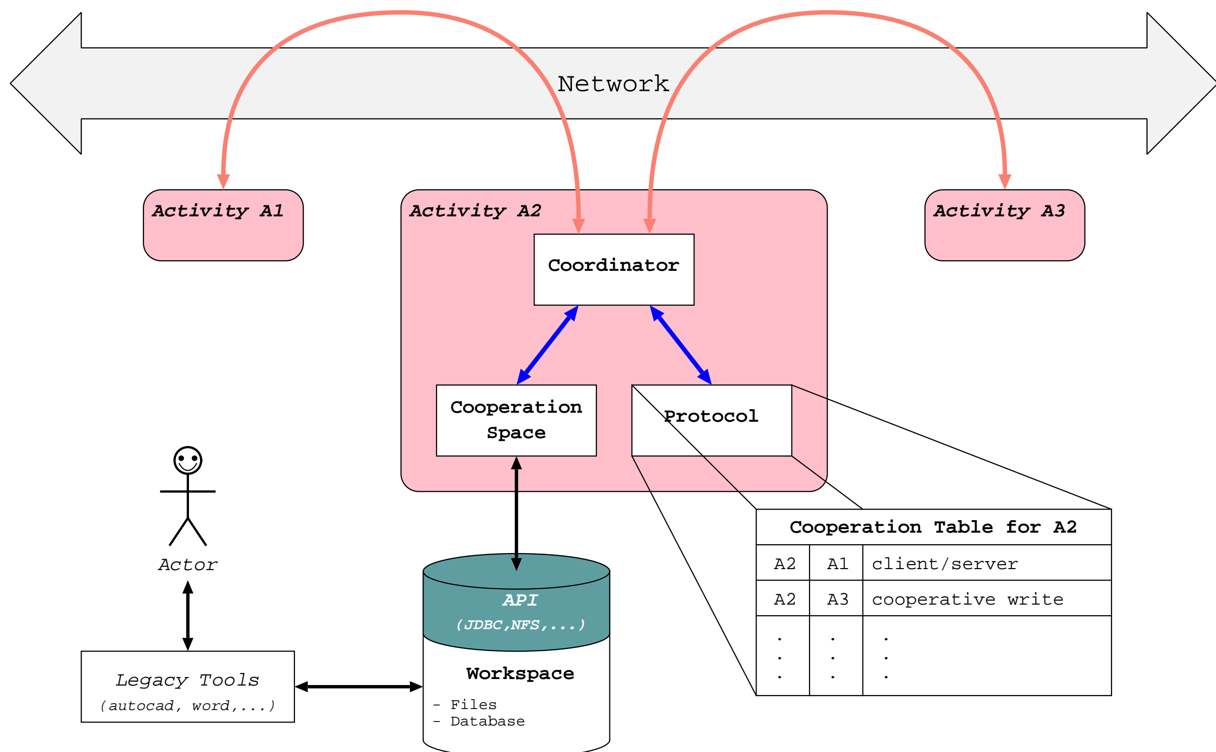

Figure 4: Framework Architecture

4.2 A Cooperation Service

Our objective is to provide a toolkit to build distributed cooperative applications. We don't want to develop yet another proprietary system, but to define a set of cooperation services instead. In other words, we intend to describe what are the basic services needed to build distributed cooperative applications. We identified four main services: a cooperation space service, a workspace service, a protocol service, and a coordination service.

Obviously, the core of our framework is the service related to protocols. As mentioned above, an activity protocol is defined by a cooperation table which shows the protocol to be used for communicating with a given activity. So cooperation paradigms can be viewed as basic generic cooperation bricks we compose to specify the protocol for an activity. This should provide the following functionalities:

- definition of basic generic cooperation bricks (according to some correctness criteria)
- protocol negotiation between activities
- cooperation table management
- control of data exchanges
- conflict detection and notification (this includes the ability to find and explain what caused the conflict)

To implement our cooperation services, we chose to conform to CORBA [11], which has became a standard in the distributed object community. From the developer point of view, this means that all the aspects related to object distribution (possibly over a heterogeneous network of stations) are taken into account by the ORB (Object Request Broker). Moreover, our services can use and be used by other CORBA services. So they easily can be integrated into existing CORBA distributed systems, which is a key feature of our framework for cooperation support.

A prototype of our framework is being developed using Java [12] as programming language and JacORB [13] as Object Request Broker (which is itself coded in Java). Thus we are independent from the underlaying operating system (Unix, Windows 95/NT,...).

5 CONCLUSION

This paper has introduced our approach to build cooperative applications. In a first time, a map of the interactions between the activities of the application is drawn; in a second time, the application is generated thanks to a set of cooperation services. If this approach is ambitious and difficult to achieve in its globality, our experience [14,15,16] demonstrates that it is feasible with a limited set of cooperation protocols (those introduced in the motivating).

Our objective now is to extend the approach by incorporating new cooperation behaviors. We are generalizing the principles developed previously to define a more large set of basic cooperation bricks in order to better cover the characteristics of a lot of concurrent engineering applications [17]. The more difficult problem we tackle is to find out new correctness properties to validate the integration of the new protocols corresponding to the new cooperation behaviors.

REFERENCES

[1] Continuus/CM: Change Management for Software Development. http://www.continuus.com/developers/ developersACED.html

[2] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In Jacky Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 194-214. Springer-Verlag, October 1995.

[3] Atria Software Inc. ClearCase product summary. Technical report, Atria Software Inc., 24 Prime park Way, Natick, Massachusetts 01760, 1994.

[4] N. Belkhatir and J. Estublier. ADELES-TEMPO: An Environment to Support Process Modelling and Enaction. In J. Kramer A. Finkelstein and B. Nuseibeh, editors, *Software Process Modeling and Technology*. Research Study Press, 1994.

[5] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing surveys*, 13(2):186-221, 6 1981.

[6] J. Elliot Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, 1981.

[7] Goodman N., Beeri C., Bernstein P.A. A model of concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230-269, 1989.

[8] P.K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models. 19(3):451-491, 1994.

[9] K. Benali, J.C. Bignon, C. Godart, and G. Halin. Cooperation models in co-design: application to architectural design. *In Proceedings of ICD&DSS*, Maastricht, Holland, July 1998.

[10] M.A. Rosenman and J.S. Gero. Modeling multiple views of design objects in a collaborative CAD environment. *Computer-Aided Design*, 2893):193-205, 1996.

[11] OMG. The Common Object Request Broker Architecture and Specification. Technical Report 2.0, Object Management Group, 1995.

[12] SUN. The Java(tm) language: an overview. ftp://ftp.javasoft.com/docs/java-overview.ps, 1994-1995.

[13] Gerald Brose. A Java Object Request Broker. Technical Report B 97-2, University of Berlin, 1997.

[14] 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.

[15] 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.

[16] P. Molli. COO-Transactions: Enhancing Long Transaction Model with Cooperation. In *7th Software Configuration Management Workshop (SCM7), LNCS*, Boston, USA, May 1997.

[17] K. Benali, G. Canals, C. Godart, and S. Tata. An Approach for Developing Cooperation in Project-Enterprises. *In Proceedings of the 3rd International Conference on the Design of Cooperative Systems,* Cannes, France, May 1998.