COO approach to support cooperation in software developments

G. Canals
C. Godart
F. Charoy
P. Molli
H. Skaf

Indexing terms: Cooperation, Consistency, Concurrency control, Constraints, Software engineering environment, Transactions

Abstract: The COO system proposes a framework to organise the cooperation between developers of complex software systems. The key idea of COO is to base software process correctness on a safe transaction model: COO promotes an original advanced transaction model which integrates some general properties that define a very permissive core synchronisation protocol, and process specific knowledge that allows the gearing of the core protocol towards process characteristics.

1 Introduction

People involved in a software development process share common objectives and interact to reach these objectives. They cooperate by carrying out in parallel, different activities that modify shared objects, and progress together towards their goal, the sum of which reach the global objective. This paper focuses on the approach we designed to maintain the consistency of a repository when shared by multiple cooperative software processes. In particular, we detail the concurrency control protocol we used to monitor the accesses made by multiple activities. This protocol can be tailored to particular application semantics by adding constraints on objects states. Thus, the COO framework provides support to complex and hybrid cooperation policies.

2 Overview and basic concepts of the COO approach

Cooperation during the development of a software system can be concretely supported by different but complementary medias such as meetings, mails, object sharing etc. In the following, we consider mainly asynchronous cooperation by object sharing, where users interact by invoking tools to modify shared objects. More precisely, we consider processes that cooperate by exchanging some partial results during their execution through a common repository. Synchronising these accesses to the repository is an important challenge in supporting the enactment of cooperating software processes.

2.1 Transactional approach

When processes cooperate and interact by exchanging some partial results during their execution, uncontrolled interactions can corrupt the consistency of the repository (e.g. dirty reads, lost updates). Efficient awareness and direct interaction between the users can reduce the risks and the frequency of these uncontrolled interactions. This is the approach generally supported by many cooperative systems [1]. However, when the tasks or the objects they manipulate are complex or numerous, or when there are many people with different roles involved in the collaboration, this approach is not sufficient to ensure consistency of the shared objects. Thus, a key issue is to provide the environment with concepts and mechanisms to maintain this consistency even in the case of multiple cooperative interactions between several parallel processes. We approach this problem by founding the correctness of execution on a safe transaction model that releases application programmers from the burden of interaction programming. Our approach promotes the use of transactional software processes, i.e. software processes whose execution is encapsulated within transactions. The transaction model we develop for this purpose uses cooperation policies based on a hybrid synchronisation strategy: a COO transactional software process is described by a fixed part, which applies for every process, and a variable part which varies from a process to another. More precisely, a process is defined by a hybrid policy that incorporates:

- a set of 'syntactic' rules that do not depend on the processes being synchronised, called the core protocol. This protocol, detailed in Section 3.2, encodes a set of cooperation patterns that applies to many software engineering activities. It is the main component used to synchronise the interactions between parallel processes.
- a set of process dependent rules which may vary from a process to another. These rules are expressed as constraints on software artifacts and process steps, and are detailed in Section 3.3.

In other words, the core protocol defines a sphere of security in which a set of well identified syntactic properties are preserved. This sphere is then restricted by the semantic rules in order to converge towards consist-
ent software artifacts. The main interest of the core protocol is that it largely simplifies the programming of process interactions.

2.2 Basic concepts to support cooperation
As introduced previously, cooperation in COO is supported by the coordination of activities that share data items stored in a repository. In this way, transactional processes cooperate by exchanging values of shared data items during their executions. These exchanges are controlled by the specialised protocol that monitors the accesses to the repository.

2.2.1 Object sharing: This is provided by a copy/modify/merge paradigm. Different transactions share a given object by owning a copy of that object in their private workspaces. These copies can be simultaneously modified by the different transactions without any interaction. They can also be transferred back to the repository and then accessed by the other transactions. Different copies of the same object can be re-conciliated by a merge operation initiated by one of the transactions that use the object. This operation occurs under the responsibility of users and produces a new object value that incorporates the local modifications and the changes carried out by the other transactions.

2.2.2 Intermediate results: These form a central concept in the COO framework. Any process result produced before the termination of the process and that correspond to uncommitted values are qualified as intermediate values, in contrast with final results that correspond to committed values produced at the termination of the process. The important difference between final and intermediate results is that while final results must respect process model rules, intermediate results can escape these rules: their production is not required to satisfy constraints, cooperation can start between processes on unfinished and maybe inconsistent objects. This is particularly useful for long duration, iterative processes such as software processes which can produce multiple successive versions of the same objects that can be used by other processes to start their own work.

2.2.3 Consistency: In this context it is offered by a protocol founded on the two following points:
1. The object values committed by a transactional process when it terminates respect some integrity constraints
2. A transactional process can access uncommitted values provided that it latterly accesses the corresponding committed values before Committing to its own modifications.

In other words, different cooperating processes can freely exchange uncommitted values during their execution, but the protocol ensures that committed values are always produced from committed values.

2.2.4 Hybrid policies: These are used to maintain this strategy for consistency. The respective roles of the two parts of a hybrid cooperation policy are the following. The core protocol is in charge of controlling interactions between parallel processes. It asserts general properties on these interactions. The process rules govern the execution of individual processes and complement the core protocol by putting constraints on the production of software items and by describing how a given process must locally react to interactions with other processes. In particular, they describe what actions have to be done or re-done when new object versions are accessed by the process, and how these new versions are integrated with the local software artifacts. This may result in merging object versions or in replying some sequences of actions. Of course, the human agent will be highly involved in this process.

3 Building cooperative development policies
This Section details in more depth, the construction of cooperative software development policies. We will first emphasise the core protocol and then discuss the addition of process rules.

3.1 Cooperation paradigms
To characterise different cooperative situations that may occur during the development of large software systems, we have pointed out three paradigms of cooperation within development activities. These paradigms and combinations of them cover a large set of cooperative executions representative of real development situations. An important requirement is that the core protocol must support all of them efficiently and correctly.

The 'client-server' paradigm corresponds to the case in which a process, hereinafter called the client reads several successive values produced by another process, the server. This basic paradigm can be combined to fit with more complex situations. Among them, the ones where the client-server dependency form a cycle, need to be underlined. Two cases of cyclic relationships can be distinguished. The case where all the relationships in the cycle concern the same object is called a 'cooperative write'. It corresponds to the case where two or more processes collaborate to the production of a common object by modifying and exchanging values of that object. The other case where the cycle of relationships concern different objects is called 'server-reviewer'.

3.2 Core protocol
The core protocol is a set of syntactic rules that organise data exchanges independently of any process semantics. Here, we give an abstract description of the protocol, regardless of any architecture. We assume that processes use two basic operations to access object values in the repository. The first is read which copies an object value from the repository into the process workspace: \( v = \text{read}(o_j) \) copies the software item \( o_j \) into the local process variable \( v \). The second operation is write: \( \text{write}(o_i, v) \) copies the value of variable \( v \) from the process workspace back to the software item \( o_i \) in the repository.

3.2.1 Some properties of the core protocol:
Before going in more detail about the protocol, two of its properties need to be highlighted. The first is that the protocol uses neither locks nor suspends processes. The consequence is that objects are always accessible and the protocol never hinders the users from working. The second property is that the protocol never imposes
the abortion of a process. This property prevents the loss of some work due to concurrency control reasons. We think that these two properties are very important in making this protocol effective.

3.2.2 Protocol rules: The protocol rules ensure properties on sequences of interleaved read and write operations issued by several parallel processes. It is based on making a clear distinction between final (committed) and intermediate (uncommitted) results. This is achieved by the use of a dedicated operation called \( IR \) – write to produce intermediate results. From that point, the strategy used by the protocol is the following: accesses a process makes to different objects are monitored; when the process tries to terminate, the protocol checks if it accessed a final value for all of these objects. If not, the termination is delayed and control returns to the user who has the responsibility to integrate the corresponding final value(s) in his or her work. The rules are:

1. A result produced before the end of a process 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 a process is a final result. All final results are produced atomically during the execution of the terminate operation.
3. A process that produces an intermediate result must produce a corresponding final result. The protocol collects all the objects that were ‘\( IR \) – written’ by the process and automatically produces a final result for each of them during the termination phase of the process.
4. If a process reads an intermediate result, then it must read the corresponding final result. The system maintains dependency relationships between processes to memorise the fact that a process has read an intermediate result from another. When a process \( A_1 \) reads the intermediate value of an object \( x \) produced by a process \( A_0 \), then a dependency \( A_1 \rightarrow A_0 \) is created. When the process \( A_1 \) reads a value of \( x \) and \( A_0 \) is terminated (i.e. when it has produced its final results), then the dependency is removed (\( A_1 \rightarrow A_0 \)).
5. A process cannot terminate if it is still dependent on another. If a process 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 process remains active.

This first set of rules maintains the general property introduced earlier for all client/server situations. However, we can verify that these rules in some cases lead to a deadlock: when the dependency set contains cycles, no process can terminate. These cases are very common since they correspond to cooperative write and server/reviewer situations. To support these situations, the protocol provides an additional set of rules to solve these deadlocks. These rules are dedicated to the termination of sets of cyclically dependent processes. The strategy is to make them terminate simultaneously when a consensus on the final value has been reached. Of course, the previous rules still apply between the process group and other processes. The rules that implement this strategy act as a cooperative commit protocol inspired from two-phase commit [2]. A complete and formal description of the core protocol can be found in [3, 4].

Two additional points need to be clarified about this protocol. The first one concerns object deletion. Suppose an object produced by a server and consumed by a client as an intermediate value that is latterly found as unnecessary and deleted by the server. On the one hand, no real final value can be produced, but on the other hand this deletion must be propagated to the client. We support these cases by the introduction of dummy object versions that indicate the deletion and that are produced as a final value for the deleted objects. The second point concerns processes involved in multiple cycles with different objects. The protocol in its actual form leads to a simultaneous termination of all the cycles that share one or more processes. An other solution consists of splitting the shared processes, but it is not actually implemented.

3.3 Adding process rules

Process rules are expressed as constraints on object states and on state sequences. These constraints are expressed in the form of logical temporal formulae and can be local to each process or global to the repository. Local constraints are used to govern the enactment of each process and to ensure the consistency of their final results. Global constraints are used to coordinate the execution of the different processes.

3.3.1 Constraints description: Constraints are described in COO with a subset of the temporal logic introduced in [5] restricted to only bounded future temporal operators. This ensures that constraints can be transformed into deterministic transition graphs with finite state sequences. Using this restricted form of logic allows us to describe both static constraints that express integrity conditions and dynamic constraints that express transition rules. The same algorithm is used in both cases to check their validity.

Constraints are expressed using first order formulae combined with the following temporal operators from, always, sometimes, before, and until. As we do not have sufficient space for a complete and formal description of these constraints, we just give here two representative examples. More on this topic can be found in [6, 7]. An integrity constraint can be described as:

- **from P1 always P2 until P3** which means 'when P1 becomes true, then P2 must remain true until P3 becomes true'. In constraint, a transition rule is expressed as:
  - **from P1 sometimes P2 before P3** which means 'if P1 become true, then P2 must become true before P3 becomes true'.

3.3.2 Integration with the core protocol: An important issue in COO is to integrate constraints checking with the core protocol. We use a violation prevention approach [5] to maintain the constraints. This means that the constraints are checked before the updates of a process are actually applied to its workspace and to its parent workspace. This technique follows the same philosophy as the core protocol since we do not need to abort processes because of constraints: in the case of a constraint violation, only the terminate operation of the process which wants to terminate is rolled back, but the process itself remains active.

Another important point is the treatment of the intermediate results which can escape constraints and are not expected to be consistent.
To simplify, we have identified three main cases which can prevent the termination of an activity:

- the process violates its local constraints: its goal has not been reached or some dynamic constraints are not in a valid state. In this case, the process must continue its execution until it reaches a valid state for termination (we suppose that a process can always reach its goal, or in other words, that users at their disposal the activities to reach this goal and know how to use them).

- the new state that would be produced would violate some constraint in the repository. In this case, the process can wait for a new state of the repository to try a new termination. It may also iterate its actions to reach its goal with new values that would be consistent in the repository context.

- some objects involved in the constraints the process can violate are in an intermediate state in the repository. In this case, the constraints cannot be evaluated. The process has to wait for the final values of these objects to terminate. This last case may lead to deadlocks when two processes are waiting for their respective final values. This is solved in the same way as in the core protocol. The two processes are grouped and freely collaborate in order to produce their results at the same time to enforce the constraints.

4 About the use of the COO approach

This Section gives some details about the use of the approach depicted in the previous Section in a particular environment. A short example is then developed to illustrate different points.

4.1 Using the COO approach

The COO approach described in Section 3 is sufficiently general to be used in different contexts. In particular, the core protocol and the constraints can be used to complement an existing software configuration management system such as CVS [8] or ClearCase [9]. These systems provide support for parallel work and sometime prevent the users from some mistakes (e.g. lost updates) but do not really offer a general consistency criterion as our approach does.

We have experienced the approach in the context of a hierarchical process-centred environment [10, 11]. In the COO system, processes can be hierarchically decomposed into sub-processes. Each process is described by a goal and pre/post conditions on the operations and sub-processes that can be started during its execution. In addition, each process corresponds exactly to one private workspace. Workspaces are thus also organised as a hierarchy and the workspace tree exactly maps out the process tree. The workspace of a process serves as the repository for its sub-processes and the transfers between a process and its sub-processes are supported by checkin/checkout operators. In that context, each workspace is managed by a particular hybrid policy, made of an instance of the core protocol, complemented by the constraints attached to the process owning the workspace. This hybrid policy defines rules on the use of the transfer operators that operate between the local workspace and the sub-workspaces. Consistency of the exchanges between processes is thus maintained over the global tree on a node per node basis.

4.2 Short example

A short example has been developed in order to illustrate how our approach works. This example is illustrated in Fig. 1. The purpose of this process model is to govern the production of a tested software configuration (box A). This task is decomposed into two sub-tasks: one is in charge of developing and testing individual modules (B) and the other is in charge of integrating them and testing the whole configuration (C). Different constraints and goals are attached to the processes; two of them are described here using the formalism introduced before:

- **C1:** process B: a software module needs to be individually tested when it has been modified:
  
  \[
  c1(m:module)\]
  
  always-from new-version(m) sometimes tested(m)

- **C2:** process C: when a new version of a module is integrated to the current configuration, a complete test of the configuration must be carried out:
  
  \[
  c2(m:module,c:soft-conf)\]
  
  from new-version(m) and integrated(m,c) sometimes tested(c) before terminated(c)

![Fig. 1 Small process model hierarchy](image)

![Fig. 2 Client/server situation](image)

Consider the schedule presented in Fig. 2 that corresponds to the example in Fig. 1: a B process serves several versions of a module to a C consumer (grey ovals correspond to read/write operations on which the protocol is evaluated).
In this execution, the two processes work simultaneously. The B process updates and produces a first intermediate version of its module. This version is immediately integrated with the configuration by the C process. It then tests the new configuration and tries to terminate (point 1). This termination is aborted by the core protocol because, at this point, the C process is dependent on the B process. At the same time, the B process has continued its work and again produced a new version of the module which is not used by the C process (since this is not mandatory on intermediate results). Finally, the B process terminates and produce a final value of its module. This value is integrated to the configuration by the process C which tries again to terminate (point 2). This termination is aborted by the constraints manager, due to constraint C2 which is not true at this point. Finally, process C runs the configuration test and terminates (point 3).

5 Related work and conclusion

The work presented in this paper is based on applying transaction technology to the support of cooperative processes in a software development environment with the objective of consistency maintenance. It proposes an advanced transaction model founded on a correctness criterion that does not impose isolation of parallel processes. This criterion is complemented by the use of constraints that allows to tune it to particular application semantics. This hybrid approach is original and combines the flexibility offered by constraints together with the security offered by the core protocol. This approach surpasses the limits of traditional transaction models based on serialisation and isolation [12] that have a limited applicability in the case of long-term, cooperative software development applications. Indeed, isolation prevents any interaction between processes from occurring during their execution. In contrast, these interactions are sought and favoured in the context of our cooperative processes.

Recent trends in transaction technology have focussed on extending conventional transaction models to enhance their applicability to new application domains [13]. In particular, several new approaches base correctness of execution on a semantic and application dependent criterion (e.g. [14, 15]). This approach is similar to the one used in many software engineering environments (e.g. [16, 17]) which delegate consistency maintaining to the process models. These approaches appear to be more difficult to implement in real use than ours since they suppose that all the possible interactions and synchronisation are described. As a consequence, process design becomes very intricate in this context. The actual state of the practice in software development corresponds to the use of software configuration management systems (e.g. RCS [18], CVS [8]). These systems provide support for parallel work with some minimal guarantees (e.g. no lost updates in the better cases) but do not offer a general strategy for consistency maintaining as our approach does.

Concerning future work, we are undertaking some important work on extending the core protocol with a goal of enlarging its applicability by taking into account more cooperation paradigms. In particular, we are thinking about adding a Split operator [19]. Indeed, splitting processes would be very useful in order to support partial commits and in the case of processes involved in multiple cyclic dependencies. Splitting them would allow the separation of the different cycles and their termination. The protocol and the properties it maintains have been formally described and verified using the ACTA framework [20] and VDM specifications. This will facilitate extensions of the model and will allow us to derive different protocols from the same kernel and to formally study their mutual integration [21].

This formal work will also serve as a basis to build a recovery strategy. At the moment, the only feature we provide for recovery from a failure (either physical or logical) is that every COO concept is represented in the repository as a persistent object. This is not sufficient to correctly handle software process failures. We are thus in search of more appropriate strategies to support software process recovery and evolution: forward recovery strategies as in the Contract model [15] are under consideration, combined with the splitting and the use of partial commits to confine recovery only on faulty objects.

Finally, another point which is well engaged concerns distribution in our framework. At this time, the COO system works on a LAN network. Our goal is to extend the system to allow the connection of different sites through a large scale network, such as Internet.

6 References

1 BAECKER, R.M.: 'Readings in groupware and computer-supported cooperative work' (Morgan Kaufmann Publishers, 1993)
2 MOLLI, P.: 'Environnements de développement cooperatif', Thèse en informatique, Université de Nancy I - Centre de Recherche en Informatique de Nancy, 1996
5 LIPECK, U.W., and FRENJ, D.: 'Construction of deterministic transition graphs from dynamic integrity constraints' in VAN LEEUWEN, J. (Ed.): 'Graph theory concepts in computer science' 1989, LNCS 344, pages 166-179
7 SKAF, H.: 'Une approche hybride pour gérer la cohérence dans les environnements de développement coopératif'. Thèse en informatique, Université de Nancy I - Centre de Recherche en Informatique de Nancy, 1997
13 ELMAGARMD, A.K. (Ed.): 'Database transaction models for advanced applications' (Morgan Kaufman, 1992)


