CAHIER DU LAMSADE

Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la Décision
(Université Paris-Dauphine)
Unité de Recherche Associée au CNRS ESA 7024

A COORDINATION MECHANISM TO PREVENT THE VIOLATION OF DISTRIBUTED CONSTRAINTS

CAHIER N° 164
septembre 1999

Marie-José BLIN [1]
Françoise FABRET [2]

received: February 1999.

[1] *LAMSADE, Université Paris-Dauphine, Place du Maréchal De Lattre de Tassigny, 75775 Paris Cedex 16, France (blin@lamsade.dauphine.fr).*

[2] *INRIA, 78153 Le Chesnay, France (Francoise.Fabret@inria.fr).*

CONTENTS

Un mécanisme de coordination pour éviter la violation de contraintes distribuées

Résumé

Certaines applications informatiques complexes nécessitent la contribution de plusieurs équipes spécialisées dont les travaux doivent respecter un ensemble important et évolutif de contraintes sur les données ainsi que des échéances serrées. L'utilisation de l'approche transactionnelle traditionnelle pour préserver la cohérence des données peut conduire à défaire et refaire certaines parties du travail et entrainer une perte de temps et d'argent. Dans certains cas, cette situation n'est pas acceptable. Nous proposons une approche alternative consistant à organiser le travail à partir des contraintes sur les données de telle sorte que lorsque le travail est fini les contraintes sont nécessairement respectées. Un modèle conceptuel de travail coopératif et un langage de définition de protocole de travail sont présentés. Nos propositions sont appliquées à la spécification de systèmes informatiques à base de composants.

Mots clés: coordination, collecticiel, workflow, contraintes de compatibilité de données

A Coordination Mechanism to Prevent the Violation of Distributed Constraints

Abstract

Some applications present specific characteristics: work distributed between several teams, a lot of changing data constraints to respect, short time delays. The use of the traditional transactional approach to preserve the data consistency may lead to undo and redo some parts of the work and so, may induce a waste of time and a waste of money. In some cases, this situation is not acceptable. We propose an alternative approach which consists of organizing the work from the constraints, so that there is no need of a constraint checking phase. A conceptual model of cooperative work and a language to define work protocol are presented. Our proposition is applied to the specification of component-oriented systems.

Key words: coordination, groupware, workflow, distributed compatibility constraints

1 Introduction

A lot of computer systems are composed of a great number of elements of different nature, for example, programs, hardware, manuals and documents which are progressively stored in a component base. More and more often, these "component-oriented" systems are built by putting together existing components which are, if necessary, adapted to the specific environment in which the system will be integrated. These adaptations lead to the creation of several versions of some components and to the improvement of the reusable component base. The building of a new system is performed in two steps. First, each element of the system is specified either by reusing the specifications of an existing component version, or by creating new specifications. Second, every specified component version is entirely built. Generally, components of the reusable component base are not independent, for example, component x uses component y. Thus, it is imperative that the component base stores not only the component themselves, but also compatibilities and incompatibilities between component version specifications. These compatibility descriptions constitute a constraint base.

Due to the great number of components to put together and to their diversity, the building of a new system requires a wide range of specialized skills. This construction is achieved by several teams coordinated by a person who plays the role of integrator. The integrator overall knows the set of components in the component base and the environment of the future system. Each team is in charge of a subset of components and of the building of a part of the system (a sub-system). Due to time constraints, as many tasks as possible are simultaneous. However, all constraints between component versions have to be respected: constraints between component versions managed by a single team as well as those between objects managed by different teams.

This paper focuses on the problem of constraint enforcement. Thus, we are only interested in the specifications phase of the systems. The traditional concept of transaction has been extended to support long-duration activities and to solve the problem of constraint enforcement in design applications. [Bernstein, Newcomer, 1998] discusses several advanced transaction models, e.g.: Split-Joint Transactions [Pu, 1988], Flexible Transactions [Elmagarmid and al., 1990], Acta [Chrysanthis, Ramamritham, 1991], Sagas [Garcia-Molina and al., 1991], Contract [Waechter, Reuter, 1992], Open Nested Transactions [Weikum, Schek, 1992].

The main idea of these models is to relax the well-known properties of atomicity, consistency, isolation and durability of the conventional transaction model which would make data unavailable for long time (since transactions may run for days or weeks). Most advanced transaction models allow transactions to be composed of other nested transactions forming a transaction tree. Results of nested transactions are visible to the other sibling transactions or even to external ones. Some models allow the specification of dependencies between transactions. Acceptable states for termination of a transaction in which some sub-transactions may be aborted may be specified by the designer. When concurrency conflicts or failures happen, the compensation concept is used in place of the standard rollback: inner transactions are associated with compensating transactions which leave the database in a consistent state.

Advanced transaction models were developed from a database point of view. Their main concern is the preservation of database consistency. If constraints are violated, work is to

be compensated. That leads to a waste of time and thus to a waste of money. In certain situations, this solution is not easily acceptable. One alternative to transactional models is to organize the work so that there is no need of a constraint checking phase. Instead of verifying data consistency to ensure the consistency of work being performed on the data, work consistency is controlled to be sure of data consistency.

Our solution follows this approach. It is in line with cooperative work studies and it proposes a computational coordination mechanism for distributed specifications. This mechanism fits the definition given in [Schmidt, Simone, 1996]. It consists of a protocol which coordinates the work and an artifact in which the protocol is objectified. The protocol stipulates and mediates the articulation of the cooperative work. Changes to the state of the protocol induced by one actor of the cooperative work (a person or a process) are conveyed to the other actors. The execution flow of the protocol may be disturbed by user interventions like stopping, restarting or iterating an activity.

Our work integrates both workflow [Jablonski, Bubler, 1996] concepts and what we call advanced groupware concepts. Worflows assist business processes which are generally composed of many automatic and manual individual tasks [Kim, Paik, 1997]. Groupware offers supports for managing communications and shared environments between participants of a group working on a common goal (electronic mail, video-conferencing, shared document editing, awareness mechanisms). Advanced groupware concepts concern the definition of cooperative work modes like, for example, iterative conversations between participants, discussions where each participant alternately develops its propositions and the different propositions are communicated to all the group, negotiation [Munier, Shakun, 1988].

The objective of a cooperative work management system is to assist the user in the modeling of cooperative work and in its execution. This paper is only concerned with the modeling aspect. It describes a model that represents cooperative work modes and flows of tasks based on these work modes. As an example, the model is applied to the component-oriented system building application.

The remainder of the paper is organized as follows: a real system (CORSSE) and its management are described in section 2 to well explain the needs of teams which build component-oriented systems; sections 3 and 4 describe the formal data and coordination models respectively which are illustrated by examples extracted from CORSSE; in section 5, we apply these formal models to CORSSE and show how an adapted articulation of the work leads to constraint enforcement; section 6 exposes some existing cooperative work models; section 7 concludes the paper and presents future work.

2 Motivating example

A concrete example CORSSE (Component-ORiented System Specification Environment) inspired from a real case is presented to explain the needs of teams that build component-oriented systems,. First, the component and constraint base are described. Then, the creation process of a new system is detailed.

The component base of CORSSE. CORSSE is devoted to distribution management. It allows the construction of specific systems for different types of stores from small supermarkets to hypermarkets. The systems may be installed in different countries with different management rules, and of course different languages. The different customers of

the systems may have specific standards, specific management procedures and needs, and different technical environments.

A distribution management system is composed of software, hardware, documentation and services. Each of these parts contains a great number of components. For example, hardware contains specific or/and general components and additional components, which are used to build or to test the specific hardware. These additional components may be general or ad hoc for the system. Specific and general components are servers, workstations, cash registers and informative tills. Each component may be composed of other components, for example, a cash register is composed of cables, one printer and one or several screens. Another example concerns the software: in addition to specific programs, the system may include general software like a documentation formatter, print tools, test tools, a database management system, a specific compiler, or a graphic user interface generator.

Systems differ from one another by their components and also by the versions of these components. When CORSSE was created, it concerned few systems, at most one for each customer. Progressively, new versions of hardware and software components, of documentation and new systems were created, for new customers, for maintaining the installed systems or for taking into account new needs or new environments. All the component and their versions used in the different systems compose the reusable component base of CORSSE. This base is incrementally built and continually evolves.

The constraint base of CORSSE. Compatibility constraints may exist between versions of different components and a system must respect them. An example of such constraints is given below.

The following object versions are compatible:

1. Version NCR of the scanner and version AU33_ND Board 3 of the program TCF

2. Version PEACH of the scanner and version AU33_SD Board 3 of the program TCF

3. Version PEACH of the scanner and version AU33_AIF Board 3 of the program TCF

4. Version PEACH of the scanner and version AU32_N Board 3 of the program TCF

5. Version NCR of the scanner and version AU32_S Board 3 of the program TCF

6. Version NCR of the scanner, versions DASSAULT, ICL and *new*[1] of the check reader, versions DASSAULT 1200 bauds and ICL 1200 bauds of the card-reader

7. Version NCR of the scanner, version FPI and *nil*[2] of the keyboard

8. Version PEACH of the scanner, version DASSAULT and *new* of the check reader, version GEMPSY 9600 bauds of the card reader, version FPI of the keyboard

9. Version GEMPSY 9600 bauds of the card reader and version PS2 of the keyboard

[1]The symbol *new* means that a new version of the object may be specified.
[2]The symbol *nil* means that the object may be absent.

10. Versions 7.04 and 7.05 of the program PA and version 3.19 of the function 3GLIF

As new component versions are created and added to the reusable component base, their compatibility with the other component versions of the base is tested. Consequently, the compatibility constraint set is updated, most of the time by adding new constraints.

Creation of a system. The different components of the reusable component base are managed by several specialized teams which may work in different places and different countries. For example, teams located in a central place manage the components common to all the customers, and one or several teams in each country are in charge of the components specific to the customers of the country.

The construction of a new system follows a pre-established manufacturing process (Figure 1) during which each team concerned creates a part of the system (subsystem) and verifies compatibility constraints among the subsystem components. Then, a specific team integrates all the parts, verifies compatibility constraints between them and tests the new system on a platform identical to the customer technical environment. Finally, the system is installed at the operational site.

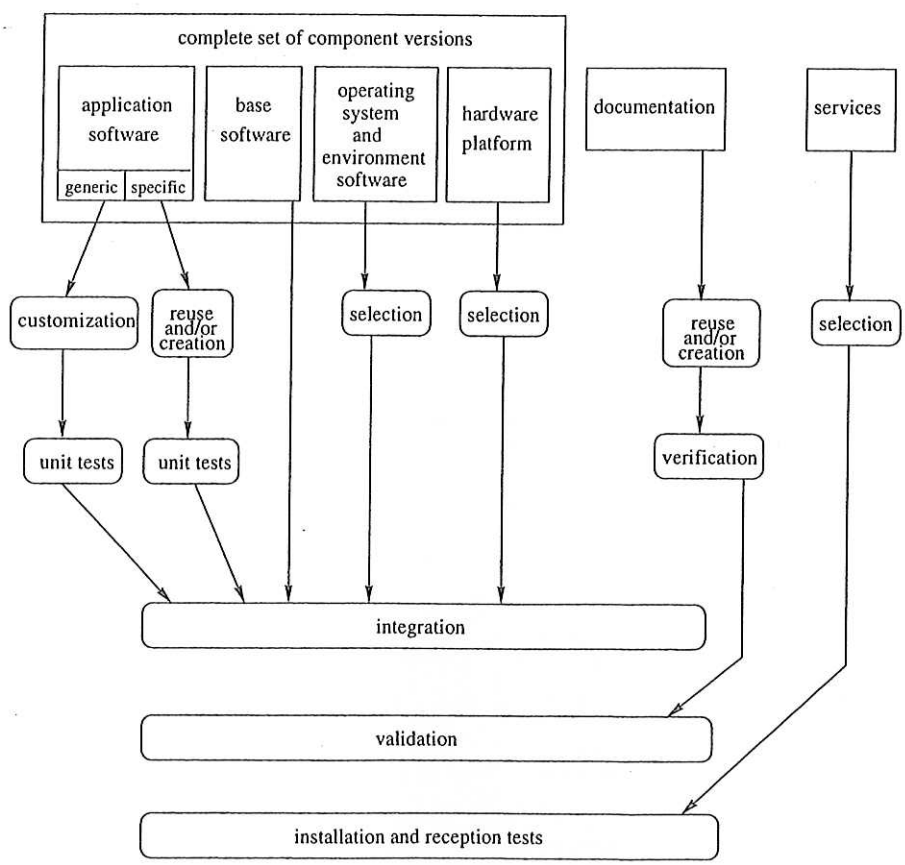

Figure 1: the creation process of a new system

3 Formal data model

The construction of a system is called a *project.*A project uses a subset of the component base (which we call the project component base) and an associated set of compatibility constraints. This compatibility constraint set is computed from the general set of constraints according to the characteristics of the system to be built. This section formalizes the different notions intuitively introduced in section 2 by the concepts of configuration, reduced constraint set, constraint violation and system. One additional concept (constraint family), used in section 5, is also defined. The different concepts are illustrated by an example extracted from CORSSE.

3.1 Configuration

A configuration is a set of multiversion objects as defined in the object-oriented database system O2 [Ardent, 1998]. Formally, a configuration is defined by a triplet $< \mathcal{O}, \mathcal{V},$ *version*$>$ where:

- $\mathcal{O}$ is a set of objects,
- $\mathcal{V}$ is a set of object versions,
- *version* is a mapping from $\mathcal{O}$ to $2^{\mathcal{V}}$ which associates each objet o in $\mathcal{O}$ with a subset of $\mathcal{V}$ representing the set of versions of o. The version mapping satisfies the exclusive property, i.e., for any pair (o, o') of objects, *version* (o) and *version* (o') are disjoint.

Intuitively, $\mathcal{V}$ represents all the known versions of objects in $\mathcal{O}$ and *version* provides the association between the versions and the objects.

Totally defined configuration. A configuration is totally defined if for any o in $\mathcal{O}$, *version* (o) is a singleton.

3.2 Constraints

Constraint properties. A constraint set specifies four type of properties of the components and component versions:

1. transitivity or compatibility between versions of different components
2. commutativity or equivalence of versions of a component
3. possible absence or mandatory presence of components in the systems
4. possibility to insert new versions of components in the systems

For example, the properties specified in the sub-set of constraints 1, 5, 6 and 7 of section 2 are:

- All the component versions cited in the four constraints are compatible
- Versions AU33_ND Board 3 and AU32_S Board 3 of the program TCF are equivalent
- Versions DASSAULT, ICL and any new other version of the check reader are equivalent

- Versions DASSAULT 1200 bauds and ICL 1200 bauds of the card reader are equivalent
- Version FPI and any new other version of the keyboard are equivalent
- If one of the components, scanner, check reader, card reader and program TCF is present in the system in one of the cited versions, the other components have to be also present in the appropriate versions. The keyboard may be absent (specified by *nil*).
- A new version of the check reader may be created (specified by *new*).

So, to highlight the properties of the subset of constraints (1, 5, 6, 7), the subset will be rewritten as follows:

Version NCR of the scanner, versions AU33_ND Board 3 and AU32_S Board 3 of the program TCF, versions DASSAULT, ICL and *new* of the check reader, versions DASSAULT 1200 bauds and ICL 1200 bauds of the card-reader, version FPI and *nil* of the keyboard

Intuitively, we can see that the rewritten constraint subset is derived from the original sub-set by grouping all compatible component versions and all equivalent versions.

The rewritten constraint set is called *a reduced constraint set* which is formally defined as follows.

Reduced set of constraints. Given a set of constraints $\mathcal{R}$ and a constraint c in $\mathcal{R}$, $\mathcal{O}_c$ denotes the set of objects in c and $version_c$ (o) with o in $\mathcal{O}_c$, the version set of object o involved in c.

$\mathcal{R}$ is a reduced set of constraints if, $\forall$ (c_i, c_j) $\in \mathcal{R}$,

- $\mathcal{O}_{c_i}$ and $\mathcal{O}_{c_j}$ are disjoint, or
- $\mathcal{O}_{c_i} = \mathcal{O}_{c_j}$ and there exists at least two objects o and o' in $\mathcal{O}_{c_i}$ such that $version_{c_i}$ (o) $\cap$ $version_{c_j}$ (o) $= \emptyset$ and $version_{c_i}$ (o') $\cap$ $version_{c_j}$ (o') $= \emptyset$

Appendix A provides the updating algorithm of a reduced set of constraints when a new constraint is added.

Constraint family. Given a reduced set of constraints $\mathcal{R}$, a constraint family $\mathcal{F}$ is the subset of $\mathcal{R}$ where all constraints refer exactly the same set of objects. Formally $\mathcal{F}$ is such that:

$\forall$ c_i in $\mathcal{F}$ and c_j in $\mathcal{R}$ - $\mathcal{F}$, $\mathcal{O}_{c_i}$ and $\mathcal{O}_{c_j}$ are disjoint.

The set of objects occurring in a constraint family $\mathcal{F}$ is called an *object family* noted $\mathcal{O}_{\mathcal{F}}$.

Constraint violation. Let $\mathcal{R}$ be a reduced set of constraints, and c a constraint in $\mathcal{R}$. Let a configuration $\Sigma = < \mathcal{O}, \mathcal{V}, version >$, then Σ violates c if:

- either some objects are missing in Σ with respect to c, or
- two of the objects of Σ are in incompatible versions with respect to c.

Formally, given a set of constraints $\mathcal{R}$ and a constraint c in $\mathcal{R}$, a configuration $\Sigma = < \mathcal{O}, \mathcal{V}, version >$ violates c if one of the two conditions below is verified:

1. $\mathcal{O}_c \cap \mathcal{O} \neq \mathcal{O}_c$
2. $\mathcal{O}_c \cap \mathcal{O} = \mathcal{O}_c$, and

   $\exists$ (o_i, o_j) in c with i $\neq$ j, such that *version* (o_i) $\subseteq$ *version*$_c$ (o_i) and *version* (o_j) $\not\subseteq$ *version*$_c$ (o_j)

The absence of an object in Σ is assumed to be indicated by the presence of the version *nil* of this object in the configuration.

We will say that Σ is a *consistent configuration* with respect to $\mathcal{R}$ if Σ does not violate any constraint in $\mathcal{R}$.

3.3 System

Given a reduced set of constraints $\mathcal{R}$, a *system* is a configuration totally defined and consistent with respect to $\mathcal{R}$.

3.4 Example

Let be a configuration Σ with:

$\mathcal{O}$ = {program TCF, scanner, card reader, check reader}
$\mathcal{V}$ = { AU33_SD Board 3, PEACH, DASSAULT, GEMPSY 9600 bauds }
version (program TCF) = {AU33_SD Board 3}
version (scanner) = {PEACH}
version (check reader) = {DASSAULT}
version (card reader) = {GEMPSY 9600 bauds}

Given the following set of constraints $\mathcal{R}$ ={c, c3, c4} with:

c = { (program TCF, [AU33_SD Board 3, AU32_N Board 3]), (scanner, [PEACH]), (check reader, [DASSAULT, *new*]), (card reader, [GEMPSY 9600 bauds]), (keyboard, [FPI]) }
c3 = {(program TCF, [AU33_ND Board 3]), (scanner, [NCR]), (check reader, [ICL, DASSAULT, *new*]), (card reader, [ICL 1200 bauds, DASSAULT 1200 bauds]), (keyboard, [FPI, *nil*]) }
c4 = { (program PA, [7.04, 7.05]), (function 3GLIF, [3.19])}

$\mathcal{R}$ is reduced because:

- $\mathcal{O}_c$ and $\mathcal{O}_{c4}$ are disjoint, and
- $\mathcal{O}_{c3}$ and $\mathcal{O}_{c4}$ are disjoint, and
- $\mathcal{O}_c = \mathcal{O}_{c3}$ and there exists two objects (program TCF and scanner) in $\mathcal{O}_c$ such that *version*$_c$ (program TCF) $\cap$ *version*$_{c3}$ (program TCF) = $\emptyset$, and *version*$_c$ (scanner) $\cap$ *version*$_{c3}$ (scanner) = $\emptyset$

The configuration Σ violates the constraint c because the object keyboard does not exist in this configuration. Thus, Σ is inconsistent with respect to $\mathcal{R}$.

Two constraint families $\mathcal{F}1$ and $\mathcal{F}2$ are defined over $\mathcal{R}$:

$\mathcal{F}1 = \{c, c3\}$ with

$\mathcal{O}_{F1}$ = {program TCF, scanner, check reader, card reader, keyboard}

$\mathcal{F}2 = \{c4\}$ with

$\mathcal{O}_{F2}$ = {program PA, function 3GLIF}

Consider the reduced set of constraints $\mathcal{R}$. Let the configuration Σ' contain the same objects and object versions than Σ plus the object keyboard in its version FPI. Σ' is a system.

4 The coordination Model

Complexity and instability of the component and constraint bases, and also the turnover of the teams make a team unable to know all the constraints, especially those involving elements managed by other teams but also those involving the elements managed by the team itself.

The composition of the teams and the modes of coordination between them depend on the teams themselves and on the general characteristics of the system to be built. Teams work differently if the system has to be quickly and reliably built or if it constitutes a new prospective platform.

So, a coordination model must obey the following requirements:

- to allow the driving of the relationships between teams according to the constraints,
- to help the teams to respect the constraints among the components they build,
- to be easily modified when changes occur in the component and constraint bases,
- to be easily specified according to the teams, their work and their composition, to the properties of the system and to the available time for building it.

This section presents a formal coordination model which answers to these needs. First, the principles of the driving of the relationships among teams are described. Then, the concepts of working modes and coordination protocol are defined and illustrated by examples extracted from CORSSE.

4.1 Principles of the driving of the relationships between teams

In what follows, a team is an agent (possibly human) in charge of an object or of a group of objects specified together. Of course, a team may be actually a group of agents, which will functionally be considered as only one.

Teams do not directly communicate with each other and a team does not necessarily know the other teams. A coordinator is in charge of the teams coordination and drives the task sequencing among the teams (Figure 2). It has information about the order in which the teams have to work (the coordination protocol described later).

A coordinator and the teams communicate by messages. All the messages exchanged between the coordinator and the teams from the beginning of the work are recorded in a message history log.

Messages consist of a message identifier and a message value. A message sent by the coordinator to a team contains information about the work to do. When the team has finished the work, it sends a message to the coordinator containing the result of its work. At this point, the coordinator searches for the next teams to activate; to do this, it uses the coordination protocol and information about its execution and the message history. Then, for each of the selected teams, it calls an application-dependent function which computes the message to send to it. This function uses the message history and data specific to the application (for example, the constraints in CORSSE). Examples of functions are provided in the next subsection.

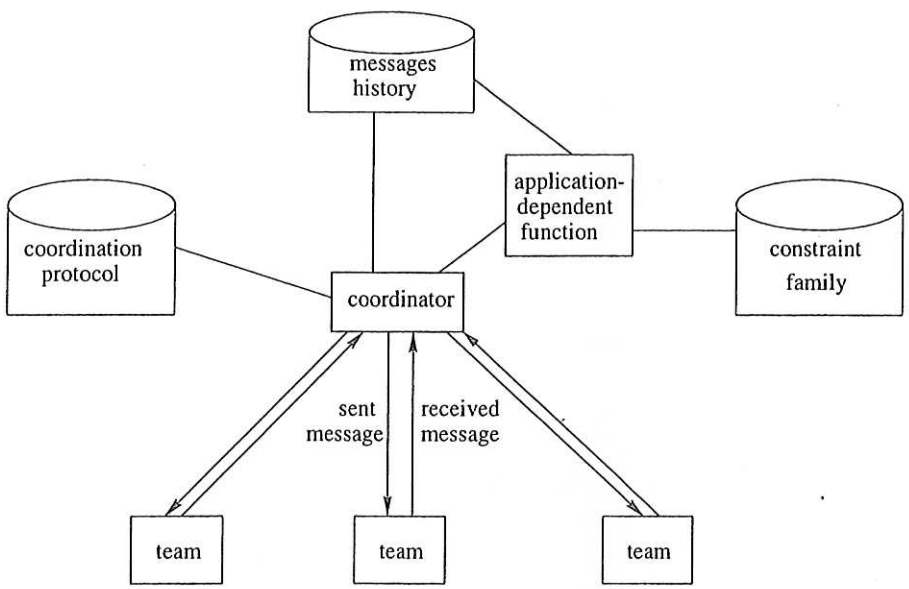

Figure 2: communication between a coordinator and teams

4.2 Working mode

The work required of the teams may be different according to the teams themselves, the general objectives and the state of the progress of the global work. Consequently, several types of messages may be sent to the teams in which case several functions will be implemented, one function for each type of message to compute.

A working mode defines the messages received and sent by the teams in a specific collaboration situation. It is associated with one application-dependent function. The coordination

protocol described later in this section provides the workflow among teams.

Formally, a working mode is defined by a triplet <identifier, context, function name > where:

- identifier is the name of the working mode,
- context is a data structure used by the application-dependent function. This structure contains the description of the messages sent to the teams and received from them. It may also contains the declaration of specific variables,
- function name indicates an application-dependent function which computes messages for the teams. This function takes the identifier of a team and returns a message to send to it.

A working mode is defined a priori, independently of the teams to which it will be applied and of the systems to be built. Two examples of working modes are presented below.

Example 4.1 The working mode "discussion". *In this mode, a team receives propositions about possible versions of the objects it is in charge of. It computes and sends its choices. A weight is affected to each of the choices to represent the team's preferences. We suppose that the teams accept the propositions they receive. The working mode "discussion" is used when teams have to discuss together before specifying objects.*

The definition of the working mode "discussion" is:

identifier: *discussion*

context:
sent_message: {< *proposition message, configuration* >}
received_message: <*team identifier; set of weighted choices* > *with*
weighted choice:= < *a configuration, a weight* >

compute_discussion_message *(set of objects): sent_message*

Sent_message provides the history of the teams' propositions from the beginning of the discussion and for each of the propositions, the related computed configuration of the team.

Received_message contains pondered solutions computed from the propositions received.

Compute_discussion_message function selects, in the message history, the propositions sent by the different teams from the beginning of the discussion. It then uses the constraints to compute, for each proposition, the possible versions of each object identified by input parameters. The computed message contains each proposition and the associated computed configuration and the structure of the expected response (received_message structure). This structure will be used by the team to present the reply.

Example 4.2 The working mode "decision". *In the preceding example, a team receives several possible configurations and makes pondered choices from the propositions. In the working mode "decision", a team receives only one proposition (a configuration) and chooses one version of each object from it.*

The definition of the working mode "decision" is:

identifier: *decision*

context:
sent_message: <configuration available >
received_message: < a system > where
each object version belongs to the configuration available to the team.

compute_decision_message *(set of objects): sent_message*

Sent_message provides to the team the configuration inside which one version for each object will have to be chosen.

Received_message specifies the version chosen for each object.

The function compute_decision_message uses the constraints to compute the possible versions of each object identified by input parameters. The computed message contains the available configuration and the structure of the expected response.

4.3 Coordination protocol

A coordination protocol defines the working modes to apply to each team and the relationships between the teams. A formal definition of a protocol is presented below. Appendix B provides the protocol execution algorithms.

Formal definition of a protocol. A protocol for a set of teams $\mathcal{E}$ is defined by a sextuplet ¡ $\mathcal{T}$, $\mathcal{V}$, $\mathcal{F}\nabla\dashv\}$, $\mathcal{I}]$, $\mathcal{E}]$, $\mathcal{D}$ ¿ where:

- $\mathcal{T}$ is a set of pairs $< C, e >$ where C is a working mode and e is a team of $\mathcal{E}$. Each of these pairs will be called a task in the following,
- $\mathcal{V}$ is validity rule which provides the possible and/or mandatory sequencing of working modes for each team,
- $\mathcal{F}\nabla\dashv\}$ is a set of fragments recursively built from fragments or tasks assembled by applying one assembly schema: the serial assembly or the parallel assembly,
- $\mathcal{I}]$ is a set of control expressions which, for each fragment, specify iteration conditions of the fragment,
- $\mathcal{E}]$ is a set of control expressions which, for each fragment, specify exit conditions of the fragment,
- $\mathcal{D}$ is a data structure used in the control expression of the fragments. This structure may also contain the declaration of specific variables.

The simplest fragments are composed of tasks assembled in a serial or in a parallel manner. More complex fragments are obtained by recursively applying the serial assembly and/or the parallel assembly to fragments. In a serial assembly, the set of components is ordered

and each of them is executed according to its order. In a parallel assembly, the components are executed in parallel.

The execution of a fragment is implicitly iterative and iterations are controlled by the definition of iteration conditions. The definition of exit conditions provides conditions of the fragment execution termination upon exceptions.

Example 4.3 [

```
[
    [(discussion e1), (discussion e2) (iterations <3) (enddis1)] |
    [(discussion e3), (discussion e4) (iterations <3) (enddis2)]
(iterations < 1) ()],
[(decision e1), (decision e2), (decision e3), (decision e4)(iteration < 1) () ]

(iterations < 1) (stop)]
```

This example shows a protocol composed of two fragments assembled in a serial manner.

1. *The first fragment is composed of two parallel discussion fragments: teams e1 and e2 discuss in a serial manner during three iterations. In parallel, teams e3 and e4 also discuss in a serial manner during three iterations. The discussion between e1 and e2 is exceptionally terminated if the message "enddis1" arrives. Similarly, the discussion between e3 and e4 is exceptionally terminated if the message "enddis2" comes.*

2. *At the end of all, a decision fragment starts in which each team serially decides. This fragment is not iterative and nothing can interrupt its execution.*

The protocol execution is exceptionally terminated if the message "stop" comes.

Valid, complete protocol. Given a set of teams $\mathcal{E}$ and a protocol P, P is valid if, for each team occurring in the protocol, the validity rule is satisfied. It is complete with respect to $\mathcal{E}$ if each team of $\mathcal{E}$ appears in at least one task of the protocol.

Example 4.4 *Consider the validity rule of the form*

discussion* decision

where symbol $*$ represents the standard Keene operator. Intuitively, this rule specifies that any team participates in one and only one decision task that is possibly preceded by one or more discussions including the team (the discussion is optional). Let us remark that this validity rule implies that any basic fragment including one decision task will be executed only once.

Given the protocol of example 4.3. The succession of tasks for each team is:

e1: discussion discussion discussion decision
e2: discussion discussion discussion decision
e3: discussion discussion discussion decision
e4: discussion discussion discussion decision

As the validity rule is satisfied for each team occurring in the protocol, the protocol is valid.

Consider the set of teams $\mathcal{E}$ = {e1, e2, e3, e4, e5}. The protocol is not complete with respect to $\mathcal{E}$ because it contains no task including e5.

5 The Compute_Protocol Algorithms

In the previous section, a very general coordination framework is proposed. In what follows, this framework is used for specifying coordination protocols in the context of CORSSE application.

Our goal is to statically construct a valid and complete protocol for a constraint family. The protocol construction uses two working modes and one validity rule. The working modes are *discussion* and *decision* as set in examples 4.1 and 4.2. Thus, the tasks are of the form <discussion set of objects> or <decision set of objects>. The set of objects involved in a task is handled by one team. At execution time, the input messages will contain the set of objects to be handled (plus additional information on these objects), and the output messages will contain the results of the work of the teams.

The validity rule is of the form discussion* decision as in example 4.4. The completude of a protocol is verified in relation to the set of objects of the constraint family as in the example below.

Example 5.1 *Take the family constraint of example 3.4: $\mathcal{F}1$ = {c, c3} with:*

c = { (program TCF, [AU33_SD Board 3, AU32_N Board 3]), (scanner, [PEACH]), (cheque reader, [DASSAULT, new]), (card reader, [GEMPSY 9600 bauds]), (keyboard, [FPI]) }
c3 = { (program TCF, [AU33_ND Board 3]), (scanner, [NCR]), (cheque reader, [ICL, DASSAULT, new]), (card reader, [ICL 1200 bauds, DASSAULT 1200 bauds]), (keyboard, [PS2]) }

and

$\mathcal{O}_{F1}$ = {program TCF, scanner, cheque reader, card reader, keyboard}

Let P be the protocol:

[[(discussion {scanner, cheque reader}), (discussion {card reader}) (iterations < 3) ()],
[(decision {scanner, cheque reader}), (decision {card reader}), (decision {keyboard}). (decision {program TCF}) (iterations < 1) ()](iterations < 1) ()]

This protocol contains six tasks gathered into two serial fragments. The tasks are:
<discussion {scanner, cheque reader}>,
<discussion {card reader}>,
<decision {scanner, cheque reader}>,
< decision {card reader}>,
<decision {keyboard}>, and
<decision {program TCF}>.

As each object of F1 occurs in at least one task of P, the protocol is complete.

We propose two algorithms (basic and optimized compute_protocol algorithms) that statically compute a coordination protocol for a constraint family F. The algorithms take a protocol P that is not necessarily complete (it may be empty), a set of additional tasks T and two default fragment control expressions (Ie for iteration conditions and Ee for exit conditions). Our goal is to produce a complete and valid protocol for F by completing P with the tasks in T. Let us remark that, depending on P and T, the construction is not always possible. The feasibility condition is that every object of F participates in a decision task occurring in P or in T. In what follows we assume that this condition is satisfied.

Generally, there are several ways to complete P with respect to T. So, the algorithms use heuristics on task scheduling. Both algorithms implement different heuristics but apply the following common rules: 1) the discussion tasks of T are serially assembled into one discussion fragment which precedes any decision task, 2) any decision task which only includes non discussed objects cannot precede one task including at least one discussed object (decisions are brought closer to discussions).

Each algorithm is briefly explained below and illustrated by an example. Details can be found in Appendix C. Inputs to the algorithms are: a valid protocol P for a constraint family, an ordered set of discussion tasks *Disc*, a set of decision tasks *Dec*, a default iteration expression Ie, a default exit expression Ee. Optimized compute_protocol algorithm uses, additionally, the constraint family F. The output of both algorithms is a valid and complete protocol P'.

Basic compute_protocol algorithm

In this algorithm, all the tasks and fragments are serial. First, a discussion fragment composed of the ordered set of discussion tasks *Disc* is appended to the input protocol P. Then, a decision fragment is added for each task of *Dec* involving an object occurring in *Disc*. The protocol is achieved by a decision fragment for each task in *Dec* involving objects which do not occur in any task of *Disc*. Each fragment of P is controlled by the default iterative and exit expressions.

Example 5.2 *Take the family constraint of example 3.4. Consider:*

- *P an empty initial protocol*
- *the set of tasks Disc = { discussion {scanner}, discussion {card reader} }*
- *the set of tasks Dec = { decision {scanner, cheque reader}, decision {card reader}, decision {keyboard}, decision {program TCF} }*
- *the default iteration expression Ie = (iterations < 1)*
- *the default exit expression Ee = ()*

The use of basic compute_protocol algorithm gives the following final valid and complete protocol:

[

[(discussion {scanner}), (discussion {card reader}) (iterations < 1) ()],
[(decision {scanner, cheque reader}) (iterations < 1) ()],
[(decision {card reader}) (iterations < 1) ()],
[(decision {keyboard}) (iterations < 1) ()],
[(decision {program TCF}) (iterations < 1) ()]

(iterations < 1) ()]

Optimized compute_protocol algorithm

With this algorithm, parallel assembly is used as much as possible. As in the basic compute_protocol algorithm, a serial discussion fragment composed of the ordered set of discussion tasks *Disc* is first appended to the input protocol P. Then, decision fragments are added following two considerations: 1) tasks involving objects occurring in a discussion task are put before tasks involving objects which do not occur in any discussion task, 2) decision tasks are added to the protocol according to their descending probability of reducing the number of constraints between remaining objects.

The selection of the most discriminating object of a set of objects may be derived using expected values as in the example below.

Example 5.3 *Let F be a constraint family containing 6 constraints and the objects a and b. The object a occurs in 3 versions (a_1, a_2 and a_3) in F and object b in 2 versions (b_1 and b_2). The table below provides:*

- *in the first column, each object version,*
- *in the second column, the number of constraints of F in which the object version occurs, and*
- *in the third column, the choice probability of the version of object.*

a_1	*3*	*0.7*
a_2	*1*	*0.2*
a_3	*2*	*0.1*
b_1	*3*	*0.5*
b_2	*3*	*0.5*

The mean number of constraints that we can hope to eliminate after the choice of one version is:

*for the object a: 3*0.7 + 1*0.2 + 2*0.1 = 2.4 constraints*
*for the object b: 3*0.5 + 3*0.5 = 3 constraints*

Thus, the most discriminating object, in this example, is b.

Example 5.4 *Let us use the optimized compute_protocol algorithm with the input of example 5.2. To select the most discriminating object of an object set, a function as the one described in example 5.3 is introduced. The set of objects occurring in Dec is scanner, cheque reader, card reader, keyboard, program TCF. If the most discriminating object in this set is scanner, the specification of the version of the scanner reduces the set of constraints to only one constraint. So, once the object scanner is specified, the specification of all the remaining objects may be run in parallel.*

The final valid and complete protocol will be:
[

[(discussion {scanner}), (discussion {card reader}) (iterations < 1) ()],
[(decision {scanner, cheque reader}) (iterations < 1) ()],
[(decision {card reader}) | (decision {keyboard}) | (decision {program TCF})
(iterations < 1) ()]

(iterations < 1) ()]

6 Related Work

This section focuses on the modeling of coordinated work and its needs.

Several papers in cooperative work concern the modeling of workflows. Besides the work already cited in the introduction [Clarke, 1996], [Jablonski, Bubler, 1996], [Kim, Paik, 1997], the needs of workflows are analyzed in [Schmidt, Simone, 1996]. The authors define a coordination mechanism as "a construct consisting of a coordinative protocol (an integrated set of procedures and conventions stipulating the articulation of interdependent distributed activities) on one hand and on the other hand an artifact in which the protocol is objectified". "A coordinative protocol is a resource for situated action in that it reduces the complexity of articulating cooperative work by providing a precomputation of task interdependencies which actors, for all practical purposes, can rely on to reduce the space of possibilities by identifying a valid and yet limited set of options for coordinative action in any given situation". The malleability quality of a workflow model, i.e., the capacity of the model to support dynamic changes, flexible process definitions and graceful handling exceptions, is underlined in [Schmidt, Simone, 1996] and [Kim, Paik, 1997].

As mentioned in the introduction, some researchers adopt advanced transaction models to specify workflows. The disadvantages of this approach were developed in [Alonso and al., 1995] and [Rusinkiewicz, Sheth, 1995]. In advanced transaction models, users and programs are not considered. Thus, mapping between activities and actors, and users actions cannot be modeled by this type of approach.

Several specific models for workflows have been proposed. All provide a language for task specifications. Thus, users can precise programs used to execute tasks, concerned human actors and related roles, grouping of tasks in activities, start and exit conditions of activities, data structure used, activity execution dependencies and workflow execution failure cases. Some of workflow models are only conceptual. They often use a graphical representation. Others allow to define workflows and to implement them as well (agent or rule based models).

Conceptual worflow models provides a graphical representation of the flow of activities. This representation is completed by a definition, in a formal language, of the different elements of the workflow: WFDL in [Casati and al., 1995] in which actors and roles concepts do not exist, METEOR in [Sheth and al., 1996], ATREUS in [Grifoni and al., 1997] which provides a lot of possibilities to define activity dependencies, ICN in [Kim, Paik, 1997].

An example of a rule-based model is provided by [Gokkoda and al, 1997]. Workflows are implemented by using the Acta transaction model and the rule specification language provides Acta-like facilities to declare task dependencies and compensated activities.

[Leymann, Roller, 1998] and [Divitini, Simone, Schmidt, 1996] propose agent-based models. Both paper concentrate on agent communications. But, the first paper focuses on the implementation of agent communications while the second one is concerned with the specification of communications. In more details, [Leymann, Roller, 1998] describes a conceptual workflow architecture based on persistent queues. Each agent has an input and an output queues. Communications between agents are realized by messages put in the agent input queues. Messages are deleted from the input queues by its consumer. The results of agent actions are messages inserted by the agents in their output queue.

[Divitini and al., 1996], present a language, IL, which allows to define the interoperability between agents in a multi-agent-based CSCW systems. IL integrates the malleability and linkability qualities of computational coordination mechanisms. In particular, special working modes, like conversation (what we called discussion in our work), awareness and iteration are treated as independent and linked coordination mechanisms. IL provides low-level means to define communications between the different agents of a coordination mechanism (awareness with or without acknowledgement of receipt, sending of information, task activation, warning) and between different coordination mechanisms. IL allows to define conditions associated with the communications. The real executed protocol will depend on the satisfaction of the conditions (malleability quality).

7 Conclusion

Some applications like the one described in this paper present specific characteristics: work distributed between several teams, a lot of changing data constraints to respect, short time delays. The use of the traditional transactional approach to preserve the data consistency may lead to undo and redo some parts of the work and thus, may induce a waste of time and a waste of money. In some cases, this situation is not acceptable. We propose an alternative approach which consists of organizing the work from the constraints, so that there is no need of a constraint checking phase. Instead of verifying data consistency to ensure the consistency of work being performed on the data, work consistency is controlled to be sure of data consistency.

A conceptual model of cooperative work and a language to define work protocol are presented. Our cooperative model is founded on the following principles: 1) a coordinator ensures the work protocol execution, 2) teams do not directly communicate among each other, 3) the coordinator ensures the communication among teams by messages, 4) exchange of messages across teams and the coordinator are asynchronous, 5) the coordinator is sensible to external events. The user defines working modes between participants of a group like collective discussion or serial decisions. The protocol definition language allows to define serial and parallel tasks and group of tasks based on pre-defined working modes, and iterative and exception conditions.

Our support considers the nine components of a cooperation model defined in [Clarke, 1996]: 1) several partners, 2) communication among partners -about choices made on the objects, 3) communication about communication and management of communication - maintenance of the history of messages among partners and communication of it to partners, 4) definition of a goal - building of a system -with specific characteristics, 5) goal-oriented working -in a continuous sequencing way, 6) definition of actions necessary to reach the goal - choices into a list of possible alternatives, 7) coordination of actions of different partners, 8) shared standards and guides -constraints between object versions

and object version manufacturing ordering, 9) a reward system -when a partner makes a choice, it is definitive and the system built is valid at the first attempt.

Our proposition was applied to the creation of a new component-oriented system. After modeling the constraints on the reusable components, a work protocol from "discussion" and "decision" working modes, a list of tasks and heuristics about task scheduling was automatically generated. This protocol can always be afterwards changed by the user.

A working protocol may also be generated for the maintenance of an existing system. Changes in the component base generally lead to update the constraint base. So, from the set of added and modified constraints, a working protocol may be generated to drive the specification of components to be changed.

There are many open issues which should be attacked:

1. In the application chosen, the compatibility constraint set is such that the specification of the different objects may be ordered so that, at any time, the valid versions of an object can be computed from the versions of the preceding objects. We are thinking to other applications using other kinds of constraints, particularly constraints on execution of tasks.

2. Our protocol is statically generated. The protocol definition language provides exception handling, for example human interventions, which may change the normal progress of the work. But it would be interesting to add facilities that allow dynamic changes of the protocol during its execution, for example: its restart from a recovery point, the serial execution of tasks initially planned in parallel, the modification of fragments according to the results of the execution of preceding ones.

3. We are going to implement a prototype. Some points are not yet decided. They concern:

   - *the management of the reusable component base.* During the building of a system, one coordinator by constraint family is created. Each coordinator controls the execution of the protocol which leads to the specification of the objects involved in the family. The project component base which has to be accessed by the coordinators may be centralized or temporary distributed between them.
   - *the management of workspaces.* When the building of a system is finished, the integrator has to access to the whole system. So, even if several coordinators controlled the cooperative work, all the sub-systems have to be put together. In case of the restart of a part of the protocol, the concerned sub-systems have to be communicated to the coordinators and the whole system updated.
   - *the different nature of processes.* The control system has a central and several decentralized processes. The decentralized processes are the coordinators. The central process initializes their work, accesses to the reusable component base, and manages the integration of the sub-systems.
   - *the protocols.* We are thinking to automatically generate E-C-A rules from a protocol definition and to use an application development toolkit like in [Bouzeghoub and al., 1998], to control its execution and to implement the coordinators.

8 Acknowledgements

We are grateful to Claudia Bauzer-Medeiros and Genevieve Jomier for their interesting remarks and suggestions.

References

AFCET, 1994, *Enquête sur la pratique de la collectique (groupware) en France*, rapport d'études, septembre 1994

G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, C. Mohan, 1995, *Advanced Transaction Models in Workflow Contexts*, Research Report RJ 9970 (87929) 7/31/95, IMB Research Division

Ardent Software, 1998, *O2 Reference Manual*, 7 rue du Parc de Clagny, 78035 Versailles Cedex, France

P. Bernstein, E. Newcomer, 1998, *Principles of Transaction Processing*, Morgan Kaufmann Publishers

M. Bouzeghoub, F. Fabret, M. Matulovic-Broque, E. Simon, 1998, *A Toolkit Approach for Developing Efficient and Customizable Active Rule Systems*, Report of Esprit Project LTR DWQ no: 22469, November 1998

F. Casati, S. Ceri, B. Pernici, G. Possi, 1995, *Conceptual Modeling of Workflows*, in Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Approach, Australia, Springer-Verlag, Lecture Notes in Computer Science

P. Chrysanthis, K. Ramamritham, 1991, *A Formalism for Extended Transaction Model*, Proceedings of the Seventeenth International Conference on Very Large Data Bases.

A. A. Clarke, 1996, *A Theoretical Model of Cooperation*, Proceedings of COOP'96, June 12-24, Juan-les-Pins, France

M. Divitini, C. Simone, K. Schmidt, 1996, *Abaco: Coordination Mechanisms in a Multi-agent Perspective*, Proceedings of COOP'96, June 12-24, Juan-les-Pins, France

A. K. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz, 1990, *A Multidatabase Transaction Model for Interbase*, Proceedings of the Sixteenth International Conference on Very Large Data Bases.

H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem, 1991, *Modeling Long-Running Activities as Nested Sagas*, In Proceedings IEEE Spring Compcon

E. Gokkoda, M. Altinel, I. Cingil, E. N. Tatbul, P. Koksal, A. Dogac, 1997, *Design and Implementation of a Distributed Workflow Enactment Service*, in Proceedings of Second IFCIS International Conference on Cooperative Information Systems, Kiawah Island, South Carolina, USA

P. Grifoni, D. Luzi, P. Merialdo, F. L. Ricci, 1997, *Atreus : a Model for the Conceptual*

Representation of a Workflow, in Proceedings of the Eighth International Workshop on Database and Expert Systems Applications, September 1-2, Toulouse, France

S. Jablonski, C. Bubler, 1996, *Workflow Management: Modeling Concepts, Architecture and Implementation*, International Thomson Computer Press

K.-H. Kim, S.-K. Paik, 1997, *Practical Experiences and Requirements on Workflow*, in Coordination Technology for Collaborative Applications, W. Conen and G. Neumann, editors, Springer-Verlag

F. Leymann, D. Roller, 1998, *Building a Robust Workflow Management System with Persistent Queues and Stored Procedures*, in Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, Orlando, Florida

B. Munier, M. F. Shakun, 1988, *Compromise, Negotiation and Group Decision*, D. Reidel Publishing Compagny

C. Pu, 1988, *Superdatabases for Composition of Heterogeneous Databases*, IEEE Proceedings of The Fourth Conference on Very Large Data Bases

M; Rusinkiewicz, A. Sheth, 1995, *Specification and Execution of Transactional Workflows*, in Modern Database Systems, Won Kim, editor, Addison Wesley

K. Schmidt, C. Simone, 1996, *Coordination Mechanisms: Towards a Conceptual Foundation of CSCW Systems Design*, JCSCW, vol. 5, no. 2-3

A. Sheth, K. Kochut, J. Miller, D. Palaniswami, J. Lynch, D. Worah, S. Das, C. Lin, I. Shevchenko, 1996, *Supporting State-wide Immunization Tracking using Multi-Paradigm Workflow Technology*, in Proceedings of the 22nd International Conference on Very Large Data Bases, Mumbai(Bombay), India

H. Waechter, A. Reuter, 1992, *The ConTract Model*, in Database Transaction Models for Advanced Applications, A. K. Elmagarmid, editor, Morgan Kaufmann Publishers

G. Weikum, H.-J.Schek, 1992, *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*, in Transactions Models for Advanced Applications, A. K. Elmagarmid, editor, Morgan Kaufmann Publishers

Appendix A. Updating algorithm of a reduced set of constraints when a new constraint is added

input: a reduced set of constraints $\mathcal{R}$ and
a new constraint c;

output: a new reduced set of constraints $\mathcal{R}'$

Initialization: $\mathcal{R}' \leftarrow \mathcal{R}$

i ← 1
k = |R'| + 1

/* *The function are_reductible takes two constraints a and b in parameters and returns true if one of the reducibility conditions is verified:*
1. $\mathcal{O}_a = \mathcal{O}_b$ *and there exists exactly one object o in* $\mathcal{O}_a$ *such that* $version_a$ *(o)* $\neq$ $version_b$ *(o)*
2. $\mathcal{O}_a \neq \mathcal{O}_b$ *and there exists at least one object o in* $\mathcal{O}_a \cap \mathcal{O}_b$ *such that* $version_a$ *(o)* = $version_b$ *(o)* */

/* c'_i *is a constraint in* $\mathcal{R}'$ */
while not are_reductible(c, c'_i) and i < k
  i ← i + 1;
end while

if i = k then /* *the new constraint c is reducible with any constraint in* $\mathcal{R}'$. *It is added as it is to* $\mathcal{R}'$ */
  $\mathcal{R}' \leftarrow \mathcal{R}' + c$;

else
  /* *Constraints c and* c'_i *are reducible. A new constraint* c'_k *is computed and added to* $\mathcal{R}'$. *The set of objects involved in* c'_k *is composed of all the objects involved in both constraints c and* c'_i. */
  $\mathcal{O}_{c'_k} = \mathcal{O}_c \cup \mathcal{O}_{c'_i}$

/* *For each object of* c'_k, *compute its set of versions* */
for each o in $\mathcal{O}_{c'_k}$
  if o $\in \mathcal{O}_c \cap \mathcal{O}_{c'_i}$ then
    /* *o is present in both constraint c and* c'_i. *So, the set of versions of o in constraint* c'_k *is composed of the set of versions of o involved in both constraints.* */
    $version_{c'_k}$ (o) = $version_c$ (o) ∪ $version_{c'_i}$ (o);

  elseif o $\in \mathcal{O}_c$ - $\mathcal{O}_{c'_i}$ then
    /* *o is only present in c. So, the set of versions of o in constraint* c'_k *is the same as in c.* */
    $version_{c'_k}$ (o) = $version_c$ (o);

  otherwise
    /* *o is only present in* c'_i. *So, the set of versions of o in constraint* c'_k *is the same as in* c'_i. */
    $version_{c'_k}$ (o) = $version_{c'_i}$ (o);

/* *Deleting of the constraint* c'_i *from* $\mathcal{R}'$ */
$\mathcal{R}' \leftarrow \mathcal{R}'$ - c'_i;

Appendix B. Protocol execution algorithms

Coordinators execute protocols by using the three algorithms below that detail the execution of a task and of a serial and a parallel fragment.

Task execution algorithm

/ When a task has to be executed, an init event is generated by one of both fragments execution algorithms (see the following algorithms). The task is the event parameter. The form of the task is (C, e) where C is the identifier of a working mode and e is a a team */*

on init_task event
    input: a task of the form (C, e)

    / a sent_message is calculated and sent to e */*
    send the message returned by C.compute_message(e) to e;

/ When e finishes its work, it sends a receive message */*

on receive message from e
    exit from task;

Serial fragment execution algorithm

/ When a serial fragment has to be executed, the coordinator generates an init_serial_fragment event. The fragment is the parameter of the event. */*

on init_serial_fragment event
    input: a serial fragment $Frag$ composed of an ordered set of tasks or an ordered set of fragments,
    an iteration control expression Ie and
    an exit control expression Ee;

    / if the iteration control expression is true, the execution of the fragment starts */*
    if Ie then
        / first(Frag) returns the first component of Frag. This component may be a fragment or a task */*

        if first($Frag$) is a fragment then
            send init_fragment event to first($Frag$); */* this event will be processed by the coordinator which will test if first(Frag) is a serial or a parallel fragment and will generate an appropriate init_fragment event (init_serial_fragment or init_parallel_fragment event /**

        else */* first(Frag) is a task */*
            send init_task event to first($Frag$);
    else */* the end of fragment is detected */*
        exit from $Frag$;

```
/* When a task finishes, it generates an exit event */
on exit from a task

    /* Frag is the fragment containing the task which has generated the exit event */
    /* next(Frag) returns the next component of Frag */
    if next(Frag) = ∅ then
        exit from Frag; /* execution of Frag is finished */

    else
        send init_task event to next(Frag); /* start the execution of the following
        task */
```

Parallel fragment execution algorithm

```
/* When a parallel fragment has to be executed, the coordinator generates an init_parallel_
fragment event. The fragment is the parameter of the event. */

on init_parallel_fragment event
    input:  a parallel fragment Frag composed of a set of tasks or a set of fragments,
            an iteration control expression Ie and
            an exit control expression Ee;

    /* if the iteration control expression is true, the execution of the fragment starts */

    if Ie then
        if first(Frag) is a fragment then
            send init_fragment event to each of the fragments composing Frag;
            /* each of these events will be processed by the coordinator which will test if
            the concerned fragment is a serial or a parallel fragment and will generate an
            appropriate init_fragment event (init_serial_fragment or init_parallel_fragment
            event /*

        else /* first(Frag) is a task */
            send init_task event to each of the tasks composing Frag;
    else
        exit from Frag; /* execution of Frag is finished */

on exit from task

    /* the message history is used to search for the exit event from each task composing
    Frag */
    if an exit has been received from each task of Frag then
        exit from Frag;

    /* if some tasks have not yet sent the exit event, the end of fragment execution is
    deferred */
```

Appendix C. Basic and optimized compute_protocol algorithms

Both algorithms take a valid protocol P for a family constraint, an ordered set of discussion tasks $Disc$, a set of decision tasks Dec, a default iteration expression Ie and a default exit expression Ee. They compute a valid and complete protocol for F. The optimized compute_protocol algorithm uses additionnally the constraint family to calculate the discriminating power of each object.

Basic compute_protocol algorithm

input: a valid protocol P,
an ordered set of discussion tasks $Disc$,
a set of decision tasks Dec,
a default iteration expression Ie and
a default exit expression Ee;

output: a valid and complete protocol P';

Initialization: $P' \leftarrow P$;

/ Compute the discussion fragment. A serial fragment composed of the ordered discussion tasks of Disc is added to P'. */*

if $Disc \neq \emptyset$ then
let $Frag$ denotes the fragment of the form $[(Disc)\ (Ie)\ (Ee)]$
$P' \leftarrow P' + Frag$;

/ Compute the decision fragments. Each decision fragment contains one task of Dec. The decision fragments are ordered in P' in the following manner: first, each decision task of Dec involving at least one object involved in Disc is added to P' ordered with respect to the objects in Disc, then the remaining tasks of Dec are considered in the same order as they appear in Dec. */*

let O' denotes the set of objects occurring in $Disc$ ordered with respect to their order in $Disc$;

while $Dec \neq \emptyset$

let O denotes the objects occurring in Dec
let o be the object of O chosen as follows:
if O' is not empty, then
o is the first element of O';
else,
o is the first element of O;

/ Selection of the decision task of Dec involving o */*

let d denotes the decision task of Dec containing o

/ Compute a decision fragment with the selected task and add it to P' */*
let $Frag$ denotes the fragment of the form [(d) (Ie) (Ee)]
$P' \leftarrow P' + Frag$

/ Deletion of the selected decision task from Dec */*
$Dec \leftarrow Dec - d$

/ Deletion of all the objects involved in the selected decision task from O' */*
remove every object occurring in d from O';
end while

return P';

Optimized compute_protocol algorithm

input: a constraint familly F
a valid protocol P for F
an ordered set of discussion tasks $Disc$
a set of decision tasks Dec
a default iteration expression Ie
a default exit expression Ee;

output: a valid and complete protocol P';

Initialization: $P' \leftarrow P$
$F' \leftarrow F$;

/ Compute the discussion fragment. A serial fragment composed of the ordered discussion tasks of Disc is added to P'. */*

if $Disc \neq \emptyset$ then
let $Frag$ denotes the fragment of the form [$(Disc)$ (Ie) (Ee)]
$P' \leftarrow P' + Frag$;

/ Compute the decision fragments. Each decision fragment contains one task of Dec. The decision fragments are ordered in P' in the following manner: first, each decision task of Dec involving at least one object involved in Disc is added to P' with respect to the discriminating power of the objects in Disc, then the remaining tasks of Dec are added ordered with respect to the discriminating power of their objects. */*

$O' \leftarrow$ set of objects occuring in $Disc$;
while $Dec \neq \emptyset$

let O denotes the objects occurring in Dec
/*remove from constraints in F' all the objects but objects of O , then reduce F' */
$F' \leftarrow$ simplify(F', O)
$F' \leftarrow$ reduce_set(F')

/ If it remains only one constraint in F', a parallel fragment composed of all the decision tasks remaining in Dec is added to P' */*

if | F' | = 1 then
  for each d_i in Dec do
    $frag_i$ = [(d_i) (Ie) (Ee)];
  $P' \leftarrow P' + frag_1 \mid ... \mid frag_n$ (with $n =$ | Dec |)
  $Dec \leftarrow \emptyset$;

else
/ F' contains several constraints. Decision fragments are computed in the following manner: first, the objects involved in Disc are selected according to their descending discriminating power and each decision task of Dec involving at least one of these objects are added to P' according to this order. At each iteration, F' is updated in deleting the objects used from F' and reducing F'. When all the decision tasks of Dec involving objects involved in Disc are used, the same process is executed on the remaining tasks of Dec: the objects involved in the remaining tasks of Dec are selected according to their descending discriminating power and the corresponding tasks in Dec are added to P'. The process ends when only one constraint remains in F'. */*

  let o be the object of O chosen as follows:
    select the most discriminating objects in O'
    select the most discriminating objects in O
  if several objects are selected then
    randomly pick one of them
  let d denotes the decision task of Dec containing o
  let $Frag$ denotes the fragment of the form [(d) (Ie) (Ee)]
  $P' \leftarrow P' + Frag$

  / Updating of Dec and O' before iterating */*
  $Dec \leftarrow Dec - d$
  if o $\in O'$ then
    $O' = O'$ - o;
end while

return P';