July 2003, Vol.18, No.4, pp.495–504 J. Comput. Sci. & Technol.

Towards a Uniform Cooperative Platform: Cova Approach and Experience

SHI MeiLin (史美林) and JIANG JinLei (姜进磊)

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P.R. China

E-mail: {shi, jjlei}@csnet4.cs.tsinghua.edu.cn

Received February 21, 2003; revised April 25, 2003.

Abstract A cooperative platform called Cova is presented, which aims to uniformly model a wide range of cooperation scenarios and to reduce the groupware design and development work. To achieve these goals, Cova provides two facilities, namely a specification language for developers to describe various cooperation modes uniformly and a run-time system that provides some general-purposed services guaranteeing the semantics specified. With these facilities, the developers can then concentrate on the application specific functions rather than the control mechanisms. Therefore, the development efficiency is promoted. This paper details the design and implementation issues of the platform including the model and the specification language, platform architecture, transaction management, service integration and so on. Application development with Cova platform is also covered.

Keywords CSCW, Cova, transaction model, service integration, XML

1 Introduction

Progress made in computer networks has resulted in many new applications, among which is computer supported cooperative work (CSCW). Due to its potentials in improving group work efficiency, CSCW has received much attention. As a result, numerous groupware[1] and meta-groupware[2] systems have been invented. Though these systems do greatly facilitate people's cooperation, their applicability is limited as most of them are designed only for supporting a specific cooperation mode (e.g., single user activity, synchronous or asynchronous cooperation). Studies on people's everyday life have shown that people often cooperate in hybrid mode, which means different modes are deployed at various scenarios even within the same application.

Research experience on CSCW shows that a language support is necessary to deal with the flexibility and complexity required by CSCW systems. As a result, Cova[3] is proposed, which aims to uniformly model a wide range of cooperation scenarios and give system developers the maximum flexibility while minimizing their endeavors in groupware design and development. To achieve these goals, Cova provides the following facilities.

1) A specification language for developers to describe various cooperation semantics.

2) A run-time system providing various general-purposed services that guarantee the semantics specified by the specification language.

3) A set of APIs for developers to access the services supplied by the run-time system.

Compared with other meta-groupware systems such as COCA[4], GroupKit[5], Rendezvous[6] and MMConf[7], Cova has the following distinguished features.

1) Cova is based on a uniform coordination model which makes it capable of describing a wide range of cooperative processes, e.g., synchronous, asynchronous, autonomous, and integrated ones. This capability is short in other meta-groupware systems with WoTel[8] as an exception.

2) Object-oriented paradigm is used to describe a cooperative process with a clear separation and seamless integration of the computation and coordination parts. The coordination part of Cova has full knowledge about the semantics of the computation part, which makes the system able to do some advanced control which otherwise is impossible[9].

3) Cova bridges the gap between process modeling and instance enactment. Information produced during runtime is encapsulated as special objects

This work is supported by the National Natural Science Foundation of China under Grant No.60073011, the National High Technology Research and Development 863 Program of China under Grant No.2001AA113150 and 985 Project (High Speed Network Based CSCW Application and Development Environment) of Tsinghua University.

that can be used in process definition. In this way, extra flexibility can be obtained.

4) Cova is more scalable in the sense that it takes interoperation and service integration into account from the very beginning. Legacy or new services can be introduced into the system easily according to the requirements of application.

This paper details the philosophy inside Cova design and implementation. The rest of the paper is organized as follows. In the coming section, we explain the formal model and the corresponding specification language. Then implementation issues are depicted in Section 3. Section 4 demonstrates the cooperative application development based on the facilities provided. At the end of the paper, conclusion is made and future work is also presented.

2 Formal Model and Specification Language

Cova is a complete programming model[10] in the sense that it provides both the computation model and the coordination model. This section looks into the formal model and the specification language.

2.1 Object Model

Cova object model is a modified version of the one defined in the ODMG Specification 2.0. It supports inheritance, polymorphism, encapsulation, and nested or local class definition. Cova object model provides 10 primitive types, namely Boolean, byte, char, tiny, short, int, long, float, double and void, and 5 collection types, namely set, bag, list, array and dictionary (see the ODMG specification for their semantics). With these types, users can specify the structure and operation semantics of the objects handled by activities. In other words, the object model provides a type system for coordination.

2.2 Coordination Model

Cova coordination model consists of two basic concepts — process and activity. They play a role similar to that of workflow model in workflow management systems. Process, which describes the properties of all stages of a cooperation scenario, is the basic organization unit in Cova. Its formal definition is as follows.

Definition 1 (Process). *A process is a* 4*-tuple* $\langle pid, A, M, s\rangle$*, where*

- *pid is the name of the process, and it is unique in the system;*
- *A is a set of activities belonging to this process;*
- *M is a set of users and/or computer programs that can manage the process;*
- $s \in A$ *is the start activity name. The start activity will be created first when the process is instantiated. It acts as process entrance. Each process should have a start activity. As for process exit, it is implied by the rules in activity definition. A process can have more than one exit.*

An activity is a piece of work that contributes to the cooperation. It has its goal, life cycle and rules regarding state transition and interaction with the environment. The formal definition of Cova activity is as follows.

Definition 2 (Activity). *An activity is a* 6*-tuple* $\langle aid, t, d, pc, P, R\rangle$*, where*

- *aid is the name of the activity. It is unique within a process.*
- *t is the activity type. It can be atomic or composite. For a composite activity, it has its own internal structure as that of processes. The introduction of composite activity makes it possible to hierarchically define process and thus enhances flexibility.*
- *d is the type name of the object manipulated by the activity (called activity object, denoted by o). It is defined by the object model and describes how the data are organized and how operations are implemented. It is through d that the computation and coordination parts of Cova are seamlessly integrated.*
- *pc is the precondition placed on the activity. The corresponding activity object cannot be manipulated until pc is met. Whenever the internal state of an activity or a process changes, pc will be evaluated.*
- *P is a set of users and/or computer programs that can access the activity object.*
- *R is a set of rules that specify what action should be taken upon the occurrence of a specific event. Each element of R has a form of* $\langle e, c, a\rangle$*, where c is a Boolean expression defined on an activity or a global object and a is an invocation to a method defined on an object. When an event e occurs, if the condition c is met, the action a will be executed. Rules in our system are divided into three categories. They are as follows.*

◇ **Time-related rules**, *which reflect the time constraints imposed on activities. For example, an activity should report its status* 2 *hours after start.*

◇ **Method invocation rules**, *which specify the way in which variables are accessed. For example, method* M_1 *must be executed before* M_2 *is executed.*

◇ **State transition rules**, *which specify the dependencies among activities. For example, activity* A_1 *will not start until* A_2 *aborts/completes.*

The introduction of rules makes it possible to describe both control and data flows among different activities and thus asynchronous cooperation is supported. In addition, rules of the activity model are optional, i.e., an activity may have no rules or rules of a certain type. Without rules specified, an activity alone can describe (depending on the number of participants) a single user activity or synchronous cooperation. It is the activity model that makes our system capable of describing different cooperation modes uniformly.

Compared with other models, Cova coordination model has the following distinct features besides the uniformity[9].

• It has a loose mathematical structure, which imposes less limitations on how various activities are related while maintaining equal, if not more powerful, expressiveness.

• It is oriented to a process being enacted rather than a process template. Thus, information generated dynamically during process enactment can be used in process modeling. In this way, flexibility is enhanced.

• It allows hierarchical process definition as explained above.

2.3 Specification Language

Many coordination-oriented languages and models[10] adopt the idea to separate the computation and coordination parts to achieve portability and to support heterogeneity. Cova also favors this idea. As a result, the specification language is divided into two parts, namely Cova Object Description Language (CODL in short) and Cova Coordination Description Language (CCDL in short).

CODL implements the Cova object model, that is, it provides the language constructs for describing the structure of activity object and specifying its methods. Quite similar to other object-oriented programming languages such as Java and C++, the basic unit of CODL is class definition. Besides the traditional components (e.g., variable declaration, assignment, exception handling, flow control and so on), the following four additional components are defined for collection query and manipulation.

• **foreach** (... **in** ...), a statement used to navigate through an object of collection type.

• **insert/insert into** ... **{at** ...**}** **values** ..., a statement used to insert new elements into an object of collection type. For collections with index (i.e., list, array and dictionary), the clause "at···" can be specified.

• **delete** ... **where** ..., a statement used to delete from a collection the elements that satisfy the condition supplied.

• **update** ... **where** ..., a statement used to update certain elements in a collection.

All components make CODL a fully-featured object query and manipulation language capable of describing a wide range of objects.

CCDL provides the language constructs for describing cooperation scenarios. It is the innovative part of Cova language. Similar to CODL, CCDL also adopts object-oriented paradigm. The syntax for a process is as follows:

ProcessDeclaration ::= **process** IDENTIFIER [**extend** IDENTIFIER] [**start** IDENTIFIER]
ProcessBody
ProcessBody ::= '{' [ProcessFieldDeclarations] '}'
ProcessFieldDeclarations ::= ProcessFieldDeclaration [ProcessFieldDeclarations]
ProcessFieldDeclaration
::= Modifiers {ProcessDeclaration | ClassDeclaration | ActivityDeclaration}

A process declaration begins with the keyword **process** followed by a unique IDENTIFIER as its name. A process can optionally inherit (specified by keyword **extend**) from another process (called super process), whose name is specified by the IDENTIFIER following the keyword extend. The start activity of a process is specified by the keyword **start** and the following IDENTIFIER. Process designers are free to specify the start activity. With no one specified, the start activity will be the first one declared in the process. Within the process body are definitions of classes (in CODL), nested processes or activities (in CCDL). A nested process is also a process, but it is invisible out of the process where it is defined.

There are two activity types in our system and thus, activity definitions have two relevant forms. The syntax for composite activity is as follows, where the first IDENTIFIER is the given name of the activity and the second one the name of a process (called sub-process), which can be defined locally or globally.

ActivityDeclaration ::= **activity** IDENTIFIER **as** IDENTIFIER';'

Composite activity provides another way to process reuse besides process inheritance. For instance, designers can predefine some standardized domain-specific processes and invoke them within other application-specific processes when needed. In this way, much work can be saved. What's more, system flexibility is also enhanced.

The syntax for atomic activity is given below.

ActivityDeclaration ::= **activity** IDENTIFIER [**handle** IDENTIFIER] [**startwhen** Expression]
ActivityBody
ActivityBody ::= '{' [ActivityFields] '}'
ActivityFields ::=ActivityField [ActivityFields]
ActivityField ::= ParticipantDeclaration | TriggerDeclaration

From the rules above, we can see that an atomic activity can be divided into a header and a body. The activity header consists of three parts. The first part includes the keyword activity and an IDENTIFIER, which specifies the name of the activity. The second part is an optional handle clause, which specifies the class name (via the second IDENTIFIER) of the activity object. The third part is an optional startwhen clause, which specifies the condition (given by Expression) under which the activity becomes active. Within the activity body is a set of activity fields, which specifies who can access the activity object (via ParticipantDeclaration) and how the activity communicates with the others (via TriggerDeclaration). The syntaxes of ParticipantDeclaration and TriggerDeclaration are as follows and we will not explain them in details due to space limitation.

ParticipantDeclaration ::= **users** MemberDeclarations ';'
MemberDeclarations ::= MemberDeclaration [',' MemberDeclarations]
MemberDeclaration ::= Expression
TriggerDeclaration
::= **trigger** [IDENTIFIER **as**] Actions EventDeclaration [**where** Expression] ';'
Actions ::= MethodCall [',' Actions]

In the first section, we have mentioned that Cova bridges the gap between process modeling and instance enactment by encapsulating information produced during runtime as special objects. These special object variables are listed below.

• An object called process and of type CProcessInstance (defined in Cova library). This variable represents a current running instance of a process. One can manipulate the running instance using this object.

• Activity object corresponding to an atomic activity. An activity object has the same name as the activity and its type is specified by the IDENTIFIER following the keyword handle (refer to activity declaration above).

• Activity instance object. This object is of type CActivityInstance (defined in Cova library) and its name has the form AX.activity, where AX is the corresponding activity name.

With the above variables, some complex constraints can be specified during build time. For example, we can easily specify that the participants of activity A_2 are the same as those of A_1 by adding the following statement to the body of A_2.

users A_1.**activity**.GetParticipants();

3 Platform Implementation

Based on the formal model, we have implemented a general platform, which can be used to develop cooperative applications. In this section we will examine the platform architecture and some advanced topics such as synchronous cooperation support, transaction management and service integration.

3.1 Platform Architecture

CSCW applications cover a wide spectrum and each application may have its own specific requirements. In addition, systems may evolve as time goes on. So as a platform, it must take these into account. Thus, we devise the platform architecture as illustrated in Fig.1. It is based on Cova run-time system with two additional components, namely Interface Services and Supporting Services. The whole platform forms three layers and they are explained as follows.

1. Interface service layer

This layer externalizes the functionalities of Cova run-time system and makes it possible to develop applications based on the platform. Services in this layer are explained below.

• Worklist manager: it is used to manage workitems (e.g., retrieve all workitem names, add a new workitem to or remove an existing one from the worklist of certain users).

• Workitem handling: it is used to fulfill various tasks (e.g., create a new instance of some process, terminate a process instance, and open an activity object).

• Instance monitor: it is used to track the cooperation progress.

• Process manager: it is used to maintain (save or delete) process or class definitions.

• System configuration: it is used to configure the platform.

These services are supplied to developers as a set of APIs similar to workflow application programming interfaces recommended by Workflow Management Coalition (WfMC). With these APIs,

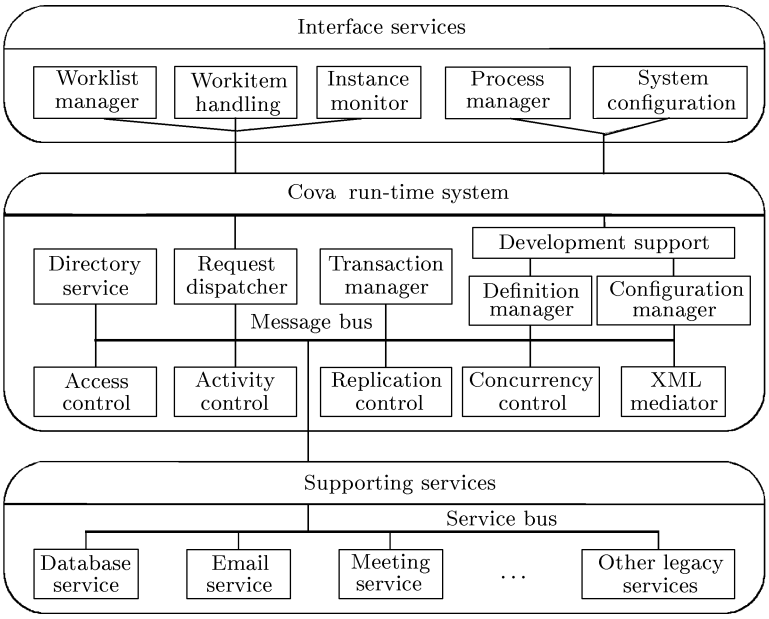

Fig.1. Cova platform architecture.

developers can develop new applications using the services provided by Cova run-time system.

2. Cova run-time system layer

This layer is the core of the platform. It provides various general-purposed services that implement complex cooperation control mechanisms and guarantee the cooperation semantics specified by the Cova specification language. Services in this layer are glued together via "message bus", which takes messages produced by one service and then forwards them to the appropriate input message queue of the other services. Below are functions of some services. As for functions of the other services, please refer to [3].

• Activity control: it is in charge of creation, activation, deactivation, synchronization and termination of activities as well as message exchange between them. It is activity control service that enables asynchronous cooperation of relatively long duration.

• Replication control: in real implementation, Cova adopts fully-replicated architecture. The purpose of replication control is to maintain the consistency of replicas when they are opened, closed or saved dynamically by multiple users and/or applications.

• Concurrency control: it is used to maintain the causal dependencies among operations and the logical-equivalence of the results produced at different sites under fully-replicated architecture. The algorithm adopted is oodOPT[11], which utilizes operation semantics to transform the operations before executing them. This is possible because the coordination part of Cova has full knowledge about the semantics of the computation part as mentioned in Section 1.

• XML mediator: it aims to tackle data inconsistency issue resulted from heterogeneous platforms and different software vendors during system interoperation and service integration. We will examine it later in this section.

• Transaction manager: it is responsible for coordinating all running activities and guaranteeing data integrity. It is the key to support mission-critical applications.

• Definition manager: it maintains process related information (e.g., process, class or user definitions), which provides the basis for cooperation.

• Configuration manager: it is used to maintain system configuration information such as service access point, available underlying services and so on. With this service, we can tailor the platform to adapt to specific applications.

• Request dispatcher: it receives user requests as well as external events and then dispatches them (via message bus) to appropriate modules to handle.

3. Supporting service layer

This layer is exploited to meet the requirements on application specific functions and system evolution by integrating application-specific services or deploying new ones. It is an open layer in the sense that new services can be added to and existing ones can be removed from it (via system configuration

service). Various (legacy or new) services are accessed uniformly via a "service bus", which works in the way similar to that of ORB in CORBA architecture. Service integration issues are detailed later in this section.

A platform constructed in this way has the following features.

• It can reduce the application development work based on the specification language, APIs and services supplied because the specification language provides constructs for describing cooperation scenarios, the services implement the complex cooperation control mechanisms, and the APIs make it possible to access the services.

• It can adapt to system evolution and wide application demands by service integration.

3.2 Synchronous Cooperation Support

It has been mentioned previously that Cova can support both synchronous and asynchronous cooperations. While asynchronous cooperation is enabled by activity control, synchronous cooperation is enabled by replication and concurrency control. In this section, we will examine how it is accomplished.

In real implementation, to reduce response time as well as to increase reliability, Cova platform adopts a fully-replicated architecture that consists of a centralized server and fully replicated clients. Users use Cova client to access activity objects stored at the server. The outstanding feature of Cova client is that it provides object replication and concurrency control mechanism, which makes it possible to support synchronous cooperation. Replication control algorithm automatically replicates the latest activity object state to local site. It also ensures that all involved clients have identical copies at the end of the procedure. Below is shown the replication control procedure for opening an object.

Algorithm 1. Replication Control Procedure for Opening an Object

```
Input
  oid: Id of the activity object to open
  sid: Id of the site issuing the request
Output
  True: If the object is opened successfully
  False: Otherwise
{
  if (oid is not found) //no such an object
    return False;
  if (oid has been opened) {//join the session
    r ← the real-time session corresponding to oid;
    foreach (s ∈ S_r) //synchronization
      Send STOP_SENDING_REQUEST to s;
    Wait for STOPPED from each s ∈ S_r; //end of
    //synchronization
    Retrieve latest object state from some s ∈ S_r;
    Send class definition and latest object state to
    sid;
    Wait for OBJECT_CONSTRUCTED from sid;
    S_r ← S_r + {sid};
  }
  else { //Object is not opened, so create a new
       //session
    Create a real-time session r;
    Send the class definition and object state to sid;
    Wait for OBJECT_CONSTRUCTED from sid;
    S_r ← {sid};
  }
  foreach (s ∈ S_r) //restart
    Send RESTART to s;
  return True;
}
```

The replication control procedure for saving or closing an object is similar to the above except that there is no need to send the latest state to client. In addition, we should point out that the presented procedure has been simplified with no consideration of exceptions. At the end of replication control procedure, all clients have the identical copy of the object. After that, each participant manipulates the local copy independently. To achieve awareness, operations generated at each client are also multicast to the other clients. During this process, oodOPT provides the function to keep the consistency of these replicas and the results produced by the same operation at different clients. In this way, synchronous cooperation is well supported. Here we should point out that replication control provides a good basis for oodOPT, which will not produce correct results if the original object copies are different.

3.3 Transaction Management

As organizations get increasingly dependent on the cooperative systems to carry out their daily activities, it becomes an important issue to keep the system reliable and thus transaction management is introduced into the platform. Transaction support by itself is an old yet young research topic. By old, we mean that there have been well-established concepts such as ACID properties and serializability, which were developed in relational database field and have extended to support advanced applications[12,13]. By young, we mean that there are still a lot of basic challenges, especially

in CSCW domain. It is widely acknowledged that the so-called traditional transactions are unsuitable for cooperative applications. While a considerable number of new transaction models (e.g., SAGAS[14], CoAct[15], Flex[16]) have been proposed in the research literature over the years in order to deal with the challenges, and although they have many theoretically interesting features, they tend to be too complicated to implement and use. From our point of view, the reason for this dilemma lies in the lack of effective approach to describing cooperation and thus the run-time system knows little to do. With Cova specification language, we believe, the bottleneck is broken, at least to some extent.

The transaction model adopted in our platform is CovaTM[17]. A CovaTM transaction is viewed as one execution of a cooperative process with its sub-transaction corresponding to activity. Formally, it can be defined as follows.

Definition 3 (CovaTM Transaction). *A CovaTM transaction, denoted by T, is a 5-tuple* $\langle ST, PP, TP, AS, TM \rangle$, *where*

• *ST is the set of all sub-transactions of T. For each sub-transaction, we need to specify its type and participants. Three types available are CP (compensable), UC (uncompensable) and CT (composite). Sub-transaction of CP or UC type corresponds to atomic activity, while sub-transaction of CT type, which is also a CovaTM transaction, corresponds to composite activity. The introduction of CT type reduces the control complexity and enhances flexibility. Moreover, sub-transactions in CovaTM may be reactivated after submission thus forming a graph rather than a tree. ST can be directly obtained from process description.*

• *PP is the set of all precedence predicates defined on ST. A precedence predicate is a Boolean function defined on the space of execution states and specifies the execution dependencies among sub-transactions. In CovaTM, two basic dependencies defined are positive dependency and negative one. A positive dependency between sub-transactions* t_1 *and* t_2 *exists if* t_1 *cannot be executed until* t_2 *succeeds. Whereas a sub-transaction* t_1 *negatively depends on* t_2 *if* t_1 *has to wait until* t_2 *has aborted before it can start. PP is specified by state transition rules.*

• *TP is the set of all temporal predicates of ST. It reflects time constraints on transaction. TP can be obtained from time-related rules.*

• *AS is the set of all acceptable states of T. Usually there is more than one way to achieve a business goal, so T may have multiple acceptable states. AS is also specified by state transition rules.*

• *TM is the set of administrators (the M in the formal process definition) of T. It is up to transaction administrators to handle emergent circumstances such as occurrence of undefined exceptions and failures.*

The outstanding feature of CovaTM lies in the following. 1) It allows alternative execution paths (there may be many ways to fulfill the same task) as well as dynamic user intervention (accomplished via interface service). Therefore, it can adapt to environment changes and incomplete cooperation description. 2) Integrated exception handling and recovery are provided. Once an exception occurs during execution, it is captured by Exception Handler. If the exception is recognizable (i.e., the handling rules have been specified), it is handled accordingly. Otherwise it is reported to transaction administrators for further handling. During this process, transaction recovery may be needed to withdraw some work that has been done. In CovaTM, compensation-based transaction recovery is exploited. For more details, please refer to [17].

3.4 Interoperation and Service Integration

As business cooperation becomes more and more pervasive, interoperations between cooperative systems are inevitable. On the other hand, a good platform should cover as many applications as possible. To meet the requirements, XML Mediator and service integration are introduced.

The common issue faced by system interoperation and service integration is the data inconsistency resulted from heterogeneous platforms and different software vendors. The traditional solution is to develop a unified access interface to the heterogeneous data through schema reconciliation and explicit data transformations[18]. However, this tight integration requires intensive human efforts and practices have witnessed the failure of such projects. Though the emergence of distributed object technologies (e.g., CORBA, DCOM and EJB) facilitates interoperation and integration, they focus on wrapping the components and routing primitive requests rather than the actual mediation. We argue that the launch of XML will change the situation essentially due to the inherent advantages in data organization and expression. Indeed, XML has been used successfully to normalize data consumed by different sources.

In our platform, XML Mediator is introduced to reconcile heterogeneous data via XML messages.

The structure of XML message is shown in Fig.2. It consists of three parts, namely message envelope, message header and message body. Message envelope defines communication related parameters such as message source and target addresses as well as message ID. Message header specifies message attributes such as names of sender and receiver, message type. Message body contains application specific request. For each XML message, we should also specify the schema used, which defines application specific data types adopted within message body.

```
⟨XMLMessage schema="schema1.xsd"⟩
  ⟨Envelope⟩
    ⟨MessageID⟩12345⟨/MessageID⟩
    ⟨Target⟩192.168.1.1⟨/Target⟩
    ⟨Source⟩192.168.1.10⟨/Source⟩
  ⟨/Envelope⟩
  ⟨Header⟩
    ⟨Receiver⟩John⟨/Receiver⟩
    ⟨Sender⟩Jack⟨/Sender⟩
    ⟨Type⟩CreateProcessInstance⟨/Type⟩
  ⟨/Header⟩
  ⟨Body⟩
    ...
  ⟨/Body⟩
⟨/XMLMessage⟩
```

Fig.2. XML message format.

With XML message defined, the system can then handle the requests from various applications automatically. Although wrapping existing systems is needed, it is still much cheaper than that of object-oriented methods. In addition, this loosely coupled way enhances the system flexibility.

Cova platform exploits service integration to enhance system functionality and applicability. Besides data integration issue, which has been solved by XML message as above, service integration should solve the following additional issues.

• Service description: service description is used to specify a service offered by a system. It concerns general properties of service as well as service interface. This is the foundation of transparent service invocation.

• Service registration: the precondition for a service to be found is to make it known publicly at some place. Service registration is exploited to achieve this goal.

It can be achieved via the following procedure.

1) Wrap the legacy system or build a new one with some public interfaces exported as a dynamic link library (DLL), say pnative.dll, in other programming languages.

2) Write some codes in CODL to describe the interface specification. A rule that must be obeyed for interface specification is that the interface name as well as the number of parameters must be the same as defined in DLL in step 1). Moreover, their parameter types must be compatible.

3) Enroll the service. As a result, a registration item is added to the system configuration file, which has the form of "CEMail=pnative.dll, plib.p, CEMailAgent", where CEMail is the name of service implemented or encapsulated in pnative.dll and CEMailAgent is the interface specification defined in Cova source file plib.p. Based on the service description, service bus can load the corresponding module when needed. All services available can be obtained from directory service.

At the end, we should point out that our platform also provides a way (via system configuration interface) to dynamically load/unload services, which makes the platform tailorable and thus platform adaptability is enhanced.

4 Application Development: A Case Study

In this section, we will illustrate how to develop a general workflow management system (WfMS) with Cova platform. Note that it is also feasible to develop other applications based on the platform. More examples can be found in [9]. This is possible owing to the features of the Cova specification language and run-time system.

According to Workflow Management Coalition (WfMC), WfMS is a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic. As Cova platform is based on a formal model similar to that of WfMS and Cova run-time system has provided the cooperation control functions, the construction is straightforward. For example, activity control serves as workflow engine and interface services act as workflow APIs. Workflow specifications can be described by the Cova specification language. The architecture of the resulting system is illustrated in Fig.3, where functions directly available are marked with bold line. All the work involves application specific functions rather than cooperation control. Table 1 lists all the steps needed to construct a WfMS by Cova, where steps needed in traditional way are also given for comparison.

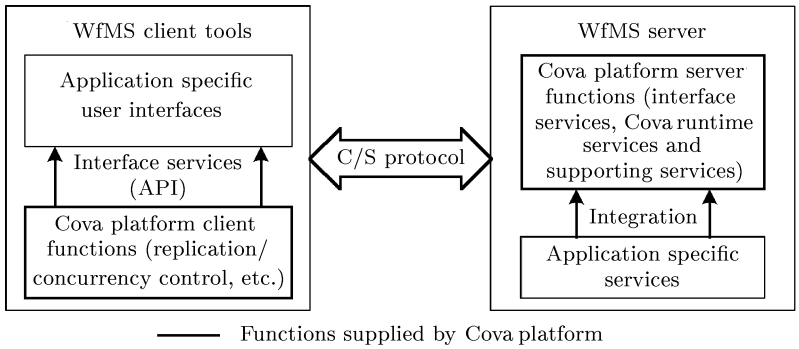

Fig.3. Workflow management system architecture by Cova platform.

Table 1. Steps Needed to Develop a WfMS

Step	By Cova	In Traditional Way
1	Requirement analysis	Requirement analysis
2	Model description in Cova language	Model design
3	–	Design and implementation of workflow engine, APIs, process management, etc.
4	Client tools development with APIs	Client tools development from scratch
5	Deployment	Deployment

We can see both approaches should analyze application requirements first. Otherwise the resulting system cannot achieve its goal. In traditional way, once the requirements are figured out, a formal model should be designed and based on it, the management system is developed. In more details, developers should implement workflow engine, provide workflow APIs and supply process management functions. It is a complicated task and full of risk factors (e.g., errors or deficiencies are easily introduced). With Cova platform, since the formal model has been defined and the necessary services for process enactment and management have been supplied by the run-time system, all endeavors needed are 1) converting the application requirements to process description in Cova specification language, and 2) compiling the codes got and saving the results to Cova server, and 3) implementing required services and integrating them into the server following the procedure presented in Subsection 3.4. Therefore, the development efficiency is greatly enhanced. Client tools development in traditional way should implement the communication protocol between server and clients. This is not needed for Cova because the protocol has been shipped with interface services. So the efficiency is also enhanced. Moreover, system developed in this way has the following features as supplied by Cova.

1) It provides good support for process reuse via process inheritance and composite activity. As we all know, it is time-consuming to define a complex process even if working collaboratively. With this functionality, process designers' burden is eased.

2) It can support synchronous activities as explained previously. This function is either short in most workflow products or implemented in other non-uniform ways. As a result, concurrency control is difficult if it is not impossible.

3) It can interoperate with other cooperative or legacy systems fluently due to the inherent support for XML message and service integration.

4) It can support mission-critical tasks since transaction management is supplied by the platform. Most workflow products fall short of this functionality.

5 Conclusions and Future Work

We have shown a platform capable of uniformly modeling cooperative scenarios in different modes and enhancing the efficiency of groupware development. The way adopted by Cova can be summarized as follows.

Firstly, based on an abstraction of groupware systems, a formal model is defined, with which various aspects of groupware systems can be investigated. This forms a solid foundation of the platform. Based on the formal model, Cova specification language is devised for uniformly describing cooperation scenarios. Extra flexibility is also gained by its clear separation and seamless integration of the computation and coordination parts of a cooperation process, and introducing run-time information into process description.

Process definitions alone are not enough for supporting cooperation, so Cova run-time system is implemented, which provides some general-

purposed necessary services (e.g., activity control, replication and concurrency control) for process enactment. On account of diverse application domains and requirements, transaction management and service integration are also introduced to enhance platform applicability. These services are made available through interface services to facilitate application development.

By now the platform is largely completed and can run some simple applications. Indeed, based on the interface services provided, we have established a command line tool for users to fulfill their tasks. In addition, two examples (one is the workflow management system given in Section 4 and the other is a co-authoring system) are under development. Our future work includes but is not limited to the following.

• Performance measurement. Since the codes are interpreted, obviously it will lower system performance. In addition, the concurrency control algorithm oodOPT will reduce system performance further for much calculation is required to transform an operation. Therefore, a careful performance evaluation is necessary especially when developing large-scale applications. However, we argue that compared with the flexibility gained, the performance decrease is worthy.

• Process analysis and optimization. The purpose is to ensure process correctness and make the process more effective.

• Support to decentralized servers. Currently there is only one centralized Cova server in our platform. This makes the control mechanisms simple and easy to implement. However, it may become a bottleneck of the platform. One way out is to deploy various cooperative services on top of grid infrastructure. We believe with the grid facilities, such a solution would be cost-effective.

References

[1] Ellis C, Gibbs S, Rein G. Groupware: Some issues and experiences. *Commun. ACM,* 1991, 34(2): 38–58.

[2] Ellis C, Wainer J. A conceptual model of groupware. In *Proc. ACM Conf. Computer Supported Cooperative Work,* Smith J B, Smith F D, Malone T W (eds.), Chapel Hill: ACM, 1994, pp.79–88.

[3] Yang Guangxin, Shi Meilin. Cova: A programming language for cooperative applications. *Sci. China, Ser. F,* 2001, 44(1): 73–80.

[4] Li D, Muntz R. COCA: Collaborative objects coordination architecture. In *Proc. ACM Conf. Computer Supported Cooperative Work,* Poltrock S, Grudin J (eds.), Seattle: ACM, 1998, pp.179–188.

[5] Roseman M, Greenberg S. GroupKit: A groupware toolkit for building real-time conferencing applications. In *Proc. ACM Conf. Computer Supported Cooperative Work,* Turner J (ed.), Toronto: ACM, 1992, pp.43–50.

[6] Patterson J F, Hill R D *et al.* Rendezvous: An architecture for synchronous multi-user applications. In *Proc. ACM Conf. Computer Supported Cooperative Work,* Halasz F (ed.), Los Angeles: ACM, 1990, pp.317–328.

[7] Crowley T, Milazzo P *et al.* MMConf: An infrastructure for building shared multimedia applications. In *Proc. ACM Conf. Computer Supported Cooperative Work,* Halasz F (ed.), Los Angeles: ACM, 1990, pp.329–342.

[8] Weber M, Partsch G, Hock S *et al.* Integrating synchronous multimedia collaboration into workflow management. In *Proc. Int. ACM SIGGROUP Conf. Supporting Group Work*, Hayne S C, Prinz W (eds.), Phoenix: ACM, 1997, pp.281–290.

[9] Yang Guangxin. A uniform meta-model for modeling integrated cooperation. In *Proc. ACM Symp. Applied Computing,* Lamont G B, Haddad H *et al.* (eds.), Madrid: ACM, 2002, pp.322–328.

[10] Gelernter D, Carriero N. Coordination languages and their significance. *Commun. ACM,* 1992, 35(2): 97–107.

[11] Yang Guangxin, Shi Meilin. oodOPT: A semantics-based concurrency control framework for fully-replicated architecture. *J. Comput. Sci. Technol,* 2001, 16(6): 531–543.

[12] Elmagarmid A K *et al.* (eds.). Database Transaction Models for Advanced Applications. San Mateo: Morgan Kaufmann, 1992.

[13] Jajodia S, Kerschberg L (eds.). Advanced Transaction Models and Architectures. Boston: Kluwer Academic Publishers, 1997.

[14] Garcia-Molina H, Salem K. SAGAS. In *Proc. ACM SIGMOD Int. Conf. Management of Data,* Dayal U (ed.), San Diego: ACM, 1987, pp.249–259.

[15] Rusinkiewicz M, Klas W *et al.* Towards a cooperative transaction model — The cooperative activity model. In *Proc. Int. Conf. Very Large Data Bases,* Dayal U, Gray P, Nishio S (eds.), Zurich: Morgan Kaufmann, 1995, pp.194–205.

[16] Elmagarmid A K, Leu Y *et al.* A multidatabase transaction model for InterBase. In *Proc. Int. Conf. Very Large Data Bases,* McLeod D, Sacks-Davis R, Schek H J (eds.), Brisbane: Morgan Kaufmann, 1990, pp.507–518.

[17] Jiang Jinlei, Yang Guangxin, Wu Yan, Shi Meilin. Cova-TM: A transaction model for cooperative applications. In *Proc ACM Symp. Applied Computing,* Lamont G B, Haddad H *et al.* (eds.), Madrid: ACM, 2002, pp.329–335.

[18] Shegalov G, Gillmann M, Weikum G. XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB J,* 2001, 10(1): 91–103.

SHI MeiLin got his B.S. degree in computer science in 1962 from Tsinghua University. His major research interests focus on computer supported cooperative work (CSCW) and computer networks. He is currently a full professor at the Department of Computer Science and Technology, Tsinghua University.

JIANG JinLei got his B.S. degree in computer science in 1999 from Tsinghua University, where he is now a Ph.D. candidate. His research interests include computer supported cooperative work (CSCW), workflow management systems (WfMS), advanced transaction processing.