A Uniform Meta-Model for Modeling Integrated Cooperation

Guangxin(Gavin) Yang
Bell-Labs Research, Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ 07974, USA
gxyang@acm.org

Abstract

Process modeling is a central topic of research on Computer Supported Cooperative Work (CSCW). Uniformity and flexibility in work representation and process enactment are two primary goals. We develop a novel meta-model capable of modeling uniformly a wide range of cooperation scenarios. We will discuss the key elements of the model and its computerized formalism, the Cova programming language, and its runtime system. We will describe in this language several typical cooperation scenarios to illustrate how integrated cooperation and other cooperation scenarios can be described and supported with our model.

Keywords

Process Modeling, CSCW, Groupware, Cova

1 INTRODUCTION

Recent advances of research on supporting cooperative work have resulted in numerous *groupware* [3] and *meta-groupware* [2, 8] systems, which have different runtime environments and application areas. Although these systems greatly facilitate interactions among widely distributed cooperators, their separation and independence from each other hinder the cooperation among users of different systems. For example, the artefact created in a synchronous co-authoring tool could hardly be used in any finely granulated manner by an asynchronous message passing system because the latter usually has no semantic information of the artefact.

The difficulty comes from the fact that each system is designed to support a specific type of cooperation. Little attention has been paid to how different types of cooperation can be supported in a comprehensive way. However, cooperation in real world settings is usually a combination of *single user* activities, *synchronous* and/or *asynchronous* cooperation. For example, even a simple *document review* process may be decomposed into an *authoring* activity and a *reviewing* activity, which may be enacted sequentially. Either of them may be carried out by only one user on his/her own, or by a group of cooperators via synchronous editing or asynchronous authoring.

We call this type of cooperation as *integrated cooperation*, which denotes a process that involves single-user, synchronous or asynchronous multi-user activities, and other integrated cooperation. To the best of our knowledge, there are only a few prototypes, e.g. WoTel [17], that are capable of supporting a specific type of integrated cooperation. Unfortunately, no meta-groupware system is capable of supporting the development of this type of applications.

One may think that integrated cooperation can be supported in a way similar to that of WoTel by defining a set of interfaces among different groupware systems to bridge the islands so that they can be integrated into another. Though this seems to be a quick solution, it is not a perfect one. The reason lies in the fact that each system has a different meta-model for describing cooperation processes it supports. Even though some systems may have similar meta-models, the formalisms that are used to describe them are different. These differences make it difficult for one system to be fully and flexibly integrated into another one through interfaces. For example, lacking the semantic information of the objects in the integrated system makes it difficult for the integrating system to take full advantage of these objects.

This paper addresses the challenge to support integrated cooperation from a more systematic point of view. We propose a novel meta-model for uniformly describing integrated cooperation processes as well as *single-user, synchronous*, and *asynchronous* processes. We show with several examples how this meta-model can be used to model a wide range of cooperation scenarios. The runtime environment based on this model will also be discussed to show how the semantics of the key constructs of this meta-model are implemented.

Process modeling by itself is an old yet young research topic. By *old*, we mean that there have been well-established concepts, e.g. *activities, triggers, roles*, etc, which were developed in various fields, e.g. *Computer Supported Cooperative Work, Software Engineering, Parallel and Concurrent Computing* and so on. By *young*, we mean there are still lots of basic challenges [1]. Modeling integrated cooperation is an important one of them. We will use the aforementioned conventional terms as the basic constructs of our meta-model. However, we will see that they will be assigned with novel meanings and semantics. We will also show that the mechanisms to implement these semantics are quite different from those used in other areas.

Our meta-model is based on an abstract of groupware systems [9]. The abstraction defines a framework within which various aspects of groupware systems can be investigated. The coming section is focused on the description of this meta-model. The 3rd section gives a brief introduction to its computerized formalism, the Cova programming language and system. Four examples will be given in the 4th section to illustrate how the meta-model can be used to describe uniformly different cooperation scenarios. Comparison with other related work, conclusions and directions for future work are presented in the last two sections.

2 THE META-MODEL

As we have stated, our meta-model is determined to be *uniform* in the sense that it should be able to model different cooperation scenarios, i.e. synchronous, asynchronous, integrated, even single user activities, in a uniform way and with great flexibility. Mathematically, a cooperation scenario described with our meta-model is a *set*, with each element describing a piece of work that contributes to the cooperation. Among these described, there are what the piece of work is, how it is being done, what it will receive from and send to other elements, etc. As in other models, the scenario is called a *process* and the description a *process definition*, which will be used to control the enactments of process instances. An element in a process is called an *activity*, which is the basic construct of our meta-model. Though the terms seem to be quite ordinary, we will show how they are different from other definitions.

2.1 Uniform Activities

An activity in our model is defined as *an active computing entity with a goal, a life cycle, and rules regarding how its state changes and how it interacts with its environment*. It is a *computing entity* in the sense that it maintains a set of states which records the results of the piece of work it describes and has an execution thread, whose execution is driven by the *input* messages from the activity's external environment. The *goal* of an activity is the desired state in which the piece of work is considered to be completed. By *active*, we mean the activity will, under certain conditions, generate *output* messages actively to its external environment, which are all the activities that communicate with it.

Intuitively, an activity plays a role similar to a secretary that holds the information related to a particular piece of work and enforces the rules that control accesses to the information. These that could access the information are called *activity participant*, who may be a (group of) human being(s) or computing process(s) capable of performing a certain action. Activity participants, under certain constraints, will access the state variables maintained by the activity to make it change little by little to the desired states. However, these accesses are not arbitrary, but under the guidance of the activity rules. Eventually participants will determine that the goal of the activity is reached. At this time, the activity may generate outputs for other activities to share the results it produces. The computation model embodied by an activity is shown in Figure 1.

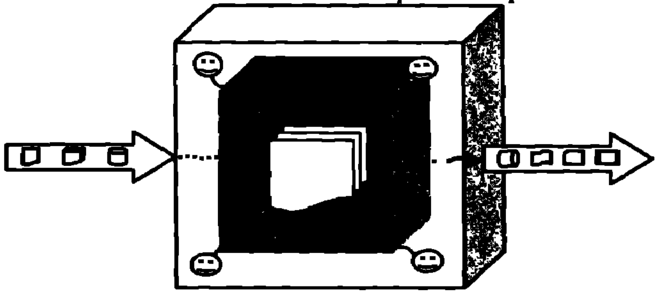

Figure 1. A uniform activity. What lies in its center is a set of variables that can be accessed through different interfaces and in different modes. They record the progresses and the final results of the activity. The curved arrow lines indicate accessing threads, which may be either simultaneous or exclusive (different line styles). The block arrows with small shapes (messages) inside are the communication channels among activities.

2.1.1 The Life Cycle

The concept of the *life cycle* of an activity is similar to that of a process or a thread in an operating system. An activity may navigate through different stages, each of which may have different effects on how the state variables it maintains can be accessed. In our model, there are six stages defined, i.e. *initiated, active, suspended, stabilized, aborted,* and *completed.*

initiated is a transient stage which becomes *active* or *suspended* after an activity is initialized. The difference between *active* and *suspended* is whether the participants can access the state variables. Only the variables maintained by an *active* activity are accessible, either exclusively or simultaneously, by one or multiple cooperators from one or multiple network sites. In this way, an activity is made capable of supporting both single user activity and synchronous cooperation.

An activity becomes *stabilized* when the participants think the goal of this activity has been reached. It is different from *completed* in that a *stabilized* activity can be reactivated when the participants later realize that the goal is not actually reached or a new goal is raised. In either case they need to further access the variables to push them into new states. Since this often occurs in real-world settings, by introducing this new stage, the activity model is made more flexible.

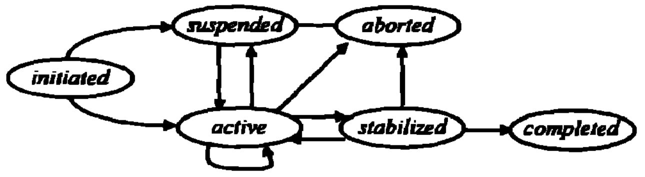

Figure 2. Transitions among different stages of an activity.

The transition paradigm among different stages of an activity is shown in Figure 2. For simplicity, the actions that cause these transitions are not shown in the figure. These actions can be either internal or external. Internal actions are generated by the runtime system. For example, the transition from *initiated* to *active* or *suspended* is done by the runtime system according to the activation rules (see below) and the state variables. External actions are usually raised by participants. For example, they may abort an *active, suspended, stabilized* activity. In all cases a transition to *aborted* will occur.

2.1.2 Activity Rules

Rules are a widely adopted mechanism in many fields.

Generally they specify under what circumstances what will occur, what are allowed, what are denied, etc. Typical examples include the trigger definitions in an active database system [6], the access policies in a network node, the product rules in an AI system, etc. They differ in their purposes, the formalisms, the constructs and their semantics, how the semantics are implemented, etc.

Our model uses rules for three purposes. The first one is to specify under what conditions an activity will become *active*, thus makes its state variables accessible. Any state change of the activity causes the runtime system to check this condition to see whether it needs to transit the activity into the *active* or *suspended* stage.

The second purpose is to define the ways in which state variables are accessed. Details include *who*, under *what conditions*, can take some *actions* in an activity. For example, some of these rules may specify whether simultaneous accesses to the state variables are allowed, while others may specify something like access control policies. Actions include *retrieving* and *updating* the state variables, transiting activity to a new stage, or a combination of them. Defining access manners at the activity level allows a cooperator to play different roles in different activities, thus provides more flexibility than other models where these policies are defined globally.

The third purpose is to define *when* and *under what conditions* an activity may send some *messages* to other activities. This type of rule is used to build the communication channels among activities for asynchronous information sharing and synchronizing the stages of different activities. A message in our model is a parameterized action on the receiving activity, whose states may be changed after the action is executed. Since the activity knows all its semantics information, messages are composed with the states of the activity and some globally accessible information in a finely granulated manner. With these semantics information, more flexible and sophisticated rules can be formed, thus provides much more flexibility than these models where the semantics information is unavailable.

While rules can be used to define the cooperation policies, they may impose some limitations on the cooperation, which will then lead to inflexibility. In our model, rules are *optional*, which means that an activity can have no rules or only some types of rules. Therefore, if there are no communication rules defined, an activity alone can be used describe either a single user activity or synchronous cooperation. With the communication rules, asynchronous cooperation can also be described. It is in this sense that we call the activity a uniform model.

2.2 Integrated Processes

Based on the uniform activity model, an integrated process is defined as a set of inter-related activities. The inter-relationship among activities is established by their communication rules. We call it an integrated process model because it has the capability of modeling integrated cooperation processes as discussed in the introduction. We will show how this is done in the forth section.

The integrated process model has several distinct features. The first one is its loose mathematical structure. Compared with other models, e.g. the Petri Net -based [14], the specialized grammar [20], the directed graph [21], which have more rigid logical structures, a loose logical structure imposes less limitations on how different activities are related to each other while maintains equal, if not stronger, expressive power. For example, it should be easy to show that any process modeled with a directed graph can be modeled with our integrated process model.

The second feature would be the fact that our meta-model is oriented to a cooperation process being enacted, not a process template in existing systems that is used to guide the enactment of process instances. Based on this orientation, a lot of dynamic information that is generated during process enactment can be used in process modeling. With the dynamic information, we can get extra flexibility otherwise not available. For example, based on the name of the author of a document, we may define the participant of a review activity to be the author's supervisor in the system directory.

The third feature would be its hierarchicalness. Similar to that each activity represents a piece of work, a process as a whole represents a larger piece of work. The state of a process is naturally defined as the combination of all the states of its component activities. It records how the process is progressing and what the latest results are. In this sense, a process can also be regarded as an activity, which can then be used as a component of another process. Therefore, process definitions can be nested to form a hierarchical structure.

3 *CovA* LANGUAGE AND SYSTEM

We have discussed the basic ideas and constructs of our meta-model. However, without a formal method, it could hardly be used to model any cooperation processes for practical use. Our approach is to develop a programming language based on this meta-model and its runtime system, which implements the semantics of the language constructs and provides a set of computation services for enacting cooperation processes. Due to page limitations, the discussions here will be kept as brief as possible.

3.1 A Brief Overview

The language, *CovA*, gets its name from the bold letter in the phrase '*Co*operati*v*e *A*pplications'. Similar to many coordination-oriented languages and models [14], *CovA* also adopts the idea to separate the computation and coordination parts of a process. The first feature that distinguishes *CovA* from other languages is that its coordination part has full knowledge about the semantics of the computation part, e.g. how it is structured, how it can be operated, etc. With this information, the runtime system is able to do some advanced controls which otherwise is impossible. For example, it can be used for more flexible concurrency control when an object is accessed simultaneously by multiple cooperators [10].

The second feature is that the computation part in our model is not an independent program that has its execution logic. Instead it provides only the descriptions on how a piece of data is structured and how the operations on it are implemented. How the operations are used is up to the cooperators. Generally, they can use an arbitrary combination of these operations to access that piece of data. If the combinations are viewed as execution logic, then each cooperator can 'build' a virtual program that best fits his/her needs to finish his/her piece of work. Obviously, this will greatly increase the flexibility compared with other solutions where only some specific programs can be used.

3.2 Object Description Language

As we have mentioned, an activity maintains a set of state variables. There are many design alternatives on how these variables are described. We have chosen to implement an enhanced version of ODMG's Object Model [18] for this purpose. The *COVA* Object Description Language or CODL in short, provides the language constructs for describing the structures of an object and implementing its methods, which, as we have mentioned above, becomes part of the computation description of a cooperation process. Similar to other object-oriented programming languages, these descriptions appear as class definitions. Due to page limitations, here we will not list its syntax rules. We will give some examples of class definitions later.

3.3 Coordination Description Language

Based on CODL, the *C*OVA *C*oordination *D*escription *L*anguage or CCDL in short, provides the language constructs for describing the components of the model outlined in the 2nd section. It is based on CODL in the sense that many descriptions in CCDL use the descriptions in CODL. In CCDL, a process description is defined with the following syntax:

```
ProcessDeclaration ::= PROCESS QualifiedName [EXTENDS
        QualifiedName] ProcessBody
ProcessBody        ::=         '{'          [ClassDeclarations]
        [ProcessDeclarations] ActivityDeclarations '}'
ActivityDeclarations ::= ActivityDeclaration [ActivityDeclarations]
```

The first rule states that a process definition begins with the keyword *process* and a process name. Optionally, it can have a super process, from which all definitions in the process body are inherited. Thus CCDL is also object-oriented. Besides a set of activity definitions, the process body may contain an optional set of class definitions (given in CODL) and an optional set of nested process definitions (given in CCDL), both of which may be used in the activity definitions.

Activity definitions may have two different forms. The first one is given with the following syntax:

```
ActivityDeclaration ::= ACTIVITY QualifiedName [HANDLES
        QualifiedName]      [STARTSWHEN       Expression]
        ActivityBody
```

Each activity has a name, which is unique within a process. The optional HANDLES clause specifies the name of a class, whose definition is given in the ClassDeclarations section. The activity maintains an instance of this class as its state variables, which we call it *activity object*. Participants can use its methods to access the activity object. The STARTSWHEN clause specifies with a Boolean expression under what condition the activity may become active, thus allows its activity object to be accessed.

The activity body is defined with the following rules:

```
ActivityBody       ::= '{' [ActivityAttributes] '}'
ActivityAttributes ::= ActivityAttribute [ActivityAttributes]
ActivityAttribute  ::= ParticipantDeclaration | RoleDeclaration |
        TriggerDeclaration
TriggerDeclaration ::= TRIGGER CovaMethodCallList WHEN
        EventDeclaration [WHERE Expression] ';'
```

It contains the rules on how the activity object can be accessed (given by ParticipantDeclaration and RoleDeclaration) and how the activity communications with other activities (given by TriggerDeclaration). The common point of these declarations is that they all may contain expressions composed with method calls to and attributes of the activity objects and the other two specialized object, **process** and **activity**, which refer respectively to the process and activity. We will discuss how they are used in the example sections.

The second form of activity definition is given by the following syntax:

```
ActivityDeclaration ::=   ACTIVITY    QualifiedName    AS
        QualifiedName;
```

The second QualifiedName is the name of a process definition, whose activities become activities of this process. In this way, a process can be nested in another process. In this way the hierarchical ness of processes is achieved.

3.4 Cova Runtime System

Process definitions alone are not enough for supporting cooperation. Based on the uniform meta-model, the Cova runtime system (CovaRT) implements the semantics of process definitions and provides a set of computation services that are necessary for enacting integrated cooperation. It adopts a hybrid architecture that consists of centralized servers and fully replicated clients. Cooperation processes are maintained as the first-class entity in Cova servers. The Activity Control service of CovaRT provides all the functions needed for process control, e.g. *creation*, *cancellation*, *suspension*, generating and executing messages for activities, flexible transaction management, and exception handling. These functions are implemented as the Cova Transaction Management model or CovaTM in short. More details about CovaTM can be found in [11].

The fully replicated clients are the key component for participants to access activity objects. As part of a process, activity objects are kept at a Cova server. When it becomes accessible, a participant may 'open' it through a Cova client, which runs at the participant's site. At this time, the Cova client gets the latest state and class definition of the object and keeps them locally. It has an interface through which the application used by a participant can retrieve the state of the object and execute an operation on it.

The uniqueness of Cova clients lies in their capabilities in object replication and concurrency control. When an activity object is accessed simultaneously by multiple participants, the client for a late comer follows a procedure to work with other clients and the server to get the latest state of the activity object. At the end of this procedure, all clients have identical replicated copies of the object. Each cooperator accesses the locally replicated copy independently. Operations generated at each client are multicast to other clients for awareness. Concurrency control is needed to keep the consistency of these replicas and the results produced by executing an operation at different clients. We have developed a fully optimistic concurrency control model, CovaCM, which guarantees the consistency based on the semantics of object structure and operations. Details about CovaCM can be found in [10].

Activity, replication, and concurrency control are the three core services provided by CovaRT. They are essential for supporting integrated processes. Several other services, such a system directory, access control, are also implemented to make CovaRT more practical for real applications [8].

4 EXAMPLES

Now it is time for us to give several examples to show how our model, language, and runtime satisfy the requirements outlined in the introduction. Four examples, a single user application, a synchronous application, an asynchronous application, and an integrated application will be discussed one by one. Our purpose in designing these examples is to show how our model, language, and runtime work in various cooperation scenarios. Therefore we will keep them as simple as possible.

4.1 A Single User Application

Let's begin with the simplest case. A user may work by him/herself on a document, e.g. a business report, a technical paper, etc. This scenario is modeled as the process given in Figure 3. The class CDocument defines semantics of its structure and all possible operations. The activity AAuthoring maintains a CDocument object as its state variable. The first activity rule defines a role that has access to all the three operations. The second rule states that only the creator of the process can access the activity object, i.e. the document, with a role defined by the first rule.

After a PSingleUserAuthoring process is created at a CovaRT server, a CDocument object will be created for the activity and become accessible. With an UI application

```
process PSingleUserAuthoring
{
    class CDocument {
        protect list<char> m_lcText;
        public void Insert(char ch, int pos) {
            insert ch into m_lcText at pos;
        };
        public void Delete(int pos) {
            delete from m_lcText at pos;
        };
        public string GetText() {
            return string(m_lcText);
        };
        ...
    }
    activity AAuthoring handles CDocument
    {
        role Author as
            {execute on Insert,Delete,GetText};
        users as process.GetCreator()
            actas Author;
    };
}
```

Figure 3. Cova codes for modeling a Single User Application.

```
process PCoAuthoring
{
    class CDocument {//same as in Figure 3
        ...
    }
    activity AAuthoring handles CDocument
    {
        role Author as
        {execute on Insert,Delete,GetText};
        synchronous group as anyone
            actas Author;
    };
}
```

Figure 4. Cova codes for modeling a synchronous co-authoring process.

built on the Cova Client, the process creator can get a copy of the document together with its class definition and work on it. His/her operations are translated into calls to the methods of the activity object, which may change its state. The new state can then be transferred to the server and kept there persistently for later accessing. The process models a single user activity in the sense that only one user, i.e. the process creator, can work on the document.

4.2 A Synchronous Process

A synchronous process is one in which multiple cooperators work together on a document to put it into a desired state. Figure 4 gives an example of this type of cooperation. The process definition states that any user in the system can participate and work with others on a CDocument object.

Compared with the definition given in Figure 3, we can see that the only modification is that the participant declaration is changed from a *user* to a *synchronous group*. This makes it possible for other users to access the CDocument object with Cova client-based UI applications while the object is being accessed. In this case, the CDocument object and its class definition are equally replicated at these Cova clients. A joining Cova client works with the server and these clients accessing the same object to make sure they hold a reasonable identical state of the object when the joining procedure is over. Starting from this point, method calls passed to each Cova client will be multicast with a reliable multicast transport service to all other related clients. Each Cova client follows the concurrency control algorithms of CovaCM [10] to make sure the requirements for the consistency model of a replicated architecture are always satisfied.

4.3 An Asynchronous Process

This example simulates a very simple workflow. A user may write a document and send it to his/her supervisor for review. The supervisor may send back some comments for revision. The process may loop a number of times until a document is approved. This scenario is described by the Cova codes in Figure 5. The process definition is inherited from the one defined for the single user application. A new class, CReview is defined for the review activity. AAuthoring is enhanced by adding a communication rule, which will send the prepared document to the 2nd activity AReviewing when the author submits it.

```
process PDocumentReview extends PSingleUserAuthoring
{
    class CReview extends CDocument{
        public list<char> m_lcComments;
        public void SubmitDocument(string doc) {
            m_lcText = doc;
        }
        public void WriteComments(string comments) {
            m_lcComments = comments;
            m_lcText     = "";
        }
        ...
    }
    extending activity AAuthoring
    {
        trigger AReviewing.SubmitDocument(AAuthoring.GetText())
            when submit;
        ...
    };
    activity AReviewing handles CReview
            startswhen (this.GetText() != "")
    {
        role Reviewer as { execute on WriteComments};
        users  actas Reviewer as
            system.Directory.GetSupervisor(
             process.GetActivity("AAuthoring").GetParticipant());
        trigger AAuthoring.Append(AAuthoring.m_lcComments)
            when submit;
        ...
    };
}
```

Figure 5. Cova description of an asynchronous document review process.

Upon receiving the document, AReviewing will become active. The supervisor of the author will be notified to open the document and write the comments. After the supervisor's submission, the comments will be appended to the CDocument object maintained by AAuthoring. Later, the author will see it and revise his/her document accordingly for another submission.

This example shows two more language features, i.e. process inheritance and system objects. In our example, PDocumentReview inherits all the class and activity definitions of PSingleUserAuthoring. Several predefined system objects, e.g. **system**, **process**, and **activity** (returned by GetActivity(...)), are also used. They are implemented to provide some information about the runtime environment, which will make process modeling more flexible.

4.4 An Integrated Process

Our last example is a scenario where both synchronous and asynchronous cooperation exist. In this scenario, a department head asks his/her employees to write a report for his/her review. They employees work on the report synchronously as described by PCoAuthoring. The report is sent asynchronously to the department head for review and comments are sent back for revision. This scenario is modeled by the Cova codes in Figure 6. The only difference between this definition and the one in Figure 5 is that the super process is changed from PSingleUserAuthoring to PcoAuthoring so that simultaneous access to the report is supported.

```
process PIntegratedDocReview extends PCoAuthoring
{
    //Everything is the same as PDocumentReview
    ...
}
```

Figure 6. Cova codes for modeling an integrated review process.

5 COMPARISONS

We have described with examples a uniform meta-model for modeling a wide range of cooperation scenarios and its runtime support. At a first sight, one may wonder how it is different from many seemingly similar ones, such as the concurrent object model, the multi-agent model, CORBA, and many coordination models and languages for parallel and distributed computing [7]. Due to page limitations, we can not give a detailed comparison here. Our general answer to the question is that our model is targeted at a different goal, i.e. modeling and supporting a wide range of scenarios of interactions among cooperators, which are usually human beings. Therefore, the semantics and mechanisms that implement them are completely different.

Many of the models mentioned above, however, aim at the coordinating multiple computing entities. Basically these models provide mechanisms to facilitate the communication and coordination among multiple concurrent computing entities. Obviously, these computing entities are quite different from human beings. For example, they are more 'patient', which means that they won't 'mind' waiting until the resources they need become available. Within this context, the semantics of coordination and the mechanisms that implement these semantics are totally different from the ones in our model. For example, the centralized tuple space of Linda provides a shared space for exchanging messages among multiple computing processes. A lock-based mechanism is used for synchronization, e.g. a process that retrieves a tuple that is not in the space will wait until the tuple becomes available.

It has been extensively discussed in CSCW literature that the centralized architecture and lock-based mechanisms are generally not suited for supporting cooperation [19]. Coordination and communication among computing entities aim mainly at increasing the speed and performance, not the flexibility of interactions among cooperators. Despite their diversity in formalisms, platforms, and many other things, these models are targeted at an area that is completely different from the one targeted by our model.

The second question that is often asked is how our model is different from other models and systems that are developed specifically for modeling and developing cooperative work, such as the process models for software engineering [1], DCWPL [15], COCA [5], GroupKit [16], COLA [12], various WfMSs, etc. As we have stated, we aim at a uniform meta-model. Obviously, process models for software engineering are domain-specific. These models do capture some important elements that are useful for supporting cooperative work. However, due to their domain specific nature, many important topics, such as advanced optimistic concurrency control, transaction management, are ignored or not well addressed. Systems such as DCWPL, COCA, etc. can be used to develop only a specific type of cooperative applications, thus lack the uniformity of our model. At the same time, since they only provide constructs for describing the coordination part of a process and have little knowledge about the computation part, finely granulated controls based on the semantics of the computation part are unavailable in these systems.

6 CONCLUSIONS AND FUTURE WORK

We have shown a meta-model capable of modeling uniformly cooperation processes in different modes. We have also developed a computerized mechanism, the Cova language, for describing cooperation processes based on this model. Our approach is different from others by its capability of uniform modeling, its clear separation and tight integration of the computation and coordination parts of a cooperation process, and its capability of introducing runtime information into process description.

There are, however, many topics for further research. For example, how good in practice the model and system will be when they are used to model and develop large-scale applications. Our concerns come from the fact that the codes of class methods are interpreted. This may result in low performance. Another interesting topic would be how our model could be used for process analyzing and optimizing. Based on our model, we are developing a mathematical tool, CoAuto, which is a specialized automaton, for describing cooperative processes mathematically. Our basic idea to do the analyzing and optimizing according to a transition graph generated from the mathematical description of a process. Details of this research will be reported in other publications.

ACKNOWELEDGES

The work on which this paper is based was done at Tsinghua University during the author's PhD study, which was supervised by Prof. Shi, Meilin and supported by several grants from China National Science Foundation and China High-Tech Research and Development Plan. The author is greatly grateful to the anonymous reviewers for their valuable comments.

REFERENCE

1. B Curtis, M I Kellner, and J Over. Process modeling. *Comm of the ACM*, 1992, 35(9):75-90
2. C A Ellis, J Wainer. A Conceptual Model of Groupware. In: *Proc of ACM Conf on CSCW*, Chapel Hill, 1994, 79-88
3. C A Ellis, S J Gibbs, G L Rein. Groupware: some issues and experiences. *Comm of ACM*, 1991, 34(1):39-58
4. C Z Sun, C A Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In: *Proc of ACM Conf on CSCW*, Seattle, 1998, 59-68
5. D Li, R Muntz. COCA: Collaborative Objects Coordination Architecture. In: *Proc of ACM Conf on CSCW*, Seattle, 1998, 179-188
6. D R McCarthy, U Dayal. The architecture of an active database management system. In: *Proc of ACM SIGMOD Conf on Mgmt of Data*, Portland, 1989, 215-224
7. G A Papadopoulos, F Arbab. Coordination models and languages. CWI Report, 1998. Available at: http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R9834.ps.Z
8. G Yang, M Shi. *Cova*: An object-oriented programming language for cooperative applications. *Science in China, Series F*, 2001, 44(1):73-80
9. G Yang. Modeling cooperative work: towards a uniform meta-model. *Bell-Labs Technical Memo*, 2001.
10. G Yang. On semantics-based concurrency control in fully-replicated architecture. *Bell-Labs Technical Memo*, 2001.
11. J Jiang, G Yang, W Yan *et al.* CovaTM: a transaction model for cooperative applications. In: *Proc of ACM SAC*, Madrid, 2002,
12. J Trevor, T Rodden, G Blair. COLA: A lightweight platform for CSCW. In: *Proc of European Conf on CSCW*, Milan, 1993, 15-30
13. K Salimifard, M Wright. Petri net-based modelling of workflow systems: An overview. European Journal of Operational Research, 2001, 134(3):664-676
14. N Carriero, D Gelernter. Coordination Languages and their Signicance . *Comm of the ACM*, 35 (2), 1992 : 97 – 107.
15. M Cortes, P Mishra. DCWPL: A programming language for describing collaborative work. In: *Proc of ACM Conf on CSCW*, Cambridge, 1996, 21-29
16. M Roseman, S Greenberg. Groupkit: A groupware toolkit for building real-time conferencing applications. In: *Proc of ACM Conf on CSCW*, Toronto, 1992, 43-50
17. M Weber, G Partsch, S Hock, et al. Integrating synchronous multimedia collaboration into workflow management. In: *Proc of ACM SIGGROUP Conf on Supporting Group Work*, Phoenix, 1997, 281-290
18. R G G Cattell, D Barry, D Bartels, et al. The Object Database Standard: ODMG 2.0. San Mateo: Morgan Kaufmann Publishers, 1997
19. S Greenberg, D Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In: *Proc of ACM Conf on CSCW*, Chapel Hill, 1994, 207-217
20. S G Natalie, S P Daniele, and P Remo. Generalized process structure grammars (GPSG) for flexible representations of work. In: *Proc of ACM Conf on CSCW*, Boston, 1996, 180-189
21. W Du, J Davis, M Shan. Flexible specification of workflow compensation scopes. In: *Proc of ACM SIGGROUP Conf on Supporting Group Work*, Phoenix, 1997, 309-316

Guangxin Yang got his bachelor, master, and doctor's degrees in Computer Science all from Tsinghua University, Beijing, P.R.C. He is currently a Member of Technical Staff with Bell-labs Research located in Murray Hill, New Jersey. His research interests include CSCW, Programming Language and System, Database, and Workflow, Next Generation Internet, Networking Protocols, etc. He is a member of ACM.