A CONTRACT-EXTENDED PUSH-PULL-CLONE MODEL FOR MULTI-SYNCHRONOUS COLLABORATION

Hien Thi Thu Truong, Claudia-Lavinia Ignat, Pascal Molli

▶ **To cite this version:**

Hien Thi Thu Truong, Claudia-Lavinia Ignat, Pascal Molli. A CONTRACT-EXTENDED PUSH-PULL-CLONE MODEL FOR MULTI-SYNCHRONOUS COLLABORATION. International Journal of Cooperative Information Systems, 2012, 21 (3), pp.221-262. 10.1142/S0218843012410031 . hal-00761038

HAL Id: hal-00761038

https://inria.hal.science/hal-00761038v1

Submitted on 4 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

International Journal of Cooperative Information Systems
© World Scientific Publishing Company

A CONTRACT-EXTENDED PUSH-PULL-CLONE MODEL FOR MULTI-SYNCHRONOUS COLLABORATION

HIEN THI THU TRUONG
Inria Nancy-Grand Est, Université de Lorraine,
Villers-lès-Nancy, 54600, France,
hien.truong@inria.fr

CLAUDIA-LAVINIA IGNAT
Inria Nancy-Grand Est,
Villers-lès-Nancy, 54600, France,
claudia.ignat@inria.fr

PASCAL MOLLI
Nantes University,
Nantes Cedex 3, 44322, France,
pascal.molli@acm.org

Received (22 February 2012)
Revised (6 August 2012)

In multi-synchronous collaboration users replicate shared data, modify it and redistribute modified versions of this data without the need of a central authority. However, in this model no usage restriction mechanism was proposed to control what users can do with the data after it has been released to them. In this paper, we extend the multi-synchronous collaboration model with contracts that express usage restrictions and that are checked *a posteriori* by users when they receive the modified data. We propose a merging algorithm that deals not only with changes on data but also with contracts. A log auditing protocol is used to detect users who do not respect contracts and to adjust user trust levels. Our contract-based model was implemented and evaluated by using PeerSim simulator.

Keywords: multi-synchronous collaboration; contract model; usage control; push-pull-clone model; trust; log auditing.

1. Introduction

Collaboration between a large number of users has emerged for years in the research domain of CSCW (Computer Supported Cooperative Work). Collaboration can be *synchronous*[13], *asynchronous* or *multi-synchronous*.[12] Synchronous (or real-time) collaboration mode allows communication in an instantaneous manner with bounded time and changes performed by one user are transmitted immediately to other group members. Asynchronous or non-real time mode conversely makes no assumption about the time intervals involved between interactions among users.

Multi-synchronous collaboration was introduced firstly by Dourish.[12] Multi-synchronous environment allows private working in cycles of divergence and convergence. Users work simultaneously in isolation in their workspaces and user changes are not visible to others until they decide to synchronize. Shared data diverges when users work in isolation and converges later after synchronization of shared data. Multi-synchronous collaboration model was used not only in research work such as SAMS[28], DSMW[37] but also in practical applications such as Distributed Version Control Systems (DVCS) (e.g. Git[23], Mercurial[32]) and Microsoft SharePoint Workspace. [9]

In multi-synchronous collaboration model, it is very difficult to control what users will do with the data after it has been released to them and to ensure that they will not misbehave and violate usage policy. From the view of deontic logic, usage policy can be expressed in terms of obligation, permission and prohibition. We model these concepts as contracts. The main issue addressed by this paper is how contracts can be expressed and checked within multi-synchronous collaboration model and what actions can be taken in response to users who misbehaved.

We consider, as an example, implicit contracts in DVCS which is an instance of multi-synchronous working environment. At the beginning, DVCS systems were mainly used by developers in open-source code projects but nowadays they started to be widely adopted by companies for source code development. In open source projects, usage restriction is expressed in the license of the code, while in closed source code projects, it is expressed in the contracts developers sign when accepting their job. In both cases, usage restrictions are checked *a posteriori* outside the collaborative environment with social control or plagiarism detection. As a result of observations concerning usage violation, trustworthiness on the users who misbehaved is implicitly decreased and collaboration with those users risks to be ceased. We aim at building a contract-based model that can express explicitly usage restrictions which are checked within a collaborative environment.

Access control mechanisms do not address the issue of usage restriction after data was released to users. Traditional access control mechanisms prevent users from accessing to data and granted rights are checked before access is allowed. It has been shown that these access control mechanisms are too strict.[11] There exist some optimistic approaches[46] that can control access *a posteriori*. In these approaches, if user actions violate granted rights, a recovery mechanism is applied and all carried-out operations are removed. Usually, this recovery mechanism requires a centralized authority which ensures that the recovery is taken by the whole system. However, a recovery mechanism is difficult to be applied in decentralized systems such as DVCS where a user has no knowledge of the global network of collaboration. Roughly speaking, access control mechanisms aim at ensuring that systems are used correctly by authorized users with authorized actions. Rather than ensuring such a strong security model, we target a flexible approach based on contracts that can be checked after users gained access to data and based on trust management mechanisms that

help users collaborate with other users they trust.

Push-Pull-Clone (PPC) is one of paradigms supporting multi-synchronous collaboration. In PPC model, users replicate shared data, modify it and redistribute modified versions of this data by using the primitives push, pull and clone. Users clone shared data and maintain in their local workspaces this data as well as changes done on it. Users can then push their changes to many channels at any time they want, and other users who have granted rights may pull these changes from these channels. By using pull primitives replicas are synchronized.

The main issue in designing a contract-based multi-synchronous collaboration is that contracts are objects that are part of the replication mechanism. In our contract-based model each user maintains a local workspace that contains shared data and contracts for the usage restriction of that data as well as changes on data. Changes done locally on the data together with specified contracts are shared with other users. Algorithms for merging and for conflict resolution have to deal not only with data changes but also with contracts. For checking if users respect contracts, a log auditing mechanism is used. According to auditing results, users adjust their trust levels assigned to their collaborators. To our best knowledge, there is no existing collaborative editing model based on contracts which allows auditing and updating trust levels during collaboration process. Major contributions of this paper are as follows:

- A PPC model extended with contracts for multi-synchronous collaboration that we call the C-PPC model. The proposed model ensures consistency of the shared document.
- A log auditing mechanism to detect user misbehavior in contract-based collaboration model.
- A set of experiments to evaluate the performance of the C-PPC model and the log auditing mechanism by using a peer-to-peer simulator.

This paper is an extension of our previous published work.[49] We present a real world motivating example for our approach and more details about deontic concepts that are used in the formalisation of the contract model. In addition, we classify different types of contract conflicts and present the solution to deal with these conflicts. Furthermore, we complete the C-PPC model with a log auditing protocol and provide some additional discussion around this model.

The paper is structured as follows. We start by presenting an overview of our proposed approach in Section 2. We then describe the C-PPC model in Section 3, including representation of logs of operations and formalisation of contracts expressed inside the model. In section 4, we present aspects of collaborative process over C-PPC model: logging changes, push and pull protocols, consistency model, log auditing and trust assessment mechanisms. We report some experimental results of simulation to evaluate the efficiency of our model in Section 5. In Section 6, we review related approaches and point out their differences from our work. We end the paper with some concluding remarks and directions for future work in Section

7.

2. Approach Overview

We consider a collaboration model that requires high levels of respect and trust among users. To create a trustful and respectful collaborative environment, we introduce collaboration contracts that will be used in collaborative interactions. In this section, we present an overview of our C-PPC model which extends the PPC model with contracts.

Let us give a simple example to illustrate how C-PPC model works in a real world example of collaboratively building a photo collection. Nowadays it is very common that people share photos. Photos help people stay in touch with their family and friends all around the world. They take photos in their daily life experience coming from many contexts such as from a wedding to a vacation or from a local meeting to an international conference. After an event where many people took many photos, people who joined the event want to share those photos with each other. By that way they can remember all little moments they might have missed during the event. Together users can build a great photo-based story for the event in which they participated.

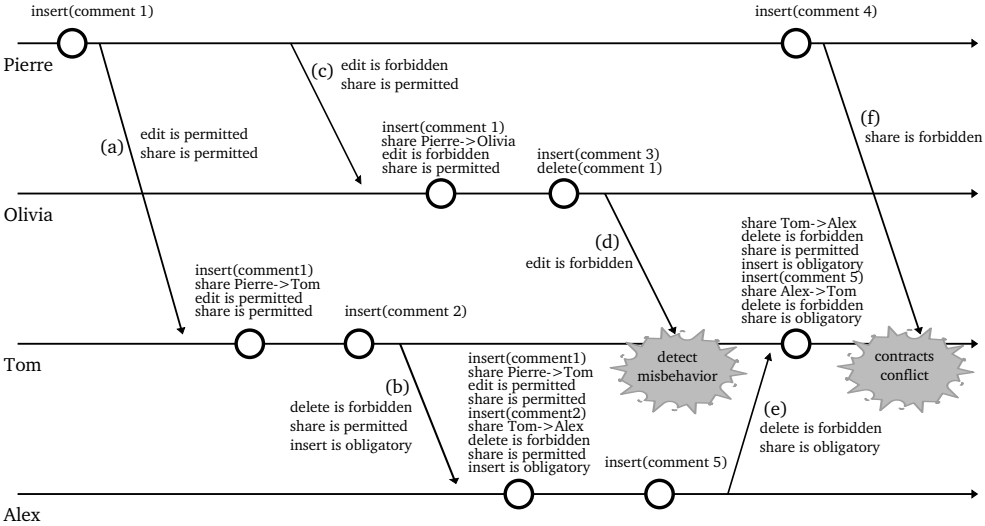


Fig. 1. A Push-Pull-Clone collaboration scenario of four users over a photo X between *Time 1* and *Time 2*. User actions are recorded chronologically.

We assume a friend-to-friend network of four users Alex, Tom, Olivia, and Pierre. We suppose that users can edit photos by means of adding and deleting comments. At the beginning, the network is built based on social trust between users and connections are established only between users who trust each other. Users trust

their collaborators with different trust levels that are updated according to their collaboration experience. For instance, Pierre trusts both Tom and Olivia, however, with different trust levels, and thus he gives them different contracts over the shared photo. For example, Pierre gives Tom the permission to edit and to share, while he gives Olivia only the permission to share the photo. Receivers are expected to follow these contracts; otherwise, their trust levels will be adjusted once misbehavior is detected. We log changes that users do on the photo collection with contracts that they receive from others when they receive their changes. Assume that in the time interval [*Time 1*, *Time 2*], Alex, Tom, Olivia, and Pierre perform their local changes on the collection.

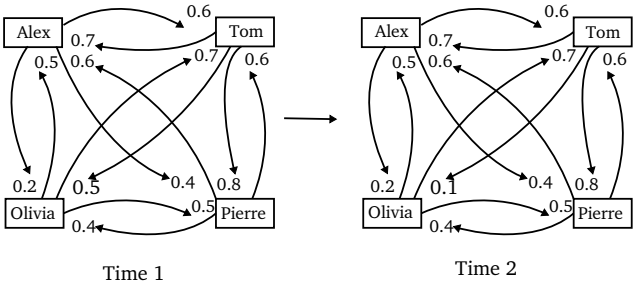


Fig. 2. Trust values at two different times *Time 1* and *Time 2*.

Let us assume that user trust values at a time instance *Time 1* are shown in Fig. 2 in which values are real numbers ranged in the interval [0, 1]. However, trust values change over time based on user's assessment. The values we take in this example show the implicit trust each user has on others. A user has no global knowledge of the trust values that each user assigns to others, but knows directly only the trust values he assigned to other users.

Our model uses push, pull and clone as native direct pair-wise communication primitives between users. To work with others, a user simply sets up a local workspace, and uses trusted channels to *push* her changes to trusted friends. Other users can then get the photo by cloning (executing a *clone* primitive) it into their workspaces. In this way, they have independent local workspaces for the shared data. They do their changes on their local replicas and publish changes by executing a *push* primitive. The user then executes a *pull* primitive to get the changes into his local workspace. Push, pull, and clone primitives are used for efficient distributed collaboration and they were already implemented in distributed version control systems such as Git and Mercurial. We assume the system uses a FIFO channel between two users for changes propagation to guarantee that messages are received in the order they are sent. This order can be preserved by using logical timestamps to sort messages into chronological order.

In Fig. 1 and Fig. 2, after *Time 1*, Pierre trusts Tom with a trust value 0.6.

Pierre makes a change on photo X by adding a new comment *comment 1*. He shares his change on photo X with Tom with the contract {*edit is permitted*, *share is permitted*} (step a). The change and the contract are pushed to a communication channel with Tom. As it is the first time that Tom initiates a communication with Pierre, he has to clone photo X with changes from Pierre. Tom now has in his local workspace a clone of the photo collection from Pierre, on which he can work in isolation. In the example, Tom adds another comment to the photo *comment 2*. Since he has the right to distribute further the data, he then shares it with his trusted friend Alex (step b). Tom wants Alex to redistribute the collection of photos only after adding her own comment on photo X. He therefore shares his data with the contract {*delete is forbidden*, *share is permitted*, *insert is obligatory*}. Concurrently, Pierre collaborates also with Olivia. He trusts her less than Tom and he wants to forbid her from editing the photo while allowing her to share it. Pierre specifies the contract {*edit is forbidden, share is permitted*}. Olivia thus has no right to edit photo X after she receives it from Pierre (step c).

Olivia violates the contract she received from Pierre by deleting Pierre's comment and adding her comment, *comment 3*, to photo X. She continues to collaborate with Tom by specifying the contract that he is forbidden to delete any comment on the photo (step d). As soon as Tom receives the changes from Olivia, he discovers that she misbehaved. He thus updates the trust value on Olivia. The Fig. 2 shows that her trust level at *Time 2* is decreased to 0.1.

In parallel, Pierre adds a new comment, *comment 4*, to photo X. He wants the comment is kept private except for Tom. He thus shares the photo with this new comment with his friend Tom, but with the restriction of not sharing it further, {*share is forbidden*} (step f). Tom receives the data with the new contract from Pierre. This contract conflicts with the contracts he holds after synchronizing with Alex (step e) which is {*share is obligatory*}. As Tom wants to be able to further modify and share the photo, he decides to resolve the conflict, for example, by discarding the changes from Pierre, or by negotiating to relax the prohibition of sharing.

In the example, Alex behaves well by always respecting contracts she has received. She adds *comment 5* to photo X and shares it with Tom by restricting him not to delete the added comments with the contract {*delete is forbidden*, *share is obligatory*} (step e). After these user interactions in the time interval [*Time 1*, *Time 2*], user trust values are illustrated at *Time 2* in Fig. 2.

We have given an illustration of how the C-PPC model can be used for the collaboration between four users Alex, Tom, Olivia, and Pierre on the assumption that users trust each other at different trust levels. We move next to the formal representation of our C-PPC model.

3. C-PPC Model

In this section, we describe main parts of our target model: users, logs of operations on shared document and contracts between users.

3.1. *Users*

Users are main participants in the C-PPC model. They are connected in a collaborative network based on the trust they have in other users. However, users are not uniformly trusted by others. Each user keeps at his local site the replica of shared documents and works locally on these replicas. For the sake of simplicity we consider that all users (also called sites throughout this paper) are collaborating on a single shared document.

3.2. *Document, Changes and Logs*

The system keeps a document as a log of operations that have been done during the collaborative process. The log maintains information about user's contributions to different parts of the document and when these contributions were performed. The outcome of collaboration is a document that could be obtained by replaying the *write* operations such as *insert*, *delete*, *update* from the log. Two users can write independently on the shared document. Changes are propagated in weakly consistent manner that a user can decide when, with whom and what data is sent and synchronized. Push, pull and clone communication primitives are operated on FIFO channels for allowing an ordered exchange of operations done on document replicas. A replica log contains all operations that have been generated locally or received from other users. Logs are created and updated at user sites. The log structure is defined in the following definitions.

Definition 3.1. (Event) Let $\mathbb{P}$ be a set of operations $\{$*insert, delete, edit, share*$\}$ that users can generate; and let $\mathbb{T}$ be a set of event types $\{$*write, communication, contract*$\}$. An event e is defined as a triplet of $\langle evt \in \mathbb{T},\ op \in \mathbb{P},\ attr \rangle$, in which *attr* includes attributes which are in form of $\{$*attr_name*, *attr_value*$\}$ to present additional information for each event.

Definition 3.2. (Log) A document log L is defined as an append-only ordered list of events in the form $[e_1, e_2, \ldots, e_n]$.

Users store operations in their logs in an order that is consistent with the generated order. The event corresponding to a *share* operation of type *communication* is issued when a user pushes his changes and it is logged at the site of receiver when this one performs a pull. This *share* event can be followed in the log by an event of type *contract* representing usage policies for the shared data.

In Fig. 3 we give an example of a log containing a single event that has three attributes. The log is presented in XML format. The *write* event refers to insert operation and belongs to *write* type. The event has attributes $\{by, Pierre\}$ (done

```
<log> <!-- at local site of Pierre -->
  <event>
   <evt>write</evt>
   <op>insert</op>
   <attr>
      <by>Pierre</by>
      <content>comment 1</content>
      <gsn>1</gsn> <!--generation timestamp -->
   </attr>
  </event>
</log>
```

Fig. 3. An example of log with one event in XML format.

by Pierre), $\{content, comment1\}$ and $\{GSN, 1\}$ for the sequence number when the event is generated.

The event attribute *GSN* (*generate sequence number*) of an event is assigned at the site where event was generated. The event attribute *RSN* (*receive sequence number*) of either a *communication* or a *contract* event is assigned at reception of this event by receiving site. We will discuss how to use these sequence numbers later in Section 4.

3.3. *Contract*

A contract expresses usage policies which one user expects others to respect when they receive and use shared data. Contracts are built on the top of basic deontic logic[55] with the normative concepts of obligation, permission and prohibition representing what one ought to, may, or must not do.

Contract in C-PPC model is different from traditional usage policy that is presented accompanied with application systems. For example, W3C platform for privacy preference P3P[52], which uses preference exchange language, APPEL[51], is an industry standard that provides a method for users to gain control over the use of personal information collected by web sites they visited. Our approach does not require additional platform to express contracts. Instead, contract is a part of replication system and it is built over operations within application domain.

3.3.1. *Symbolism of deontic concepts*

Let us start first with our review of deontic logic. Deontic logic is one of four main groups of modalities that the philosopher Von Wright mentioned in his papers[55] (the alethic models or models of truth, the epistemic models or models of knowing, the deontic models or models of obligation and the existential models or models of existence). Deontic logic is used as the logic of rights and duties. In the deontic

models, the first preliminary concept is the *act* that is pronounced obligatory, permitted and forbidden. The word *act* is used for properties and not for individuals, for example, steal or smoke are acts. The negation of a given act is performed by a subject, if and only if, it does not perform the given act. For example, the negation of the act of answering a question is the act of not answering it.

The philosopher von Wright considers the concept of permission is true on formal grounds, then defines the concept of the obligatory and the forbidden. The deontic concepts that are applied to a single act follows.

> "If an act is not permitted, it is called *forbidden*. For instance, theft is not permitted, hence it is forbidden. We are not allowed to steal, hence we must not steal."
>
> "If the negation of an act is forbidden, the act itself is called *obligatory*. For instance: it is forbidden to disobey the law, hence it is obligatory to obey the law. We ought to do that which we are not allowed not to do."
>
> "If an act and its negation are both permitted, the act is called *indifferent*. For instance: in a smoking compartment we may smoke, but we may also not smoke. Hence smoking is here a morally indifferent form of behavior. [..] Indifference is a narrower category than permission. Everything indifferent is permitted but everything permitted is not indifferent. For, what is obligatory is also permitted , but not indifferent."
>
> (Deontic logic[55], pages 3-4).

The norms derived from deontic concepts of the permitted, the obligatory and the forbidden are permission, obligation and prohibition, respectively. We build the contract on the top of these norms and handle them in distributed manner. For the permission, we distinguish a permission that can be either a strong permission or a weak permission.[56] The permission, which is an exception of an obligation and a prohibition, is a strong permission. The permission which follows from the absence of a prohibition, is a weak permission. If an act is strongly permitted then its negation is permitted, whereas if an act is weakly permitted then its negation is forbidden. With a strong permission, a subject always can choose to perform the act or not, while it is not the case for weak permission.

We illustrate the concepts of deontic logic model in Fig. 4. We will summarize how these deontic norms are symbolized as a formal logic following von Wright model.

We use A to denote a name of an act and $\sim A$ is used as a name of its negation. The proposition that the act named by A is permitted is expressed in symbols by P_A. The proposition that the act named by A is forbidden, which is the negation of the proposition that it is permitted, is symbolized by $\sim P_A$ or F_A. The proposition that the act named by A is obligatory, which is the negation of the proposition that the negation of the act is permitted, is symbolized by $\sim P_{\sim A}$. We use simpler

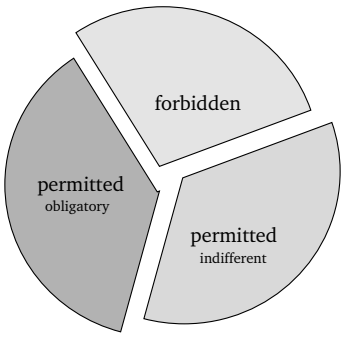


Fig. 4. Three deontic concepts that cover all possibilities assigned to an act. The act is only either forbidden, obligatory or permitted indifferently.

symbol for the obligatory, O_A. The proposition that the act named by A, which is called indifferent, is symbolized by $(P_A)\&(P_{\sim A})$. In this symbolism, P, O, F are called the deontic operators. Sentences of the type "P *name of act(s)*" are called P-sentences. Similarly, we might have O-sentences and F-sentences. Also we have "permitted", "obligatory", "forbidden" as deontic values.

The above deontic operators apply to a single act with what we call an atomic name. Since we can define the conjunction, disjunction, implication of two given acts to be what we call a molecular name, we can apply deontic operators to pairs of acts as well. If A and B denote acts, then $A\&B$ is used as a name for their conjunction, $A \vee B$ as a name for their disjunction, $A \rightarrow B$ as a name of their implication.

Considering the distribution property of deontic operators, wrong conclusions might be taken with respect to the application of these operators. We first consider negation operation. If the act A is permitted, we can conclude nothing to the permitted, forbidden or obligatory as character of $\sim A$. Sometimes $\sim A$ is permitted, sometimes not. From the Fig. 4 we can see that A might be obligatory or indifferent. If A is obligatory as well as permitted, then $\sim A$ would be forbidden. If A is what we call indifferent, then $\sim A$ is also permitted. For example, in smoking compartment, smoking and not-smoking is permitted. But in the non-smoking compartment, not-smoking is permitted but smoking is forbidden.

We next consider distribution property in the conjunction of two acts. If both A and B are permitted, it does not mean $A\&B$ is permitted because doing either of them may commit us not to do the other. For example, it is permitted to promise to give a thing and it is also permitted not to give a thing, but it is forbidden to promise to give a thing and then not give it.

We finally consider distribution property in the disjunction of two acts. If at least one of the acts A and B is permitted, it follows that their disjunction $A \vee B$ is permitted. When both acts are forbidden, their disjunction is forbidden.

The von Wright deontic model includes also laws of deontic logic. A true proposi-

tion, that a certain molecular P-/O-sentence expresses a deontic tautology, is called a law of deontic logic. We summarize below these laws from notions of permission and obligation. It should be noticed that the combining force order follows that "$\sim$" is stronger than "&", "&" is stronger than "$\vee$", and "$\vee$" is stronger than "$\rightarrow$".

- Two laws on the relation of permission and obligation:
  (1) P_A is identical with $\sim O_{\sim A}$.
  (2) O_A entails P_A.
- Four laws for the dissolution of deontic operators:
  (1) $O_{A\&B}$ is identical with $O_A \& O_B$.
  (2) $P_{A\&B}$ is identical with $P_A \vee P_B$.
  (3) $O_A \vee O_B$ entails $O_{A\vee B}$.
  (4) $P_{A\&B}$ entails $P_A \& P_B$.
- Seven laws on commitment (doing an act commits to do another act if the implication of one to another is obligatory):
  (1) $O_A \& O_{A\rightarrow B}$ entails O_B. This law is intuitively obvious. If doing an act that is obligatory commits us to do another act, then this act is obligatory also.
  (2) $P_A \& O_{A\rightarrow B}$ entails P_B. If doing what we are free to do commits us to do another act, then this act is permitted to do also.
  (3) $\sim P_B \& O_{A\rightarrow B}$ entails $\sim P_A$. This is a vice versa law of the law above. If doing an act commits us to a forbidden, then this act is forbidden also.
  (4) $O_{A\rightarrow B\vee C} \& \sim P_B \& \sim P_C$ entails $\sim P_A$. This law is an extension of above law. Doing an act that commits us to a choice of forbidden alternatives, then this act is forbidden also.
  (5) $\sim(O_{A\vee B} \& \sim P_A \& \sim P_B)$. This law means it is impossible to oblige to choose between forbidden alternatives.
  (6) $O_A \& O_{(A\&B)\rightarrow C}$ entails $O_{B\rightarrow C}$. This law means that if doing two acts, one of which being obligatory, commits us to do a third act, then doing the second act commits us to do the third act.
  (7) $O_{\sim A\rightarrow A}$ entails O_A. If it is failure to perform an act commits us to perform it, then this act is obligatory.

Combining two acts into a molecular act might lead to the incompatible. Two acts are incompatible if their conjunction is forbidden. For example, reading and smoking both are not permitted in library, so they are incompatible.

3.3.2. *From deontic modalities to contracts*

Based on deontic concepts, we formalize contract in our target model as follows.

Definition 3.3. (contract primitive). For a set of n possible operations $\mathbb{P} = \{op_1, op_2, \ldots, op_n\}$, a contract primitive is denoted by a deontic operator followed

by a write or a communication operation. A contract primitive is an event in log that takes deontic operators P (the permitted), O (the obligatory), F (the forbidden) as modality attributes (so-called modal). If *op* is an operation in $\mathbb{P}$ then the contract primitive c_{op} based on *op* is denoted as: F_{op} (doing *op* is forbidden), O_{op} (doing *op* is obligatory), and P_{op} (doing *op* is permitted). When we use the generic notation c it means that the contract c can refer to any operation.

Definition 3.4. (contract). A contract C is a collection or a set of contract primitive(s) which are built on the operations of $\mathbb{P}$. It is denoted as $C_{\mathbb{P}} = \{c_{op_1}, c_{op_2}, \ldots, c_{op_n}\}$. Alternatively, we can use the notation $C = \{c_{op_1}, c_{op_2}, \ldots, c_{op_n}\}$ for a contract.

For example, in Fig. 1, Pierre inserts a comment into photo X and gives it to Tom with a contract $C_{\{edit,\ share\}} = \{P_{edit},\ P_{share}\}$ (edit and share are permitted) with two single contract primitives P_{edit} and P_{share}. When a user shares data by means of a push primitive, at the site of the receiver, a share event is logged with attributes representing users who sent and received the changes. Moreover, contract events are logged describing the contracts received. In Fig. 5, we illustrate the representation of the log at site of Tom after he cloned *photo X* from Pierre.

If we have n contract primitives, we can obtain a contract by merging these contract primitives. For instance, if we have two contract primitives $c_1 = P_{op_1}$ (op_1 is permitted) and $c_2 = O_{op_2}$ (op_2 is obligatory), then we can build the contract $C_{\{op_1,op_2\}} = \{P_{op_1},\ O_{op_2}\}$. Concerning merging contract primitives to obtain a contract, we consider two following axioms:

(A1) $C = \{c_1\}$ & $(c_1 \rightarrow c_2) \longrightarrow C = \{c_1, c_2\}$ (*deducibility*)

(A2) $C = \{c^1_{op}, ..., c^n_{op}\}$ & $(c^1_{op} \succ c^2_{op} \succ \ldots \succ c^n_{op}) \longrightarrow C = \{c^1_{op}\}$ (*priority*)

("$\succ$" denotes a higher priority relationship between two contract primitives or two operations).

The axiom (A1) shows the consequent deducibility. We assume a set of inference rules can be defined among the contract primitives in the system. Following this axiom, if $c_1 \in C$ and $c_1 \rightarrow c_2$ then $c_2 \in C$. At a certain time if the user u has a contract C that contains c_1 and respecting c_1 commits to respecting c_2 then even if c_2 is not explicitly given to user u, c_2 is added to C. This is helpful to reduce the number of contract primitives given at a certain sharing time since users do not have to specify contracts that can be inferred from other contract primitives based on deducible rules of system settings. For instance, if we suppose that $O_{edit} \rightarrow P_{insert}$, then $C = \{O_{edit}\}$ & $(O_{edit} \rightarrow P_{insert}) \longrightarrow C = \{O_{edit}, P_{insert}\}$. This means if a user receives an obligation to edit, she will have the permission to insert automatically since the setting of inference rule $O_{edit} \rightarrow P_{insert}$ holds. Another example, if system allows $P_{edit} \rightarrow P_{insert}$ then $C = \{P_{edit}\}$ & $(P_{edit} \rightarrow P_{insert}) \longrightarrow C = \{P_{edit}, P_{insert}\}$.

The axiom (A2) rules the merging process of different contract primitives referring to a same operation. If we have n contract primitives referring to an operation *op* with priority order $c^1_{op} \succ \ldots \succ c^n_{op}$ then these contract primitives can be merged

```
<log> <!-- at local site of Tom -->
  <event>
   <evt>write</evt><op>insert</op>
   <attr>
      <by>Pierre</by>
      <content>comment 1</content>
      <gsn>1</gsn>
   </attr>
  </event>
  <event>
   <evt>share</evt><op>share</op>
   <attr>
      <by>Pierre</by>
      <to>Tom</to>
      <gsn>2</gsn>
      <rsn>1</rsn> <!-- receipt timestamp -->
   </attr>
  </event>
  <event>
   <evt>contract</evt><op>edit</op>
   <attr>
      <by>Pierre</by><to>Tom</to>
      <modal>P</modal>
      <gsn>3</gsn>
      <rsn>2</rsn>
   </attr>
  </event>
  <event>
   <evt>contract</evt><op>share</op>
   <attr>
      <by>Pierre</by><to>Tom</to>
      <modal>P</modal>
      <gsn>4</gsn>
      <rsn>3</rsn>
   </attr>
  </event>
</log>
```

Fig. 5. An example of log containing contract events

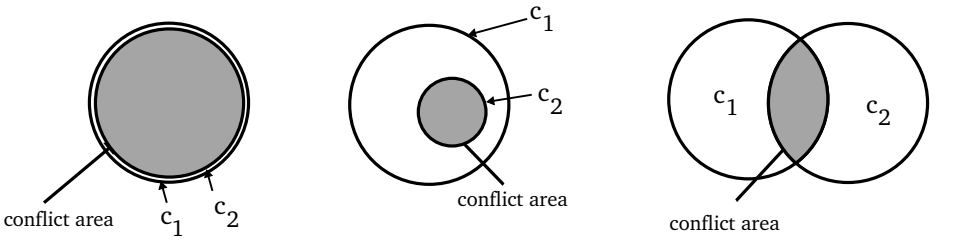
and the resulting contract is deducible as $C = \{c^1_{op}\}$. For instance, if $C = \{F_{op}, P_{op}\}$ and $F_{op} \succ P_{op}$ then it is deducible to have a contract $C = \{F_{op}\}$. This means that even though *op* is permissive with the contract primitive P_{op}, it is forbidden to perform *op* if $C = \{F_{op}\}$ and $C = \{P_{op}\}$ are merged with the condition $F_{op} \succ P_{op}$.

3.3.3. *Contract conflict*

When multiple users work on the same shared data and share their changes to one another under different contracts, it is not possible to ensure that the system will be conflict-free regarding these contracts. Therefore it is necessary to identify conflicts, to detect conflicts and to propose conflict resolution strategies.

The term *deontic conflict* and *deontic inconsistency* have been used interchangeably in the literature. In the book *On Law and Justice*, Ross[41] identifies three ways in which inconsistency in law arises: "total-total", "total-partial" and "partial-partial".

(1) Total-total inconsistency: this means neither of a pair of norms is applicable without conflicting with the other. If the conditional facts of each norm are symbolized by a circle, a total-total inconsistency occurs when the two circles coincide (Fig. 6a). In total-total inconsistency two norms are absolutely incompatible. This is thus said strong inconsistency since no norm can be performed without causing norm violations. For example, the total-total inconsistency arises when an action is simultaneously obligatory and forbidden.



(a) Total-Total inconsistency (b) Total-Partial inconsistency (c) Partial-Partial inconsistency

Fig. 6. Three ways of inconsistencies.

(2) Total-partial inconsistency: this means one of the two norms is not applicable in any case without coming into conflict with the other, whereas the other norm does not conflict in all cases with the first one. Such inconsistency occurs where one circle lies inside the other. (Fig. 6b). As an example, the total-partial inconsistency arises when an action is simultaneously permitted and forbidden.

(3) Partial-partial inconsistency: this means each of the two norms has cases that conflict with the other but also cases in which no conflict arises. This inconsistency exists when two circles intersect. (Fig. 6c). We can see this inconsistency in an

example which a person is obliged to attend a concert but the entering to the theatre without ticket is forbidden. The partial-partial inconsistency arises between two norms which are the obligation to attend and the prohibition to enter without ticket. If the person has a ticket, then she can fulfill one of two norms or both without causing violation to the other. In this case, she can enter the theatre, and by attending the concert, she fulfills the obligation requiring her to attend it. By this, no conflict arises. However, if she does not have a ticket, then she cannot act following one norm without violating the other. Without having a ticket, when she respects the prohibition to enter by staying outside, she violates the obligation to attend the concert. In contrast, when she fulfills the requirement to attend the concert, she will violate the prohibition not allowing her to enter without ticket. Through the example we see two norms that are incompatible in once case and compatible in another case.

Ross[41] also figures out that in judging inconsistencies an important part is the relationship between statutes where conflict occurs. Inconsistency is drawn (a) within the same statute or (b) between older and more recent statutes.

Concerning inconsistencies, in the normative discourse, the unrealizability is mentioned with two conditions: (i) the norm belonging to a set of norms must be individually realizable. This condition means each single norm should not be impossible to conform; (ii) however, the norms in that set of norms are not jointly realizable. This means what is prescribed by a set of norms cannot be performed simultaneously.

Definition 3.5. (contract consistency) A contract which is a collection (or a set) of contract primitives (norms of obligations, permissions, and prohibitions), is consistent, if and only if, its contract primitives are simultaneously jointly realizable.

Inconsistencies arise due to the incompatibility of the deontic operators. The deontic square of opposition (Fig. 7), which is based on Aristotle's philosophy (as stated by Moretti[30]) about logic square and first used in deontic logic by Bentham[5], depicts the relationship between norms. It shows four types of inconsistencies between four deontic modalities.

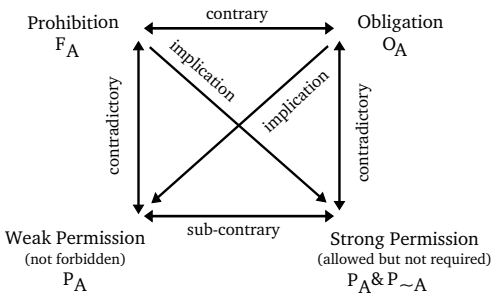


Fig. 7. Deontic square of opposition.

(1) *Contraries*: the pair obligation and prohibition forms this opposition. An action cannot be obligatory and forbidden simultaneously. This is a total-total inconsistency since both norms may be false.

(2) *Contradictories*: strong permission and prohibition, obligation and weak permission form this opposition. One norm in each pair of norms is true. An action is either permitted or forbidden as well as an action is either obligatory or omissible.

(3) *Implication*: obligation implies weak permission, prohibition implies strong permission. If an action is obligatory, then it cannot be forbidden, thus its permission is possible. Also, if an action is forbidden, then it cannot be obligatory, and thus its omission is possible.

(4) *Sub-contrary*: from the contrary of obligation and prohibition, strong permission ($P_A \& P_{\sim A}$) and weak permission P_A are contrary to each other. An action may be performed if it is not forbidden, as well as it may be omitted if it is not obligatory.

According to this square, there are three possibilities for an action that is either forbidden, obligatory or indifferent (permitted but not obliged) (this is consistent with what is shown in Fig. 4). From this square we can observe that the contrary relationship between prohibition and obligation raises the real conflict (total-total inconsistency). The situation when an action is simultaneously obliged and forbidden influences behaviors in conflicting fashion in the sense that it is impossible to do the action that is compliant with one norm without conflicting the other. For the pair permission and prohibition, we adopt the view that their contradictory is an inconsistency but not a real total-total conflict. This comes from the fact that a permission may not be acted on, so no real conflict occurs between permission and prohibition. Therefore from our view, real conflict rather than normal inconsistency arises only between obligation and prohibition. In our model, we assume users should be able to perform actions and therefore contracts must not be inconsistent or they must not contain any inconsistency or conflict between their contract primitives.

Even though each contract is conflict free, conflicts may arise when two contracts are merged during synchronization phase. The fact that two users assert two contract primitives that are inconsistent is quite frequent. It is even possible for one and the same user to assert two inconsistent contract primitives. If we want that users collaborate in contract-compliant manner, we must resolve conflicts. The important thing is to identify inconsistencies and to examine the techniques used to remove them.

Definition 3.6. (contract conflict) Two contracts C_1 and C_2 conflict if at least one contract primitive $c_i \in C_1$ conflicts with one another $c_j \in C_2$. Contract primitives are conflicting between O_{op} and F_{op}. Besides, two contracts are inconsistent to each other if at least one contract primitive $c_i \in C_1$ is inconsistent with one another $c_j \in C_2$.

3.3.4. *Conflict resolution*

We present in this section our solution to deal with inconsistencies of contracts. There is no fixed principles for conflict resolution. In order to ensure the consistency of contracts in the system, conflicts are resolved based on several criterias. Contracts, among which a real conflict arises, cannot co-exist in the contractual system, hence they must be avoided. One way to do this is by means of negotiation between users. In this case, the system should inform contracting partners about their contractual situation and what are the conflicting contracts on which operations. Then contracting partners decide how obligations and prohibitions can be "relaxed" in order to allow additional options for further actions. For example, Figure 8 depicts the case that conflict need to be negotiated to precede further work of contracting partners. In the figure we can see even the conjunction of two obligations might create conflict. Say, *a night club is obliged to close emergency exit to prevent crimes quit for drugs* (O_{op_1} from police department), and *a night club is obliged to open emergency exit* (O_{op_2} from fire safety). In this case $O_{op_1 \& op_2}$ is inconsistent and two obligations are not realizable at the same time. In current work, we consider only the inconsistency between deontic operators (P, F and O) and not yet between semantic content of actions under those operators.

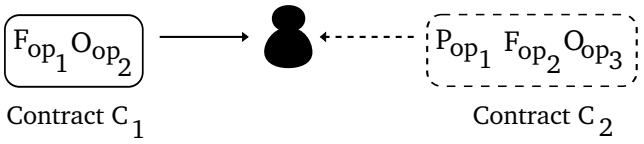


Fig. 8. A scenario when a user adopts two contracts that are inconsistent with each other. The user holds Contract 1 and a coming one Contract 2. The conflict between O_{op_2} and F_{op_2} needs to be resolved to proceed further actions.

In case of partial inconsistencies only, inconsistent contracts can co-exist. For example, in Fig. 8, if we eliminate the conflict between O_{op_2} and F_{op_2}, there still remains the partial inconsistency between F_{op_1} and P_{op_1}. In this case if the user is not performing op_1 then F_{op_1} and P_{op_1} are both respected and therefore no real conflict occurs. Similarly, if F_{op_1} and P_{op_1} are both given by the same user and P_{op_1} overrides F_{op_1} (we discuss overriding rules later) then no real conflict occurs. Even though this type of inconsistency can exist in the system, it is better to reduce inconsistency possibilities. Users should be informed about their contractual situation when a synchronization is performed. This helps them choosing between different contracts the ones that give them better benefit. In what follows we describe a method for ordering contracts based on the order of operations associated with them.

3.3.5. *Ordering contracts*

Contract primitives are naturally interrelated and interdependent and there is no hierarchy between them. Thus, some priorities can be established in terms of particular objectives and they vary depending on particular applications. One way to select a contract in the case of conflict is to assign orders to various contract primitives, and sort them in ascending or descending order and then compare contracts composed of them. The hierarchical ordering of contracts enables users to give preferences to some contracts over others. In that case, the set of contract primitives is not a normal set but a partially ordered set and the ordering relations are intrinsic to the contracts in the system.

Depending on the operation types, the order of contract primitives is given as follows. Operations are categorized to different groups according to their types. Given two operations op_1 and op_2 with the priority order $op_1 \succ op_2$, then the order of contract primitives with the same deontic operator is assigned according to the order of operations, $c_{op_1} \succ c_{op_2}$. If the contract primitives associated with operations belong to different groups, then we determine a combined order for each, based on the order within group and the order of the group.

We formalize the method to order single operations as well as sets of operations. Our method is inspired from the work of Cholvya and Hunterb.[8] We present below the ordering of operations within single category and across multiple categories, and next is our solution to compare contracts.

- Ordering categories of operations: Each category includes a set of operations referring to a specific kind of action. Let us consider two categories $\Lambda_1 = [\alpha_1, ..., \alpha_m]$ and $\Lambda_2 = [\beta_1, ..., \beta_n]$. The ordering of Λ_1 and Λ_2 is given based on the priority of them in a particular system. In addition, the order of Λ_1 and Λ_2 implies the order of every operation of Λ_1 and Λ_2. For example, $\Lambda_1 > \Lambda_2 \Rightarrow \alpha_i > \beta_j \ \forall \alpha_i \in \Lambda_1, \beta_j \in \Lambda_2$.
- Ordering operations within a single category: Operations in a single category of actions can be put in a hierarchical order specific to a particular system. For two operations α_1 and α_2, their orders should be either $\alpha_1 > \alpha_2$ or $\alpha_1 < \alpha_2$. For example, (*add-comment* > *read*) in category *edit*.

To compare two contracts, let $\mathbb{P}$ be a set of n operations that could be ordered as [op_1, op_2, ..., op_n] from the highest to the lowest priority conforming to the orders of operations within each category (if any) and between categories as presented above. Let $\mathbb{S}$ be a set of n-digit ternary numbers from 0 to $3^n - 1$ and a contract C composed of m contract primitives built over operations of $\mathbb{P}$, $C = \{c_1, c_2, ..., c_m\}$, $c_i = P_{op_j}|O_{op_j}|F_{op_j}$, $c_i \in C$, $op_j \in \mathbb{P}$, $1 \leq i \leq m$, $1 \leq j \leq n$ as a list of contract C where contract primitives are ordered following operation order.

To order contract, we set norms in some kind of hierarchy, some is regarded as more basic than others. Without losing generality, let us assume that deontic operators are ordered as $P \succ O \succ F$; also operations in $\mathbb{P}$ are put in order $op_n \succ$

$op_{n-1} \succ \ldots \succ op_1$. A mapping from C to $\mathbb{S}$ results in $s \in \mathbb{S}$. For each $op_j \in \mathbb{P}$, $1 \leq j \leq n$, we set:

- If $\exists c_i = P_{op_j}|O_{op_j}|F_{op_j} \in C$ then:
  (1) if $c_i = P_{op_j}$ then $s[j]_3 = 2$, where $1 \leq i \leq m$;
  (2) if $c_i = O_{op_j}$ then $s[j]_3 = 1$, where $1 \leq i \leq m$;
  (3) if $c_i = F_{op_j}$ then $s[j]_3 = 0$, where $1 \leq i \leq m$;
- If $\nexists c_i \in C$, $1 \leq i \leq m$, then $s[j]_3 = 2$. This case presents the absence of any contract specified on operation op_j. We take the positive view that the absence of obligations and prohibitions implies the permission. Therefore, $s[j]$ is set value as 2, as same as in case that op_j is permitted.

The comparison of two contracts C_1 and C_2 is based on the comparison of their corresponding digital numbers $[s_1]_3$ (mapped from C_1) and $[s_2]_3$ (mapped from C_2). We have $(C_1 > C_2) \Leftrightarrow (s_1 > s_2)$ and vice versa.

For instance, given a set $\mathbb{P}$ of two operations (n=2) in the order $op_2 \succ op_1$, and we want to compare two contracts $C_1 = \{O_{op_1}, P_{op_2}\}$ and $C_2 = \{O_{op_2}\}$. The 2-digit ternary numbers $s_1 = [12]_3$ and $s_2 = [21]_3$ are mapped from C_1, C_2 to $\mathbb{S}$. Since $s_2 > s_1$, so that $\{O_{op_2}\} > \{O_{op_1}, P_{op_2}\}$, hence, $C_2 > C_1$.

This ordering mechanism helps users to make decision in case of inconsistencies to choose the contracts with more benefits. It is important to make users aware of what is added to the system might introduce inconsistency. Furthermore, in a peer-to-peer network with no central authority that maintains the consistency of contracts, once conflicts are detected, they should be resolved or adapted by users.

3.3.6. *Repealing contracts*

In addition to adding contracts to data when it is shared with collaborators, our approach supports removal of given contracts. We consider the overriding rule to repeal contracts issued in the past.

Overriding rule allows that an old contract is replaced by a new one. In this case the new contract overrides the old one. The contract primitive c_2 overrides c_1 if both c_1 and c_2 are given by the same sender to the same receiver and c_2 was sent later than c_1. We can express this by c_2 overrides $c_1 \Longleftrightarrow (c_1.op = c_2.op)$ and $(c_1.attr.by = c_2.attr.by)$ and $(c_1.attr.to = c_2.attr.to)$ and (c_2 was received after c_1).

Let us present an example of contract overriding when Tom realizes that the operation op under the contract primitive $c_{op} = F_{op}$ he gave to Alex some time ago should not be forbidden any longer because conditions that made the prohibition of performing op have changed. He wants to permit Alex to do op. Since previous changes performed together with given contracts were logged and shared with many users, the only solution for removing the prohibition is by compensation. Tom can override the prohibition by giving a new contract to Alex. Once the new contract is accepted by Alex, the prohibition is removed for her.

With this compensation solution, the addition of new contract might introduce

new inconsistencies or might lead to wrong conclusion as mentioned in previous section of deontic symbolism. We notice that inconsistencies would arise at receiver side when a sender tries to repeal an old contract primitive by asserting its negation. Therefore overriding a contract is not just simply adding its negation. This could make contracts in system inconsistent. An ideal system, thus must help users to be aware of any conflict when they repeal a contract, for example, by providing awareness mechanism about conflict.

There is an alternative to remove old contract without introducing a new contract. Rather than negating a contract users might reject its validation.[3] This helps to avoid inconsistency (notice that rejecting is not the same as negating, while with negation we assert another contract primitive to the system for a negation while with rejection we just simple add an event to confirm the revocation of an old contract primitive). However, we do not adopt this solution in our current work as the compensation solution seemed more appropriate to our logging mechanism.

4. Collaborative Process

This section describes the basic protocols of collaborative process over C-PPC model: logging changes, pushing logs containing document changes and contracts, and merging pairwise logs.

4.1. *Logging Changes*

Each site maintains a local *clock* to count events (*write*, *communication*, and *contract*) generated locally or received from remote sites. When changes are made or received, they are added to log in the following manner:

- When a site generates a new *write* event e, it adds e to the end of its local log in the order of occurrence and augments its clock. The *clock* value is assigned to attribute *GSN* (*generate sequence number*) of event e (i.e. $e.attr.GSN = clock$).
- When a site receives and accepts (from now and afterward we simply say a site receives a remote log since we do not proceed further in case users reject the remote log) a log from another site, events from the remote log that are new to its local log are appended at the end of the local log in the same order as in the remote log.
- When a user shares a document with another user, she sends a *communication* event followed by some contracts, which are logged by receiving user. We denote by e one of these events (communication or contract). At time of reception, receiver assigns his local clock to attribute *RSN* (*receive sequence number*) of e, (i.e. $e.attr.RSN = clock$).
- We assume that a user is unwilling to disclose to other collaborating users all *communication* and *contract* events that she has given to a certain user. Thus *communication* and *contract* events are not kept in the log of the

sender. Moreover, even if a site sends those events to other sites, receiving sites could refuse integration of remote changes. In this way, sending sites would contain events that are not accepted by receivers. Therefore, *communication* and *contract* events are not logged by sending site.

- An event e is said *committed* by site u when it is added (logged) to local log of u in one of the following cases:
  (1) e is a *write* event generated and saved (kept in log) by u.
  (2) e is a *contract* or *communication* event given to u by another site v. Recall that sending site does not keep *contract* or *communication* events in its local log.

An important feature of C-PPC model is that changes of one site are not propagated to all other sites since user trust levels are different and sites might receive different contracts for the same document state. We discuss the consistency of proposed model based on the CCI consistency model[47] which requires preserving *causality*, ensuring document *convergence* and preserving user *intention*.

Concerning causality preservation, our model deals with two causal relationships (denoted as $\xrightarrow{c}$): *causal* relation (based on *happened-before* defined by Lamport[21]) and *semantic causal* relation.

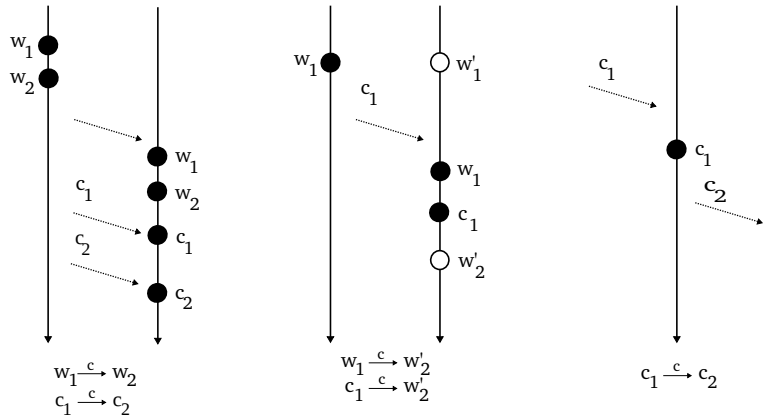


Fig. 9. Causal relations between events (w_i represents for *write* events and c_i represents for *contract* or *communication* events).

- *Causal relation:* two events e_1 and e_2 are in a causal relation, denoted as $e_1 \xrightarrow{c} e_2$, if:
  (1) for two events of the same type (i.e. two *write* events, two *contract* events or

two *communication* events) e_1 and e_2 generated by the same site, if e_1 was committed before e_2 then $e_1 \xrightarrow{c} e_2$. For example, for two write events we have $(e_1.attr.by = e_2.attr.by)$ and $(e_1.attr.GSN < e_2.attr.GSN)$ and $(e_1.evt = e_2.evt = write) \Longrightarrow e_1 \xrightarrow{c} e_2$ (see example Fig.2.(a)). For two contract events we have $(e_1.attr.to = e_2.attr.to)$ and $(e_1.attr.by = e_2.attr.by)$ and $(e_1.attr.RSN < e_2.attr.RSN)$ and $(e_1.evt = e_2.evt = contract) \Longrightarrow e_1 \xrightarrow{c} e_2$ [see e.g. Fig.9(a)].

(2) for two events generated by different sites, e_1 generated by site u and e_2 generated by site v, $e_1 \xrightarrow{c} e_2$ if e_2 is committed after e_1 has been received (or committed) at site v [see e.g. Fig.9(b)]

- *Semantic causal relation*: Two contract events e_1 and e_2 are said to be in a semantic causal relation if e_1 is received by a site before that site sends e_2 to another site. The contract event one site gives to other sites should depend on her current contracts: $(e_1.evt = e_2.evt = contract)$ and $(e_1.attr.to = e_2.attr.by)$ and $(e_1.attr.RSN < e_2.attr.GSN) \Longrightarrow (e_1 \xrightarrow{c} e_2)$ [see e.g. Fig.9(c)].

The above causal relations between events are used in the auditing mechanism for detection of users that did not respect the given contracts.

In C-PPC model, logs are propagated by using anti-entropy[10] which ensures the *happened-before* relation between events as defined by Lamport[21] without using state vectors[25] or causal barriers.[34] We say that event e_1 *happened-before* e_2, denoted as $e_1 \xrightarrow{hb} e_2$, if e_2 was generated on some site after e_1 was either generated or received by that site. The *happened-before* relation is transitive, irreflexive and antisymmetric. Two events e_1 and e_2 are said concurrent if neither $e_1 \xrightarrow{hb} e_2$ nor $e_2 \xrightarrow{hb} e_1$.

Two events that are in a causal or semantic causal relation are also in a happened-before relation.

We define a *partially ordered set* (poset) $H = (E, \xrightarrow{hb})$ where E is a ground set of events and $\xrightarrow{hb}$ is the *happened-before* relation between two events of E, in which $\xrightarrow{hb}$ is irreflexive and transitive. We call H as an event-based history in our context. Given a partial order $\xrightarrow{hb}$ over a poset H, we can extend it to a total order "$<_t$" with which "$<_t$" is a linear order and for every x and y in H, if $x \xrightarrow{hb} y$ then $x <_t y$. A linear extension L of H is a relation $(E, <_t)$ such that: (1) for all e_1, e_2 in E, either $e_1 <_t e_2$ or $e_2 <_t e_1$; and (2) if $e_1 \xrightarrow{hb} e_2$ then $e_1 <_t e_2$. This total order preserves the order of operations from a partial order set H to the linear extensions on the same ground set E.

We call these linear extensions as individual logs observed by different sites. The Fig. 10 shows an example of a history and its congruent linear extensions.

In collaborative systems, where multiple sites collaborate on the same shared data object, we can consider that the global stream of activity of all sites is defined by a partially ordered set of events. Each site, however, maintains a single log as its local observation and synchronization. It can see only events in local workspace

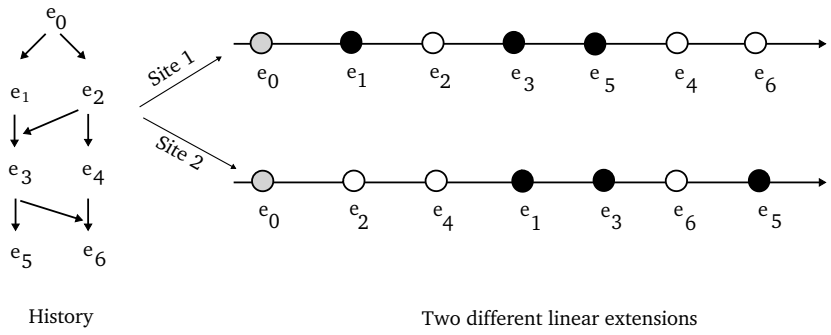


Fig. 10. An example of history and logs

that it generated locally or received from other sites. The site keeps therefore an individual log as a linearization of history built on a subset of a ground set of operations. There are remaining events of global history built on entire ground set of events that are not visible for the site.

4.2. *Pushing Logs containing Contracts*

A key advantage of weakly consistent replication by relaxing data consistency is that the protocol for data propagation can accommodate contracts to let users decide with whom to reconcile. Anti-entropy[10,33] is an important mechanism to achieve eventual consistency among a set of replicas. Basic anti-entropy allows two replicas to become updated by sending updates generated at one replica to other replica. Anti-entropy guarantees causal order of events which specifies that if an event is known to a site then any event preceding that event is already known to the site.

In addition to propagation of changes, since in C-PPC model sites may have different levels of trust in other sites and the trust relationship may change during the collaboration, contracts are given to restrict usage on the shared document when a user shares a document to another user. The user pushes her log as follows:

- Since a document is shared as a log of events, therefore to send a contract for document usage control, the contract is attached at the end of the log.
- In sharing, a user specifies a new contract; however, she cannot specify a higher contract than what she currently holds. For instance, if a user u currently holds a contract C on the document d, she only can share d with another user with a contract C' where $C' \leq C$ (contracts are compared as presented in section 3.3.5).
- A user cannot specify a new contract which conflicts with her current contract. For instance, if user u has contract $C = \{O_{op}\}$, then she cannot add F_{op} to C.
- The contracts a user specifies to two distinguished users might be different. These two users do not not know the contract of the other user as far as they do not collaborate with each other.

During the collaborative process the log of each site grows and the document and contracts are updated each time a user synchronizes with other users.

4.3. *Pulling and Merging Pairwise Logs*

The collaboration involves logs reconciliation. Consider that a user u receives a remote log L' from a remote user v through anti-entropy propagation consisting of events from site v that site u did not see since their last synchronization. u has to elect new events from L' to append to her log L.

Function isMerged(u, v, L, L', CT, CT')

```
1  if Trust(u, v) is low then
       // v is distrustful
2      result ← Reject;
3  else
4      if ct' ∈ CT' conflicts ct ∈ CT then
5          if conflict is resolvable then
6              result ← Merge;
7          else result ← Reject;
8      else
9          result ← Merge;
10 return result;
```

A site might receive a remote log with conflicting contracts. In case of unresolvable conflicts, the user decides either to reject the remote document version or to leave the local version to accept new one. The function *isMerged* checks for conflict before merging. It checks if remote log *L'* sent by user v can be merged with local log L of user u. The function takes as arguments the log L of the local site u and the remote log L' of the remote site v containing new events since their last synchronization. Given these logs, the current contract holded by u and the contract that v gives to u when L' is sent can be computed. We denote these contracts by CT and CT' respectively (CT is the contract holded by u and CT' is the contract given to u by v). A site neither merges nor creates a new branch if the sender is distrustful. We consider a dominance of contract if a user revokes an old contract and replaces it by a new one. For instance, the old contract F_{share} received by site v from site u can be replaced by a new one P_{share}. Two logs can be merged if no conflict is found.

If the result returned by *isMerged* function is *Merge* (merging can be performed), we perform synchronization by using our proposed merging algorithm. We assume the merging algorithm ensures causality not only between *write* events but also between *communication* events and *contract* events. We next discuss in detail how to ensure causality.

To determine the total order of events committed by one site, we use the *"commit*

sequence number" CSN. In merging function, commit sequence number *CSN* is used to track the last event committed by one site.

As we mentioned before, the attributes of event e, $e.attr.GSN$ and $e.attr.RSN$ record the values of the clock of its generation and its receipt, respectively. Note that every event has *GSN* attribute assigned before log is propagated, but *RSN* attribute is assigned to *communication* and *contract* events at the receiving site during the synchronization. The value of *CSN* of an event e committed by site u is computed as follows:

- If e is a *write* event generated by u, the commit sequence number *CSN* is assigned the value of attribute $e.attr.GSN$. The site who committed e is extracted from e's attribute $e.attr.by$.
- If e is a *communication* event or a *contract* event given by a site a to a site v and committed by site v, the commit sequence number *CSN* is assigned the value of attribute $e.attr.RSN$. The site who commits e is extracted from attribute $e.attr.to$.

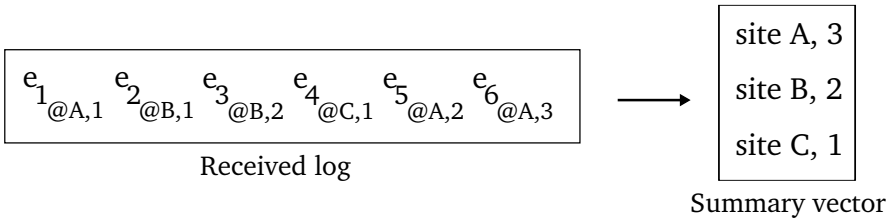


Fig. 11. An example of Summary Vector

The fact that the merging function ensures that new events are added only to the end of log enables the property that if the log of a site u contains an event e committed by v with a commit sequence number *CSN*, then it contains all the events committed by v prior to e. In order to avoid merging events that have been already integrated, we use a summary vector SV which has the maximum size equal to the number of users. The summary vector of site u (SV_u) keeps the highest commit sequence number *CSN* of each site $v \neq u$ known by u in its components $SV_u[v]$ (see the example in Fig.11). A summary vector is a set of time-stamp of commit sequence numbers, each from a different user indexed by site identifiers. This allows a site u to correctly determine that an event from site v should be merged into local log if its *CSN* is higher than the current entry value of SV corresponding to its belonging site. The summary vector used here is different from the version vector used in weakly consistent replication to maintain causal relationship between events where the size is the number of sites and the version vector needs to be exchanged together with the corresponding operation. Instead, summary vector is maintained locally at sites and its size is the number of other sites whose events are known to

the site.

Function merge(L, L', clock)

```
1  for i = 1 to sizeof(L') do
2      e ← L'[i] ;
3      if e.evt = write then
4          CSN ← e.attr.GSN;
5          site ← e.attr.by;
6      else
7          if e.attr.RSN = null then
8              e.attr.RSN ← clock;
9              clock ← clock + 1;
10         site ← e.attr.to;
11         CSN ← e.attr.RSN;
12     if CSN > SV[site] then
13         append e to the end of L;
14         SV[site] ← CSN;
15 return L;
```

It is possible to replay *write* events from log to get document state. We can use any existing CRDT approach[35,53,39] in which concurrent operations can be replayed in any causal order as they are designed to commute in order to ensure document consistency. The complexity of function *merge* is $O(n)$ where n is the size of the remote log L'.

4.4. *Log Auditing and Trust Assessment*

Compliance checking whether user actions in collaborative system comply with contracts is an important part of our C-PPC model. This question is done through logging and auditing mechanisms that are principle in many systems supporting observation. Log auditing is an approach that adopts *a posteriori* enforcement. It complements *a priori* access control, in order to provide a more flexible way of controlling compliance of users after the fact. In this subsection we present log auditing procedure and trust assessment based on auditing results.

4.4.1. *Auditing principles*

Before presenting our auditing procedure for C-PPC model, we clarify some principles concerning our auditing mechanism.

(1) Users can perform auditing of the log in order to make misbehaving users accountable for their actions without the need of any central authority. In this way the dependence on an online entity that provides auditing logs is overcome. However, the disadvantage of the mechanism is that users have no knowledge

about global actions done by all other users in order to completely assess if a particular user behaved well or not. Our auditing mechanism is therefore based on incompleteness evidence. Though this assumption could be claimed as a drawback, it is suitable to human society where a person is assessed only based on some of her noticed behaviors.

(2) Logs that reflect actions done by users and that are input to the auditing mechanism must be maintained correctly. Even though avoiding log tampering is impractical in distributed environments, tampering detection is possible. We have proposed using authenticators for detecting log tampering.[50] Log tampering is detected at time of synchronization before the log is accepted by receivers.

(3) How to use log auditing result and treat data resulted from misused actions? When a user discovers other users that misbehaved, she updates their trust levels. Users use trust models to manage their friend reputation. The trust levels obtained from auditing result are used as input data for a trust model. We focus only on using auditing result to update trust values. We exclude any further aspects of trust models such as how to propagate personal view of trust among users or how to use external resources to assess trust values or how to aggregate trust values.

These above principles distinguish our auditing mechanism from other prior approaches. In following subsections we identify situations when contracts are violated and then provide the log auditing mechanism.

4.4.2. *Contract violation*

In this subsection we specify the three types of attacks that might lead to contract violation.

- Malicious users tamper logs to eliminate or modify contracts or other events in the log. We consider that a user u is *malicious* if she re-orders, inserts or deletes events in the log that consequently affects auditing result. For instance, u removes some obligations that she does not want to fulfill. The log auditing mechanism assumes logs are authenticated. Any tampering should be detected by the log authentication mechanism.
- Malicious users perform actions that are forbidden by the specified contracts. These action events are labelled as *bad*.
- Users neglect obligations that need to be fulfilled. For instance, a user receives an obligation *"insert is obligatory"* but she never fulfills this obligation. If at a given moment a log auditing mechanism is performed and no event that fulfills the obligation is found, we cannot claim that the user misbehaved. He might fulfill the obligation at a later time. The given obligation is labelled as *unknown* meaning that the obligation has not yet been fulfilled. Once the obligation is fulfilled, the *unknown* label is removed.

Users are expected to respect given contracts. If a user respects all given contracts, then she will get a good trust value assessed by others. Ideally, if a user misbehaves in one of the three ways mentioned above, his misbahaviour should be detected by other users. The auditing mechanism returns a trust value that is computed from the number of events labelled with *good*, *unknown* and *bad*. Note that this manner of computing trust values does not distinguish an accidental attack from an intentional attack. In order to make users aware of unintentional misuses, the system prevents users in case a contract is violated by reminding them the obligations they hold.

4.4.3. *Log auditing*

An initial idea of our proposed log auditing mechanim was proposed in our previous work [48]. Our auditing procedure aims at finding contract violations and making users accountable for their actions by adjusting their trust levels following a trust metric. The general idea of the auditing procedure is to browse the log and check each event appearing in the log whether it conforms to given contracts. For each violation of a particular user found, we increase the number of *bad* events counted for the user. Similarly for each obligation that is not yet fulfilled, we increase the number of *unknown* events. This statistic of contract violations by a user over all events that are audited is used to compute the trust level of this user.

Procedures *updateAuditState* and *audit* present auditing protocol and trust computation in details when a user u audits actions of all other users, say v, who appears in the log. In these procedures, G_v and Q_v are used to keep a set of contracts and a set of obligations which user v holds, respectively ($Q_v \in G_v$). At the initial step, $G_v = \emptyset$ and $Q_v = \emptyset$. For each event e in the log L, the procedure *updateAuditState* checks its event type, contract or write event. If e is a contract given to user v then it is added to G_v. Moreover, if e is an obligation, it is counted as *unknown* event until an event that fulfills it will be found. If e is a write or a communication event performed by user v, it is checked if it complies with or violates contracts in G_v. In the procedure for updating auditing state, for each user v, *numberOfBadEvents[v]* and *numberOfUnknownEvents[v]* are used to count the number of *bad* and *unknown* events that are audited, respectively (remaining events are considered *good*). *auditedEvents*[v] is used to count the total number of audited events. All users v audited by u are inserted in set V. At the initial step of *audit* procedure, these variables: *numberOfBadEvents[v]*, *numberOfUnknownEvents[v]* and *auditedEvents[v]* are set equal to 0, and the set V is set empty.

Procedure *audit* takes as input the local log L of user u and the position in the log *lastCheckedPos* identifying the last event checked in the previous auditing mechanism. L is browsed to check whether each behavior of other users is correct. When log analysis is finished, trust values of all audited users v in V are recomputed based on auditing results. By doing this, their accountability is made through updating their trustworthiness.

Procedure updateAuditState(e)

```
1  if (e.evt =' contract') then
2      v ← e.to;
3      G_v ← G_v ∪ {e};
4      if e overrides c in G_v then
5          G_v ← G_v \ {c};
6      if (e.attr.modal =' O') then
7          Q_v ← Q_v ∪ {e};
8          numberOfUnknownEvents[v] + +;
9  else
10     v ← e.by;
11     if e violates G_v then
12         numberOfBadEvents[v] + +;
13     if e fulfills c in Q_v then
14         Q_v ← Q_v \ {c};
15         numberOfUnknownEvents[v] − −;
16 V ← V ∪ {v};
17 numberOfAuditedEvents[v] + +;
```

A user can perform log auditing at any time at local site and trust values are updated personally. Log analysis has polynomial order of n time complexity $O(n)$ with n is the number of events that are audited. In case auditing creates significant overhead, users might skip auditing some parts of log which were done by highly trusted users. However, in case these users behave badly, they are discovered only in a next auditing phase.

Procedure audit($L, lastCheckedPos$)

```
1  for i = lastCheckedPos + 1 to length(L) do
2      e ← i^th event in L;
3      updateAuditState(e);
4  foreach v in V do
5      re-compute trust for user v;
```

In order to manage trust levels, we need a decentralized trust model. The trust level of a user assessed by one another could be aggregated from log-based trust, reputation trust and recommendation trust. Trust computation varies from trust models. In order to provide a complete trust model, in our future work we will propose a trust metric based on the log auditing result.

5. Evaluation

In this section, we present the evaluation of our proposed model by performing some experiments using a peer-to-peer simulator and give some discussions around the

proposed model.

5.1. *Correctness of C-PPC Model*

The C-PPC model uses operation-based optimistic replication. The core data structure used in the model is a partially order log. Events (write, communication and contract) are communicated using anti-entropy protocol which ensures causality. The document is achieved correctly if and only if the log was not tampered. This is an assumption of our model. Our solution about the construction and verification of authenticators to secure log are presented in our previous work.[50] Authenticators prevent re-ordering of log events and therefore causality is preserved. If log was tampered, receiving site might discard it and the trust level of the site that misbehaved would be decremented.

Concerning the document convergence, as C-PPC model uses CRDT for commutative operations, it ensures that in the presence of different contracts received by different sites when the same set of *write* operations was executed at those sites, their copies of the shared document are identical. However, the shared document might be in different states on two sites since the shared document is not uniformly distributed due to the use of contracts and the trust levels of users. And finally, concerning the property of intention preservation of C-PPC model, it is ensured by causality preservation and CRDT algorithm.

The C-PPC model supports multi-synchronous collaboration which allows simultaneous work in isolation workspace even when network is disconnected and user changes are propagated and synchronized with reconnection. The extension of using contract for PPC model made the condition that the logs are synchronized more complex due to the arising of contract conflict. However, users can use log auditing mechanism to detect any conflict of contracts and logs are synchronized together if and only if all conflicts are resolvable. Conflicts can be resolved by the rejection of the owner or by the overriding of user with high role in system (the order of users can be voted between users in system).

5.2. *Experiments*

Due to the unavailability of real data traces of collaboration including contract, we evaluate the feasibility of C-PPC model through simulation using PeerSim simulator.[29] We focus first on the ability of detecting misbehaving users; then we estimate the overhead generated by using contract.

5.2.1. *Setup*

We setup the simulation with a network of 200 users in which a number of users are set as honest users, and the remain are set as misbehaving users. The portion of honest/misbehaving users in different experiments varies depending on the purpose of evaluation.

For simulating process, we generate randomly the data flow of collaboration during the simulation. The data flow includes operations, contracts and users with whom to share. The network topology with which users share log to their neighbors are built randomly by the simulator. One interaction is defined as a process of sharing a log with a specified contract, from one user to another one. Since the total number of interactions generated should be pseudo uniformly distributed over all users, we let one user perform sharing with not more than 3 other users at each step. Similarly, the number of operations and contracts generated by one user each time is at most 10 operations and 3 contracts (if we consider only 3 types of actions in our system: insertion, deletion and sharing).

Each node in network represents for one user. Between two interactions, nodes generate local operations randomly but must follow its current contract. Nodes keep their contractual state temporarily to generate correctly operations. However not every node respect its contracts. While honest nodes generate allowed operations, misbehaving nodes generate operations that violate their contracts. This data is used to evaluate our algorithms of detecting misbehavior.

Since contracts are generated randomly with only limited condition that they should not bigger than node's current contract (contracts are ordered as in previous section), conflict certainly arises in simulation between contracts of different nodes. As nodes in simulator cannot behave human acts, we omit negotiation protocol for contracts. Furthermore, to simplify we do not allow neither total-total inconsistencies nor partial-total inconsistencies for contracts hold by nodes. Contract conflict thus are detected before logs are synchronized. Once conflict are found, logs are rejected to be merged and the node which detected conflict waits for next cycle or for other nodes which send log without conflicting (see our Algorithm isMerged). With this restriction logs are always maintained under consistent contracts.

5.2.2. *Experiment 1 - Misbehavior detection*

To evaluate the ability of misbehavior detection, we check first the ability to detect a selected misbehaving user according to the total number of interactions performed by all users. The estimation is performed on a collaborative network of 200 users with 60 misbehaving users (30% of users are misbehaving users). The auditing process is performed after each synchronization with another user. We select randomly one misbehaving user to be audited and we analyze the percentage of users that can detect him. Fig. 12 shows the results recorded after each cycle. We can see that the misbehaving user is detected by a few users at the beginning and then the number of users that detect his misbehavior increases along with the increasing of number of interactions.

Second, we check the percentage of misbehaving users that can be detected. We select randomly one honest user from the network to observe the percentage of misbehaving users that she can detect. Fig. 13 shows the result according to the number of synchronizations done by the selected user with others. We can see from

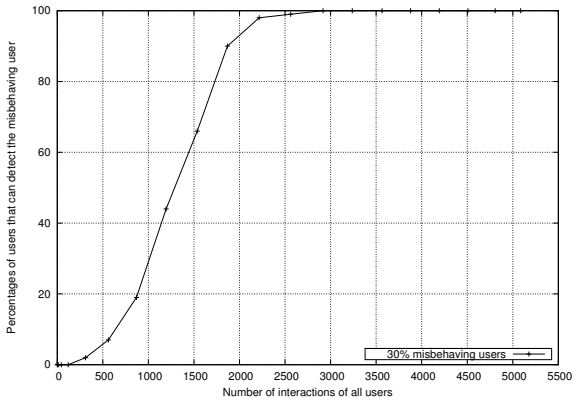


Fig. 12. Ability to detect one selected misbehaving user with respect to the total number of interactions in a collaborative network of 200 users with 30% of them are misbehaving users.

the graph that up to 20% of misbehaving users are detected after the first four synchronizations (auditing is done four times), and after the fifth synchronization more than 80% of misbehaving users are detected. We can see a drastic change in the figure between the fourth and the fifth synchronization. That change is due to a synchronization of the log of selected user with a remote log that contains misbehavior of most remaining misbehaving users. This can occur in distributed networks of random topology where clusters of collaborating users exist. Once an interaction occurs between two users belonging to such clusters, misbehaving users of the two clusters are discovered. Only about 10% of misbehaving users may require more interactions to be detected. From results in Fig. 13 we can see that the ability to detect misbehaving users depends also on the topology of collaborative network. In the future work we will perform more experiments to evaluate how topology would affect the detection.

In order to have a global view about the evolution of the percentage of detected misbehaving users, we compute the average value of detected misbehaving users over all users of the collaborative network. Fig. 14 shows, on average, the percentage of misbehaving users that are detected by one user. We perform the experiment in case of a low, medium and high population of misbehaving users in the network (respectively 5%, 30%, 80% of misbehaving users). The results show that the system still functions well in case of a high/low population of misbehaving users.

5.2.3. *Experiment 2 - Overhead estimation*

We conduct this experiment to evaluate the time overhead generated by using contract for the synchronization and auditing mechanism. We compare two models: with and without contract. To be able to make the comparison between these two models, we follow the same data flow. In the model without contract, the synchro-

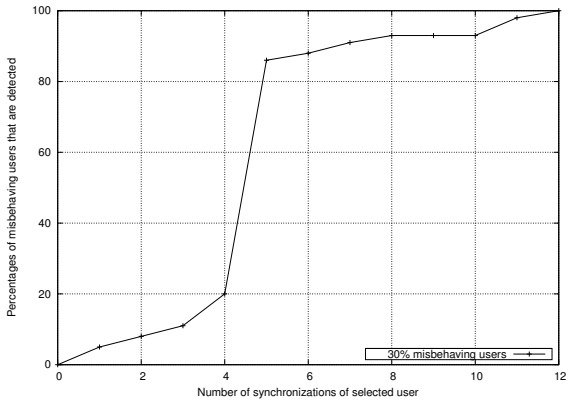


Fig. 13. Percentage of detected misbehaving users with respect to the number of synchronizations done by selected honest user.

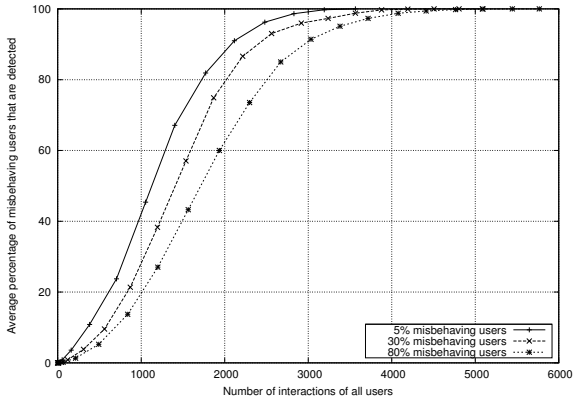


Fig. 14. Average percentage of detected misbehaving users with respect to to the total number of interactions in the collaborative network.

nization mechanism requires merging logs of write events only. In the model with contract the synchronization mechanism requires merging logs of write events and contract events. Additionally, an auditing mechanism for user misbehavior detection has to be applied.

We compute for each model the total time (T) of all the synchronizations performed by a given user to build the same state of document, $\mathrm{T} = \sum t_i$, where t_i is the time required for the i^{th} synchronization. Fig. 15 shows the result according to the number of write operations in the local log. From these results we can see that the time overhead generated by using contract is reasonable since the difference of time overhead computed for two models increases slowly with an increasing of number of events.

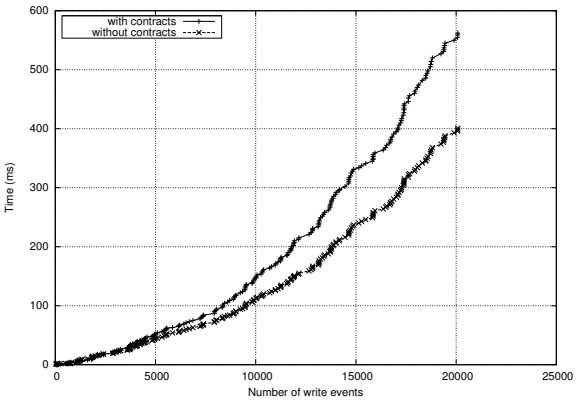


Fig. 15. Synchronization time with growing of number of *write* events

5.3. *Discussion*

In this section we discuss some potential limitations in our work. First, contract-based collaboration does not offer a solution for plagiarism and violation of contracts outside of the system. Beyond *write*, *communication* and *contract* events that a computer system could log, there are always side channels that can work around the logging. For example, a malicious user could replay *write* events from a log to create a new document and then share it and claim herself as being the owner. Or a malicious user could reveal the content of the document outside of the system by using communication means such as email, telephone call and chat, these actions being not logged by the system. These violations can be detected by humans or by using plagiarism techniques, however, this is out of the scope of this paper. The proposed model uses contract as a means to express data usage restrictions that helps to protect data privacy and to build a trustworthy collaborative environment.

A second limitation of our approach is how to deal with the growing size of the log during collaborative process. The log should be ultimately truncated so that it does not grow without bound. That requires some additional constraints and consensus of collaborators. Once the cycle of collaboration grows big after a long period, log of operations can be converted to state of document. After this conversion the log is emptied and all contracts recorded are not kept any more. Removing all contracts is reasonable since the behavior kept a long time ago might not be suitable to evaluate trust level at present. At the moment we do not consider log truncation in the proposed model.

Third limitation is that we have not fully explored a wide range of contracts that can be specified in our collaboration model. In C-PPC model, contracts are based on a basic deontic logic including permission, prohibition, and obligation. They can be combined with operators from temporal logic to express time dimension of contracts, however, we will consider this in our further work.

Next, we discuss the ability to apply the C-PPC model to multiple documents rather than a single one. Our approach is a general solution and thus it is applicable to multiple documents. We can keep a single log for operations over multiple documents. As an example, the single log is kept for operations of different files in a source code project using Mercurial distributed version control system. Each file has its unique identifier in the project, so we can keep operations with an additional attribute of file identifier to distinguish them. In each interaction of sharing, contract can include multiple contract primitives that refer to rights and duties of user on different files at the same time. For editing operations on documents, we use CRDT approaches, so concurrent operations can be replayed in any order without making document content diverge. When our approach is applied to multiple documents, the approach works without considering file system operations such as moving a file or renaming a file. We can allow these file operations in our approach if a solution of CRDT for file systems is proposed. However, this is out of scope of this paper.

6. Related Work

Our work is related to several topics in the area of privacy and data management in multi-synchronous working environment such as contract-based models, usage control models, access control models, trust management and log auditing for log-based collaboration. In this section we briefly survey the most relevant works and point out the differences of our work with respect to these approaches

The contractual approach is useful for a wide range of applications, such as resource management, cooperative task execution, cooperative work in distributed systems and software engineering. Traditionally a contract is an agreement between two or more persons about actions that are performed. Contracts also regulate behavior when persons cooperate or use shared resources. Contracts exist in many systems. It is either implicit in communication protocols, software licenses, downloading and sharing policies in P2P file-sharing systems or explicit in paper-based contracts of using network services. The push-pull-clone model for collaborative editing source code was adopted in distributed version control systems but users are uniformly trusted and there are no contracts specified during collaboration. Wikipedia features an informal contract-based model where contracts are checked by crowd sourcing. Anybody can edit according to rules that are checked *a posteriori* by other people. In contract-based models rules have to be explicitly expressed and checked by the system. Existing work has focused primarily on either contract models for individual aspects or collaboration models with implicit contracts. Some works proposed a contract framework (see, for example Contract Framework [44,43], Contract Net protocol [45], Agreement Framework [40]) for negotiating and controlling resource usage in a distributed system and engaging to solve the connection problem between nodes with tasks to be executed simultaneously. Contracts express the terms under which nodes in network promise (obligation) to offer and to get payment with regard of exchanging resources. The contract model which deals with contract

specification including of the server, the client, the resource, the negotiation, the signature scheme is different from our contract-based C-PPC model. Rather than focusing on how to express contracts, in our model, we regard contracts as events of deontic modals and operations. We deal with contract specification, merging contracts and conflict resolution between conflict contracts. In order to guarantee the fulfillment of contracts, we adopt the same view as the contract model in Shand et al.[44] on using a distributed trust model to audit participant actions if they conform contracts.

Contractual approach is also adopted in software engineering. Métayer et al.[27,26] proposed a set of methods and tools to define software liabilities among parties. Proposed framework includes the formal definition of liability and the analysis of log files to verify contractual liability *a posteriori* to make parties accountable. The proposed framework is similar to our approach regarding *a posteriori* accountability based on log analysis. However, the contractual framework is not possible to be applied to multi-synchronous collaboration systems where contracts are replicated and synchronized as in our C-PPC model.

Access control mechanisms are designed to limit which authorized users can access to and use data or resources in a computer system. This checking is performed before the access is allowed. Role-based access control (see, for example, RBAC[42], OrBAC[1], OASIS[4], NIST[15]) simplifies the specification and management of security policies within an enterprise. Most RBAC models allow permissions to be assigned to a functional role or set of roles which are hierarchically organized. The idea of incorporating attributes to RBAC models to provide more flexible RBAC was presented in Goh et al.[16], Kumar et al.[20] Our contract-based model is notably different compared to access control mechanisms. The model gives access first to data without control but with restrictions that are verified *a posteriori*.

In the field of access control, usage control is regarded as an extension of data protection beyond access control.[36] Usage control policies can be enforced by using a detective enforcement or a preventive enforcement.[14] Our work belongs to the category of detective enforcement usage control mechanisms. The C-PPC model does not help to prevent users from violating contracts; instead it makes users aware of received contracts and of other users that violated contracts. The auditing result will be used to evaluate user trustworthiness. We formalize contracts upon basic terms of right and obligation which are common in many existing works (see UCON model).[17,57] Unfortunately existing usage control models do not support multi-synchronous collaboration where users can work concurrently on shared documents. Therefore, they do not deal with merging policies and resolving conflicts among contracts.

Purpose-based access control, another approach for privacy preserving access control based on the notion of purpose, has made a significant impact on many access control systems. Purpose is a central concept in many privacy access control models for database systems[2,22] and the notion of purpose was clearly defined in Byun et al.[6] The concept of Hippocratic databases was introduced by Agrawal et

al.[2] for privacy protection within relational database systems. The proposed structure Strawman Architecture consists of privacy policies and privacy authorizations to define usage purposes. Lefevre et al.[22] presented an approach of enforcing privacy policy in database systems to let providers have control over their users on what they are allowed to see their personal data and for what purpose. Byun et al.[6] presented a model in which purpose information is associated with given data element to specify the intended use of the data element to give user in the context of relational databases. Even though these approaches define privacy policies of which purpose data can be accessed at a later time, the real access is only given after the access purpose was checked against the intended purposes was associated to the data item at a prior time. This is different from our contract-based model where contract is checked *a posteriori*. The purpose-based access control is pessimistic in considering that users are not trusted in requesting data for the right purpose. Conversely, our contract-based model is optimistic and allows users to use data first and auditing is performed later. The optimistic model is more suitable to collaborative environments where users need a certain level of mutual trust to collaborate with each other. Another major difference is that all these approaches were mainly applied for centralized database systems where policies can be verified by a central authority. In our approach we applied a contract-based model for multi-synchronous working environment where there is no central log that can be audited.

C-PPC collaboration model is closely related to the approach proposed by Wobber et al.[54] for ensuring security and privacy in a weakly consistent replication system where users are not uniformly trusted. Access control policy claims are treated as data items. The guards added to replication protocol enforce specified policies at synchronization step. A replica must check whether the requested action is allowed by the policy and then decide whether to accept or deny updates. In this approach, each replica is a local authority that maintains current policies. This is similar to our approach where we let each user perform self-auditing based on local view of other users actions. However, the approach of Wobber et al. only expresses rights but not obligations that each replica should follow. Moreover, only the author of an item can define the policy associated to it and hence there is no requirement to resolve conflicts between policies. In our approach, we need to deal with policy conflicts as multiple contributors can specify different contracts on the shared document. Moreover, the system uses a state-based replication where each site applies updates to its replica without maintaining a change log rather than an operation-based replication as in our work.

Trust management is an important aspect of the solution that we proposed. The concept of trust in different communities varies according to how it is computed and used. Our work relies on the concept of trust which is based on past user behaviors. [31] With C-PPC model users first bring social trust into the system. However trust is not immutable and it changes over time. Thus trust should be managed by using a trust model. A trust model includes three basic components[24]

that are gathering behavioral information, scoring and ranking peers and rewarding or punishing peers. Most of existing P2P trust models (e.g. EigenTrust model[18]) propose mechanisms to update trust values based on direct interactions between peers while we use log auditing to help one user evaluate others either through direct or indirect interactions. We are not aware of any existing trust model that takes log auditing result into trust assessment.

Log auditing technique is a general principle in systems supporting observation. Keeping and managing event logs is frequently used for ensuring security and privacy. This approach has been studied in many works. In Cederquist et al.[7], a log auditing approach is used for detecting misbehavior in collaborative work environments, where a small group of users shares a large number of documents and policies. In Kruhow et al.[19,38], authors present a logical policy-centric framework for behavior-based decision making. The framework consists of a formal model of past behaviors of principals which is based on event structures. However, these models[7,19,38] require a central authority that has the ability to observe all actions of all users. This assumption is not valid for a purely distributed PPC collaboration. The complexity of our log auditing mechanism compared to centralized solutions comes from the fact that each user has only a partial overview of the global collaboration and can audit only users with whom he collaborates. Therefore, a user can take decisions only from the information he possesses from the users with whom he collaborates.

7. Conclusion

We have presented a contract extended push-pull-clone model (C-PPC) for multi-synchronous collaboration where users share their private data with some contracts that receivers should comply and rust levels are adapted according to users' past behavior regarding conformance to given contracts. Changes on shared data performed by users and contracts given when data is shared are logged in a distributed manner. We formalised the notions of contracts expressed inside the C-PPC model. We proposed a merging algorithm that deals not only with changes on data but also with contracts and a conflict resolution mechanism among contracts specified in parallel by multiple contributors. A mechanism of log auditing in distributed manner is applied during collaboration and users who did not conform to given contracts are detected and made accountable by having their trust levels decremented. We implemented the proposed collaboration model with a number of simulations using PeerSim simulator. Experiment results show the feasibility of our model. Some directions of future work include proposing a trust metric that is suitable for our C-PPC model and exploring a wider range of contracts that can be specified by users.

Acknowledgments

This research was partly funded by the ANR national research grant STREAMS (ANR-10-SEGI-010).

References

1. A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, Lake Como, Italy, June 2003.
2. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic Databases. In *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002*, pages 143–154, Hong Kong, China, August 2002. VLDB Endowment.
3. C. E. Alchourron and E. Bulygin. The expressive conception of norms. In R. Hilpinen, editor, *New Studies in Deontic Logic*, pages 95 – 124. D. Reidel Publishing Company, 1981.
4. J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5:492–540, November 2002.
5. J. Bentham and H. L. A. Hart. *Of Laws in General*. University of London, Athlone Press, 1945.
6. J.-W. Byun and N. Li. Purpose based access control for privacy protection in relational database systems. *The International Journal on Very Large Data Bases*, 17:603–619, July 2008.
7. J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini. Audit-based Compliance Control. *International Journal of Information Security*, 6(2):133–151, March 2007.
8. L. Cholvya and A. Hunterb. Merging requirements from a set of ranked agents. *Knowledge-Based Systems*, 16(2):113 – 126, March 2003.
9. M. Cooperation. Sharepoint - Collaboration Software for the Enterprise. `http://sharepoint.microsoft.com`.
10. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth annual ACM Symposium on Principles of Distributed Computing, PODC'87*, pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.
11. P. Dewan and H. Shen. Controlling Access in Multiuser Interfaces. *ACM Transactions on Computer-Human Interaction*, 5(1):37–62, March 1998.
12. P. Dourish. The parting of the ways: divergence, data management and collaborative work. In *Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, ECSCW'95, pages 215–230, Norwell, MA, USA, 1995. Kluwer Academic Publishers.
13. C. A. Ellis, S. J. Gibbs, and G. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, January 1991.
14. S. Etalle and W. H. Winsborough. A posteriori compliance control. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, SACMAT '07, pages 11–20, New York, NY, USA, 2007. ACM.
15. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4:224–274, August 2001.

16. C. Goh and A. Baldwin. Towards a more complete model of role. In *Proceedings of the third ACM workshop on Role-based access control*, RBAC '98, pages 55–62, New York, NY, USA, 1998. ACM.
17. J.Park and R.Sandhu. The UCON-ABC usage control model. In *ACM Transactions on Information and System Security*, pages 7(1):128–174, 2004.
18. S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the 12th International Conference on World Wide Web, WWW 2003*, pages 640–651, Budapest, Hungary, May 2003. ACM Press.
19. K. Krukow, M. Nielsen, and V. Sassone. A Logical Framework for History-based Access Control and Reputation Systems. *Journal of Computer Security*, 16(1):63–101, January 2008.
20. A. Kumar, N. Karnik, and G. Chafle. Context sensitivity in role-based access control. *SIGOPS Operating Systems Review*, 36:53–66, July 2002.
21. L. Lamport. Times, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
22. K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 108–119. VLDB Endowment, 2004.
23. J. Loeliger. Collaborating with Git. *Linux Magazine*, June 2006.
24. S. Marti and H. Garcia-Molina. Taxonomy of Trust: Categorizing P2P Reputation Systems. *Computer Networks*, 50:472–484, March 2006.
25. F. Mattern. Virtual Time and Global States of Distributed Systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, October 1989. Elsevier Science Publishers B. V.
26. D. L. Métayer. Formal methods as a link between software code and legal rules. In *Proceedings 9th International Conference on Software Engineering and Formal Methods, SEFM 2011*, pages 3–18, Montevideo, Uruguay, November 2011.
27. D. L. Métayer, M. Maarek, V. V. T. Tong, E. Mazza, M.-L. Potet, N. Craipeau, S. Frénot, and R. Hardouin. Liability in software engineering: overview of the LISE approach and illustration on a case study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*, pages 135–144, Cape Town, South Africa, May 2010.
28. P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain. Sams: Synchronous, asynchronous, multi-synchronous environments. In *Proceedings of the Seventh International Conference on CSCW in Design, CSCWD'02*, pages 80–84, Rio de Janeiro, Brazil, 2002.
29. A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer, P2P'09*, pages 99–100, Seattle, WA, September 2009.
30. A. Moretti. Why the logical hexagon? *Logica Universalis*, 6:69–107, 2012.
31. L. Mui, M. Mohtashemi, and A. Halberstadt. A Computational Model of Trust and Reputation. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS 2002*, pages 2431–2439, Waikoloa, Big Island, Hawaii, January 2002. IEEE Computer Society.
32. B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.
33. K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 288–301, New

York, NY, USA, 1997. ACM.
34. R. Prakash, M. Raynal, and M. Singhal. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing*, 41(2):190–204, March 1997.
35. N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.
36. A. Pretschner, M. Hilty, and D. Basin. Distributed Usage Control. *Commun. ACM*, 49:39–44, September 2006.
37. C. Rahhal, H. Skaf-Molli, P. Molli, and S. Weiss. Multi-synchronous collaborative semantic wikis. In *Proceedings of the 10th International Conference on Web Information Systems Engineering*, WISE '09, pages 115–129, Berlin, Heidelberg, 2009. Springer-Verlag.
38. M. Roger and J. Goubault-Larrecq. Log Auditing through Model-Checking. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations, CSFW 2001*, pages 220–234, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
39. H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, March 2011.
40. M. Roscheisen and T. Winograd. A Communication Agreement Framework for Access/Action Control. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 154–, Washington, DC, USA, 1996. IEEE Computer Society.
41. A. Ross. *On Law and Justice*. Berkeley:University of California Press, 1959.
42. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29:38–47, February 1996.
43. B. Shand and J. Bacon. Policies in accountable contracts. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), 5-7 June 2002, Monterey, CA, USA*, pages 80–91. IEEE Computer Society, 2002.
44. B. N. Shand. Trust for resource control: Self-enforcing automatic rational contracts between computers. Technical Report UCAM-CL-TR-600, University of Cambridge, 2004.
45. R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.
46. G. Stevens and V. Wulf. A New Dimension in Access Control: Studying Maintenance Engineering Across Organizational Boundaries. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW 2002*, pages 196–205, New Orleans, Louisiana, USA, November 2002. ACM Press.
47. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.
48. H. T. T. Truong and C.-L. Ignat. Log Auditing for Trust Assessment in Peer-to-Peer Collaboration. In *Proceedings of the 10th International Symposium on Parallel and Distributed Computing, ISPDC 2011*, Cluj-Napoca, Romania, July 2011. IEEE Computer Society.
49. H. T. T. Truong, C.-L. Ignat, M.-R. Bouguelia, and P. Molli. A contract-extended push-pull-clone model. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2011*, pages 211–220, Or-

lando, FL, USA, October 2011.
50. H. T. T. Truong, C.-L. Ignat, and P. Molli. Authenticating Operation-based History in Collaborative Systems. In *Proceedings of the ACM International Conference on Supporting Group Work, Group 2012*, Sanibel Island, Florida, USA, October 2012. ACM. (to appear).
51. W3C. World Wide Web Consortium (W3C) - A P3P Preference Exchange Language 1.0 (APPEL1.0). `http://www.w3.org/TR/P3P-preferences/`.
52. W3C. World Wide Web Consortium (W3C) - P3P: The Platform for Privacy Preferences. `http://www.w3.org/P3P/`.
53. S. Weiss, P. Urso, and P. Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, August 2010.
54. T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based Access Control for Weakly Consistent Replication. In *Proceedings of the 5th European Conference on Computer Systems, EuroSys 2010*, pages 293–306, Paris, France, 2010. ACM Press.
55. G. H. V. Wright. Deontic logic. *Mind*, 60(237):1–15, January 1951. Oxford University Press.
56. G. H. V. Wright. *Norm and action : a logical enquiry*. Routledge and Kegan Paul - Humanities Press, London, New York, 1963.
57. X. Zhang, M. Nakae, M. J. Covington, and R. S. Sandhu. Toward a usage-based security framework for collaborative computing systems. *ACM Transactions on Information and System Security*, 11(1):1–36, 2008.