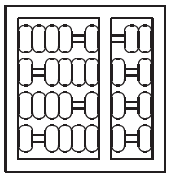

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

A Synchronization Manager Component for Workflow Management Systems

Gustavo Kasprzak *M. Beatriz F. Toledo*
Itana M. S. Gimenes

Technical Report - IC-03-007 - Relatório Técnico

April - 2003 - Abril

A Synchronization Manager Component for Workflow Management Systems

Gustavo Kasprzak Maria B. F. de Toledo* Itana M. S. Gimenes †

April 14, 2003

Abstract

Workflow Management Systems (WfMS) have been adopted as a practical solution to automate the execution of *business processes* in order to use resources efficiently and make organizations more competitive. Workflow activities usually have a very coarse granularity, are long-lived and business process dependent. The semantics of activities, including the semantics of associated invoked applications and resources, are not known by the WfMS. As a result, inconsistencies may arise from the concurrent use of shared applications/resources by the workflow activities. Thus, we propose a workflow activity synchronization mechanism to deal with these inconsistencies using a component-based solution. We present the specification of a software component called *Synchronization Manager Component*, a prototype implementation and a case study carried out within the context of the WorkToDo environment.

1 Introduction

Workflow Management Systems (WfMS) have been adopted as a practical solution to automate the execution of *business processes* in order to use resources efficiently and make organizations more competitive.

According to the *Workflow Reference Model* [2] developed by the *Workflow Management Coalition* (WfMC), a *process definition* representing a business process is composed by a sequence of interdependent *activities*. An activity represents the computer automation of a logical step within a workflow and its execution could require a shared external (software) application or a shared resource. A WfMS can instantiate process definitions and manage the execution of these instances together with their respective activities.

Workflow activities usually have a very coarse granularity, are long-lived and business process dependent. The semantics of activities, including the semantics of associated invoked applications and resources, are not known by the WfMS. As a result, inconsistencies may arise from the concurrent use of shared applications/resources by workflow activities

*Institute of Computing, Universidade Estadual de Campinas, Caixa Postl 6176, 13083-970, Campinas - SP.

†Department of Informatics, Universidade Estadual de Maringá, Av. Colombo 5790, 87020-900, Maringa - PR.

belonging to the same or eventually to different workflow instances.Thus, we propose a workflow activity synchronization mechanism to deal with these inconsistencies using a component-based solution.The component was specified and a prototype was implemented. A case study was carried out within the context of the WorkToDo environment [10], a WfMS compliant with WfMC.

The rest of the paper is organized as follows. Section 2 discusses a small example and the advantage of a component-based approach to address this issue. Section 3 describes the Synchronization Manager Component specification. The prototype implementation and the case study based on the WorkToDo system is presented in Sect. 4. Section 5 describes related work while Sect. 6 ends the paper with conclusions.

2 Motivation

Consider two workflow activities which read/update a certain file in a file system. If these activities execute concurrently inconsistencies may arise. To avoid this, the shared resource should be accessed in a exclusive way and the execution of one activity should only start after completion of the other.

There are two feasible alternatives to control the execution of activities that may cause inconsistencies. The first enhances the file system adding a layer to manage concurrent accesses to files. The latter requires synchronization constraints to be specified at process definition level with the WfMS being responsible for interpreting them and controlling activities accordingly. We believe that it is desirable to have a general purpose facility to deal with any kind of activity/resource.

Some of the proposed work in the area [3, 8, 4] suffer from the drawbacks below:

- are highly dependent on specific workflow models and implementations;
- demand complex process definition languages;
- require a great effort on workflow designers;
- and present low potential for re-usability.

We focus here on a component-based solution that, in addition to solving the synchronization problem, would eliminate the drawbacks above. A component can be defined as an independent software unit that encloses both its project and implementation [5, 14, 15]. It communicates with external environment through well-defined interfaces. The component-based approach is important as it increases re-usability potential. The proposed component follows the WfMC and the most relevant work principles in the area. Thus, the objective is that any WfMS implementation may integrate an implementation of the proposed component as a synchronization solution.

3 The Synchronization Manager Specification

The *Synchronization Manager Component* (SMC) described below aims at ensuring synchronization requirements of activities within a workflow management system. The activities may belong to the same workflow instance (intra-workflow synchronization) or different instances (inter-workflow synchronization). Each activity must be associated with a class that determines the synchronization requirements for that activity. Thus compatibility information amongst activity classes must be provided as a list of conflicting classes. Activities of conflicting classes may not be executed concurrently in order to guarantee a controlled access to resources. For example, consider two activities, A and B, where A belongs to the synchronization class called "Activity Class 1" while B belongs to the class "Activity Class 2". The conflicting class list of "Activity Class 1" contains "Activity Class 2" and vice-versa. Therefore A and B are incompatible and must not execute concurrently.

This section presents the SMC specification following UML [16]. The use cases present the situation within which the component is used. The internal architecture presents the object classes that are part of the SMC and their relationship. The following sections present the SMC interfaces and raise some deadlock detection issues.

3.1 Use Cases

Two kind of actors can interact with the SMC: human and software actors. Thus, three scenarios of usage are possible:

1. Usage by WfMS administrator(s) (human actor) for the creation and destruction of conflicting classes and SMC configuration;
2. Usage by WfMS users (human actor) for querying about SMC configuration;
3. Interaction(s) with the workflow engine(s) (software actor) for querying about SMC configuration and invocations of the synchronization operations.

The use case diagram of the Fig. 1 shows these three scenarios. Note that invocations to synchronization operations can come from different workflow engines, possibly from different WfMS.

3.2 Internal Architecture

The SMC was developed considering a distributed environment, a large number of activity classes and a large number of activities within each class.

The class diagram of SMC is presented in Fig. 2. It is composed of two major object classes: the classes *ConflictingClassManager* and *ConflictingClass*. The first class creates instances of the second and manages information about them including, for instance, the host where each conflicting class instance was created. Each of these instances executes at a fixed network node whose location must be known by each WfMS requiring their services. A conflicting class manager instance is accessed through a specific interface — called *IConflictingClassMgt* — that contains the following operations: create, destroy, retrieve

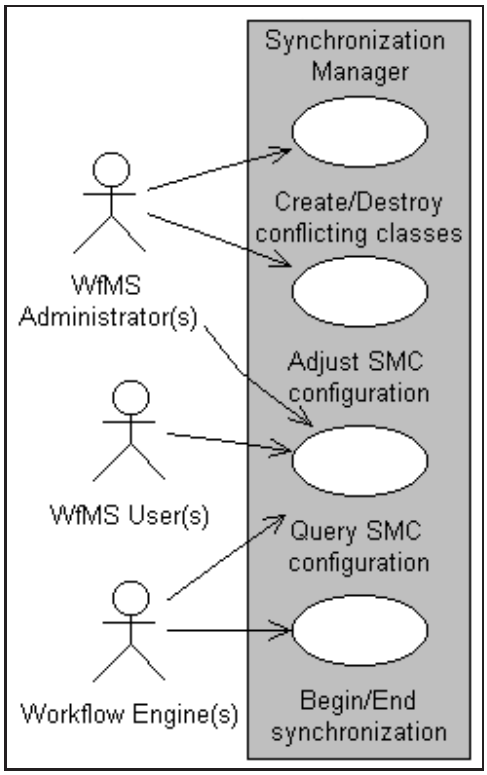

Figure 1: Use Case diagram showing the three SMC sceneries of usage.

and retrieve all conflicting classes. Usually, this class is associated with the WfMS administrator (graphical) interface that uses it to control the SMC and must only be accessed by him/her, although different access policies could be implemented.

The ConflictingClass class models the concept of activity classes. In a given moment, an instance of the SMC can have several instances of this class, each one representing an activity class. Moreover, it is also associated with two auxiliary classes, which are the classes *ActiveActivityTable* and *ConflictingClassTable*. The first stores the number of activities of this specific activity class under execution. The other stores the list of conflicting classes of this specific activity class. The class ConflictingClass has four fields that can be described as follows:

Name: the activity class name (e.g. "Activity Class 1" used in the example);

Description: textual description of the activity class that consists of, for example, characteristics of the activities that belong to this class;

Resources: information about the (hardware and software) resources that are used in the execution of activities of this class;

Host: the host name where the ConflictingClass was instantiated.

According to the three sceneries of usage presented in Fig. 1, the operations of the Conflicting class are distributed into three distinct interfaces: *IConflictingClassConfigAdm*, which contains the operations that the workflow administrator can use to query and adjust the ConflictingClass configuration; *IConflictingClassConfigUsr*, which contains the operations that other workflow users can use to query and adjust (in a restricted way) the ConflictingClass configuration; *ISynchronization*, which contains the operations related with the activity synchronization. The first two interfaces contain ordinary operations for querying and updating. Thus, our focus is on the specification of the *ISynchronization* interface. The class diagram of the Fig. 2 shows the described architecture.

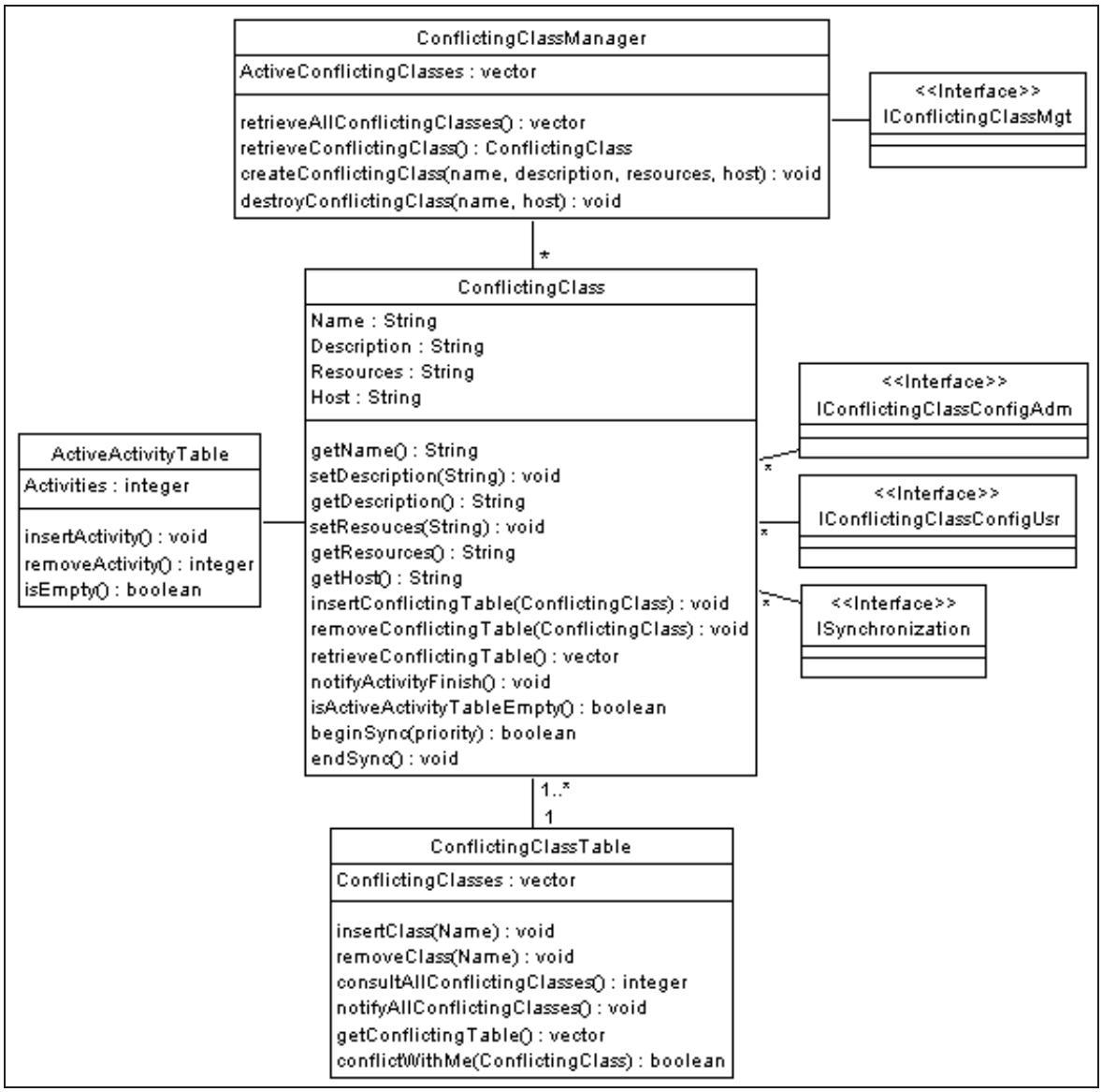

Figure 2: Class diagram of the SMC architecture.

3.3 The ISynchronization Interface Operations

Each new invocation to an operation of this interface causes a new thread to be created. The interaction between workflow engines that require activity synchronization and an instance of the class ConflictingClass occurs through the operations *beginSync* and *endSync*. Table 1 describes these and the other remaining operations. The SMC can be used by those workflow engines that support the activity class concept and the invocation of the synchronization operations must be coupled with their scheduling algorithms.

Table 1: Description of the ISynchronization operations.

Operation	Textual Description
beginSync	Just before scheduling the activity for execution, the workflow engine responsible for the activity must invoke this operation on its activity class instance. The operation may return synchronization success or synchronization failure. It has one parameter specifying priority used to avoid starvation.
endSync	When the activity finishes, the workflow engine responsible for the activity must invoke this operation on its activity class instance to confirm the activity completion.
notifyActivityFinish	This operation notifies other activity class instances about the completion of all activities of that class.
isActiveActivityTableEmpty	This operation checks whether the activity class has activities under execution.

A workflow engine invoking the beginSync operation for an activity X obtains *synchronization success* (`TRUE`) if there are no incompatible activities under execution (activities of conflicting classes). Note that it is possible that X activity class be incompatible with itself, what means that activities of this class must be executed with mutual exclusion. If there are not incompatible activities under execution, X can be safely scheduled. Otherwise, the workflow engine receives a *synchronization failure* (`FALSE`) and the X execution must be postponed. Synchronization failures are handled in an implementation dependent-way by re-invoking the beginSync operation until synchronization success is obtained or a number of retries is reached.

3.3.1 The beginSync Operation

The high-level algorithm that describes this operation is presented in Fig. 3.

To better illustrate synchronization, consider the example in the beginning of this section with the activities A and B of synchronization classes "Activity Class 1" and "Activity Class 2" respectively. Suppose that these activity classes have no other activities under execution

```
if Active Activity Table is not empty and Activity class does not conflict with itself
then
  Inserts the activity into Active Activity Table;
  Returns TRUE;
else
  Queries all conflicting classes about the number of activities under execution;
  if the query result is zero then
    Inserts the activity into Active Activity Table;
    Returns TRUE;
  else
    sleeps and waits a notification about the completion of all activities of conflicting
    classes;
    After the notification returns FALSE;
  end if
end if
```

Figure 3: High-level algorithm for the beginSync operation.

yet. In a given moment, activity A becomes ready for execution and the workflow engine responsible for A begins the synchronization procedure, described in the steps bellow:

1. The workflow engine invokes the beginSync operation on the ISynchronization interface of the "Activity Class 1";
2. "Activity Class 1" verifies if its Active Activity Table is empty;
3. As there is not any activity of this class under execution, the Active Activity Table answers true;
4. "Activity Class 1" queries all conflicting classes in its list of conflicting classes. For each conflicting class in the list (in this case just the "Activity Class 2"), "Activity Class 1" invokes the isActiveActivityTableEmpty operation to find whether or not there are activities under execution;
5. As there are not conflicting activities executing, "Activity Class 1" inserts A into its Active Activity Table;
6. The operation returns success;
7. And A can be safely initiated.

The sequence diagram in the Fig. 4 shows the steps above. If before this beginSync invocation, the "Activity Class 1" already had one or more activities under execution, after step 3, it would check if it conflicts with itself, and if not, A would be inserted into the Active Activity Table and the operation could return success avoiding unnecessary queries on other conflicting classes.

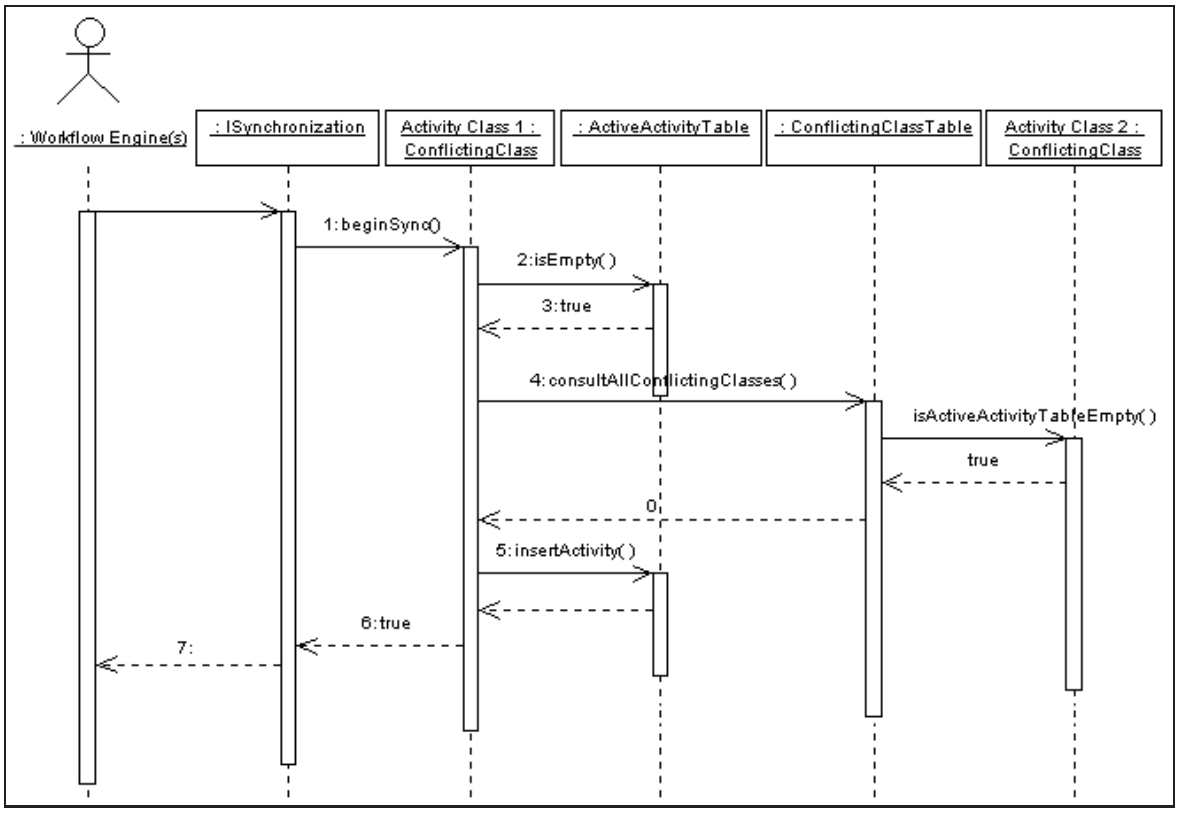

Figure 4: Sequence diagram when activity A beginSync is executed with success.

In case of synchronization failure, the thread that answers the beginSync request can have its execution suspended. For instance, consider a new scenery for our example where activity B of "Activity Class 2" is under execution and the workflow engine responsible for A invokes beginSync for it. The execution flows until step 3 in the diagram of Fig. 4. After step 4, "Activity Class 2" informs that it already has an activity (B) under execution and the thread executing beginSync for A is suspended in step 5 until the end of all activities of "Activity Class 2". While B is executing, another activity of the same activity class can eventually starts its execution. When all activities of Activity Class 2 finishes, the thread executing beginSync for A resumes its execution and a synchronization failure is returned to the workflow engine. This situation is presented in the Sequence Diagram of Fig. 5.

The workflow engine receiving a synchronization failure may re-invoke the beginSync a number of times until synchronization success is obtained or a number of retries is reached.

3.3.2 The endSync Operation

When an activity finishes, the workflow engine responsible for it invokes the endSync operation on the suitable activity class. The activity class that receives the request removes the corresponding activity from the Active Activity Table and then, if it was the last activity of this class under execution, the class notifies all conflicting classes. Suspended threads will

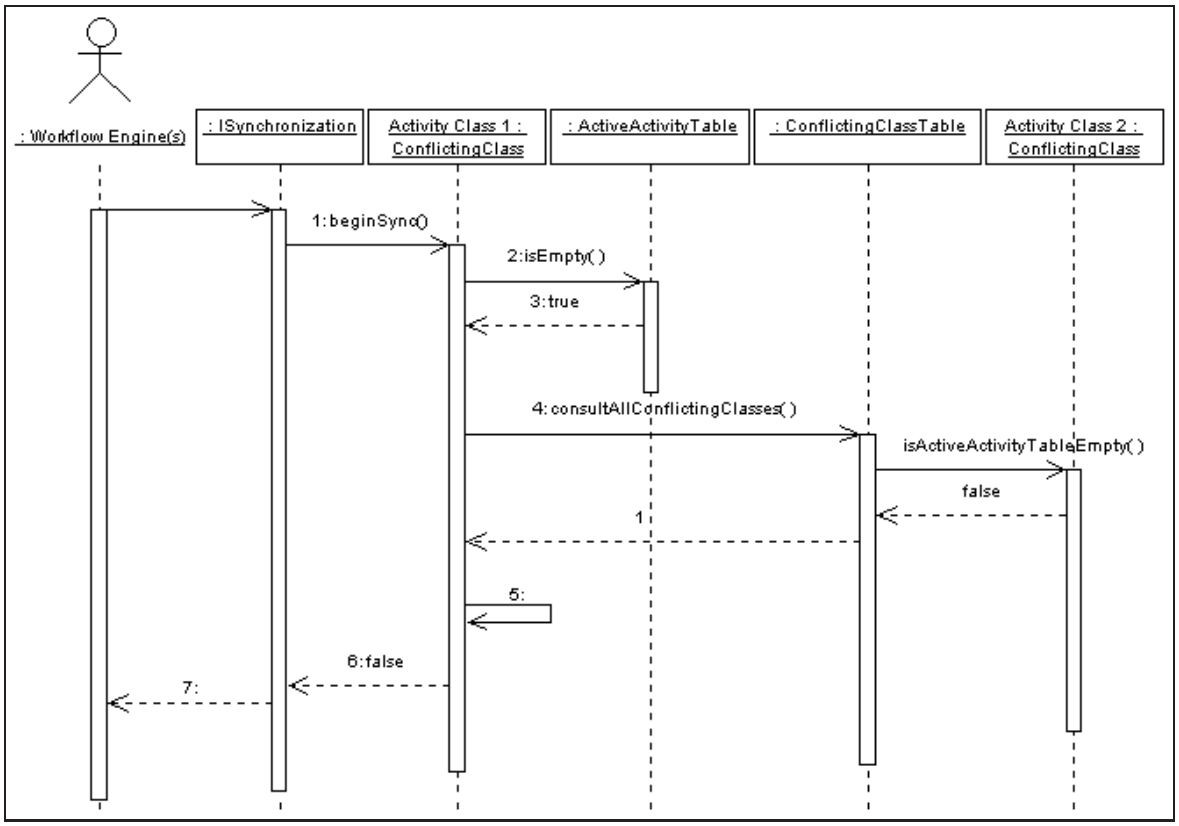

Figure 5: Sequence diagram when activity A beginSync is executed with failure.

now be able to proceed.

3.4 Deadlock Detection and Handling

Deadlocks may occur as the operations of the ISynchronization interface are executed in a critical region. For example, consider Activity Classes 1 and 2 and a concurrent request beginSync received by each. Activity Class 1 tries to query Activity Class 2 and vice-versa. Thus the deadlock situation is established.

There are several solutions for deadlocks in distributed systems [6]. The solution adopted here is based on *timeout*. When a thread is created to answer a beginSync request, another thread called *monitoring thread*, is also created at the same moment. The monitoring thread is put to sleep and it remains in this state for a pre-defined period. When it wakes up, it verifies if the beginSync thread is still executing and, if this is true, it interrupts the beginSync thread execution and finishes.

Using this technique, the deadlock situation between the two classes above would be broken. When one of the monitoring threads (for Class 1 or 2) resumes, it detects that the beginSync thread is still executing and kills it. Thus, the other beginSync thread involved in the deadlock can continue its execution.

Another important issue is starvation. As the beginSync operation may be killed and retried, starvation may occur. To avoid it, the thread that executes beginSync is given a greater priority each time a beginSync returns with failure and is re-executed.

4 Implementation

To validate the Synchronization Manager Component specification we have implemented it and integrated with a workflow management system. Before presenting more details about the SMC implementation, we will describe the WorkToDo [9, 10], the workflow management system used in this case study.

4.1 WorkToDo

The WorkToDo was originally designed to address the problem of using WfM systems in wireless communication environments [9, 10], where connections are unstable and non-permanent.

Its process model is a subset of the model proposed by the WfMC defining which activities should be executed, in which sequence, by whom and with what data. A process is a set of activities and dependences among them; each activity has a set of input and output data and a processing entity responsible for its execution (a user or an application). Each user belongs to a role that defines a group of users with some set of characteristics. Activities may be tasks (elementary actions) or sub-processes. Tasks are classified into three categories: *automatic*, which are executed by a software invoked automatically by the WfMS (e.g., executing a transaction on a database); *Semi-automatic*, which are executed by a human assisted by a software (e.g., writing a document using a text editor, scanning a photograph); *Manual*, which are executed exclusively by a human (e.g., filling in a form, sending a letter).

WorkToDo uses dependence rules to model the control flow of a process. A dependence rule contains the conditions that must be reached to allow the execution of an activity. These conditions are described in terms of activity state transitions. A rule is composed by zero or more terms containing an activity name and a state. These terms may be operands of boolean operators (*and*/*or*).

A *Workflow Definition Language* (WDL) is provided to allow process definitions.

Furthermore, WorkToDo has a distributed architecture shown in Fig. 6 composed by the following elements:

User and Role Manager: is responsible for managing users and roles;

Definition Manager: is responsible for managing process definitions;

Instance Manager: maintains information about process instances under execution and those already finished;

Process Manager: coordinates a process execution. It verifies which activities are ready to execute, initiates their execution and collects their results, verifying if these results

allow the execution of other activities. There is one process manager for each executing process instance.

Task Manager: controls the execution of an automatic task.

Worklist Manager: controls and maintains a list with all the activities that can be executed by a given user.

The WorkToDo is in conformance with the WfMC Reference Model. Scheduling and controlling of activities is done by the Process Manager which incorporates the functionality of the workflow engine. The interface with users and applications provided by Worklist Managers and Task Managers corresponds to Client Applications and Invoked Applications respectively.

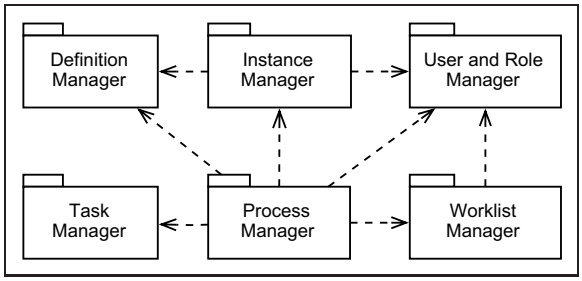

Figure 6: Package diagram showing the WorkToDo high-level package structure and relationships.

The WorkToDo may be implemented using any programming language and any distributed platform that supports remote communication, however the current prototype uses Java and CORBA. Both of them were used for the SMC implementation as well (see Sect. 4.2).

4.2 SMC Prototype Implementation

The SMC prototype was implemented using OrbixWeb [7] (IONA CORBA implementation) and Java [13] (JDK 1.2).

The choice for Java was motivated by its property to be multi-platform, allowing the system execution on different hardware and operating systems. This is an important aspect in a workflow system, where many users access the system from different machines.

CORBA was chosen because it provides transparent remote object access making the development of distributed applications easier. Furthermore, it is a relatively stable technology and has been used successfully in many contexts.

The classes of the Class Diagram in Fig. 2 were coded into the same Java package. The classes Conflicting Class Manager and Conflicting Class (software representation of an activity class) were implemented as CORBA servers. The three different Conflicting Class interfaces — IConflictingClassConfigAdm, IConflictingClassConfigUsr and ISynchronization — were defined in distinct IDLs specifications, although the object that implements

the operations of these interfaces (the servant) is the same. Figure 7 shows the IDL specification used for the ISynchronization interface. The multi-threading property of this class was implemented using OrbixWeb *Filter* facility, which allows a server to create a new thread to answer each new invocation received. The persistence property was implemented using OrbixWeb *Loader* facility together with Java file manipulation classes and methods.

```
interface Synchronization{
boolean isActiveActivityTableEmpty();
boolean beginSync(in long priority);
void endSync ();
void notifyActivityFinish();
};
```

Figure 7: "*ISynchronization*" interface IDL definition.

Table 2 presents the number of Java classes written in the WorkToDo and SMC implementation. For each prototype, the table shows the number of automatically generated classes after IDL compilation, the number of manually codified classes, the number of modified classes in WorkToDo and the estimated code lines number for manually codified classes.

Table 2: Code information about WorkToDo and SMC implementation.

Prototype/Classes	**Automatic**	**Manual**	**Modified**	**Code Lines**
WorkToDo	162	86	7	7500
CGS	35	11	-	1000

4.3 Component Integration

As expected, the integration of SMC with the WorkToDo has occurred without any major problem. To support the concept of activity classes, both the task model (in the WorkToDo process model) and the process definition language had to be extended. A task has now, in addition to the clauses `APPLICATION`, `PRIORITY` and `DEADLINE`, another one related with its synchronization properties, the `ACTIVITY_CLASS`.

The scheduling algorithm of the process manager was changed in order to consider synchronization requirements. When an activity becomes ready to execute, the process manager invokes the beginSync operation on the appropriate conflicting class (defined by the activity class clause above). If either a synchronization failure or an exception is returned, the process manager re-invokes the operation until a *maximum retry number* is reached. If beginSync returns with success, the activity is scheduled for execution. When the activity ends, the process manager invokes the endSync operation.

This integration experience has clearly shown the feasibility of using SMC in WfMS environments without requiring a great effort.

4.4 Synchronization Example

Consider a business process of an insurance company where a customer may have several kinds of insurance policies (e.g. for life, house or car). Each kind of insurance policy is treated by a different process (instance), and the customer record is shared by them. Furthermore, for each kind of insurance policy, the company maintains for administrative purposes a *controlling file* containing information about all insurance policies. The information includes, for example, the insurance policy ID, its status and beneficiaries.

The controlling file update activity is shown in the following process definition fragment:

```
WORKFLOW LifeInsuranceCreation{
   FILE LifeInsuranceControllingFile{
      NAME "/processes/files/life_insurance_controlling.cdb";
   }
   ⋮
   TASK UpdateLifeInsuranceControllingFile : UpdateFile{
      ROLE Manager;
      IN_CONTEXT LifeInsuranceControllingFile;
      OUT_CONTEXT LifeInsuranceControllingFile;
      DEPENDS ClientAgreement -> SUCCEDED;
      DESCRIPTION "Updates the life insurance controlling file";
   }
   ⋮
}
```

The process, named `LifeInsuranceCreation`, contains the task (activity) `UpdateLife-InsuranceControllingFile`, which is an instance of the task model `UpdateFile`. The role `Manager` is responsible for this task execution and the task will only be executed after the task `ClientAgreement` is executed successfully. The only input and output data for `Update-LifeInsuranceControllingFile` is the file `life_insurance_controlling.cdb`, identified by `LifeInsuranceControllingFile` and stored at the path `/processes/files/`. The specification also contains a textual description of the task.

The task model `UpdateFile` is defined as follows:

```
TASK UpdateFile{
   TYPE Automatic;
   ACTIVITY_CLASS ControllingFileUpdate;
   APPLICATION UpdateFileScript.sh;
   PRIORITY 10;
   DEADLINE 5 MINUTES;
   DISCONNECTED_OPERATION false;
}
```

The task is automatic and its related application is `UpdateFileScript.sh`, a shell script whose characteristics are defined separately. The definition includes the task priority and

deadline for its execution. The clause `DISCONNECTED_OPERATION` indicates if the task may be executed in disconnected mode (`true`) or not (`false`).

The `ACTIVITY_CLASS` task attribute indicates that the task belongs to activity class `ControllingFileUpdate`. To guarantee synchronization requirements, the SMC instance integrated with WorkToDo must have an instance of `ControllingFileUpdate` activity class executing. The beginSync operation will be invoked on this activity class and will return with success if no other activity of this class is executing. Thus exclusive access of the controlling file is guaranteed.

In this example, another synchronization requirement could be imposed upon the activities that access the shared customer record. The solution would be very similar to the one above. These activities would belong to the same activity class and should execute in a mutual exclusive way.

5 Related Work

The synchronization and coordination of concurrent executions is not a recent problem and has been studied for a long time. Originally it was discussed in the context of operating systems to allow the simultaneous use of scarce resources by multiple users and multiple solutions were developed such as *mutual exclusion protocols*, *semaphores* and *monitors*. Next, it was studied in other areas such as database where locking and certification mechanisms [17] were proposed.

In the database area, extended transactions models [12, 4] have focused primarily on coordinating sub-transactions constituting a start point for synchronization in workflow environments.

In [1], the authors present the advantages of combining workflow management and transaction management and the use of a task compatibility specification to control concurrent executions of activities. Our approach provides the same capabilities of the compatibility matrix with the additional advantage of handling coarser granularity elements (activity classes) and thus dealing with a more compact matrix. Furthermore, this work is defined upon a centralized architecture in contrast with our proposal.

The need for mutual exclusion of concurrent activities to ensure correct interleaving is emphasized in [8] were a history-based protocol is proposed. This protocol takes advantage of semantic constructs associated with WfMS to solve some problems such as dealing with inherited restrictions and the coarse granularity of workflow specifications. Again, only a centralized environment is considered. The required synchronization specifications is based on protected operations of a process class with respect to other process classes which we consider more complex than specifying a compatibility matrix.

The work in [3] handles data inconsistencies arising due to concurrent executions and due to failures defining a uniform set of low-level mechanisms and dynamically modifying workflow rule sets for control flow. Here the synchronization specifications are dependent on particular activities belonging to particular workflows.

The design of a component-based WfMS is in accordance with the WfMC objectives of allowing different companies to develop workflow technology solutions. A task scheduling

component was proposed by Gimenes and Barroca [11] similar to the SMC component.

However, as far as we know, the synchronization issue was neither tackled using a component-based approach nor based on workflow activity classes.

6 Conclusions

In this paper, we describe the Synchronization Manager Component specification, a software component that provides synchronization capabilities to workflow management systems. Workflow activities are mapped into activities classes and are executed only if there are not concurrent incompatible activities under execution. The operations begin/end synchronization are coupled with workflow engine scheduling algorithms to guarantee the correct activity synchronization.

Furthermore, a Java/Corba prototype implementation of the SMC and a case study within the WorkToDo WfMS were presented. This experience has clearly shown the feasibility of the proposal.

We believe that the proposed work, in addition to solve the synchronization problem efficiently, eliminates some of the drawbacks of other works such as poor re-usability, complex process definition languages and complex synchronization specifications. In addition, it may be used in different and heterogeneous workflow environments in conformance with WfMC and general software engineering guidelines.

References

[1] Y. Breitbart and A. Deacon and H.-J. Schek and A. Sheth and G. Weikum: Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. SIGMOD Record (ACM Special Interest Group on Management of Data), 22, 3, 23–30, 1993

[2] D. Hollinsworth: The Workflow Reference Model, Workflow Management Coalition. D. Hollinsworth, The Workflow Reference Model, Workflow Management Coalition, TC00-1003, December 1994

[3] Mohan Kamath and Krithi Ramamritham: Failure Handling and Coordinated Execution of Concurrent Workflows. 334-341, 1998

[4] H. Wächter and A. Reuter: The ConTract Model, Helmut Waechter, Andreas Reuter The ConTract Model chapter 7 in [4] pp. 219-263, 1992

[5] Desmond D'Souza and Alan Wills: Objects, Components and Frameworks With UML: The Catalysis Approach. Addison-Wesley, 1998

[6] George F. Coulouris and Jean Dollimore: Distributed Systems: Concepts and Design. 1988, Addison-Wesley Longman Publishing Co., Inc.

[7] IONA Technologies: OrbixWeb: IONA Technologies Java ORB Implementation. April, 2002, url = "http://www.iona.com/products/orbixweb/index.html"

[8] Gustavo Alonso and Divyakant Agrawal and Amr El Abbadi: Process Synchronization in Workflow Management Systems. Symposium on Parallel and Distributed Processing, 1996

[9] L. H. Reinehr and M. B. F. Toledo: WorkToDo: Um Sistema de Gerenciamento de Workflows para Ambientes de Comunicação sem Fio. Proceedings of 20th Simposio Brasileiro de Redes de Computadores, Búzios, Rio de Janeiro, Brasil, 2002, 619–634

[10] L. H. Reinehr and M. B. F. Toledo: A CORBA-based Workflow Management System for Wireless Communication Environments. Proceedings of Confederated International Conferences: CoopIS, DOA and ODBASE, 827–844, R. Meersman and Z. Tari, Irvine, California, USA, 2002

[11] I. M. S. Gimenes and L. Barroca: Enterprise Frameworks for Workflow Management Systems. Proceedings of Software — Pratice and Experience, 755–769, 2002

[12] P. Attie and M. Singh and A. Sheth and M. Rusinkiewicz: Specifying and Enforcing Intertask Dependencies, Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin, 1993

[13] Sun Microsystems: Java Development Kit, version 1.2 API Specification. April, 2002

[14] John Cheesman and John Daniels: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2000

[15] Clemens Szyperski: Component Oriented Programming. Addison-Wesley, 1997

[16] Rainer Burkhardt: UML: Unified Modeling Language. Addison-Wesley, 1997

[17] Bernstein, P.A. and Hadzilacos, V. and Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, 1987