A Criterion to Enforce Correctness of Indirectly Cooperating Applications

G. Canals**, C. Godart*, P. Molli* and M. Munier*
CRIN/CNRS, Bâtiment LORIA, Campus scientifique
B.P. 239, 54506 Vandœuvre-lès-Nancy CEDEX
*Université de Nancy I-ESSTIN; **Université de Nancy 2;
godart@loria.fr

Abstract

Cooperative applications are expected to become commonplace in the future. We are concerned here with a special case of cooperation called indirect cooperation. The idea of the paper is that a Concurrency Control approach better fits to indirect cooperation than a Concurrent Programming one. In other words, it does exist syntactic correctness criteria which defines a large sphere of security in which application programmers are released from the burden of interaction explicit programming. This paper arguments this point of view and describes such a criterion: the COO-Serializability. It applies for a class of applications which cooperate indirectly.

Keywords: Cooperation, Concurrency control, Concurrent Programming, Correctness Criterion, Cooperative Programming, Cooperative Execution

1 Introduction

Cooperative applications are expected to become commonplace in the future. We are concerned here with a special case of cooperation that we called *indirect cooperation*. The idea of the paper is that a Concurrency Control approach better fits to indirect cooperation than a Concurrent Programming one. In other words, it exists syntactic correctness criteria which define a large sphere of security in which application programmers are released from the burden of programming interactions explicitly. This paper presents such a criterion called COO-Serializability and demonstrates its applicability in a class of applications which cooperate indirectly.

Section 2 intuitively defines the idea of indirect cooperation and introduces the principle on which we found correctness: a separation between correctness of interactions and individual correctness of cooperating activities. Section 3 defines and specify our correctness criterion. Section 4 is about the implementation of this criterion: it gives an implementable protocol, discusses some interactions between the correctness of interaction and the individual correctness of applications and describes some experimentations we have done and we are doing. Section 5 compares our proposition with some related work and finally concludes. The paper is illustrated with software engineering example which is our initial experimentation field.

2 Indirect cooperation: Definition and Correctness Principle

2.1 Indirect cooperation: definition

A cooperative application is a set of interactions between a set of active (software or human) activities. Activities interact (cooperate) when they share objects not only at the start and at the end of their execution, as in traditional database applications, but during their execution. Activities communicate through peer to peer communication channels or through cooperation spaces, especially common repositories.

Indirect cooperation In this paper, we are concerned with applications in which cooperation is mainly indirect, i.e. in which interactions are supported by asynchronous exchanges of objects which are not directly devoted to cooperation but which are products of the application. This includes a large domain of applications in which people contribute to a (rather) long term common objective. This is the case, as example, of software engineering applications in which people cooperate by sharing software artifacts like specification, source and documentation objects. More generally, this is the case of most concurrent enginnering applications like the design of a large building or the design, development and maintenance of some urban networks. We can also define indirect cooperation in contrast to close cooperation in which activities cooperate closely in the same space to a generally short term objective.

The more common way for two activities to interact indirectly is by exchanging items through a common repository.

Main properties of indirect cooperating applications These applications are:

1. cooperative, in the sense that programs want to interact in order to get some advantages from interactions with others. This is in opposition with concurrent activities, in which programs are in competition, ignore each other and meet others by hazard,

2. asynchronously cooperative, in the sense that interactions are not precisely forecast, not preprogrammed,

3. interactive: we are clearly concerned with applications in which control relies on the responsibility of human agents,

4. of uncertain development: when such an application starts its execution, the program of this execution is not completely defined. At the opposite, it is incrementally built in response to human agent initiatives and to interactions. As an example, a simple syntax correction activity can be sufficient to fix a bug in a program, but a software re-design activity can also be requested to fix a bug in another program,

5. of long duration: they can spend over a long period of time: several hours, days or months. This is generally the case of most design activities,

6. reversible. A decision, and the corresponding program execution is not irreversible. At the opposite, it can be compensated by executing a program, may be the same applied to different data, or another one statically forecast for this purpose, or finally one dynamically defined to reach a new consistent state.

Cooperation paradigms To be a little more concrete, we pointed out three situations that may occur during concurrent engineering and that we feel representative of indirect cooperation. These situations, called paradigms later, come from some observations during previous experiences in the software process field. If it is clear that they do not represent all the cooperation behaviors, we believe that these paradigms, and combinations of them cover a large set of cooperative executions.

The *server-consumer* paradigm corresponds to the case in which a process, hereinafter called the *consumer* reads values produced by another process, the *server*. When the two processes execute in parallel, such exchanges can occur one of several times before the server terminates. This situation may occur, when a sub-process which develops a document includes a section which is written under the responsibility of another sub-process; a typical situation in software development and more generally cooperative editing.

The *cooperative write* paradigm corresponds to the case in which two processes modify the same object at the same time and exchange values of this object. The two processes involved in a cooperative write relationship are simultaneously server and consumer for one another for the same object: the server/consumer relationships form a cycle on the same object.

The *server-reviewer* paradigm corresponds to the case in which a process produces values which are read and used by a second process to produce values of other objects. These values are in turn read by the first process and influence its own work. This corresponds to the case where server/consumer relationships form a cycle, like in the cooperative

write paradigm, but involves different objects. This is typical of most review/inspection processes.

2.2 Correctness principle

A concurrency control approach rather than a concurrent programming one Due to their indirect and asynchronous nature, it seems difficult for a programmer or a set of programmers to have a global view of the application and to explicitly program all the interactions between applications: 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 with these remarks, we believe that a concurrency control approach is better adapted to our class of applications than a concurrent programming one [5]. Thus, we found 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 much as possible not depending on the semantics of the application being synchronized [3]. Another argument in favor of the Concurrency Control approach is that, on the 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*: this must be done incrementally.

Two orthogonal dimensions More precisely, we distinguish as much as possible between the problem of **correctness of interactions**, or the problems related to the parallelism of executions, and the **correctness of individual work, also called consistency of products**, or the problems related to the fact that the right operation is applied to the right product at the right time, independently of interactions due to parallelism. Thus, we found the correctness of execution on two orthogonal dimensions:

- correctness of interactions is based on a syntactic criterion (a characterization of the traces of correct executions). This is addressed in section 3,

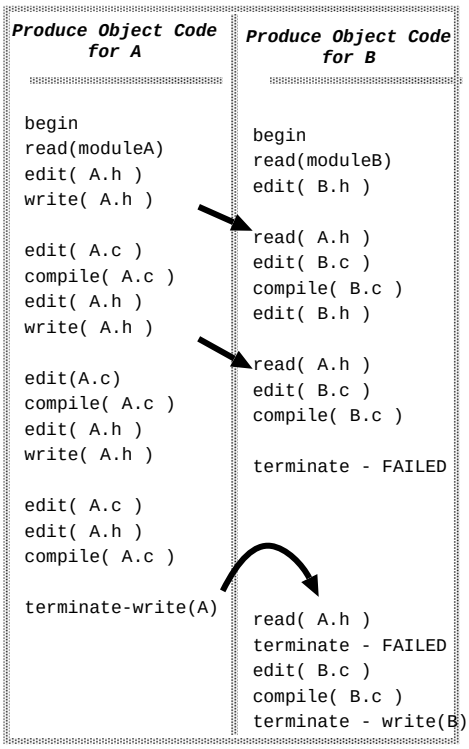

Figure 1: Correctness of interactions and correctness of products

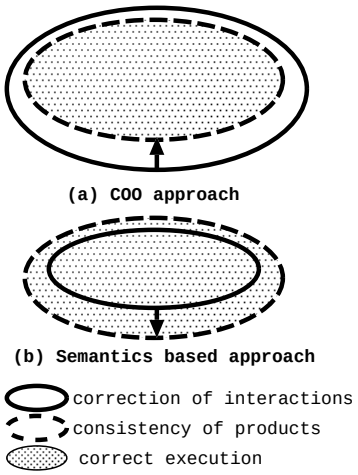

Figure 2: Correctness of executions

- consistency of products is based on semantics rules which control the ordering of the operations with regards to the states of products. Although, this is not a central topic of this paper, we consider it quickly in section 4.2.

In this context, an execution in which some activities interact indirectly is correct if it verifies both the syntactic rules related to the correctness of interactions and the semantic rules related to the consistency of products. This is the general approach used in most traditional database applications, but with one major difference: in cooperative applications, activities can exchange results when executing, while this is forbidden by traditional concurrency control algorithm. We come back on this difference in the following.

The scenario in figure 1 illustrates this principle. It simply depicts an execution in which two activities interact to produce the object code of a module A and the object code of a module B (suppose this code written in C language) with the constraint that B depends on A : the interface of A (A.h) is included in the body of B (B.c). Intuitively, this execution is correct if B is compiled with the last version of A.h. We break down this problem of correctness into two sub-problems:

1. the process in charge of developing module B can read some preliminary versions of A.h, but it must read the final value of this file; that is an instance of a general (non semantic) rule: when an activity has read an intermediate result (also called inconsistent value) of another, it must read the corresponding final result. This rule manages the correctness of interaction between A and B. In the scenario in figure 1, the first attempt to terminate issued by B in the scenario is aborted due to the fact that B has not read the final value A.h: it does not verify the syntactic rules.

2. The last value of A.h must be included in the last value of B.c (we suppose that it is implicitly done by edit(B.c)). Or in other words, the last value of B.c must be compiled with the last value A.h. This is a semantic rule directly related to the context of our application: programming in C. This rule manages the individual correctness of B. The second attempt to terminate issued by B is aborted due to the fact that B has not edited B.c with a consistent value of A.h. It does not verify this semantic rule. Another semantic rule prevents B to terminate if it has not compiled the last edited value of B.c.

To simplify, the general idea is that the syntactic criterion defines a large sphere of security which is restricted by the semantic rules (see figure 2 (a)).

3 Correctness of interactions

To characterize the set of correct interactions, we have defined a correctness criterion called *COO-serializability*. To specify it, the principle is to store the sequence of some specific events, i.e to build an history of the execution, and to verify that this history respects some well defined properties. The events that we consider are the *read* and *write* operations. An activity *read*s an object to transfer it from a repository to its workspace. The *write* operation is the reverse one which transfers an object from the activity workspace to a repository. *read*[1] and *write*[2] are abstractions for, as two examples, a *file open* and *file free* in the context of a Unix tool, or for *check_in*, *check_out* in the context of a versioned environment.

To formalize these criteria we use the notation introduced in the ACTA[3] model [18]. These notations are based on set theory and first order logic.

Before to detail our general criterion, let us describe a first criterion called *CS-serializability* which is a first step towards *COO-serializability*.

3.1 CS-serializability

CS-serializability supports the case where an activity a_1 reads one (or several) intermediate(s) result(s), also called inconsistent value(s)[4], produced

[1] $x_0 \leftarrow read(x)$ means: read the value of the object x from the repository to the variable x_0

[2] $write(x, x_0)$ means: write the object x in the repository with the value of the variable x_0

[3] for people who are not familiar with these notations, we gives the basic ACTA definitions in annex 6.1)

[4] intuitively, the idea is that a result which is not the final result of an activity is potentially inconsistent because it may not respect all the constraints of the product: as example, it is current in the first stages of a software development to share some source documents which consciously content some definitions in natural language

by another activity a_0. This is typically the case of the *server/consumer* paradigm. To illustrate that, suppose a cooperative execution of $\{a_0, a_1\}$ where data exchanges are oriented from a_0 to a_1 as in the following schedule (figure 3):

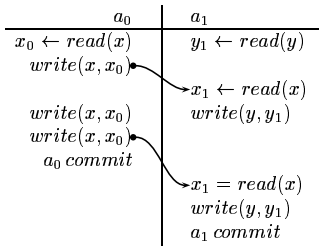

Figure 3: Cooperative history

In this execution, a_1 reads an intermediate (possibly inconsistent) value of x produced by a_0. If we observe the effect of this execution on the database and if we suppose that activities work well individually, we can conclude that only some operations are important to reason about the correctness of interactions. In fact, the same operation applied to the same objects is repeated several times. If we consider that each occurrence compensates the previous one, the highlighted execution in figure 4 has the same effect on the repository than the previous one.

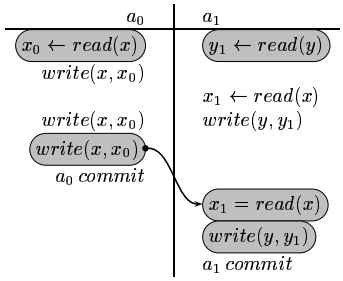

Figure 4: Serializable useful history

We can easily verify that the highlighted *useful history* in this log *is serializable*: it is equivalent to a serial execution of a_0 followed by a_1. Given that, if we suppose that a serial execution is correct, we can conclude that the whole cooperative execution is correct.

The issue now is to extract the useful sub-history from a cooperative history. To support this, we have defined to the *Last occurrence* and the *Commit Propagation* axioms.

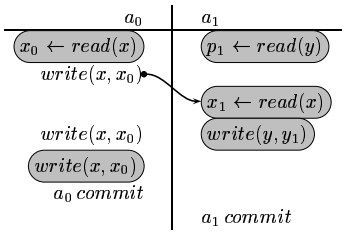

Figure 5: Serializable but not correct useful history

1. *Last occurrence axiom.* The last occurrence axiom generalizes the remark above. 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. In our example, a_0 repeats three times $write(x, x_0)$. We consider that in such a case, it is sufficient to retain in the useful history only the last occurrence of each operation sequence. The intuitive justification is that the last value of an object (we say the committed value) that an activity writes is considered as consistent by definition. This is captured by the following first axiom:

AXIOM 1 (LAST OCCURRENCE)
Let H be an history (defined in 6.1), let Q_t be a sequence of operations $q_t[ob]$ on an object ob produced by an activity t. For each object, it must exist at least one committed operation q which is not followed by an uncommitted operation in Q_t

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

This axiom defines a sub-history H' obtained from H by selecting the last occurrence of each operation sequence. However, it is not sufficient to determine a useful history equivalent to the whole execution: some other occurrences need to be added to H' as we will see now. Suppose that a_1 does not repeat the

operation $read(x)$, after applying the Last Occurrence axiom; we obtain the following execution (the sub-execution is highlighted) (see figure 5):

It is easy to see that results of a_1 are produced with a potentially inconsistent value of x. The trouble is that an operation is committed while it depends of an uncommitted operation: we observe that the operation $read(x)$ of a_1 is committed although the operation $write(x, x_0)$ of a_0 on which it depends is uncommitted. Thus, this operation should be also selected in the useful history, in addition to these selected by the Last Occurrence Axiom. This is the role of the Commit Propagation Axiom.

2. *Commit Propagation Axiom* To solve this problem, we use the ACTA $return\_value\_dep(p, q)$ predicate which allows to represent this dependency. If $return\_value\_dep$ is true, then the result of $q_{t'}[ob]$ depends on the operation $p_t[ob]$. We can instantiate the $return\_value$ predicate with read/write operations:

   (a) $return\_value\_dep(read_i[ob], write_i[ob])$ means that reading the value of ob in an activity i has an effect on the next writing of ob in the *same* activity.

   (b) $return\_value\_dep(write_i[ob], read_j[ob])$ means that reading the value of ob in an activity j depends on a previous writing of this object by an activity i.

   Now, we can introduce the second axiom of CS-serializability:

   AXIOM 2 (COMMIT PROPAGATION)
   All committed operations must depend on committed operations.

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

Using these two axioms, we are now able to specify CS-Serializability by transforming the classical definition of serializability in order to integrate the fact that activities can read inconsistent values.

DEFINITION 1 (CS-SERIALIZABILITY)
Let T be a set of activities. Let H be the log of events produced by activities belonging to T. Let $\mathcal{C}_{cs}$ be a binary relation between activities belonging to T such that:

$$
\begin{array}{l}
\forall t_i, t_j \in T, t_i \neq t_j,\ (t_i \mathcal{C}_{cs} t_j) \text{ if } \exists ob\ \exists p, q \\
\quad (commit_{t_i}[p_{t_i}[ob]] \in H \wedge commit_{t_j}[q_{t_j}[ob]] \in H) \\
\quad \wedge (conflict(p_{t_i}[ob], q_{t_j}[ob])) \\
\quad \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])
\end{array}
$$

Let $\mathcal{C}^*_{cs}$ the transitive closure of $\mathcal{C}_{cs}$; i.e.:

$$(t_i \mathcal{C}^*_{cs} t_k) \text{ if } (t_i \mathcal{C}_{cs} t_k) \vee \exists t_j (t_i \mathcal{C}_{cs} t_j \wedge t_j \mathcal{C}^*_{cs} t_k)$$

H is CS-serializable iff: $\forall t \in T, \neg(t \mathcal{C}^*_{cs} t)$

Intuitively, $t_i \mathcal{C}_{cs} t_j$ if it exists a semantical dependency between an operation of t_i and an operation of t_j and t_i precedes t_j in H. An history is CS-serializable it these dependencies do not create a cycle in the history.

It is interesting to note that if all operations are committed, the above definition is equivalent to the classical definition of serializability. In other terms, all serializable executions are CS-serializable. A serializable execution is a cooperative execution where no data exchange occurs during the execution.

To illustrate these definitions, let us consider the two following executions. Figure 6 depicts a CS-serializable execution. The figure depicts the execution analysis in three time: the first one represents the initial execution, the second one highlights the sub-execution determined by the Last Occurrence axiom and the last step highlights the sub-execution determined by the Commit Propagation axiom. This execution is CS-serializable because the useful history is serializable.

Figure 7 describes a non CS-serializable history. This example is the same as in figure 6 except that a_1 does not read the last version of x. As a consequence, the Commit Propagation axiom forces the "first" write of x by a_0 to be committed and the useful history is not serializable.

3.2 COO-serializability

Now, we deal with the general case: activities can mutually read inconsistent values each from the others, either directly through one object (case

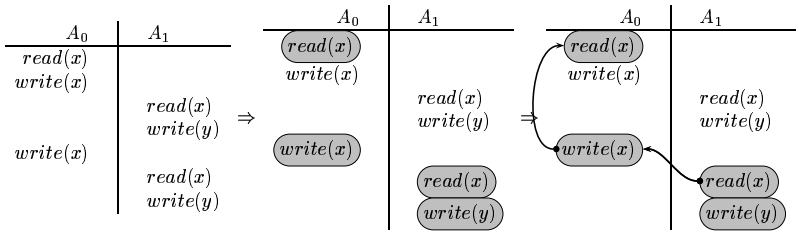

Figure 6: A CS-serializable execution

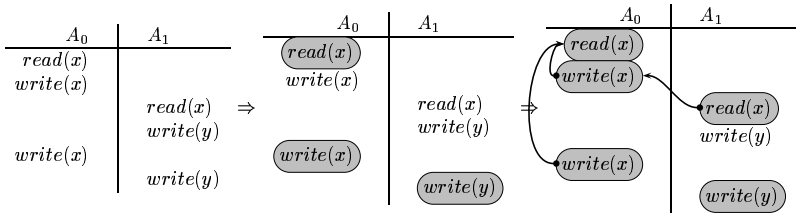

Figure 7: A non CS-serializable execution

of *cooperative write*) or indirectly through several objects (case of crossed *server/consumer* and/or *server/reviewer*). This can lead to histories that are not CS-serializable as depicted in figure 8.

More precisely, the resulting *useful history* is not serializable due to the existence of cycles in $\mathcal{C}^*_{cs}$. However, this class of execution is very representative of cooperation and thus need to be supported.

Our approach is to extend CS-serializability with the notion of *cooperative serializability* [12, 18]. The idea is to group activities implied in a cycle and to allow the members of a group to freely cooperate over some shared objects before to commit "simultaneously" one single consistent set of the shared objects (we say also to commit one state).

This is governed as follows:

1. all conflicts occurring between activities belonging to the same group are ignored,
2. CS-serializability is required between the activities which do not belong to a group with respect to all the activities in the group.

The following definitions extends CS-serializability to integrate this idea.

DEFINITION 2 (COO-SERIALIZABILITY)
Let T_c be a group of activities, $T_c \subseteq T$. Let $\mathcal{C}_{coo}$ a binary relation on T. Let H an execution of activities belonging to T.

$$\begin{array}{l}\forall t_i, t_j \in T, t_i \neq t_j \\ \quad \forall T_c \subseteq T, (t_i \mathcal{C}_{coo} t_j) \text{ iff} \\ \quad t_i \notin T_c, t_j \notin T_c (t_i \mathcal{C}_{cs} t_j) \vee \\ \quad t_i \notin T_c, t_j \in T_c (t_i \mathcal{C}_{cs} t_j \vee t_j \mathcal{C}_{cs} t_i)\end{array}$$

H is COO-serializable if $\forall t \in T \neg (t \mathcal{C}^*_{coo} t)$

Intuitively, an execution is COO-serializable if groups of activities and single activities (which do not participate to a group) are CS-serializable.

As example, in the execution depicted in figure 9, a_0 is CS-serializable with respect to $\{a_1, a_2\}$ and conflicts occurring between $\{a_1, a_2\}$ are ignored because a_1 and a_2 belong to the same group.

However, this is not completely satisfactory for two reasons.

The first is the lack of control within a group which can cause serious problems as illustrated in figure 9 where $\{a_1, a_2\}$ commits while a_2 has not read the value of y committed by a_1. Thus, the execution of $\{a_1, a_2\}$ produces a single state but a_2 is not aware of the value of y belonging to this state. Unfortunately, the last value of y may be consistent from the point of view of a_1 but not from this of a_2.

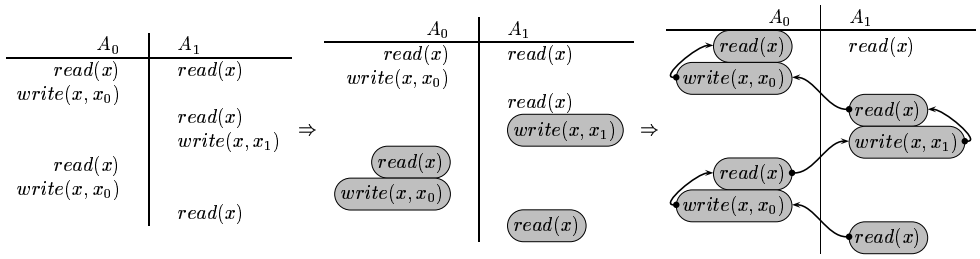

Figure 8: CS-serializability with mutual inconsistent read

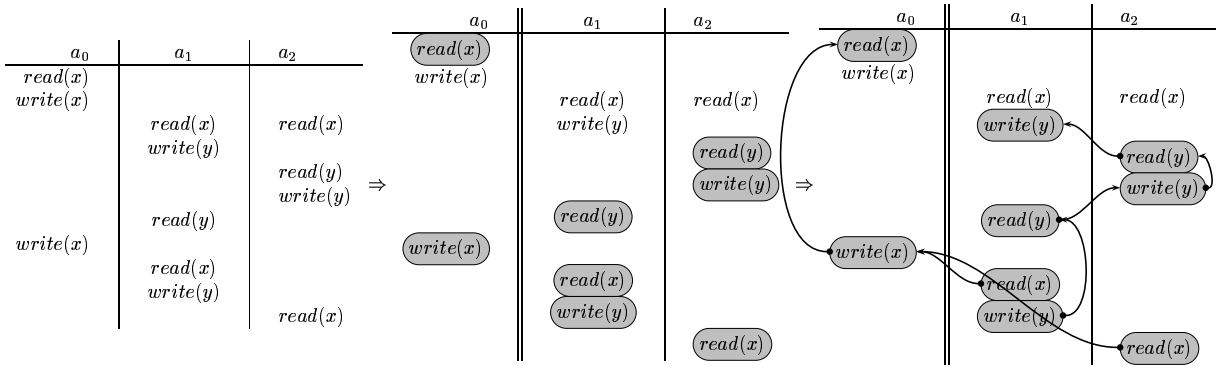

Figure 9: A non COO-serializable execution with $\{a_1, a_2\}$ in the same group (a)

Group Convergence To avoid such a case, we force activities belonging to the same group to converge to a unique final state i.e. all activities of a group which share the same object must agree on the committed value of this object. In an history, this is characterized by the fact that each activity must read the final value of the other activities of the group. This is formalized by the group convergence axiom.

Axiom 3 (Group Convergence)
Group Convergence to a unique final state:

$$\begin{aligned}&(Commit_t \in H) \wedge t \in T_{coo} \Rightarrow\\ &\quad \forall ob \forall q (State(H^{(ob)}) \neq State(H^{(ob)} \circ q)) \wedge\\ &\quad (commit_{t_i}[q_{t_i}[ob]] \in H) \wedge t_i \neq t \Rightarrow\\ &\quad \exists p\, (return\_value\_dep(q,p) \wedge\\ &\quad (State(H^{(ob)}) = State(H^{(ob)} \circ p)) \wedge\\ &\quad (q_{t_i}[ob] \rightarrow p_t[ob]) \wedge commit_t[p_t[ob]] \in H)\end{aligned}$$

Intuitively, if we consider to simplify that q is a write operation and p a read operation. This axiom means that each *write event* committed by an activity a_1 must be followed by a *read event* committed by any activity a_2 which has previously read ob (meaning of $return\_value\_dep$ in this context).

Group Condition The second problem is that it is necessary to dynamically find, by analyzing the history which activities must be grouped. This is formalized by the Group Condition axiom:

Definition 3 (Group Condition)
Let T be the set of activities. Let H be the log of events produced by activities. Let $\mathcal{C}_{dep}$ be a binary relation on T such that :

$$\begin{aligned}&t_i, t_j \in T,\ t_i \neq t_j,\ t_i \mathcal{C}_{dep} t_j \text{ iff } \exists ob, \exists p, q\\ &\quad ((p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge return\_value\_dep(p,q))\end{aligned}$$

$$t_i, t_j \in T_c \text{ iff } ((t_i \mathcal{C}^*_{dep} t_j) \wedge (t_j \mathcal{C}^*_{dep} t_i))$$

The execution depicted in figure 10 is COO-serializable: the Group Condition definition forces $\{a_1, a_2\}$ to be grouped and the Group Convergence axiom is also respected: a_2 reread the last value of y produced by a_1.

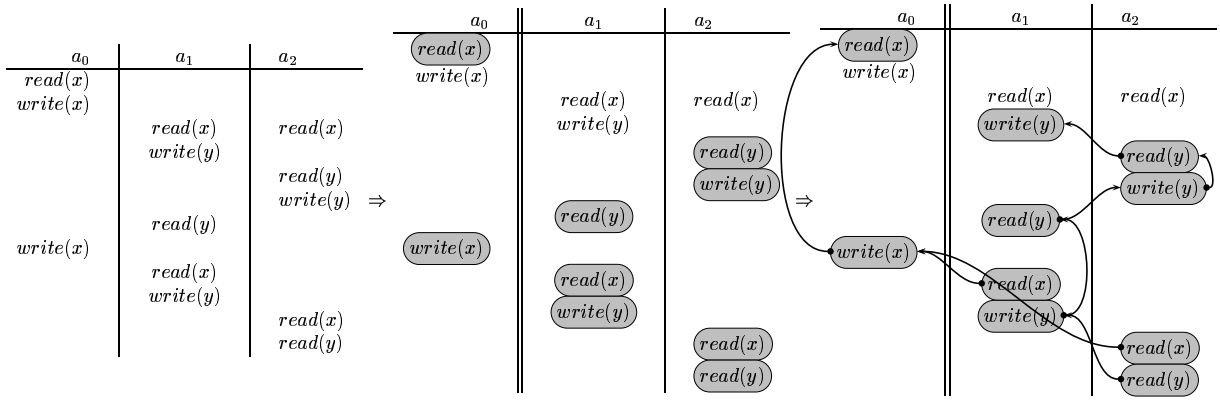

Figure 10: A COO-serializable execution with $\{a_1, a_2\}$ in the same group (b)

4 Implementing COO-Serializability

A problem with such a criteria is its implementability. We have demonstrated it by implementing a protocol that accepts only COO-serializable executions. If it accepts only COO-serializable executions, it do not accept all (correct) COO-serializable executions. This is due to the complexity of proving COO-serializability, which is at least as complex as proving serializability, which is itself a NP complete problem.

This protocol is sketched out in section 4.1 (the reader can see [13], pages 117 to 128 for a precise description of the algorithm in VDM). The interactions of this protocol with the enforcement of the semantic rules, related to the correctness of individual work, is discussed later in 4.2. 4.3 describes one application in the context of software engineering and a second in this of concurrent engineering.

4.1 The COO protocol

CS-serializability The rules to support CS-serializability are the following.

1. A result produced before the end of an activity is always an intermediate result. Users can call, at any time, the operation $IR - write$ to produce an intermediate result.

2. A result produced at the end of an activity is a final result. All final results are produced atomically during the execution of the *terminate* operation.

3. An activity that produces an intermediate result must produce a corresponding final result. The protocol collects all the objects that were "$IR - written$" by the activity and automatically produces a final result for each of them during the termination phase of the activity.

4. If an activity reads an intermediate result, then it must read the corresponding final result. The system maintains *dependency relationships* between activities to memorize the fact that an activity reads an intermediate result from another. When an activity A_1 reads the intermediate value of an object x produced by an activity A_0, then a dependency $A_1 \xrightarrow{x} A_0$ is created. When the activity A_1 reads a value of x and A_0 is terminated (i.e. when it has produced its final results), then the dependency is removed ($A_1 \overset{x}{\nrightarrow} A_0$).

5. an activity cannot terminate if it is still dependent on another. If an activity tries to terminate without reading the final value of some object after a previous access to an intermediate value of this object, the *terminate* operation is aborted and the activity remains active.

COO-serializability This first set of rules implements the CS-serializability. To implement COO-serializability, this set must be extended with

the following. They are mainly devoted to the termination of sets of cyclically dependent activities. The strategy is to make them terminate simultaneously and allow this termination to occur only when they have converged to a common value for all the objects they share:

1. Activities involved in a cyclic dependency graph form a *group* of activities.
2. A group-member activity can start a *group termination* by trying to terminate itself. The *terminate* operation in this case produces a set of potentially final results and changes the state of the activity from *active* to *ready to terminate* (RTT).
3. When a group-member activity tries to *terminate* and all the other group-members are in the RTT state, then all the activities are terminated simultaneously. Potentially final results are definitely promoted to final results.
4. When a group-member activity tries to *terminate* and there is another group-member still *active*, then it produces new potentially final results and enters the RTT state.
5. If a group member produces a new intermediate result during the group termination phase, then this termination tentative is aborted, and all the group-members re-enter the *active* state. This is the way for an activity to clearly indicate its disagreement with the object values produced by the group, and to ask for more work on the shared objects.

One can observe that, as an activity is automatically added to a cooperative group upon its participation in a cycle, we risk having all activities ending up in one big group which may in turn prevent the group to converge at all. In fact, if this phenomenon cannot be theoretically excluded, it is limited as a result of the workflow of cooperative applications which ensures globally that applications progress towards their goal. We experimented this behavior in the context of software engineering applications in which interactions are nevertheless very numerous.

One can also observe that, if the protocol imposes an activity, which has read an intermediate result x, to read the final version of x, it imposes nothing to prevent some inconsistencies to appear if the value of x is used to modify another object y. In fact, we completely delegate the propagation of final version reads, which is internal to each activity, to the evaluation of constraints as introduced in the following section. This allows a clear distinction between the correctness of interactions between activities, and the individual correctness of activities. In addition, it is to be underlined that the automatic detection of dependencies is a very tricky problem for which we have not a complete solution.

4.2 Evaluating constraints in the COO protocol context

Correctness of interactions, as defined in the above section, delimits a large sphere of security by asserting that the rules concerning the organization of data exchanges have been respected. However, there is a strong assertion at the basis of *COO-Serializability*: activities commit consistent values, or in other terms, work well individually. Coming back to our example in figure 1, it is not sufficient for the correctness of the execution to impose the last version of A.h to be read by B, as the *commit propagation* axiom imposes. We must be sure that this version is effectively integrated in the work of the *client* activity B, i.e is included in B.c which is then compiled again. This is done by taking into account some knowledge on the activity being synchronized. In this example the rule can be: "each time an include file is modified, the source files which include it must be compiled again". It is well understood that such a rule, as it is dedicated to correctness of individual work, applies locally, in the context of one activity; as it is not related to correctness of interactions, it does not apply between activities.

More generally, our approach to guarantee consistency of committed values is to base individual work correctness on semantic rules. This is in the spirit of integrity constraints in the domain of databases.

The general relationship between syntactic rules and semantic rules is expressed in figure 2-a: semantic rules are used to restrict the sphere of security defined by the correctness of interactions. Constraints evaluation occurs when an activity wants

to terminate its execution and only if the syntactic rules are enforced.

This is classical in traditional databases, but the problem is more complex due to some definitions of the above protocol, and more generally to cooperation. The COO protocol imposes classical constraint evaluation technics to evolve. This section is an overview of these problems. For a deepened view, see [21].

The first problem is that cooperation behaviors are based on the ability to share preliminary, inconsistent results. A first question which arises at this point is: what to do when a constraint cannot be immediately evaluated due to the fact that the value of an object implicated in the constraint is an intermediate result ?

Three different strategies can be considered:

1. the current activity can terminate even if all the constraints are not validated. This means that the activity delegates the responsibility of constraint validation to another active activity. This makes the hypothesis that, in the case where the terminating activity has corrupted the object base, the work of this activity can be compensated by another activity.

2. Constraints are evaluated with the last consistent values which correspond to the intermediate results which forbid the evaluation of constraints. This means that we want to compare, to assemble objects of different generations. Is it realistic in the context of long term processes as our activities are ?

3. The current activity does not terminate: it continues its execution waiting until the constraints can be evaluated. Intuitively, this means that activities cooperate to validate constraints.

The second problem is due to the grouping of activities (in fact the *Group Convergence* Axiom) which considers that activities in a cycle of dependencies must produce a unique new state of the repository. A problem occurs when the consistency of products imposes an order on the activities which are in the cycle. As an example, how to interpret a vivacity constraint like: *from a consistent state x, it is inevitable to go through a consistent state y before to enter in a consistent state z*, when the three states are committed by three activities which are grouped due to the *Group Convergence* axiom and as a consequence must produce one unique state from the point of view of correctness of interactions ?

One can explain that it is a problem of design and that somewhere the semantics rules are error prone: this can be the case, but this can also be due to the fact that correctness of interactions can group together two activities because of two dependencies which do not address the same set of objects (two sets of objects not directly semantically related each to the other). We can suppose that this will occur rarely, due to the intuitive definition of an activity, which, even if cooperating, groups together operations operating a well defined set of inter-related objects. Nevertheless, this problem can occur and we consider it as a special case of failure.

An idea to solve this problem is to split some activities, which have been previously grouped, into two or several activities in order to transform some COO-serializable sub-histories into CS-serializable histories. This principle reuses the model of Split-Transactions [16].

4.3 Putting it into practice

Our first application is Software Engineering. In fact our initial problem was to support cooperation of software developers [9]. The second application is Concurrent Engineering: the objective is to experiment our algorithm outside the context of software engineering and in the context of a federation of object bases rather than one distributed repository.

A Software engineering Environment We have developed a software engineering environment which favorizes interactions in the context of software development. This first prototype (150 000 loc) implements the COO protocol on top of an object oriented database (P-RooT which extends a PCTE based environment with object orientation).

During this work, we studied the means to tune our algorithm to fit different needs [10]. Tuning is achieved, on the one hand, by plugging different semantic rules on the core syntactic protocol as introduced above, and on the other hand by plugging a locking mechanism and locks models to restrict one

more time the set of acceptable executions. Especially, this allows to impose isolation between two processes if requested (a serializable execution is COO-serializable). We studied also the way to integrate different tuned protocols in a hierarchical way in order to fit the hierarchical organization of software processes. This work demonstrates the applicability of our approach in large software development applications, and more generally, we believe, in many real design applications.

A Concurrent Engineering Design Application A second prototype is being designed in the context of Concurrent Engineering, with examples taken in the domain of the architectural design of large buildings. The development platform is Corba + Java (JacORB [4]). The objective is to experience our model in a context outside software engineering and especially the domain of co-design on the WEB. This introduces new problems like an organization of processes following a network architecture and the ability for a process to be momentarily disconnected from the network.

5 Conclusion and Perspectives

As justified before, we believe that a concurrency control approach better fits indirect cooperation correctness than a concurrent engineering one (in opposition to, as example [2]). In this context, our approach to support cooperation is quite original: when most models [15, 1, 22] found correctness on classical serializability and uses some knowledge on the process to relax serializability (figure 2, b), we found correctness on a new criterion which defines a large sphere of security and we use knowledge to restrict this sphere of security (figure 2, a). The criterion which is the closest to our, because it allows a transaction to read uncommitted data from another, and so to import and export inconsistency, is Epsilon-serializability [19, 17]. There, correctness is based on bounding the amount of imported and exported inconsistencies. Our approach is different in the sense that although we allow the import/export of inconsistency, we require convergence and final consistency.

Our work is also related to version and configuration management [14]. In this context, we can compare our approach, with the Make tool which implements a restricted form of CS-serializability, and more generally with the Long Transaction Model [8]. There, software processes are encapsulated in transactions with an optimistic concurrency control protocol, but transactions cannot release partial results and thus cannot support cooperative executions. It can also be compared with Adèle [2] and EPOS [7], but in the first, cooperation does not rely on general properties and the second rather enters in the schema of figure 2 (b).

As introduced in section 4.2, a short term perspective of our work is to integrate the *Split-Transaction* model [16] in our model in order to allow an activity to delegate the evaluation of a constraint to another activity. Splitting an activity is also a means to suppress dependencies between activities and to obtain group convergence. In this way, it directly addresses the problem raised in section 4.1.

We have experimented our approach in the software engineering field, and we try to export it in a more general context. This implies new problems to be solved like the organization of processes in a network architecture and the need for processes to work momentarily in a disconnected way.

Another perspective of our work is to generalize our approach by studying new paradigms and new criterions. Other cooperation patterns have been introduced, as example *mutual sessions*, *turn taking*, ... in the co-authoring domain [11] and we are studying how we can support them. Other criteria exist, as aforementioned in the domain of databases, but also in this of distributed computing [20] and we are studying their relationships with, on the one hand the needs in terms of cooperation patterns, on the other hand with our criterion.

References

[1] Barghouti, N. Supporting Cooperation in the MARVEL Process-Centered SDE. *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments 17*, 5 (December 1992), 21–31.

[2] Belkhatir, N., and Estublier, J. ADELE–TEMPO: An Environment to Support Process Modelling and Enaction. In *Software Process Modelling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Research Study Press, 1994.

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

[4] Brose, G. A Java Object Request Broker. Tech. Rep. B 97-2, Université de Berlin, 1997.

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

[6] Chrysanthis, P., and Ramamritham, K. ACTA: The SAGA Continues. In *Database transaction models for advanced applications*, A. Elmagarmid, Ed. Morgan Kauffman, 1992.

[7] Conradi, R., and al. EPOS: Object-Oriented Cooperative Process Modelling. In *Software Process Modelling and Technology*, A. Finkelstein and J. Kramer and B. Nuseibeh, Ed. Research Study Press, 1994.

[8] Feiler, P., and Downey, G. Transaction-Oriented Configuration Management: A Case Study. Tech. Rep. CMU/SEI-90-TR-23 ESD-90/TR-224, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, November 1990.

[9] Godart, C. COO: a Transaction Model to support COOperating software developers COOrdination. In *4th European Software Engineering Conference (ESEC4), Garmisch, LNCS 717* (1993).

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

[11] K. Gronbaek, J. A. Helm, O. M., and Sloth, L. Cooperative hypermedia systems: A dexter based approach. *Communications of the ACM, 37(2)* (February 1994), 65–74.

[12] Martin, B., and Pedersen, C. Long-Lived Concurrent Activities. In *Distributed Object Management*, Özsu, Dayal, and Valduries, Eds. Morgan Kauffman, 1992.

[13] Molli, P. *Environnements de Développement Coopératifs*. Thèse en informatique, Université de Nancy I – Centre de Recherche en Informatique de Nancy, 1996.

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

[15] Nodine, M., Ramaswamy, S., and Zdonik, S. A Cooperative Transaction Model for Design Databases. In *Database transaction models for advanced applications*, A. Elmagarmid, Ed. Morgan Kauffman, 1992.

[16] Pu, C., Kaiser, G., and Hutchinson, N. Split Transactions for Open-Ended Activities. In *Proceedings of the 14th international conference on VLDB* (Los Angeles, September 1988), pp. 26–37.

[17] Pu, C., and Leff, A. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 16th Annual ACM Conference on the Management of Data* (Denver, May 1991), pp. 377–386.

[18] Ramamritham, K., and Chrysanthis, P. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In *Distributed Object Management*, Özsu, Dayal, and Valduriez, Eds. Morgan Kauffman, 1993.

[19] Ramamritham, K., and Pu, C. A Formal Characterization of Epsilon Serialisability. *IEEE Transactions on Knowledge and Data Engineering 7*, 6 (December 1995), 997–1007.

[20] Raynal, M., and Mizuno, M. How to find his way in the jungle of consistency criteria for

distributed objects memories. In *Proceedings of the 4th workshop on Future Trends of Computing Systems* (1993).

[21] SKAF, H., CHAROY, F., AND GODART, C. A Hybrid Approach to Maintain Consistency of Cooperative Software Development Activities. In *The Ninth International Conference on Software Engineering and Knowledge Engineering, SEKE'97* (Madrid, 1997).

[22] WACHTER, H., AND REUTER, A. The ConTract Model. In *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Ed. Morgan Kauffmann, 1992, ch. 7, pp. 219–258.

6 Annexs

6.1 ACTA Definitions

See [18] for more complete definitions.

1. A transaction accesses and updates objects by applying operations to objects. An operation returns a value and produces a state. If s is a state of the object base, $return(s,p)$ returns the value of the object after applying p in state s. $state(s,p)$ returns the state after the execution of p in the state s.

2. Calling an operation p applied to object ob in the transaction t is an event noted $p_t[ob]$. The effect of an operation is not immediately persistent. It must be explicitly committed by the operation $Commit$. It can be explicitly aborted by the operation $Abort$. An operation which is neither committed nor aborted is said $InProgress$

3. $p_{t_i}[ob] \rightarrow q_{t_j}[ob]$ is true if the event $p_{t_i}[ob]$ precedes $p_{t_j}[ob]$.

4. An history $H^{(ob)}$ of operations applied to the same object from a state s is the functional composition of these operations $H^{(ob)} = p_1 \circ p_2 \circ ... \circ p_n$ with $p_{t_i}[ob] \rightarrow p_{t_{i+1}}[ob]$

5. Two operations are in conflict from a state $H^{(ob)}$ if:
   $(state(H^{(ob)} \circ p, q) \neq State(H^{(ob)} \circ (q,p)))\vee$
   $(return(H^{(ob)}, q) \neq State(H^{(ob)} \circ (p,q)))\vee$
   $(return(H^{(ob)}, p) \neq State(H^{(ob)} \circ (q,p))$

6. $return\_value\_dep(p,q)$ is true if, $\forall t, t'$, the result of $q_{t'}[ob]$ depends on the result of $p_t[ob]$.

7. SE_t contains the set of significant events of transactions,
   IE_t contains the set of events which can initiate a transaction
   TE_t contains the set of events which can terminate a transaction

6.2 CS-serializability axioms

DEFINITION 4 (CS-SERIALIZABILITY AXIOMS)

(1) $SE_t = \{Begin, Commit, Abort\}$

(2) $IE_t = \{Begin\}$

(3) $TE_t = \{Commit, Abort\}$

(4) t satisfies the fundamental Axioms 1 à 4

(5) $View_t = H_{ct}$

(6) $ConflictSet_t = \{p_{t'}[ob] | InProgress(p_{t'}[ob])\}$

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

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

(9) $(Commit_t \in H) \Rightarrow \neg(t\mathcal{C}^*_{cs}t)$

(10) $\exists ob, \exists p, q\,((Abort_t[p_t[ob]] \in H) \wedge (return\_value\_dep(p,q) \wedge p_t[ob] \rightarrow q_{t'}[ob]))$
$\Rightarrow (Abort_{t'}[q_t[ob]] \in H)$

(11) $\exists ob\,\exists p\,(Abort_t[p_t[ob]] \in H) \Rightarrow (Abort_t \in H)$

(12) $(Abort_t \in H) \Rightarrow \forall ob \forall p((p_t[ob] \in H) \Rightarrow Abort_t[p_t[ob]] \in H)$

Axiom 1 CS-activities are associates with the three significant events: Begin, Commit and Abort.

Axiom 2 Begin is the initiation event for CS-activities.

Axiom 3 Commit and Abort events are the termination events.

Axiom 4 CS-activities satisfy fundamental axioms [6][ACTA]: a transaction cannot be initiated by several events, a transaction cannot terminate it has not been initiated, a transaction cannot be terminated by two events, only *InProgress* operations can operate objects

Axiom 5 An activity sees the current state of objects in the database.

Axiom 6 Conflicts have to be considered against all in-progress operations performed by different activities.

Axiom 7 Last occurrence of a series of identical operations are committed. (see Last Occurrence axiom page 5).

Axiom 8 A committed operation must depend from a committed operation(see Commit Propagation axiom page 6).

Axiom 9 An activity can only commit if it is not part of a cycle of $\mathcal{C}_{cs}$ developed through the invocation of conflicting operations. (see CS-serializability page 6).

Axiom 10 If an operation is aborted, then a dependent operation is aborted too.

Axiom 11 If an operation is aborted, then the responsible transaction is aborted too.

Axiom 12 If activities abort then all operations belonging to this transaction are aborted.

6.3 COO-serializability Axioms

DEFINITION 5 (COO-SERIALIZABILITY AXIOMS)
Let T an activity set. $t \in T, \forall T_c \subseteq T$.

(1) $SE_t = \{Begin, Commit, Abort\}$

(2) $IE_t = \{Begin\}$

(3) $TE_t = \{Commit, Abort\}$

(4) t satisfies the fundamental Axioms 1 à 4

(5) $View_t = H_{ct}$

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

(7) $ConflictSet_t = \{p_{t'}[ob] \mid t \in T_c \Rightarrow t' \notin T_c, InProgress(p_{t'}[ob])\}$

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

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

(10) $(Commit_t \in H) \wedge t \in T_c \Rightarrow$
$\forall ob \forall q(State(H^{(ob)}) \neq State(H^{(ob)} \circ q)) \wedge (commit_{t_i}[q_{t_i}[ob]] \in H) \wedge t_i \neq t \Rightarrow$
$\exists p\,(return\_value\_dep(q,p) \wedge (State(H^{(ob)}) =$
$State(H^{(ob)} \circ p)) \wedge (q_{t_i}[ob] \rightarrow p_t[ob]) \wedge commit_t[p_t[ob]] \in H)$

(11) $(Commit_t \in H) \Rightarrow \neg(t\mathcal{C}^*_{coo}t)$

(12) $\exists ob\, \exists p\,(Abort_t[p_t[ob]] \in H) \Rightarrow (Abort_t \in H)$

(13) $\exists ob, \exists p, q\,((Abort_t[p_t[ob]] \in H) \wedge (return\_value\_dep(p,q) \wedge p_t[ob] \rightarrow q_{t'}[ob]))$
$\Rightarrow (Abort_{t'}[q_t[ob]] \in H)$

(14) $(Abort_t \in H) \Rightarrow \forall ob \forall p((p_t[ob] \in H) \Rightarrow Abort_t[p_t[ob]] \in H)$

Coo-Activities enhance CS-activities with the notion of group of activities like in [12][Cooperative Serializability].

- Axioms 1-5 are identical to CS-activities.
- Conflicts have to be considered against all in-progress operations performed by activities that not belong to the transaction group.
- See Last Occurrence axiom page 5.
- See Commit Propagation axiom page 6.
- Axiom 10 states that activities belonging to the same group must converge to unique final state. (Group Convergence Axiom page 8).
- Axiom 11 states that an activity can only commit if it is not part of a cycle of $\mathcal{C}_{coo}$ developed through the invocation of conflicting operations. (see COO-serializability page 7).
- Axiom 12-14 are identical to CS-activities.