DisCOO, a really distributed system for cooperation

Manuel Munier, Khalid Benali, Claude Godart

DisCOO, a really distributed system for cooperation

Manuel Munier — Khalid Benali — Claude Godart

LORIA - INRIA - UMR n^{o}7503
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex - FRANCE
munier@marsan.univ-pau.fr
{benali,godart}@loria.fr

ABSTRACT. *The design or realization of any consequent project implies the involvement of several people, and sometimes even of several teams or companies. In addition to the fact that they all work on the same project, the various actors cooperate and collaborate. Indeed the realization of any project is not simply a succession of steps carried out by one actor at a time, but is the result of cooperating actors aiming at the realization of a common goal. It is the concept of concurrent engineering. Concurrent engineering requires cooperation between the various actors and exchange of the data produced by each one of them. More and more are companies using the Internet for data exchanges, the various actors are no longer forced to work in the same geographical place. One speaks then about virtual enterprise or project-enterprise if joint work lasts as long as a project. However, it is not enough just to exchange data for working together, it is also necessary to control and manage these exchanges. Collaboration involving some concurence in the work of several actors, the production of various versions of the exchanged documents implies a control of these exchanges. The objective of this article is the description of a really distributed system for cooperation. The overall philosophy of our system is the distribution of the exchanges control and the access to the exchanged data in a standard way. The paper's kernel is the formalization of the exchanges control. Adopting a transactional approach for the realization of our system for cooperation, we formalize our cooperative transactional system and our distributed criterion of correction (DisCOO-serializability) in ACTA. We use a support example to illustrate our discourse and to apply this formalization to a concrete case. We supplement description by the presentation of the DisCOO prototype (realized in Java and Prolog upon an ORB) which implements our system.*

KEYWORDS: *Cooperative systems, advanced transaction models, distributed systems*

1. Introduction

The design or the realization of any consequent project implies the involvement of several people, and probably even of several teams or companies. Different and various competencies are required. If we consider the AEC domain (Architecture, Engineering, and Construction) which will provide us with the support example for illustrating our approach, competencies to build a simple house range from the design of volumes by the architect to the specification of the structural elements by the structural engineer and encompass skills for air-conditioning realization by the HVAC engineer.

To say that these various building trade actors work on the same project is not enough. In addition to the fact that they all work, these various actors cooperate and collaborate. Indeed the realization of any project is not simply a succession of steps carried out by one actor at a time but is the result of cooperating actors aiming at the realization of a common goal. It is the concept of concurrent engineering which allows the various actors to work in synergy and permits a reduction of the time-limit for delivery and the best use of the skills of each actor.

This concurrent engineering requires cooperation between the various actors and an exchange of the data produced by each one of them. The data exchange has existed for a long time in simple forms (mail, fax, exchange of floppy disks,. . .) in business. In the current context, with the democratization of the Internet, more and more companies use this medium for data exchanges. The various actors are no longer forced to work in the same geographical place. One speaks then about virtual enterprise or project-enterprise if joint work lasts as long as a project (for example a building achievement in AEC).

However, these uncontrolled exchanges do not allow a real synergy between the various actors. It is not enough just to exchange data, it is also necessary to control and manage these exchanges. Back to our support example, sending a version of the architect's plan to the HVAC engineer is not enough to solve the problems instigated by their collaboration. Indeed the plan sent by the architect at a given moment corresponds to one version. Cooperation implying some concurrence in the work of the two actors, the production of various versions of the exchanged documents implies a control of the exchanges. This exchange control is currently possible in cooperative systems, but generally this control is centralized. In the AEC domain for instance, systems such as "cell of synthesis" or "electronic store of plans" provide actors with a centralized control and are limited to specific access rights management.

The objective of this paper is the description of a really distributed system for cooperation. We start by presenting the various categories of environments developed to assist the cooperation (section 2). Then we present in section 3 the support example which will enable us to clarify our approach. Sections 4 and 5 show the overall philosophy of our system, namely, the distribution of the exchange control and the access to the exchanged data in a standard way. Section 6 constitutes the paper's kernel and presents the formalization of the exchange control: we start by clarifying our choice of a transactional approach for the realization of our system for cooperation, then we gradually formalize the various elements of our cooperative transactional system (local history, local bases, transfer operation,. . .) and finally we describe our distributed criterion of correction (*DisCOO*-serialisability) in ACTA. Throughout this presentation, we use the support example to illustrate our discourse and to apply this formalization to a concrete case. Section 7 allows us to present the implementation of our system thanks to the *DisCOO* prototype realized in Java and Prolog on top of an ORB. We finally conclude by presenting the results of our approach and the possible perspectives for such an approach.

2. Existing environments to assist the cooperation

In the case of a relatively complex application, there is no actor who has the integral knowledge of the overall activity carried out and who may control its achievement. It is thus extremely difficult for any one actor to "manually" foresee all the consequences of a modification made to a document of the application. All these reasons imply that is essential to use environments providing sophisticated mechanisms for coordination and communication, thus allowing notification and propagation of the changes to the concerned actors. This is done with the aim of coordinating the actors and then of reducing the impact of a document modification on the overall activity.

These environments can be classified in four categories according to the way they tackle cooperation problems. First, we have the **configuration management systems** whose objective is to manage the versions and the successive configurations of the various shared data (data coherency). **Process centered environments** aim to control the successive states of the shared data and/or the sequencing of activities. In the data bases field, **transaction systems** ensure that the parallel execution of several activities (encapsulated within transactions) does not introduce inconsistency into their results. Finally **CSCW tools** ("Computer Supported Cooperative Work") are oriented towards communication and human relations of the cooperation.

2.1. *Configuration management systems*

In order to control the concurrent updates carried out by the various activities of a distributed system, it is possible to use a configuration management tool (RCS [TIC 89], ClearCase [ATR 94], Continuus, Adèle [BEL 94]). Its role is to ensure storage of the shared data, called resources, while keeping a trace of their evolution (generally through their successive versions) and controlling the concurrent accesses to the data made by the activities. For example, if two activities modify the same resource in parallel, each one of them will develop a branch of versions, starting from an initial version of this resource. In this way, each activity works on its own copy of the resource, without being disturbed by the modifications made by the other activity. When they both complete their work, an activity (possibly a third one) will be charged to merge these two branches in order to produce only one new version, meaning that the modifications made by one activity will not crush the other activity's modifications.

The role of the configuration management system will be then to memorize the fact that this new version is derived from the two preceding ones.

However, the majority of existing configuration management systems are based on a client/server architecture (centralized repository, possibly replicated and/or splited on several hosts). Thus they do not correctly answer the requirement of distribution and autonomy of the activities. Moreover, the configuration management systems are primarily concerned with the problems of concurrent accesses to a common repository: management of the versions and the configurations of shared resources. They do not define any control for the cooperation based on the exchanges among activities.

2.2. *Process centered environments*

Distinct from the configuration management tools, the process management tools are mainly directed towards the description of the correct executions in terms of successive states of a resource or correct sequence of the various activities: workflow models [COA 97, ALO 96], contract model [WAC 92], event/condition/action rules in MARVEL [BAR 92a, BAR 92b] or Adèle-Tempo [BEL 94]. This approach generally requires to describe the complete application, i.e. taking care of all the activities and all the resources. If we add the problems implied by the synchronization of the distributed activities, the obtained model becomes complex because of the inherent complexity of the working context to model.

Contrary to the existing process management tools, our objective is not to describe how the activities must work to be able to cooperate, but simply to define how the exchanges between these activities must be done. We want to enforce controls on the exchanges of results between activities, but not on the activities themselves nor on the way they produce these results.

2.3. *CSCW tools*

The Computer Supported Cooperative Work (CSCW) aims to allow groups of users to collaborate for achievement of common goals thanks to collaborative systems or groupwares. However, contrary to the configuration management systems, the process management tools or the transaction systems, a CSCW environment is not solely directed towards the maintenance of the shared objects coherency (management of the concurrent accesses). Such an environment also takes into account "human" aspects of the cooperation such as the management of a group of people, the notification mechanisms, the communication techniques (electronic mail, videoconference,. . .). BSCW [BEN 97a, BEN 97b] (sharing of information through centralized repository), Wiki[1] (collective authoring of documents via the Web) and Microsoft NetMeeting (vi-

[1] Wiki: `http://wiki.lri.fr:8080/scoop/scoop.wiki`

deo / audio conference, sharing of applications, white board, synchronous forums of discussion) are examples of such groupwares.

For the consistency control, groupware applications use the mechanisms developed for the distributed systems or the configuration management systems: locks on the objects, token passing (or "turn taking"), dependencies detection (conflicts are solved by the users). These techniques aim to ensure the coherence of the shared objects and not to coordinate the activities which cooperate.

2.4. *Transactional systems*

Within a transaction processing system [AGR 90, BER 97], each activity is encapsulated in a transaction that hides problems of concurrent accesses to shared objects. A transaction is then represented by the sequence of operations (eg: read, write) invoked on shared objects. A transaction is the unit of control: either the transaction completes and all its changes to the state of objects happen (the transaction commits), or the transaction is aborted and appears as having no effects on objects (the transaction is rolled back). This "all or nothing" property, also called atomicity, preserves the consistency of distributed systems despite concurrent accesses and failures.

The main idea behind transactional systems is to ensure that if all transactions individually execute correctly, then their interleaving (due to their concurrent accesses on shared objects) does not introduce inconsistencies in the state of shared objects. It is the purpose of a correctness criterion, which defines on the execution (history of the operations invoked by transactions) some properties that characterize "correct" executions. Within the context of administration and banking applications, a well known correctness criterion is the "serializability". It considers that the concurrent execution of some transactions is correct if this execution produces the same results (in terms of states on shared objects) than a serial execution of these transactions.

However, this correctness criterion is not suitable for us because it forces isolation of transactions: intermediate states of transactions (i.e. values of shared objects) cannot be visible for the other transactions. Thus, the serializability does not support our need of cooperation among transactions in terms of intermediate results exchanges during their execution. In the literature, new correctness criteria and transaction models were proposed to relax the isolation property between transactions.

Nested transaction model: Within the nested transaction model introduced in [MOS 81], a nested transaction is a tree of transactions. Starting at the root, each transaction can create lower-level transactions (called subtransactions[2]), which are embedded in the sphere of control of the parent. Transactions at the leaf level are flat transactions, except that they lack the durability of non-nested flat transactions: the commit of a subtransaction will not take effect unless the parent transaction commits.

[2] A subtransaction begins after its parent transaction and ends before it.

By induction, therefore, any subtransaction can finally commit only if the root transaction (corresponding to the whole acivity) commits.

Multi-level transaction model: Multi-level transactions [BEE 88] are a generalized and more liberal version of nested transactions. They allow for the early commit of a subtransaction (also called pre-commit). They assume the existence of a compensating transaction (it can be another nested or multi-level transaction), which can semantically reverse what the original subtransaction has done in case the parent transaction decides (or is forced) to roll back. Whereas nested transactions are simply an execution model that places no restriction on what the subtransactions do and which object they manipulate, multi-level transactions, however, require a hierarchy of abstract data types. First, the entire system consists of a strict hierarchy of objects with their associated operations (abstraction hierarchy). Second, the objects of layer n are completely implemented by using operations of layer $n - 1$ (layered abstraction). Lastly, there are no shortcuts that allow layer n to access objects on a layer other than $n - 1$ (discipline).

The main difference between all other extended models and multi-level transactions is that they are the only ones that have all the ACID properties, because their scheme of layering object implementation makes it possible to protect updates at lower layers by isolating higher-layer objects. This achieves isolation for the root transaction, and thereby the possibility of executing the root transaction atomically. This guarantee does not hold for the other (not flat) transaction models.

Saga model: This model [GAR 87] allows to relax the isolation property for long lived transactions. A saga is a chain of ACID transactions, and this model uses the compensation idea from multi-level transactions to make the entire chain atomic. Rules for concurrency control are the following. First, sagas are not isolated one from one another. It means that a subtransaction of one saga can view intermediate results of another saga. Second, the commit of a subtransaction is not constrained by the commit of its parent saga. Lastly, when a subtransaction is rolled back, the whole saga is aborted (using compensating transactions to semantically undo results of committed subtransactions).

Cooperative transaction model: The common trait of all these models (flat, nested or multi-level transactions, sagas) is that they are founded on the notion of structural and dynamic dependencies. In other words, all the additional structure that distinguishes the different transaction models simply denotes different protocols for deciding if and when state transitions can be externalized (committed).

In the literature, there are suggestions for other transaction models that are much more complex. In particular, their structure is defined by more criteria than just controlling the commit of updates. As an example, cooperative transactions [NOD 92] allow for explicit interactions among collaborating users on shared (design) objects. At that level of cooperation, though, notions such as atomicity are not powerful enough to model the complex rules of state transitions the application wants to support. All this becomes very much dependent on the application semantics.

Atomic objects: Another approach to achieve transaction atomicity is to confine concurrency control and recovery mechanisms within the shared objects themselves. Such objects, called atomic objects [WEI 89, WEI 84], enhance their modularity since they can be designed and tested locally, and can increase transaction concurrency by providing appropriate mechanisms to their use and semantics. Then, some global serialization protocol ensures that all atomic objects manipulated by a transaction define the same serialization order for this transaction with regard to other transactions.

3. Presentation of the support example

To illustrate our approach, we reuse the example developed in [BIG 98, BEN 99] (which is derived from the example presented in [ROS 96]). It consists in designing a one-story apartment containing a living room with a glass wall. Several designers cooperate to achieve this work, thus forming a project-enterprise. These designers share three documents: the **plan** jointly authored by the architect and the structural engineer, the **advice** written by the town-planner and, to a lesser extent in this example, the **specification of the glass wall** defined by the HVAC engineer. Aiming to simplify, each partner will be represented by a single activity (figure 1).

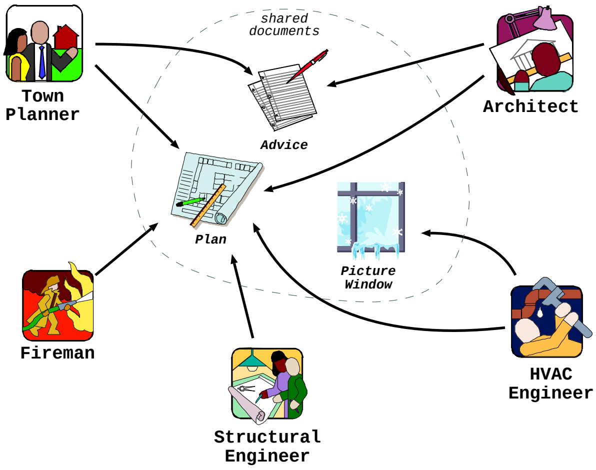


Figure 1. *Partners and shared documents*

– the **architect**: He has to design and represent the apartment's spatial organization with its walls, windows, and so on. To construct his plan, the architect takes care only 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, an so on) 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 modification of the plan provided by the architect.

– the **HVAC engineer**: He is normally in charge of the air-conditioning of the building (heating, heat insulation...) and all building fluid management. In our example, he will only intervene to specify the glass wall (materials, thickness) according to climate and apartment exposure.

– the **town planner**: He controls the architect and gives opinion or advice in return. The architect has to consider this advice and possibly modify his plan according to it. The town planner has to validate the final plan. The town planner takes care only of the town planning aspects of the apartment.

– the **fireman**: Before giving his opinion or advice, the town planner can consult a fireman to check if the apartment is in conformity with the fire emergency standards (number and location of the emergency exits, number and location of the trap doors of smoke clearing,. . .)

The various data exchanges between the partners are not constrained by the same rules. If it is desirable, in certain cases, to encourage cooperation among partners, in some other cases it can be necessary to restrain this cooperation. It is the case for example for the cooperation between the town planner and the architect: they share some documents, but only to read; no one is allowed to modify the documents of another. The cooperation rules negotiated to control these document exchanges between the various partners are presented in figure 2.

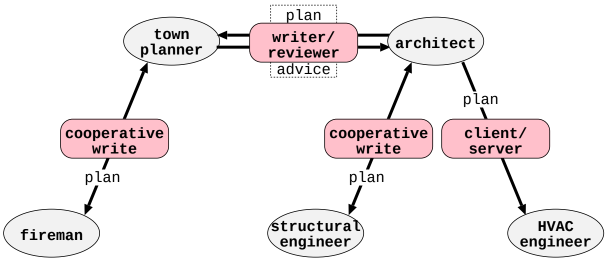


Figure 2. *Cooperation patterns negotiated for exchanging documents*

– **client/server**: The architect (the "server") provides various successive versions of the plan to the HVAC engineer (the "client"). Exchanges are only from the architect towards the HVAC engineer.

– **writer/reviewer**: The town planner (the "reviewer") reads, but does not modify the plan which is provided to him by the architect (the "writer"). He simply writes his opinion or advice he could transmit to the architect who can update his plan in return. This process can iterate.

– **cooperative write**: The architect and the structural engineer work togeher on the drawing of the plan (same logical object). During all the duration of the activity of design they modify it, possibly simultaneously, integrating the modifications made by one or the other. The goal is to produce a final version of this plan which satisfies both of them.

Our objective is to conceive an infrastructure **to coordinate**, via the use of various **cooperation patterns**, the **data exchanges** between various activities. In this aim, it is necessary to be able to store data on an activity level, to define protocols for exchange of data and to apply these protocols to control the interactions among the activities.

4. Control of the exchanges

When an activity wants to share a given object with another activity, a phase of negotiation starts to define the cooperation patterns which will control all exchanges concerning this object between these two activities. We have, at least, to ensure that the pattern that one activity wishes to use is known by the other one. The result of this negotiation is a **contract** agreed by the two activities. This contract sets the cooperation rules to respect for the sharing of the concerned object. For example $Contract[archi,hvac,\{plan\},client\_server]$ represents the agreed contract between the architect and the HVAC engineer to share the object `plan` according to the cooperation pattern "client/server". Thus, each activity of the system owns a **cooperation table** containing all the contracts signed by this activity.

activity	**partner**	**objects**	**pattern**	**role**
`architect`	`structural-engi.`	`{plan}`	cooperative write	∼
`architect`	`HVAC-engi.`	`{plan}`	client/server	server
`architect`	`town-planner`	`{{plan},{advice}}`	writer/reviewer	∼

activity	**partner**	**objects**	**pattern**	**role**
`HVAC-engi.`	`architect`	`{plan}`	client/server	client

Figure 3. *Cooperation tables for figure 2*

During a data exchange between two activities, each activity controls locally (i.e. using the information contained in its table of cooperation) that this exchange is correct according to the contract they negotiated. If a violation of the contract (or more exactly a violation of the cooperation pattern specified on the contract) is detected, this exchange is refused. Thus an activity encompasses two components dedicated to the control of its exchanges with the other activities: a **protocol** in charge of the management of the activity cooperation table; a **coordinator** controlling that all the accesses to its documents respect the protocol.

5. Data accesses

The local repository of an activity is composed of two parts: a public part, named **cooperation space**, and a private part, named **workspace**. The cooperation space contains object versions made public by the activity, i.e. versions that another activity can import. This cooperation space also contains the relations between these various versions. Data exchanges between activities will occur as transfer operatoins between their respective cooperation spaces. The workspace allows to present to the user the objects of the cooperation space in a form usable by its existing applications (`.DXF` files for Autocad or `.DOC` files for Word for example). It is the place where the applications will indeed handle the data.

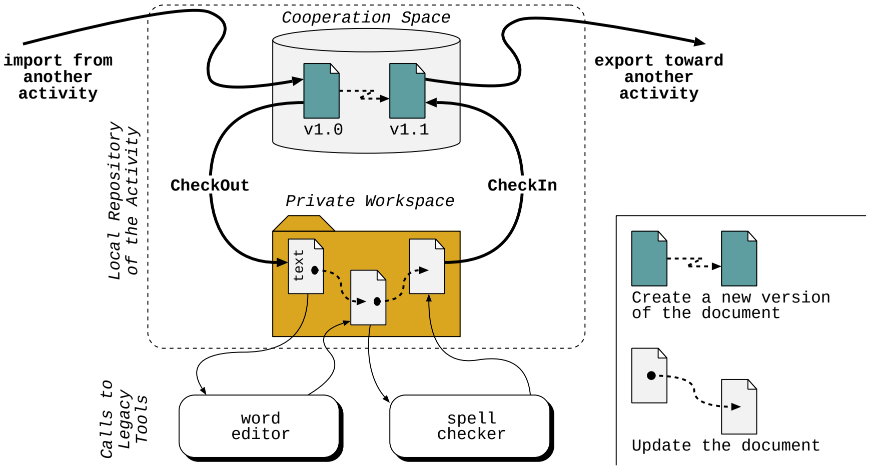


Figure 4. *Cooperation space and workspace*

Thus, the local repository of an activity is represented by two distinct components: a zone of exchange, the cooperation space, accessible to the other activities and in which are stored the shared objects; a private zone, the workspace, which provides the data on which the activity can work (use of existing or legacy tools) to achieve its task. The transfers between these two zones are carried out on the initiative of the activity (and thus of the user). In other words, it is the user who decides when he publishes his results (intermediate or final) as he decides when he integrates, in his workspace, the modifications made by the other activities on the shared objects (previously imported in his cooperation space). Our system and our approach having for objective to allow cooperation and collaborative work without changing the tools and the individual modes of production, we only manage the data in the cooperation space. The workspace is only considered as a destination for a transfer from the cooperation space or as a source of a publication into the cooperation space. From the point of view of the actors of the system, i.e. the users, it is necessary that they can use their actual applications (Autocad, Word, emacs, GCC, . . .) on the shared data. Thus, these data must be accessible in their native format, i.e. `.DXF` or `.DOC` files, or even in the form of tuples in a data base or object base in an object oriented environment (ex: CORBA).

As represented in figure 4, the user starts by importing the document (in the form of an object) from the cooperation space of another activity and stores it in his own cooperation space. In order to be able to handle it (in the form of a file for example) with his actual applications, he has to transfer (`Check_Out` operation) this document into his workspace. As the workspace is a private space, his updates are not visible by other activities. When he considers that his work is completed, he publishes (Check_In operation) a new version of this document, meaning an update of the document stored in his cooperation space. From this moment, this document (in the form of an object) can be imported by other activities.

6. Formalization of the control of the exchanges

Within such a cooperation context in distributed applications, it seems difficult for a programmer or a set of programmers to have a global view of the whole application and to program explicitly all the interactions between activities: it is necessary to release programmers from the burden of interaction programming. In other terms, it must be possible to program a large part of cooperating activities independently of each other: application programmers should be concerned with the behavior of each activity individually, not with the interactions with other activities. In relation to these remarks, we believe that a concurrency control approach is better adapted to our class of applications than a concurrent programming one. Thus, we base correctness of cooperative executions on a correctness criterion in the spirit of criteria defined for concurrency control purposes, i.e a criterion which is as far as possible not depending on the semantics of the application being synchronized [BER 81, RAM 96]. Another argument in favor of the concurrency control approach is that, on one hand, due to their uncertainty, it is not possible to assert correctness of executions of cooperative applications *a priori*; on the other hand, due to their long duration, it is not possible to do it *a posteriori*: it must be done incrementally.

6.1. *A transactional approach*

Following a transactional approach [BER 97] we view a cooperative application as a set of transactions accessing *at the same time* to a set of objects. Each activity of the application is encapsulated in a transaction that hides problems of concurrent accesses to shared objects. A transaction is then represented by the sequence of operations invoked on shared objects. However, the way of synchronizing these transactions is more complex than in classical transactional systems (administration, banking, ...) [GRA 93] in which all activities are mainly concurrent, that means they execute in isolation and are unaware of the others. In our case, we should speak rather about a *cooperation control* approach than a *concurrency control* approach.

Although existing transaction models and correctness criteria integrate some features needed for cooperation support, none of them simultaneously deal with our needs for **cooperation** (transactions are not isolated), for **distribution** (each transaction is provided with its own copy of objects it manipulates), and for **autonomy** (decentralized interaction control performed locally by transactions themsleves). Thus, in the vein of [ELM 92, JAJ 97], we work on the definition of a new extended transaction model that could support distributed cooperative applications by relaxing one (or several) ACID[3] properties ensured in classical transaction models [MUN 98, BEN 99].

6.2. *Local repositories, local histories, transfer operations*

The first step in defining our transaction model aims to deal with the distribution of the transactions (each activity of the system executes within a long-lived transaction). We provide each of them with a **local repository** to store its own copies of objects it uses. Objects are no longer kept in a single global repository accessed by all the transactions. So, unlike classical transaction models, a "logical" object in our model will possibly have several "physical instances" we have to distinguish. We also need to define the **local history** for a transaction, that means the system events which the transaction have to be aware of.

As each transaction owns copies of the object it uses, data exchanges among transactions will not be **implicit** (through concurrent accesses to a common repository), but **explicit** instead. Transactions will invoke **transfer operations** between their respective local repositories. The purpose of such an operation is to synchronize the value of two instances of a same logical object. Using transfer operations along with local repositories and histories notions, we can then define cooperation rules for an object shared by two transactions. Such a rule is only based on information that is local to each transaction, and aims to control their (transfer) operations.

Then, we will detail the impact of these changes on traditional "global" correctness criteria, i.e. correctness defined on the complete history of the whole system (cycle detection of dependencies between the activities, for instance). Our new advanced transaction model will be specified with these new notions in order to get "local" correctness criteria defined on local histories of transactions. The core idea is the following: if we ensure the correctness of the execution for each transaction with regard to all the transactions directly connected to it, this should ensure, implicitly, the correctness of the whole execution.

Transaction local repository: First, we need to formalize explicitly, within our transaction model, that we provide each transaction with its own local repository, instead of using a single (global) common repository. As we explained before, when several transactions share an object (simultaneous accesses), each one stores its own copy of this object in its own local repository. Thus, a single **logical object** will have several **instances** (one for each transaction it is used by), and we need to distinguish between them. These instances are not duplicates which are automatically synchronized by the underlying system, but true independent copies that transactions will synchronize on their own by explicitly exchanging values of these instances. Moreover, at some points during the execution, several copies of the same logical object could have different values.

[3] ACID: **A**tomicity, **C**onsistency, **I**solation, **D**urability

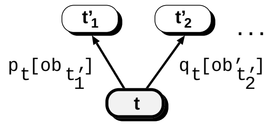
When a transaction t invokes an operation op on the object ob (classically expressed by $op_t[ob]$ in the ACTA formalism [CHR 94] to denote this event), we need to specify which copy of the object ob is concerned by this operation op. We use $op_t[ob_{t'}]$ to denote the invocation of the event $op_t[ob]$ on the copy of the object ob owned by the transaction t'. The set of all the objects ob_t is named the **local repository** of the transaction t.

For instance, when the HVAC engineer reads the plan produced by the architect in his/her local repository, this event corresponds to the $read_{hvac}[plan_{archi}]$ operation. To update his/her own copy of this plan, the HVAC engineer then performs a $write_{hvac}[plan_{hvac}]$ operation.

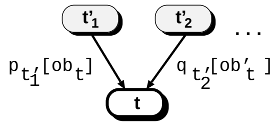
Transaction local history: Using this new syntax, we can express that a transaction can invoke an operation not only on objects within its own local repository, but also on objects stored in local repositories of the other transactions (eg: $op_{t_i}[ob_{t_j}]$ or $read_{hvac}[plan_{archi}]$). These last operations precisely allow us to formalize **interactions** among transactions, and especially data transfers between their respective local repositories.

We can now define the local history of a transaction t. It is the set of all the events that the transaction t has to be aware of. The ACTA formalism [CHR 94] is based on two event types: significant events (`Begin`, `Commit`, `Abort`, ...) and object events (invocations of operations on objects). Here, we are mainly concerned with object events to control the view of a transaction. In ACTA, the view of transaction t, represented by $View_t$, defines the objects and their states that are visible to transaction t. In other words, the view of a transaction identifies the operations the effects of which (on objects) are visible for this transaction. Moreover, as a subset of the (global) current history H_{ct} of the system, the view of a transaction preserves the partial ordering of the operations. Here are the visible operations for a transaction t:

1. operations invoked by the transaction t itself, whatever are the objects implied (local or distant): $\{p_t[ob_{t'}] \in H_{ct}\}$



2. operations invoked by some transactions $t' \neq t$ on objects stored in the local repository of the transaction t: $\{p_{t'}[ob_t] \in H_{ct}\}$



For instance, when the HVAC engineer reads the plan produced by the architect in his/her local repository, the corresponding event $read_{hvac}[plan_{archi}]$ is logged by both transactions $hvac$ and $archi$. The HVAC engineer logs this event because he/she is the invoker of this operation (cf. item 1). The architect writes this event in his/her history because this operation concerns one of the objects of his/her local repository (cf. item 2).

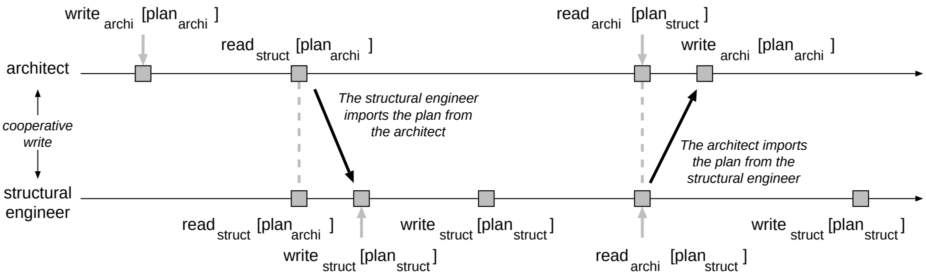

write archi [plan archi]
read struct [plan archi]
read archi [plan struct]
write archi [plan archi]
architect
cooperative write
structural engineer
The structural engineer imports the plan from the architect
The architect imports the plan from the structural engineer
read struct [plan archi]
write struct [plan struct]
write struct [plan struct]
read archi [plan struct]
write struct [plan struct]

Figure 5. *Sample execution*

Figure 5 depicts a sample execution where the architect and the structural engineer collaborate to draw a plan. This figure uses a space-time message diagram [4] to represent the execution of transactions $archi$ and $struct$. Each horizontal line corresponds to the execution of one transaction (time going from left to right), i.e. the local history of this transaction. An arrow between two transactions represents a data exchange between them, the *send* event being at the root of the arrow (eg: read operation), and the *receive* event being at the head of the arrow (eg: write event).

Within our transaction model, the view of a transaction t (denoted $View_t$) is defined as follows: $View_t = \{p_t[ob_{t'}] \in H_{ct}\} \cup \{p_{t'}[ob_t] \in H_{ct}\}$. This identifies, among all the operations of the whole history, those of which the transaction t is aware. $H_{ct/t}$ denotes this set and corresponds to the **local history** of the transaction t. For instance, the execution shown by figure 5 results in the following histories for both transactions $archi$ and $struct$:

archi	**struct**
$Contract[archi,struct,\{plan\},C/W]$	$Contract[archi,struct,\{plan\},C/W]$
$write_{archi}[plan_{archi}]$,	
$read_{struct}[plan_{archi}]$, $\longleftarrow$	$\longrightarrow$ $read_{struct}[plan_{archi}]$,
	$write_{struct}[plan_{struct}]$,
	$write_{struct}[plan_{struct}]$,
$read_{archi}[plan_{struct}]$, $\longleftarrow$	$\longrightarrow$ $read_{archi}[plan_{struct}]$,
$write_{archi}[plan_{archi}]$.	
	$write_{struct}[plan_{struct}]$.

Thus, an history H (local or global) is a series of object events (operation invocations). $H^{(ob)}$ denotes projection of the history H onto the object *ob*. For instance $H^{(ob)} = p_1 \circ p_2 \circ \ldots \circ p_n$ represents both the order of the operations invoked on the object *ob* (i.e. the operation p_i occurs before the operation p_{i+1}, and we write this $p_i \rightarrow p_{i+1}$) and their functionnal composition. In other words, the state s of the ob-

[4] Used in the field of distributed systems, such diagrams show exchanges (message passing) among network distributed processes.

ject *ob* within the history $H^{(ob)}$ is the result of the invocation of successive operations in $H^{(ob)}$ (according to the relation $\rightarrow$) from an initial state s_0 (i.e. $s = state(s_0, H^{(ob)})$, or simply $s = state(H^{(ob)})$).

Within the ACTA formalism, a transaction accesses objects of some common repository through the invocation of operations specific to objects. Each operation returns a value and produces a new state. Let s be the state of an object. $return(s,p)$ denotes the result of the invocation of the operation p on this object. The state procuced by the operation p is represented by $state(s,p)$. Two operations conflict for the state $H^{(ob)}$ (we write this $conflict(H^{(ob)},p,q)$ or simply $conflict(p[ob],q[ob])$) if their effects on this state or their returned values are not independent of the order in which they are invoked. Two operations that do not conflict are *compatible*.

As object state changes are viewed through the return values of operations, we can define dependencies between operations that conflict. When two operations conflict (i.e. the predicate $conflict(H^{(ob)},p,q)$ returns `true`), then the following predicate $return\_value\_independent(H^{(ob)},p,q)$ is `true` if the return value of the operation q does not depend on the operation p to be invoked before or after q. We can denote this with $return(H^{(ob)} \circ p, q) = return(H^{(ob)}, q)$. Otherwise, the operation q is "return-value dependent" on the operation p and we denote this with the predicate $return\_value\_dependent(H^{(ob)},p,q)$.

It is obvious that these two definitions introduced by the ACTA formalism are not based, from our transaction model point of view, on "physical" instances of objects (eg ob_t), but on "logical" objects instead (eg ob). For instance, at the level of the logical object $plan$, the reading operation invoked by the HVAC engineer on the plan of the architect (represented by $read_{hvac}[plan]$) conflicts with any update operation invoked by the architect on his/her plan ($write_{archi}[plan]$). Moreover, the predicate $return\_value\_dependent(H^{(plan)},write_{archi}[plan],read_{hvac}[plan])$ returns `true`.

As $conflict(p[ob],q[ob])$ was a shortcut for $conflict(p_{t_i}[ob],q_{t_j}[ob])$ [5], a first step is to consider $conflict(p_{t_i}[ob],q_{t_j}[ob])$ as a shortcut for $conflict(p_{t_i}[ob_{t_k}],q_{t_j}[ob_{t_k}])$ [6]. For the time being, it means that two operations can only conflict when they are invoked on the same copy of a logical object. Later, we will give the meaning of $conflict(p_{t_i}[ob_{t_{k_1}}],q_{t_j}[ob_{t_{k_n}}])$. In other words, we will define the notion of conflict between two operations p_{t_i} and q_{t_j} invoked on two distinct copies of a same logical object *ob*. And so on for the predicate $return\_value\_independent(p_{t_i}[ob],q_{t_j}[ob])$.

Our first need is to find, when a transaction t invokes an operation p on an object $ob_{t'}$, all the operations invoked by any other transaction that could conflict with $p_t[ob_{t'}]$ (the ACTA formalism uses $ConflictSet_t$ to denote this set). Thus, $View_t$ and $ConflictSet_t$ operation sets define the events that the transaction t will be allowed to invoke. More precisely, when a transaction wants to invoke a new event, preconditions of this event (derived from the axiomatic definition of the invoker transaction) are eva-

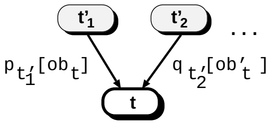
[5] Operations p and q (invoked by any two transactions t_i and t_j respectively) conflict on the object *ob*.

[6] Operations p_{t_i} and q_{t_j} conflict on any copy ob_{t_k} of the logical object *ob*.

luated against these two sets. If all of them are satisfied, the new event is executed and then logged in local histories of implied transactions (cf. transaction view definition). Otherwise, if one (or more) precondition fails, the event invocation is denied.

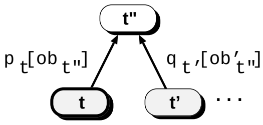
Within our transaction model, operations that could conflict with those invoked by a transaction t are defined below (the predicate $Inprogress(p)$ used below means that the operation p is executing and is neither committed nor canceled).

– operations invoked by any transaction $t' \neq t$ on objects ob_t of the transaction t: $\{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\}$



For instance, an operation invoked by the architect can possibly conflict with an operation $read_{hvac}[plan_{archi}]$ invoked by the HVAC engineer on the copy of the object $plan$ owned by the architect.

– operations invoked by any transaction $t' \neq t$ on objects of a third transaction $t'' \neq t$ if these objects appear in some operations invoked by the transaction t:



$\{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \wedge (\exists q\, q_t[ob_{t''}] \in H_{ct}) \wedge Inprogress(p_{t'}[ob_{t''}])\}$

For instance, an operation $read_{hvac}[plan_{archi}]$ invoked by the HVAC engineer can possibly conflict with an operation $read_{struct}[plan_{archi}]$ invoked by the structural engineer because they both access the copy of the object $plan$ owned by the architect.

Thus, within our transaction model, the set of operations that can conflict with an operation invoked by a transaction t is:

$$\begin{aligned} ConflictSet_t = & \{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\} \\ \cup & \{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \wedge (\exists q\, q_t[ob_{t''}] \in H_{ct}) \wedge Inprogress(p_{t'}[ob_{t''}])\} \end{aligned}$$

We can note that when we put all the objects within a single common repository (instead of using local repositories), the definitions of $View_t$ and $ConflictSet_t$ become equivalent to those found in the axiomatic definition of atomic transactions:

– $View_t = H_{ct}$ where H_{ct} denote the global current history

– $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, Inprogress(p_{t'}[ob])\}$

We have introduced the notions of local repository (several "unsynchronized" copies ob_t for a logical object ob) and local history (each transaction only has a partial view of the system) and stayed compliant with the ACTA formalism. Based on them, the next section formalizes the concept of **explicit** data transfer among transactions.

Transfer operations: Now, each transaction of our model is provided with its own local repository in which it can store local copies (instances) of objects it need to access. For instance, when two transactions $archi$ and $hvac$ share a logical object $plan$, they both have their own copy of this object ($plan_{archi}$ and $plan_{hvac}$ respectively) in their local repository. This section details the way these two transactions can exchange values of object $plan$.

As explained when we introduced new notations to identify various copies of logical objects, a transaction is not limited to the invocation of operations on its own objects. For instance, the transaction $hvac$ can invoke a $read$ operation on the copy of the object $plan$ stored in the local repository of the transaction $archi$ ($read_{hvac}[plan_{archi}]$ denotes this operation). Such operations named **transfer operations** enable transactions to exchange data between their respective local repositories.

Within a classical transaction model, an interaction between two transactions t_i and t_j on the object *ob* is denoted by $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \ \wedge \ conflict(p_{t_i}[ob], q_{t_j}[ob])$. It means that transactions t_i and t_j invoked operations $p_{t_i}[ob]$ and $q_{t_j}[ob]$ on the same object *ob*, that the operation $p_{t_i}[ob]$ occurred before the operation $q_{t_j}[ob]$ (partial order defined in H_{ct}), and that both operations conflict together. Based on notations introduced to identify copies of logical objects, what is the meaning for the following expression $(p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}]) \ \wedge \ conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$? We have two cases:

– either it is the same copy ob_{t_k} of the logical object *ob* and we come back to the case $(p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]) \ \wedge \ conflict(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}])$,
– or $ob_{t_{k_1}}$ and $ob_{t_{k_n}}$ are two distinct copies of the logical object *ob*, $(p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}]) \ \wedge \ conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$ with $t_{k_1} \neq t_{k_n}$

It is easy to deal with the first case as we can reuse existing definitions for the order relationship $\rightarrow$ and for the predicate $conflict$. So, we are more concerned with the second case, and more precisely with the meaning of $conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$. Intuitively, we can understand this as follows: *"the value of the object ob owned by the transaction t_{k_n} (i.e. $ob_{t_{k_n}}$) depends on the value of the object ob owned by the transaction t_{k_1} (i.e. $ob_{t_{k_1}}$)"*. In other words, we got a sequence of operations as the one depicted on figure 6 where transactions t_{l_i} let us "propagate", transaction by transaction, the value of the object $ob_{t_{k_1}}$ to the object $ob_{t_{k_n}}$.

Figure 6 uses a space-time message diagram to represent the execution of three transactions. Using such a graphical representation, it is easy to determine if two events are **causally** dependent: if we can find a path from one of the events to the other by going from left to right all along horizontal lines ("transaction-order" within a local history) and by following arrows ("read-from" relationships between transactions), then events are linked; otherwise they are independent one from each other.

To be more formal, a transfer operation is a set of two "basic" operations invoked by a single transaction t_l, one to access the value of the object $ob_{t_{k_i}}$ (eg: $read_{t_l}[ob_{t_{k_i}}]$), the other to modify the value of the object $ob_{t_{k_j}}$ (eg: $write_{t_l}[ob_{t_{k_i}}]$). In his way,

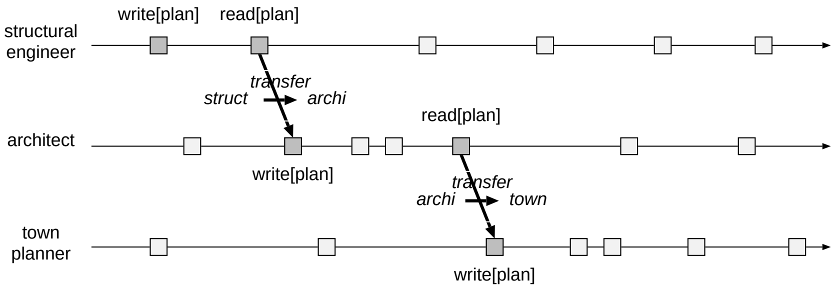


Figure 6. *Propagation of changes*

the transaction t_l transfers the content of one copy of ob ($ob_{t_{k_i}}$) inside another copy ($ob_{t_{k_j}}$). Such an operation is represented by $transfer_{t_l}[ob_{t_{k_i}},ob_{t_{k_j}}]$ and corresponds to a set of two operations, i.e. $transfer_{t_l}[ob_{t_{k_i}},ob_{t_{k_j}}] = (q_{t_l}[ob_{t_{k_i}}],p_{t_l}[ob_{t_{k_i}}])$ with $(q \in Read^{(ob)}) \wedge (p \in Write^{(ob)}) \wedge (q_{t_l}[ob] \rightarrow p_{t_l}[ob])$, where $Read^{(ob)}$ denotes the set of operations that access but do not change the value of the object *ob*, and where $Write^{(ob)}$ denotes the set of operations that update the value of the object *ob*.

These transfer operations allow us synchronize the local histories of the transactions. Read-from dependencies between transactions are built when an operation $p_{t_l}[ob_{t_k}]$ is logged by both transactions t_l (the one that invokes the operation) and t_k (the one that owns the object implied in the operation).

We can now define the semantic dependency relationship $\xrightarrow{dep}$ among the objects. For an object, this relationship determines if the value of one copy was produced from the value of another copy. This semantic dependency relationship $\xrightarrow{dep}$ expresses at the object level the causal dependency relationship that exists at the event level. Here is the meaning of a conflict between two operations invoked on two distinct copies of logical object *ob*:

$$conflict(p_{t_i}[ob_{t_{k_i}}],q_{t_j}[ob_{t_{k_j}}]) \Leftrightarrow conflict(p_{t_i}[ob],q_{t_j}[ob]) \wedge (ob_{t_{k_i}} \xrightarrow{dep} ob_{t_{k_j}})$$

In practice, the notion of conflict between two operations $p_{t_i}[ob_{t_{k_i}}]$ and $q_{t_j}[ob_{t_{k_j}}]$ invoked on two distinct copies $ob_{t_{k_i}}$ and $ob_{t_{k_j}}$ of the same logical object *ob* only serve as a basis to prove that properties ensured by our *distributed* correctness criteria presented in this section are identical to the ones ensured by classical *centralized* correctness criteria. As our distributed correctness criteria are based on local informations only (local histories of transactions), they mainly use the traditional notion of conflict between two operations within a single local history. We provided the reader with a general conflict definition to emphasize the differences between the *centralized* and *distributed* approaches.

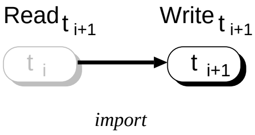
Let us go back to our support example and to the execution depicted in figure 6. ⋄
1. The structural engineer makes some changes on the copy of the plan he/she owns

in his/her local repository: this is denoted by the operation $write_{struct}[plan_{struct}]$. $\diamond$ **2.** The architect imports this new version of the plan: this is denoted by the operation $transfer_{archi}[plan_{struct},plan_{archi}]$ or, more precisely, by the two operations $read_{archi}[plan_{struct}]$ and $write_{archi}[plan_{archi}]$. So, the object $plan_{archi}$ depends on the object $plan_{struct}$, i.e. $plan_{struct} \xrightarrow{dep} plan_{archi}$. $\diamond$ **3.** Now, the town planner imports this new version of the plan available from the architect: this is the operation $transfer_{town}[plan_{archi},plan_{town}]$ or, more precisely, operations $read_{town}[plan_{archi}]$ puis $write_{town}[plan_{town}]$. So, the object $plan_{town}$ depends on the object $plan_{archi}$, i.e. $plan_{archi} \xrightarrow{dep} plan_{town}$.

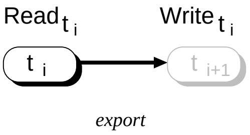
Thus, the predicate $conflict(write_{struct}[plan_{struct}],read_{town}[plan_{archi}])$ evaluates to `true` as we have $conflict(write_{struct}[plan],read_{town}[plan])$ for the logical object $plan$ and, by transitivity, we have $plan_{struct} \xrightarrow{dep} plan_{town}$ for copies of this object.

We can now present two special cases of transfer operations: a transaction imports (respectively exports) an object from (respectively towards) another transaction:

– **Import**: If we have $t_{l_i} = t_{k_{i+1}}$, then $transfer_{t_{k_{i+1}}}[ob_{t_{k_i}},ob_{t_{k_{i+1}}}] \equiv import_{t_{k_{i+1}}}[ob_{t_{k_i}}]$, i.e. the transaction $t_{k_{i+1}}$ imports (inside its object $ob_{t_{k_{i+1}}}$) the value of the object *ob* available from the transaction t_{k_i} (i.e. of the object $ob_{t_{k_i}}$).



– **Export**: If we have $t_{l_i} = t_{k_i}$, then $transfer_{t_{k_i}}[ob_{t_{k_i}},ob_{t_{k_{i+1}}}] \equiv export_{t_{k_i}}[ob_{t_{k_{i+1}}}]$, i.e. the transaction t_{k_i} exports the value of the object *ob* (i.e. $ob_{t_{k_i}}$) towards the transaction $t_{k_{i+1}}$ (i.e. inside the object $ob_{t_{k_{i+1}}}$).



Intuitively, the meaning of an import is *"the consumer decides to get the information from the producer"*, whereas the meaning of an export is *"the producer put the information to the consumer by force"*.

For instance, when the town planner imports the version of the plan produced by the architect, two operations occur: the reading of the plan from the architect ($read_{town}[plan_{archi}]$) and the update of the plan in the local repository of the town planner ($write_{town}[plan_{town}]$). When the structural engineer exports his/her plan to the architect, the situation is quite the same: $read_{struct}[plan_{struct}]$ followed by $write_{struct}[plan_{archi}]$.

Please note that when we force all the objects of the system to be stored within a single repository common to all the transactions (as it is in classical transaction models), we get back the traditional notation

$(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge conflict(p_{t_i}[ob], q_{t_j}[ob])$. Once again, the notions of local repository and local history extend the ACTA formalism and stay compliant with it.

6.3. *Distributed interaction control*

To allow transactions to cooperate through intermediate result exchanges (that means results produced while transactions are running, prone to further changes, and possibly inconsistent for the system), [MOL 96] defined a new correctness criterion: *COO*-serializability. By removing the isolation property between transactions, this criterion supports many interesting executions which are not serializable, and three new cooperation patterns in particular: client/server, writer/reviewer, and cooperative write. However, even if transactions can execute on geographically distributed sites, the control of their interactions remains centralized because the *COO*-serializability criterion is defined on the global history of the whole system. Using distribution facilities provided by our new transaction model, this section defines a new criterion, the *DisCOO*[7]-serializability, that ensures the same properties than *COO*-serializability on the whole system, but in a decentralized way.

First, we remind the reader of the axiomatic definition of the *COO*-transactions (figure 7). Then, we detail another axiomatic definition which is equivalent (both criteria accept the same set of cooperative executions), but based on events logged in local histories of transactions only.

COO **correctness criterion:**

The *COO*-serializability was defined in [MOL 96] to support cooperative executions of transactions by relaxing the isolation property. It means that transactions can cooperate, during their execution, through data exchanges within a common repository. *COO* defines two kinds of results: **intermediate results** produced by transactions at some times of their execution but prone to further changes and possibly inconsistent for the system, and **final results** that transactions produce at commit time. The *COO* criterion can be viewed as an extension of a classical correctness criterion, the serializability, to support the notion of intermediate result. Intuitively, a cooperative execution is correct with regard to the *COO*-serializability if the following synchronisation rules are satisfied:

1. A transaction that produced a intermediate result has to produce the corresponding final result.

2. If a transaction reads an intermediate result produced by a transaction, then it has to read the corresponding final result before it can commit (and produce its own final results). When a transaction reads an intermediate result from another transaction, it sets a dependency on this transaction. When the transaction reads the corresponding final result, the dependency is removed.

[7] *DisCOO* stands for Distributed *COO*.

3. When the system finds a cycle within the graph of dependencies between transactions (bidirectional intermediate result exchanges between two transactions, for instance), it groups all the transactions implied in the cycle. Transactions of such a group (denoted by T_{coo}) have to commit in an atomic way.

1. Cooperation leads to repeated occurrences of the same operations applied to the same objects. Thus, in general, the history of a cooperative execution contains several sequences of identical operations. For such a sequence, only the last occurence needs to be committed. This defines a "useful" sub-history of the execution.

$$(\mathtt{Commit}_t \in H) \ \Rightarrow \ \forall ob\, \forall Q_t[ob]$$
$$\exists q \in Q_t \ (\forall q' \in Q_t \ , q' \neq q \ , q'_t[ob] \rightarrow q_t[ob]) \ \wedge \ (Commit_t[q_t[ob]] \in H)$$

2. When an operation is committed, all the operations which it depends on have to be committed too.

$$(Commit_t[q_t[ob]] \in H) \Rightarrow$$
$$\exists p \ (\ return\_value\_dep(p_{t'}[ob],q_t[ob]) \ \wedge \ (p_{t'}[ob] \rightarrow q_t[ob]) \ ) \ \Rightarrow$$
$$(Commit_{t'}[p_{t'}[ob]] \in H)$$

3. Within a group, all the transactions have to reach a common final state for shared objects.

$$(\mathtt{Commit}_t \in H) \ \wedge \ t \in T_{coo} \ \Rightarrow \ \forall ob\, \forall q$$
$$(State(H^{(ob)}) \neq State(H^{(ob)} \circ q)) \ \wedge \ (Commit_{t_i}[q_{t_i}[ob]] \in H) \ \wedge \ (t_i \neq t) \ \Rightarrow$$
$$\exists p \ (\ return\_value\_dep(q_{t_i}[ob],p_t[ob]) \ \wedge \ (State(H^{(ob)}) = State(H^{(ob)} \circ p)) \ \wedge$$
$$(q_{t_i}[ob] \rightarrow p_t[ob]) \ \wedge \ (Commit_t[p_t[ob]] \in H)$$
$$)$$

4. Within a group, either all the transactions commit (`Commit`), or none of them commit (`Abort`). A group behaves like a single transaction.

$$\forall t_i,t_j \in T_{coo} \ , t_i \neq t_j \ , (t_i \, \mathcal{SCD} \, t_j) \ \wedge \ (t_i \, \mathcal{AD} \, t_j)$$

5. When an operation is aborted, all the operations that depend on it have to be aborted too.

$$(Abort_t[p_t[ob]] \in H) \ \Rightarrow$$
$$(return\_value\_dep(p_t[ob],q_{t_i}[ob]) \wedge (p_t[ob] \rightarrow q_{t_i}[ob])) \ \Rightarrow \ (Abort_{t_i}[q_{t_i}[ob]] \in H)$$

Figure 7. *Fundamental axioms of COO-transactions*

DisCOO distributed correctness criterion:

The *COO*-transaction model doesn't vary from classical transaction models with regard to the way transactions access objects (through a common centralized repository) and the interaction coordination defined by the *COO*-serializability correctness criterion (based on the global history of the whole system). In other words, this model needs to be aware of **all** the operations invoqued by **all** the transactions on **all** the objects to coordinate interactions among transactions.

Our work aims to decentralize this control towards transactions themselves in the form of small pieces of control defined for each transaction with regard to events logged in its local history. The *DisCOO*-serializability can be viewed as a distributed version of the *COO*-serializability. Our "correctness criterion distribution" approach means that local properties to be checked by each transaction with regard to its partners are defined in such a way that, when they are satisfied on all transactions, they ensure the same behavior of the whole system than properties of the initial "centralized" correctness criterion. This section presents a new axiomatic definition for cooperative transactions which is equivalent to 7 but based on local histories (cf. figure 8). Equivalent means that when the *DisCOO*-serializability is satisfied on each node of the transaction graph, the whole execution conforms to the *COO*-serializability (a *DisCOO*-serializable execution is *COO*-serializable too).

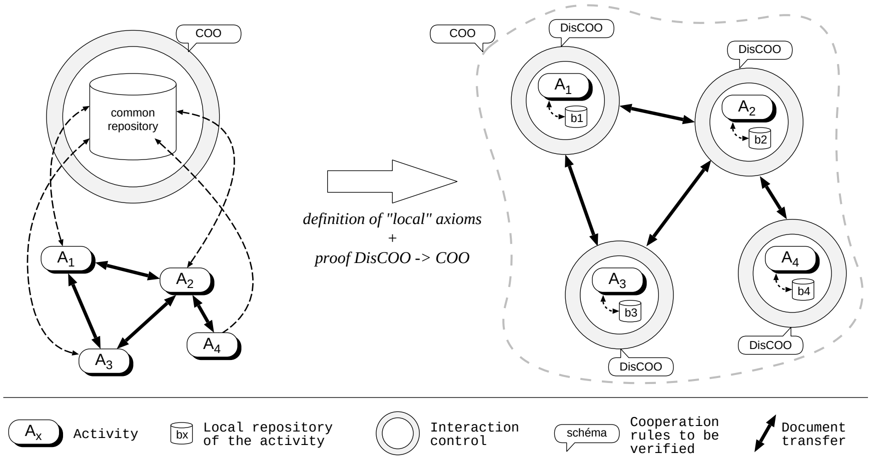


Figure 8. *How to decentralize the COO correctness criterion*

Intuitively, the first two axioms of the *COO*-transactions prevent a transaction from committing (`Commit`) if it isn't *"up to date"*, i.e. if, for some object, this transaction reads an intermediate result but didn't reread the corresponding final result (produced by a committed operation). In other words, if an operation q depends on an operation p (i.e. $return\_value\_dep(p,q)$), then the last occurence of q has to depend on the last occurence of p. This is the purpose of the predicate up_to_date defined below[8]. It denotes that the transaction t_j is *"up to date"* with regard to the transaction t_k for objects in the set O.

[8] We use rvd as a shortcut for $return\_value\_dependent$.

$$\begin{aligned} &up\_to\_date(t_j,t_k,O) \equiv \forall ob \in O \quad \forall Q_{t_j}[ob_{t_k}] \\ &\quad (\; \exists t_i \; \exists p \; rvd(p_{t_i}[ob_{t_k}], LastOcc(Q_{t_j}[ob_{t_k}])) \; \wedge \; (p_{t_i}[ob_{t_k}] \rightarrow LastOcc(Q_{t_j}[ob_{t_k}])) \;) \Rightarrow \\ &\quad (\quad \nexists p' \in Sequence(p_{t_i}[ob_{t_k}]) \quad (LastOcc(Q_{t_j}[ob_{t_k}]) \rightarrow p'_{t_i}[ob_{t_k}]) \;) \end{aligned}$$

Figure 9 presents a sample use of the up_to_date predicate based on our support example. At step 1, the last read operation invoked by the architect on the plan of the structural engineer occurred after the last write operation invoked by the structural engineer on his/her plan. Thus, the architect is *"up to date"* with regard to the structural engineer. At step 2, the structural engineer updated his/her plan after the architect accessed it. At this time, the architect is no longer *"up to date"* with regard to the structural engineer.

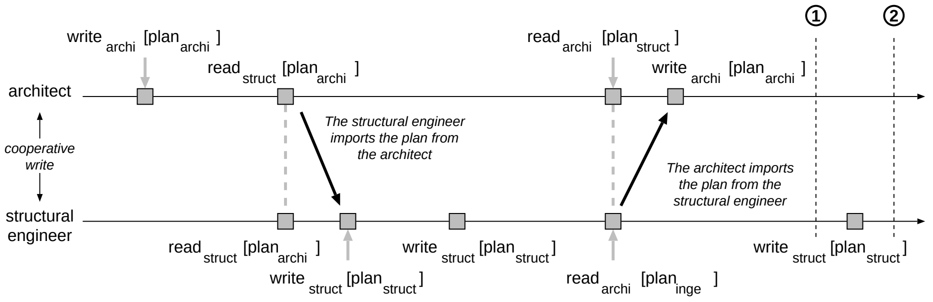

①
②
write$_{archi}$ [plan$_{archi}$]
read$_{struct}$ [plan$_{archi}$]
read$_{archi}$ [plan$_{struct}$]
write$_{archi}$ [plan$_{archi}$]
architect
cooperative write
structural engineer
The structural engineer imports the plan from the architect
The architect imports the plan from the structural engineer
read$_{struct}$ [plan$_{archi}$]
write$_{struct}$ [plan$_{struct}$]
write$_{struct}$ [plan$_{struct}$]
read$_{archi}$ [plan$_{inge}$]
write$_{struct}$ [plan$_{struct}$]

Figure 9. *Sample use of the up_to_date predicate*

The definition of the up_to_date predicate is based on two new functions: $LastOcc$ and $Sequence$. The former returns the last occurence whitin the sequence for an operation. The latter returns the ordered set of all the occurences (named the sequence) for a given operation. These functions are defined below.

Occurence sequence for an operation: An operation $p_{t_i}[ob_{t_j}]$ can be invoked several times during the execution. Even if they are all denoted by $p_{t_i}[ob_{t_j}]$ in the history, they are distinct occurences of the same operation. The set of all the occurences of the operation $p_{t_i}[ob_{t_j}]$ (ordered by the $\rightarrow$ relationship) is named the **sequence** of the operation $p_{t_i}[ob_{t_j}]$ and is represented by $P_{t_i}[ob_{t_j}]$. Given an operation, the purpose of the $Sequence$ function is to return the sequence of this operation, i.e.

$$Sequence(occ) = \{p_{t_i}[ob_{t_j}] \in H \;\mid\; occ \equiv p_{t_i}[ob_{t_j}]\}$$

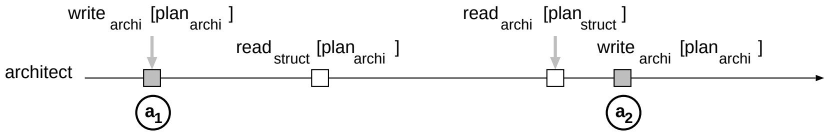

write$_{archi}$ [plan$_{archi}$]
read$_{struct}$ [plan$_{archi}$]
read$_{archi}$ [plan$_{struct}$]
write$_{archi}$ [plan$_{archi}$]
architect
a$_1$
a$_2$

Figure 10. *Sample occurence sequence*

Look at the architect for instance. He/she will update his/her plan of the apartment several times. Within the local history of the transaction $archi$, several successive

$write_{archi}[plan_{archi}]$ operations are logged (cf. operations a_1 and a_2 on the figure 10, extracted from the figure 5). The ordered set of these occurences is named the **sequence** of the operation $write_{archi}[plan_{archi}]$ and is denoted by $WRITE_{archi}[plan_{archi}]$.

Last occurence of an operation: Given the sequence of occurences for an operation, the function $LastOcc$ returns the last occurence (with regard to the $\rightarrow$ relationship) within this sequence, i.e.

$$LastOcc(S) = occ \in S \quad \text{such that} \quad \nexists\, occ' \in S\ (occ' \neq occ) \quad occ \rightarrow occ'$$

On the example depicted in figure 10, the last occurence of the operation sequence $WRITE_{archi}[plan_{archi}]$ corresponds to the second operation $write_{archi}[plan_{archi}]$.

DisCOO **correctness criterion:** The two following axioms express that a transaction will be allowed to `Commit` only if it is *up_to_date* with regard to all the transactions on which it depends, and if all these transations are committed (and so on recursively).

$$(\mathtt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \forall ob \quad up\_to\_date(t_j, t_k, \{ob\})$$

$$\begin{aligned}&(\mathtt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \\ &\quad (\ \exists p,q\ (rvd(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}]) \wedge (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]))\ ) \Rightarrow \\ &\quad (\mathtt{Commit}_{t_i} \in H_{t_k})\end{aligned}$$

Whenever the $return\_value\_dependent$ introduces a cycle within the graph of dependencies among transactions (group situation for the *COO* model), it means that transactions implied in the cycle have to be *up_to_date* with regard to each other before they commit. In other words, they need to reach an agreement on shared objects (group convergence). Take care that, when a transaction t_j depends on a transaction t_i, the second axiom only imposes to have $\mathtt{Commit}_{t_i} \in H_{t_k}$ when t_j commits ($\mathtt{Commit}_{t_j} \in H_{t_k}$). Especially, there is no need for $\mathtt{Commit}_{t_i}$ to occur before $\mathtt{Commit}_{t_j}$ (i.e. $\mathtt{Commit}_{t_i} \rightarrow_{H_{t_k}} \mathtt{Commit}_{t_j}$). They can occur **at the same time**. Thus, it avoids deadlocks in case of cycle as a solution is to have `Commit` operations of all the transactions implied in the cycle to appear at the same time in the history.

Lastly, we need to ensure that whenever an operation p is aborted, all the operations q that depend on it (i.e. $return\_value\_dep(p,q)$) are aborted too, and so on recursively. A side effect is that all transactions of a group are aborted when one of them aborts.

$$\begin{aligned}&(Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H_{t_k}) \Rightarrow \\ &\quad rvd(p_{t_j}[ob_{t_k}], q_{t_i}[ob_{t_k}]) \wedge (p_{t_j}[ob_{t_k}] \rightarrow q_{t_i}[ob_{t_k}]) \Rightarrow \\ &\quad (Abort_{t_i}[q_{t_i}[ob_{t_k}]] \in H_{t_k})\end{aligned}$$

Figure 11 sums the *DisCOO*-serializablity up in three axioms. This new correctness criterion provides transactions with more autonomy. Each transaction controls by itself its own interactions (data exchanges) with its partners, and these controls are exclusively based on events logged inside its local history. Especially, it means that the *DisCOO*-serializablity no longer uses the notion of cycle in the the graph of dependencies among transactions (we need a global view of the whole system to find cycles). The *DisCOO*-serializablity works transaction by transaction. Thus, even if we speak about groups of transactions to denote inter-dependencies among transactions, groups don't explicitely exist within the *DisCOO*-transaction model.

1. $(\texttt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \forall ob \quad up\_to\_date(t_j, t_k, \{ob\})$

2. $$\begin{array}{l} (\texttt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \\ \qquad (\, \exists p,q \; (rvd(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}]) \wedge (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]))\,) \Rightarrow \\ \qquad (\texttt{Commit}_{t_i} \in H_{t_k}) \end{array}$$

3. $$\begin{array}{l} (Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H_{t_k}) \Rightarrow \\ \qquad rvd(p_{t_j}[ob_{t_k}], q_{t_i}[ob_{t_k}]) \wedge (p_{t_j}[ob_{t_k}] \rightarrow q_{t_i}[ob_{t_k}]) \Rightarrow \\ \qquad (Abort_{t_i}[q_{t_i}[ob_{t_k}]] \in H_{t_k}) \end{array}$$

Figure 11. *Fundamental axioms of DisCOO-transactions*

To conclude this section dedicated to the *DisCOO* correctness criterion, here is an example of *DisCOO*-serializable execution (depicted in figure 12) based on our support example presented in the section 3. On the one hand, the architect and the structural engineer share the document `plan` with regard to the "cooperative write" pattern (*DisCOO* correctness criterion). On the other hand, the architect and the town planner collaborate through ducoments `plan` and `advice` with regard to the "writer/reviewer" pattern (*DisCOO* correctness criterion with some additional constraints to **direct data exchanges**).

1. The architect works on his/her plan. Then, he decides to publish a new version. We denote this operation with $write_{archi}[plan_{archi}]$. ◇ **2.** The structural engineer imports this new version to synchronize his copy of the plan. ◇ **3.** The structural engineer updates his plan and then publish a new version. ◇ **4.** The architect synchronizes his copy of the plan by importing the version of the plan produced by the structural engineer. ◇ **5.** Now, the architect can merge his changes to the copy of the plan stored in his local repository. ◇ **6.** Meanwhile, the town planner imported the version of the plan available in the repository of the architect. ◇ **7.** In this way, he/she can begin to review the plan of the architect and can produce a preliminary version of his advice. ◇ **8.** When the town planner is aware of the new version of the plan in the repository of the architect, he decides to synchronize his copy of the plan. ◇ **9.** Now, he can update his advice with regard to this new version of the plan and then publishes a new advice. ◇ **10.** The architect imports within his local repository the advice produced by the town planner. ◇ **11.** Meanwhile, the structural engineer imported the plan updated by the architect. ◇ **12.** At this time, the architect wants to commit his acivity. As the transaction $archi$ had interaction with transactions $struct$ and $town$, it needs to evaluate the three axioms of the *DisCOO*-serializability (figure 11) with regard to these two partners.

– On the side of the transaction $struct$, the last read operation invoked by the architect on the plan in the local repository of the structural engineer (step 4) occurred after the last write operation performed by the structural engineer on the same copy

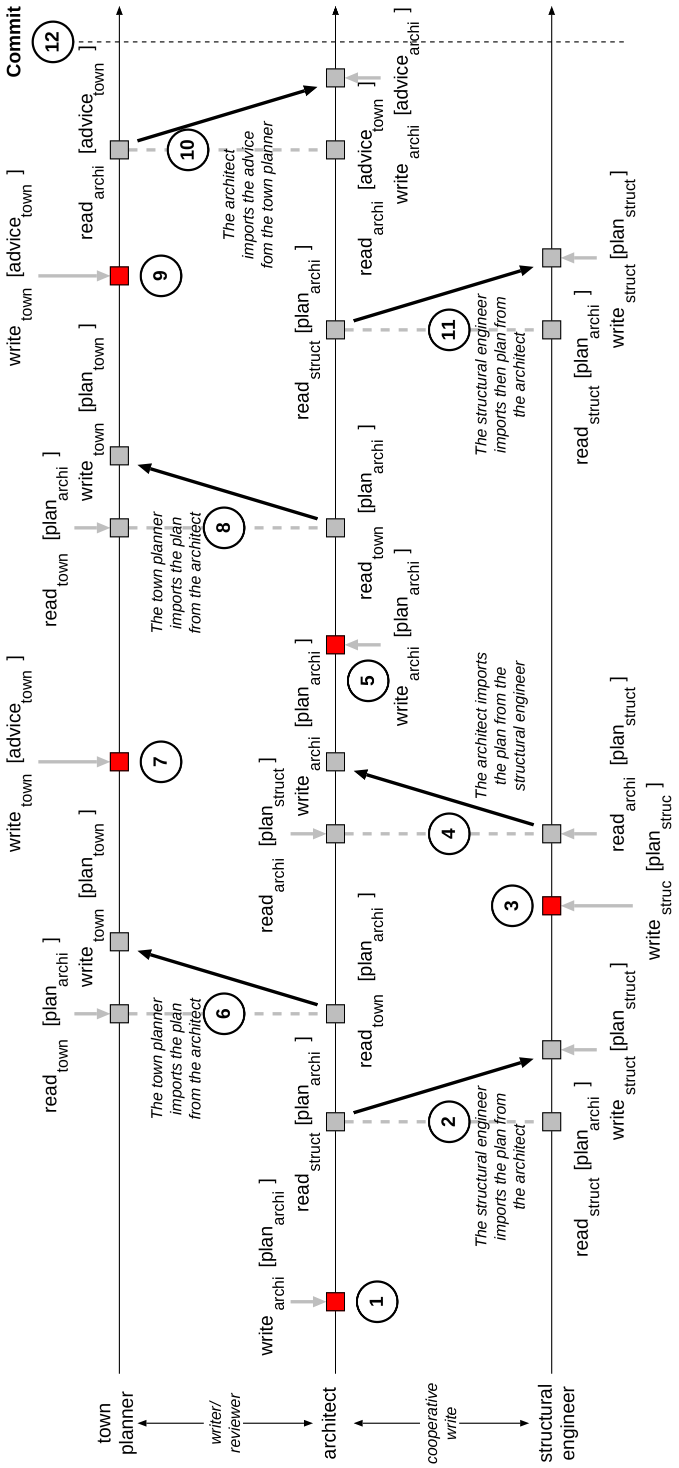

town planner
writer/ reviewer
architect
cooperative write
structural engineer
1
$write_{archi}$ [$plan_{archi}$]
2
$read_{struct}$ [$plan_{archi}$]
$read_{struct}$ [$plan_{archi}$]
$write_{struct}$ [$plan_{struct}$]
The structural engineer imports the plan from the architect
6
$read_{town}$ [$plan_{archi}$]
$write_{town}$ [$plan_{town}$]
$read_{town}$ [$plan_{archi}$]
The town planner imports the plan from the architect
3
$write_{struc}$ [$plan_{struc}$]
4
$read_{archi}$ [$plan_{struct}$]
$write_{archi}$ [$plan_{archi}$]
$read_{archi}$ [$plan_{struct}$]
The architect imports the plan from the structural engineer
7
$write_{town}$ [$advice_{town}$]
5
$write_{archi}$ [$plan_{archi}$]
8
$read_{town}$ [$plan_{archi}$]
$write_{town}$ [$plan_{town}$]
$read_{town}$ [$plan_{archi}$]
The town planner imports the plan from the architect
11
$read_{struct}$ [$plan_{archi}$]
$read_{struct}$ [$plan_{archi}$]
$write_{struct}$ [$plan_{struct}$]
The structural engineer imports then plan from the architect
9
$write_{town}$ [$advice_{town}$]
10
$read_{archi}$ [$advice_{town}$]
$read_{archi}$ [$advice_{town}$]
$write_{archi}$ [$advice_{archi}$]
The architect imports the advice fom the town planner
12
Commit

Figure 12. *Example of DisCOO-serializable execution*

of the plan (step 3). Thus the transaction $archi$ is "up to date" with regard to the transaction $struct$ (cf. axiom n^o1):

$$\begin{array}{l}(\texttt{Commit}_{archi} \in H_{struct}) \;\Rightarrow \\ \quad \forall ob \in \{plan\} \quad up\_to\_date(archi{,}struct{,}\{ob\})\end{array}$$

Due to this rvd dependency developped by operations of steps 3 and 4, the transaction $archi$ will not be able to commit before the commitment of the transaction $struct$ (cf. axiom n^o2). It means that the following axiom has to be satisfied, especially for $p = write$, $q = read$, $ob = plan$ and $t_i = struct$:

$$\begin{array}{l}(\texttt{Commit}_{archi} \in H_{struct}) \;\Rightarrow \\ \quad (\; \exists p{,}q \; (rvd(p_{t_i}[ob_{struct}]{,}q_{archi}[ob_{struct}]) \;\wedge \\ \quad (p_{t_i}[ob_{struct}] \rightarrow q_{archi}[ob_{struct}])) \;) \Rightarrow \\ \qquad\quad (\texttt{Commit}_{t_i} \in H_{struct})\end{array}$$

As the transaction $struct$ has to wait for the commitment of the transaction $archi$ too (due to steps 1 and 2), both transactions will have to commit at the same time. Please note that at the time being (step 12), the transaction $struct$ is "up to date" with regard to the transaction $archi$ about the plan, i.e.:

$$\begin{array}{l}(\texttt{Commit}_{struct} \in H_{archi}) \;\Rightarrow \\ \quad \forall ob \in \{plan\} \quad up\_to\_date(struct{,}archi{,}\{ob\})\end{array}$$

– The transaction $archi$ is "up to date" with regard to the transaction $town$ too (cf. axiom n^o1) because the last read operation invoked by the architect on the advice of the town planner (step 10) occurred after the last write operation performed by the town planner on his advice (step 9). At step 12, please note that the transaction $town$ is "up to date" with regard to the transaction $archi$ too about the plan (steps 7 and 8). As in the previous case, the axiom n^o2 will force transactions $archi$ and $town$ to be committed at the same time.

Thus, at step 12, all the three axioms of the *DisCOO*-serializability are satisfied on the local history of each transaction. It means that the transaction $archi$ can be committed, on condition that we commit (at the same time) transactions $struct$ and $town$ too.

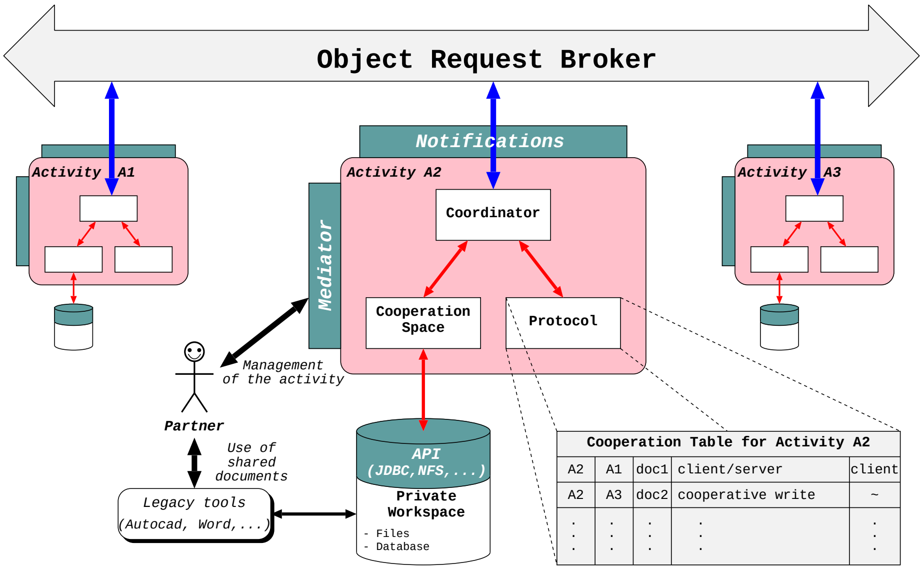
In order to illustrate the axiom n^o3, let us suppose we want to abort the operation $write_{archi}[plan_{archi}]$ invoked at setp 7. This axiom forces us to abort all the operations whose return value depends on this write operation, i.e.:

$$\begin{array}{l}(Abort_{archi}[write_{archi}[plan_{archi}]] \in H_{archi}) \;\Rightarrow \\ \quad \left\{\begin{array}{ll} & rvd(write_{archi}[plan_{archi}]{,}q_{t_i}[plan_{archi}]) \\ \wedge & (write_{archi}[plan_{archi}] \rightarrow q_{t_i}[plan_{archi}])\end{array}\right. \\ \Rightarrow \; (Abort_{t_i}[q_{t_i}[ob_{archi}]] \in H_{archi})\end{array}$$

Within the local history of the transaction $archi$, the operation $q_{t_i}[plan_{archi}]$ could be replaced with operations $read_{town}[plan_{archi}]$ and $read_{struct}[plan_{archi}]$. The abort of these two operations will be logged in the local history of transactions $town$ and $struct$ respectively. Then, according to the axiom n^o3, we have to abort write operations of step 8 and 11 too. And so on little by little.

7. Our prototype: *DisCOO*

In the perspective of not yet programming another proprietary system to support cooperation, we decided to implement our transaction model on top of a CORBA[9] Object Request Broker which provides us with transparency with regard to programming languages, operating systems and hardware. Our architecture (cf. figure 13) is defined by four main cooperation services needed to build cooperative applications: a **cooperation space** service to manage resources in the local repository of an activity; a **workspace** service to let users call their legacy tools (Word, Autocad, emacs, gcc, . . .) on their ressources viewed as files and repositories; a **coordination** service to ensure that all interactions from or toward the cooperation space are approved by the protocol service (cf. "interaction control" layer); a **protocol** service which contains all the mechanisms for the distributed correctness criteria part of our transaction model (local history, negotiated cooperation schema, methods to check cooperation schemas against the local history of the activity).



Cooperation Table for Activity A2

A2	A1	doc1	client/server	client
A2	A3	doc2	cooperative write	~
⋮	⋮	⋮	⋮	⋮

Figure 13. *Our architecture on top of an ORB*

Obviously, as introduced before, we are mainly concerned with coordination and protocol services. They are described below. For cooperation space and workspace services, our *DisCOO* prototype only provides programmers with some basic mechanisms to manage resources in order to validate the coordination and protocol services. Especially, we did not care about resource version and configuration management and that activities can only share simple file resources that we can externalize in a works-

[9] *Common Object Request Broker Architecture* [OMG 95, WEI 96]

pace (a directory) on the hard disk. Thus, users can call their legacy applications to access their resources (in the form of standard files).

Coordination service: It is one of the most important components in the design of an activity because it controls all the interactions of this activity with the other activities of the system. In other words, when an exchange occurs, the role of the coordinator is to check if this operation satisfies cooperation schemas (correctness criteria) found in the protocol component of the activity. In case of success, the operation is passed to the cooperation space of the activity to perform corresponding actions. Otherwise, the operation is denied and has no effects on the cooperation space of the activity. Thus, an exchange invoked between (the cooperation spaces of) two activities is executed only if both activities allow this operation.

Protocol service: Obviously, the core of the *DisCOO* prototype is the protocol service, and more precisely the components in charge of local history management and of cooperation schemas evaluation. Remember we defined such a cooperation schema as a set of cooperation rules or predicates based on events logged in the current local history of the activity. As an example, the cooperation schema "cooperative write" is denoted by the predicate $DisCOO$ (which uses the predicate up_to_date):

$$\begin{aligned}
&up\_to\_date(t_j,t_k,O) \equiv \forall ob \in O \quad \forall Q_{t_j}[ob_{t_k}]\\
&\quad (\ \exists t_i\ \exists p\ rvd(p_{t_i}[ob_{t_k}],LastOcc(Q_{t_j}[ob_{t_k}]))\ \wedge\ (p_{t_i}[ob_{t_k}] \rightarrow LastOcc(Q_{t_j}[ob_{t_k}]))\ ) \Rightarrow\\
&\quad (\ \nexists p' \in Sequence(p_{t_i}[ob_{t_k}]) \quad (LastOcc(Q_{t_j}[ob_{t_k}]) \rightarrow p'_{t_i}[ob_{t_k}])\ )
\end{aligned}$$

$$DisCOO(t_i,t_j,O) \equiv \begin{cases}
\forall t_1,t_2\ (t_1 \in \{t_i,t_j\},\ t_2 \in \{t_i,t_j\} - t_1)\\
\qquad (\mathtt{Commit}_{t_1} \in H_{t_2})\ \Rightarrow\ up\_to\_date(t_1,t_2,O)\\
\\
\forall ob \in O \quad \forall t_1,t_2\ (t_1 \in \{t_i,t_j\},\ t_2 \in \{t_i,t_j\} - t_1) \quad \forall t_k\ (t_k \neq t_1)\\
\qquad (\mathtt{Commit}_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2})\ \Rightarrow\\
\qquad\quad rvd(p_{t_k}[ob_{t_2}],q_{t_1}[ob_{t_2}])\ \wedge\ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}])\ \Rightarrow\\
\qquad\quad (\mathtt{Commit}_{t_k}[p_{t_k}[ob_{t_2}]] \in H_{t_2})\\
\\
\forall ob \in O \quad \forall t_1,t_2\ (t_1 \in \{t_i,t_j\},\ t_2 \in \{t_i,t_j\} - t_1) \quad \forall t_k\ (t_k \neq t_1)\\
\qquad (Abort_{t_1}[p_{t_1}[ob_{t_2}]] \in H_{t_2})\ \Rightarrow\\
\qquad\quad rvd(p_{t_1}[ob_{t_2}],q_{t_k}[ob_{t_2}])\ \wedge\ (p_{t_1}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_k}[ob_{t_2}]) \Rightarrow\\
\qquad\quad (Abort_{t_k}[q_{t_k}[ob_{t_2}]] \in H_{t_2})
\end{cases}$$

Instead of programming specific algorithms to implement all these cooperation rules, **we decided to translate them in Prolog predicates**. In this way, cooperation schemas and rules we defined in ACTA are really used as properties (Prolog predicates) evaluated on the local history of the activities. Thus, it is very easy to add new cooperation schemas within *DisCOO* or to change existing ones.

Technical aspects: All the components of *DisCOO* were developed in Java on top of an ORB which is itself coded in Java (JacORB), and the Prolog interpreter we use is written in Java too. We chose the Java programming language to be able to deploy our activities in heterogeneous environments for which an Java virtual machine exists (Windows 98/NT, Unix, MacOS, ...). It means that our prototype is neither concerned with a particular operating system (Java bytecode) nor with communication aspects (CORBA). For instance, we managed to launch several *DisCOO*-activities on a network where some stations use Windows NT while some others run Solaris or Linux.

8. Conclusion

This paper presents a model to support cooperation among distributed activities, and two results in particular. First, it is the formal definition of a new **extended transaction model**, *DisCOO*, as well as a **distributed correctness criterion**, the *DisCOO*-serializability (that supports three cooperation schemas: "cooperative write", "client/server", "writer/reviewer"). Second, we designed a **framework** to develop cooperative applications by "connecting" distributed activities together. Such a connection between two activities defines the cooperation schema used by these activities to share a given object. This architecture was put into practice as **cooperation services** within the *DisCOO* prototype.

Distributed cooperative transaction model: As we pointed out in section 2, existing environments assisting the cooperation are not suitable for distributed applications. 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. The main drawback is that clients have to be connected to this server at all times.

A second drawback of these environments concerns mechanisms they use to coordinate the activities. Configuration management systems only deal with the storage of shared objects, their versions and configurations, locks to control concurrent accesses. Process centered environments force us to describe the whole application and to make provision for all the possible interactions among activities, a task that could become very complex. As for transactional systems, they encapsulate each activity within a transaction and ensure that if all transactions individually execute correctly, then their interleaving (due to their concurrent accesses on shared objects) does not introduce inconsistencies in the state of shared objects. However, if existing correctness criteria are well suited for classical transactions (isolated, short duration), they are too strong for distributed cooperative activities.

In opposition, we developed a distributed approach. Starting from work done in *COO* about the syntactic correctness of cooperative interactions, our first goal was to go from a control of concurrent accesses (implicit interactions) to a control of data exchanges between transactions (explicit interactions). For that purpose, we defined the notions of **local repository** for a transaction, of **local history** for a transaction, of **transfer operation** between transactions. The main idea is that we provide each transaction with its own copy of the objects it need to access, and transactions cooperate by exchanging values of their copies between their respective local repositories. Moreover, these transfer operations allow us to synchronize local histories of the transactions. It was the step *autonomy for data accesses* (section 6.2). Our second goal aimed to define **distributed correctness criteria**. In other words, a transaction should be able to coordinate by itself its exchanges with the other transactions. Whereas a classical correctness criterion defines properties on the whole history of the system, a distributed correctness criterion is designed to be evaluated by each transaction with regard to its local view of the system (its local history). We gave an example of such a distributed correctness criterion: the *DisCOO*-**serializability**. If satisfied on the local history of each transaction, it ensures the same properties on the system behavior than its centralized parent, the *COO*-serializability. It was the step *autonomy for interaction control* (section 6.3).

As a result, within our model, transactions are structured as a graph where nodes denote transactions and edges denotes cooperation schemas negotiated between transactions. This is a peer-to-peer architecture where each transaction is responsible for the control of its own interactions with the others.

DisCOO **prototype:** When we designed our prototype, we avoided developing yet another proprietary system. We thought rather in terms of basic mechanisms needed to support cooperation among activities in distributed applications. To achieve this goal, we built our framework (figure 13) on top of a CORBA objet request broker and we defined four main **cooperation services**: a cooperation space to store copies of shared objects on activities, a workspace to access shared objects with legacy tools, a protocol for interaction control, a coordinator which is the interface of the activity for other activities.

Future work: In section 6 we stated that two transactions have to negotiate a cooperation schema before they can exchange data. We were only concerned with the $Contract[t_i,t_j,O,s]$ event in the local history of transactions. One of our prospects is to work on the negotiation step which results in such an event. In other words, we now want to know how and why activities choose a particular cooperation schema to control their exchanges. Such informations could be very useful when several cooperation schemas conflict (for an activity, some operations allowed by one schema are denied by another schema). One solution is to replace, on the fly, one of the cooperation schemas used by the activity, i.e. to renegotiate one of the contracts. Another prospect is to analyse cooperation behaviors to deduce new cooperation schemas that we can formalize as correctness criteria and implement within our prototype *DisCOO* (as Prolog predicates).

9. References

[AGR 90] AGRAWAL D., ABBADI A., "Transaction Management in Database Systems", ELMAGARMID A., Ed., *Database transaction models for advanced applications*, Morgan Kauffman, 1990.

[ALO 96] ALONSO G., AGRAWAL D., ABBADI A. E., MOHAN C., "Functionality and Limitations of Current Workflow Management Systems.", *IEEE Expert Journal*, 1996.

[ATR 94] ATRIA SOFTWARE INC., "ClearCase Product Summary", report , 1994, Atria Software Inc., 24 Prime park Way, Natick, Massachusetts 01760.

[BAR 92a] BARGHOUTI N., "Concurrency Control in Rule-Based Software Development Environments", PhD thesis, Columbia University, 1992, Technical Report CUCS-001-92.

[BAR 92b] BARGHOUTI N., "Supporting Cooperation in the MARVEL Process-Centered SDE", *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, vol. 17, num. 5, 1992, p. 21–31.

[BEE 88] BEECH D., "A Foundation for Evolution from Relational to Object Databases", *Proceedings of the International Conference on Extending Database Technology*, Venice, March 1988, p. 251–267.

[BEL 94] BELKHATIR N., ESTUBLIER J., "ADELE–TEMPO : An Environment to Support Process Modelling and Enaction", FINKELSTEIN A., KRAMER J., NUSEIBEH B., Eds., *Software Process Modelling and Technology*, Research Study Press, 1994.

[BEN 97a] BENTLEY R., APPELT W., BUSBACH U., HINRICHS E., KERR D., SIKKEL K., TREVOR J., WOETZEL G., "Basic Support for Cooperative Work on the World Wide Web", *International Journal of Human-Computer Studies: Special issue on Innovative Applications of the World Wide Web*, vol. 46, num. 6, 1997, p. 827–846, Academic Press.

[BEN 97b] BENTLEY R., HORSTMANN T., TREVOR J., "The World Wide Web as enabling technology for CSCW: The case of BSCW", *Computer-Supported Cooperative Work: Special issue on CSCW and the Web*, vol. 6, Academic Press, 1997.

[BEN 99] BENALI K., MUNIER M., GODART C., "Cooperation models in co-design", *International Journal on Agile Manufacturing (IJAM)*, vol. 2, num. 2, 1999.

[BER 81] BERNSTEIN P., GOODMAN N., "Concurrency Control in Distributed Database Systems", *ACM Computing surveys*, vol. 13, num. 2, 1981, p. 186–221.

[BER 97] BERNSTEIN P., NEWCOMER E., *Principles of Transaction Processing*, Morgan Kaufmann, 1997.

[BIG 98] BIGNON J., HALIN G., BENALI K., GODART C., "Cooperation models in co-design: application to architectural design", *4th International Conference on Design and Decision Support Sys tems in Architecture and Urban planning*, Maastrich, July 1998.

[CHR 94] CHRYSANTHIS P., K.RAMAMRITHAM, "Synthesis of Extended Transaction Models", *ACM Transactions on Database Systems*, vol. 19, num. 3, 1994, p. 451-491.

[COA 97] COALITION W. M., "Terminology & Glossary", report num. WFMC-TC-1011, Issue 2.0, June 1997, Workflow Management Coalition.

[ELM 92] ELMAGARMID A., Ed., *Database transaction models for advanced applications*, Morgan Kauffman, 1992.

[GAR 87] GARCIA-MOLINA H., SALEM K., "Sagas", *Proceedings of the 12th Annual ACM Conference on the Managemant of Data*, San Francisco, California, May 1987, p. 249–259.

[GRA 93] GRAY J., REUTER A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[JAJ 97] JAJODIA S., KERSHBERG L., Eds., *Advanced Transaction Models and Applications*, Morgan Kauffman, 1997.

[MOL 96] MOLLI P., "Environnements de Développement Coopératifs", Thèse en Informatique, Université de Nancy I – Centre de Recherche en Informatique de Nancy, 1996.

[MOS 81] MOSS J. E., "Nested Transactions: An Approach to Reliable Distributed Computing,", PhD thesis, MIT, 1981.

[MUN 98] MUNIER M., GODART C., "Cooperation services for widely distributed applications", *Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco Bay, USA, 1998.

[NOD 92] NODINE M., RAMASWAMY S., ZDONIK S., "A Cooperative Transaction Model for Design Databases", ELMAGARMID A., Ed., *Database transaction models for advanced applications*, Morgan Kauffman, 1992.

[OMG 95] OMG, "The Common Object Request Broker: Architecture and Specification", report num. 2.0, 1995, Object Management Group.

[RAM 96] RAMAMRITHAM K., CHRYSANTHIS P., "A taxonomy of correctness criteria in database applications", *The VLDB Journal*, vol. 5, num. 5, 1996, p. 85–97.

[ROS 96] ROSENMAN M., GERO J., "Modelling multiple views of design objects in a collaborative CAD environment.", *Computer-Aided Design*, vol. 28, num. 3, 1996, p. 193–205.

[TIC 89] TICHY, WALTER F., "RCS — A system for version control", *Software — Practice and Experience*, vol. 15, num. 7, 1989, p. 637-654.

[WAC 92] WACHTER H., REUTER A., "The ConTract Model", ELMAGARMID A., Ed., *Database Transaction Models for Advanced Applications*, Chapitre 7, p. 219–258, Morgan Kauffmann, 1992.

[WEI 84] WEIHL W., "Specification and Implementation of Atomic Data Types", PhD thesis, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA, March 1984.

[WEI 89] WEIHL W., "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types", *ACM Transactions on Programming Languages and Systems*, vol. 11, num. 2, 1989, p. 249–282.

[WEI 96] WEISS M., JHONSON A., KINIRY J., *Distributed Computing: Java, CORBA, and DCE*, Open Software Foundation Version 2.1, February 1996.