ARTICLE IN PRESS

Available online at www.sciencedirect.com

ScienceDirect

Data & Knowledge Engineering xxx (2007) xxx–xxx

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Design time analysis of multiagent protocols

Pınar Yolum

Department of Computer Engineering, Boğaziçi University, TR-34342 Bebek, Istanbul, Turkey

Received 21 July 2006; accepted 21 December 2006

Abstract

Interaction protocols enable agents to communicate with each other effectively. Whereas several approaches exist to specify interaction protocols, none of them has design tools that can help protocol designers catch semantic protocol errors at design time. As research in networking protocols has shown, flawed specifications of protocols can have disastrous consequences. Hence, it is crucial to systematically analyze protocols in time to ensure effective specification. This paper first studies and formalizes important generic properties of commitment protocols that can ease their effective and consistent development significantly. Next, we identify robustness properties of protocols that are useful in determining the applicability of protocols in different settings. Since these properties are formal, they can easily be incorporated in a software tool to (semi-)automate the design and specification of commitment protocols. Where appropriate we provide algorithms that can directly be used to check these properties in such a design tool.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Commitments; Protocols; Design tools; Analysis; Robustness

1. Introduction

Multiagent systems consist of autonomous, interacting agents. For the agents to interact effectively, their interactions should be regulated. Multiagent interaction protocols provide a formal ground for realizing this regulation. Similar to the protocols in traditional systems, multiagent protocols need to be specified rigorously so that the agents can interact successfully. Some important properties of network protocols have been studied before, where a protocol was represented as a finite state machine (FSM) [13,14]. However, FSMs are not well-suited for dynamic environments of multiagent systems [5,19,25]. Contrary to the protocols in static systems, multiagent protocols need to be specified flexibly so that the agents can exercise their autonomy by making choices or by dealing with exceptions as best suits them [25,26]. However, developing effective protocols that will be carried out by autonomous agents is challenging [15,16].

Recently, social constructs are being used to specify agent interactions. These approaches advocate declarative representations of protocols and give semantics to protocol messages in terms of social (and thus observable) concepts. Alberti et al. specify interaction protocols using social integrity constraints and reason about the expectations of agents [1]. Fornara and Colombetti base the semantics of agent communication on

E-mail address: pinar.yolum@boun.edu.tr

0169-023X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2006.12.001

commitments, such that the meanings of messages are denoted by commitments [12]. Yolum and Singh develop a methodology for specifying protocols wherein protocols capture the possible interactions of the agents in terms of the commitments to one another [25,26].

In addition to providing flexibility, these approaches make it possible to verify compliance of agents to a given protocol. Put broadly, commitments of the agents can be stored publicly and agents that do not fulfill their commitments at the end of the protocol can be identified as non-compliant. In order for these approaches to make use of all these advantages, the protocols should be designed rigorously. For example, the protocol should guarantee that, if an agent does not fulfill its commitment, it is not because the protocol does not specify how the fulfillment can be carried out. The aforementioned approaches all start with a manually designed, correct protocol. However, designing a correct protocol in the first place requires important effectiveness properties to be established and applied to the protocol. A correct protocol should define the necessary actions (or transitions) to lead a computation to its desired state. Following a protocol should imply that progress is being made towards realizing desired end conditions of the protocol. The followed actions should not yield conflicting information and lead the protocol to unrecoverable errors. That is, the protocol should at least allow a safe execution. Depending on the circumstances, the protocol should allow enough alternatives so that if one alternative fails, the protocol would still be fulfilled by following another alternative execution.

Contributions: This paper develops and formalizes design requirements for developing *commitment protocols* [22,25]. The developed design requirements capture the following concepts:

- *Effectiveness.* The transitions of a protocol should be enough to ensure that the protocol can be executed and ended successfully. To study if a protocol is effective or not, we first formalize the concepts of progress and effective progress and then derive rules that can be checked at compile time to find out if a given commitment protocol effectively progresses.
- *Consistency.* A protocol should not yield conflicting computations. The participants of a protocol should not be able to execute actions that lead the protocol to enter an inconsistent state. To study if a protocol is consistent, we study the combined effects of commitment operations to see if they can yield an inconsistent state.
- *Robustness.* A protocol is robust if the tasks that need to be carried out as a result of executing the protocol can be achieved in several ways. To study the robustness of protocols, we study the alternative execution paths in the protocol.

Whereas the design requirements are derived for commitment protocols, the underlying ideas are generic and can be applied to other social approaches as well. By usage of these requirements, inconsistencies as well as errors can be detected during design time. For the derived requirements, we provide algorithms with which each requirement can be checked against a protocol specification at design time. These requirements can easily be automated in a design tool to help protocol designers develop protocols.

Organization: The rest of the paper is organized as follows. Section 2 gives a technical background on event calculus and commitments. Section 3 reviews commitment protocols. Sections 4 and 5 develop effectiveness and consistency requirements, respectively. Section 6 shows how these requirements can be implemented in a design tool. Section 7 identifies two key classes of commitment protocols and shows methods to automatically detect them. Section 8 discusses the recent literature in relation to our work and Section 9 summarizes the results and contributions of the paper.

2. Technical background

We first give a brief overview of event calculus, which we use to formalize the design requirements. Next, we summarize Yolum and Singh's formalization of commitments and their operations.

2.1. Event calculus

The event calculus (EC) is a formalism based on many-sorted first order logic [17]. The three sorts of event calculus are *time points* (T), *events* (E) and *fluents* (F). Fluents are properties whose truth values can change

Table 1
Event calculus predicates

$Initiates(a, f, t)$	f holds after event a at time t
$Terminates(a, f, t)$	f does not hold after event a at time t
$Initially_{P}(f)$	f holds at time 0
$Initially_{N}(f)$	f does not hold at time 0
$Happens(a, t_1, t_2)$	event a starts at time t_1 and ends at t_2
$Happens(a, t)$	event a starts and ends at time t
$HoldsAt(f, t)$	f holds at time t
$Clipped(t_1, f, t_2)$	f is terminated between t_1 and t_2
$Declipped(t_1, f, t_2)$	f is initiated between t_1 and t_2

over time. Fluents are manipulated by initiation and termination of events. Table 1 supplies a list of predicates to help reason about the events in an easier form. Below, events are shown with $a, b, \ldots$; fluents are shown with $f, g, \ldots$; and time points are shown with t, t_1, and t_2.

We introduce the subset of the EC axioms that are used here; the rest can be found elsewhere [20]. The variables that are not explicitly quantified are assumed to be universally quantified. The standard operators apply (i.e., $\leftarrow$ denotes implication and $\wedge$ denotes conjunction). The time points are ordered by the $<$ relation, which is defined to be transitive and asymmetric.

(1) $HoldsAt(f, t_3) \leftarrow Happens(a, t_1, t_2) \wedge Initiates(a, f, t_1) \wedge (t_2 < t_3) \wedge \neg\, Clipped(t_1, f, t_3)$
(2) $Clipped(t_1, f, t_4) \leftrightarrow \exists a, t_2, t_3\ [Happens(a, t_2, t_3) \wedge (t_1 < t_2) \wedge (t_3 < t_4) \wedge Terminates(a, f, t_2)]$
(3) $\neg HoldsAt(f, t) \leftarrow Initially_{N(f)} \wedge \neg Declipped(0, f, t)$
(4) $\neg HoldsAt(f, t_3) \leftarrow Happens(a, t_1, t_2) \wedge Terminates(a, f, t_1) \wedge (t_2 < t_3) \wedge \neg Declipped(t_1, f, t_3)$

2.2. Commitments

Commitments are obligations from one party to another to bring about a certain condition [6]. A base-level commitment $\mathsf{C}(x, y, p)$ binds a debtor x to a creditor y to bring about a condition p [21]. When a base-level commitment is created, x becomes responsible to y for satisfying p, i.e., p should hold sometime in the future. The condition p does not involve other conditions or commitments.

A conditional commitment $\mathsf{CC}(x, y, p, q)$ denotes that if the condition p is satisfied, x will be committed to bring about condition q. Conditional commitments are useful when a party wants to commit only if a certain condition holds or only if the other party is also willing to make a commitment. It is easy to see that a base-level commitment is a special case of a conditional commitment, where the condition is set to true. That is, $\mathsf{C}(x, y, p)$ is an abbreviation for $\mathsf{CC}(x, y, true, p)$. Commitments are represented as fluents in the event calculus. Hence, the creation and the manipulation of the commitments are shown with the *Initiates* and *Terminates* predicates.

Compared to the traditional definitions of obligations, commitments can be carried out more flexibly [21]. By performing operations on an existing commitment, a commitment can be manipulated (e.g., delegated to a third-party). We summarize the operations to create and manipulate commitments [21,25]. In the following discussion, x, y, z denote agents, c, c' denote commitments, and e denotes an event.

(1) $Create(x, \mathsf{C}(x, y, p))$: When x performs the event e, the commitment c is created.
$\exists e\{Happens(e, t) \wedge Initiates(e, \mathsf{C}(x, y, p), t)\}$
(2) $Discharge(x, \mathsf{C}(x, y, p))$: When x performs the event e, the commitment c is resolved.
$\exists e\{Happens(e, t) \wedge Initiates(e, p, t)\}$
(3) $Cancel(x, \mathsf{C}(x, y, p))$: When x performs the event e, the commitment c is canceled. Usually, the cancellation of a commitment is followed by the creation of another commitment to compensate for the former one.
$\exists e\{Happens(e, t) \wedge Terminates(e, \mathsf{C}(x, y, p), t)\}$
(4) $Release(y, \mathsf{C}(x, y, p))$: When y performs the event e, x no longer need to carry out the commitment c.
$\exists e\{Happens(e, t) \wedge Terminates(e, \mathsf{C}(x, y, p), t)\}$

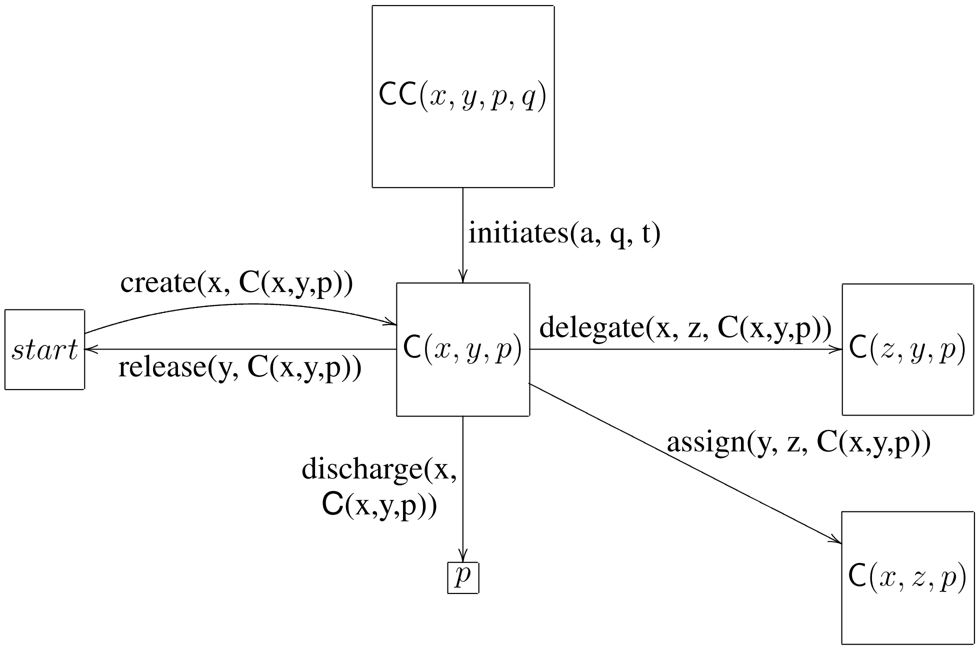

Fig. 1. Commitment transitions.

(5) *Assign*$(y, z, \mathsf{C}(x, y, p))$: When y performs the event e, commitment c is eliminated, and a new commitment c' is created where z is appointed as the new creditor.
$\exists e\{\mathit{Happens}(e, t) \wedge \mathit{Terminates}(e, \mathsf{C}(x, y, p), t) \wedge \mathit{Initiates}(e, \mathsf{C}(x, z, p), t)\}$

(6) *Delegate*$(x, z, \mathsf{C}(x, y, p))$: When x performs the event e, commitment c is eliminated but a new commitment c' is created where z is the new debtor.
$\exists e\{\mathit{Happens}(e, t) \wedge \mathit{Terminates}(e, \mathsf{C}(x, y, p), t) \wedge \mathit{Initiates}(e, \mathsf{C}(z, y, p), t)\}$

The following rules operationalize the commitments. Axiom 1 states that a commitment is no longer in force if the condition committed to holds. In Axiom 1, when the event e occurs at time t, it initiates the fluent p, thereby discharging the commitment $\mathsf{C}(x, y, p)$.

Commitment Axiom 1. *Discharge*$(x, \mathsf{C}(x, y, p)) \leftarrow \mathit{HoldsAt}(\mathsf{C}(x, y, p), t) \wedge \mathit{Happens}(e, t) \wedge \mathit{Initiates}(e, p, t)$

The following axiom captures how a conditional commitment is resolved based on the temporal ordering of the commitments it refers to. When the conditional commitment $\mathsf{CC}(x, y, p, q)$ holds, if p becomes true, then the original commitment is terminated but a new commitment is created, since the debtor x is now committed to bring about q.

Commitment Axiom 2. $\mathit{Initiates}(e, \mathsf{C}(x, y, q), t) \wedge \mathit{Terminates}(e, \mathsf{CC}(x, y, p, q), t) \leftarrow \mathit{HoldsAt}(\mathsf{CC}(x, y, p, q), t) \wedge \mathit{Happens}(e, t) \wedge \mathit{Initiates}(e, p, t)$

Fig. 1 depicts an overview of the possible transitions based on the commitment operations and axioms.

3. Commitment protocols

A commitment protocol is a set of actions such that each action is either an operation on commitments or brings about a proposition. Agents create and manipulate commitments they are involved in through the protocol they follow. As is customary in multiagent systems, we assume that actions are associated with roles. An agent can start a protocol by performing any of the actions that is allowed by the role it is playing. The transitions of the protocol are computed by applying the effect of the action on the current state. In most cases this corresponds to the application of commitment operations. Given a protocol specification, actions of the protocol can be executed from an arbitrary initial state to a desired final state. A protocol run can be viewed as a series of actions; each action happening at a distinct time point.

Example 1. We consider the Contract Net Protocol (CNP) as our running example [11]. CNP starts with a manager requesting proposals for a particular task. Each participant either sends a proposal or a reject message. The manager accepts one proposal among the submitted proposals and (explicitly) rejects the rest. The participant with the accepted proposal informs the manager with the proposal result or the failure of the proposal.

Example 2. By sending a call-for-proposals, the manager makes a conditional commitment such that if a participant sends a proposal, than the manager will decide and reply with the results of the call-for-proposals. This can be represented with CC(M, P, proposal, (accepted ∨ rejected)). By sending a proposal to the manager, a participant creates a conditional commitment such that if the manager accepts the proposal, then the participant will deliver the result of the proposal (e.g., CC(participant, manager, accepted, result)). If the manager then sends an accept message, this conditional commitment will cease to exist but the following base-level commitment will hold: C(participant, manager, result). Since the commitments can be easily manipulated, the participant can act in the following ways: (1) it can discharge its commitment by sending the result as in the original CNP (discharge), (2) it can delegate its commitment to another participant, who carries out the proposal (delegate), or (3) it can send a failure notice as in the original protocol (cancel).

Table 2 tabulates the possible actions in the protocol and the fluent each action brings about. As stated before, each action either corresponds to a commitment operation or brings about a proposition in the world. Fig. 2 gives an example protocol flow, where *M* denotes the manager and *P* and *P*1 denote participants. The contents of the states are depicted in Table 3 to ease readability.

Table 2
Actions and the corresponding commitment operations in CNP

sendCFP(M, P)	create(M, CC(M, P, proposal, reply))
sendProposal(P, M)	create(P, CC(P, M, accepted, result))
sendReply(M)	(accepted ∨ rejected)
sendResult(P)	discharge(P, C(P, M, result))
transferProposal(P, P1)	delegate(P, P1, C(P, M, result))
cancelProposal(P)	cancel(P, C(P, M, result))

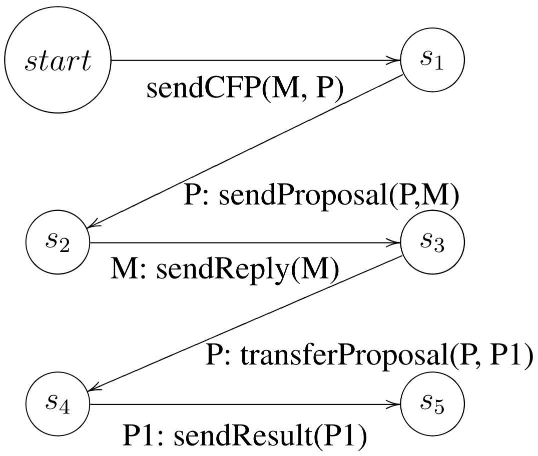

Fig. 2. CNP transitions.

Table 3
State identifiers and the corresponding fluents that hold

Start	True
s_1	CC(M, P, CC(P, M, accepted, result)), (accepted $\vee$ rejected)
s_2	C(M, P, (accepted $\vee$ rejected)) $\wedge$ CC(P, M, accepted, result)
s_3	accepted $\wedge$ C(P, M, result)
s_4	accepted $\wedge$ C(P1, M, result)
s_5	accepted $\wedge$ result

4. Protocol effectiveness

Analysis of commitment protocols poses two major challenges. One, the states of a commitment protocol are not given *a priori* as is the case with FSMs. Two, the transitions are computed at run time to enable flexible execution. That is, as described before the actions of the agents correspond to commitment actions. The commitment actions define how the operations change the commitments in the world. Based on the executed actions, the state of the commitments are derived and this state form the new protocol state.

To conclude that a protocol is effective, we need to first show that the actions in the protocol are enough to allow the protocol to generate desired states. Next, we need to show that the protocol allows making progress such that the execution of the protocol does not get stuck in cycles. Finally, we need to show that the protocol can make effective progress by yielding the desired states.

To achieve the above properties, we study the possible protocol runs that can result from a protocol specification. A protocol run specifies the actions that happen at certain time points. We base the definition of a protocol state on these time points. More specifically, a state of the protocol corresponds to the set of propositions and commitments that hold at a particular time point in a particular run.

To ease the explanation, we introduce the following notation. Let F be the set of fluents in the protocol. F is $CS \cup CCS \cup PS$ such that CS is the set of base-level commitments, CCS is the set of conditional commitments and PS is the set of propositions in the protocol. Let c be a commitment such that $c \in CS$ then $O(c)$ is the set of operations allowed on the commitment c in the protocol and $O = \{O(c) : c \in CS\}$. Since a commitment cannot be part of a protocol if it cannot be created, we do not explicitly add it to the operation set. Hence, $O(c)$ can contain five types of operations in Section 2.2, namely, discharge, cancel, release, delegate, and assign. We assume that all the propositions referred by the commitments in CS and CCS are in PS.

Definition 1. A *protocol state* $s(t)$ captures the *content* of the protocol with respect to a particular time point t. A protocol state $s(t)$ is a conjunction of $HoldsAt(f,t)$ predicates with a fixed t but possibly varying f. Formally, $s(t) \equiv \wedge_{f \in F'} HoldsAt(f,t)$ such that $F' \subseteq F$.

Two states are equivalent if the same fluents hold in both states. Although the two states are equivalent, they are not strictly the same state since they can come about at different time points.

Definition 2. The $\equiv$ operator defines an equivalence relation between two states $s(t)$ and $s(t')$ such that $s(t) \equiv s(t')$ if and only if $\forall f \in F : (HoldsAt(f,t) \iff HoldsAt(f,t'))$.

Protocol execution captures a series of operations for making and fulfilling of commitments. Intuitively, if the protocol executes successfully, then there should not be any open base-level commitments; i.e., no participant should still have commitments to others. This motivates the following definition of an end-state.

Definition 3. A protocol state $s(t)$ is a *proper* end-state if no base-level commitments exist. Formally, $\forall f \in F : HoldsAt(f,t) \Rightarrow f \notin CS$.

Generally, if the protocol ends in an unexpected state, i.e., not a proper end-state, one of the participants is not conforming to the protocol. However, to claim this, the protocol has to ensure that participants have the choice to execute actions that will terminate their commitments. The following analysis derives the requirements for effective commitment protocols.

Holzmann labels states of a protocol in terms of their capability of allowing progress [14]. Broadly put, a protocol state can be labeled as a progressing state if it is possible to move to another state. For a protocol to function effectively, all states excluding the proper end-states should be progressing states. Otherwise, the protocol can move to a state where no actions are possible, and hence the protocol will not progress and immaturely end.

Definition 4. A protocol state $s(t)$ is progressing if both of the following hold:

- $s(t)$ is not a proper end-state (e.g., $s(t) \Rightarrow \exists f \in CS : HoldsAt(f, t)$).
- there exists an action that if executed creates a transition to a different state. (e.g., $s(t) \Rightarrow \exists t' : t < t' \wedge s(t) \not\equiv s(t')$)

At every state in the protocol, either the execution should have successfully completed (i.e., proper end-state) or should be moving to a different state (i.e., progressing state).

Definition 5. A protocol $\mathscr{P}$ is *progressive* if and only if each possible state in the protocol is either a proper end-state or a progressing state.

This follows intuitively from the explanation of making progress. Lemma 1 formalizes a sufficient condition for ensuring that a commitment protocol is progressive.

Lemma 1. *Let $\mathscr{P}$ be a commitment protocol and c be a base-level commitment. If $\forall c \in CS : O(c) \neq \emptyset$, then $\mathscr{P}$ is progressive.*

Proof. By Definition 5, every state in $\mathscr{P}$ should be a proper end-state or a progressing state. If a state does not contain open commitments then it is a proper end-state (Definition 3). If the state does contain a base-level commitment, then since at least one operation exists to manipulate it, the protocol will allow a transition to a new state. Thus, the state is a progressing state (Definition 4). □

Ensuring a progressing protocol is the first step in ensuring effectiveness. If a protocol is not progressing, then the participants can get stuck in an unexpected state and not transition to another state. However, progress by itself does not guarantee that the interactions will always lead to a proper end-state. This is similar in principle to livelocks in network protocols, where the protocol can transition between states but never reach a final state [14, p. 120].

Example 3. Consider a participant P whose proposal has been accepted (hence, C(P, M, result)). Next, the participant delegates its commitment to another participant $P1$ (hence, C(P1, M, result)). As was the case in the example execution in Fig. 2. Next, participant $P1$ delegates the commitment back to participant P and thus the protocol moves back to the previous state (C(P, M, result)). Participants P and $P1$ delegate the commitment back and forth infinitely as shown in Fig. 3.

Obviously, the situation explained in Example 3 is is not desirable. It is necessary to ensure progress but this is not sufficient to conclude that the protocol is making *effective* progress.

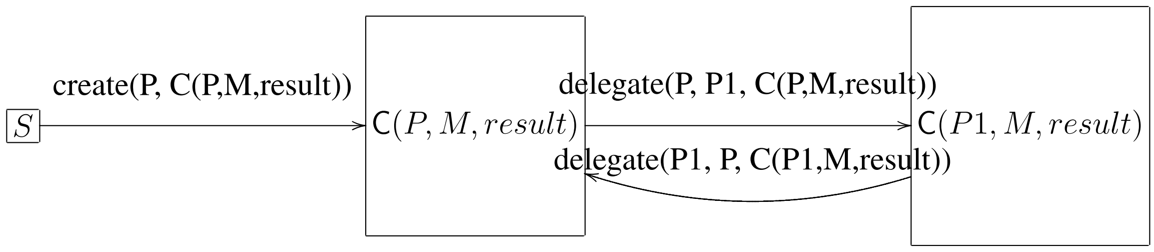
create(P, C(P,M,result))
S
C(P, M, result)
delegate(P, P1, C(P,M,result))
delegate(P1, P, C(P1,M,result))
C(P1, M, result)

Fig. 3. Infinitely delegating a commitment.

Definition 6. A cycle in a protocol refers to a non-empty sequence of states that start and end at equivalent states. A cycle can be formalized by the content of the beginning and ending states. That is, an execution sequence is a cycle if: $\exists t, t', t'' \in T : (s(t) \equiv s(t')) \wedge (t < t'' < t') \wedge (s(t) \not\equiv s(t''))$.

Definition 7. An infinitely repeating cycle is a cycle with progressing states such that if the protocol gets on to one of the states then the only possible next transition is to move to a state in the cycle [14].

In Example 3, the two delegate actions form an infinitely repeating cycle. Once the protocol gets into either state 2 or state 3, it will always remain in one of these two states.

Lemma 2. *An infinitely repeating cycle does not contain any proper end-states.*

Proof. By Definition 7 an infinitely repeating cycle only contains progressing states and by Definition 4, a progressing state cannot be an end-state.

Given a cycle, it is easy to check if it is infinitely repeating. Informally, for each state in the cycle, we need to check if there is a possible transition that can cause a state outside the cycle. This can be achieved by applying all allowed operations (by the proposition) to the commitments that exist in that state. As soon as applying a commitment operation to a state in the cycle yields a state not included in the cycle, the procedure stops, concluding that the cycle is not infinitely repeating. □

Lemma 3. *Let l be a cycle. Let $c \in CS$ be a commitment that holds at a state $s(t)$ on this cycle at any time t. If* discharge, cancel *or* release $\in O(c)$ *then cycle l is not infinitely repeating.*

Proof. A cycle is not infinitely repeating if there is a path from a state in the cycle to a state outside the cycle. Discharging, canceling, or releasing a commitment will lead the protocol to go to a proper end-state. Since no proper end-state is on an infinitely repeating cycle, the cycle will not repeat (Lemma 2). □

Example 4. In Example 3, if either participant could discharge or cancel the commitment or could have been released from the commitment, then there need not have been an infinitely repeating cycle.

Note that we are not concerned about the choices of the agents in terms of which actions to take. Looking back at Example 3, assume that agent P could also execute an action that could discharge its commitment (to carry out the proposal), but choose instead to delegate it to agent $P1$. The protocol then would still loop infinitely. However, our purpose here is to make sure that agent P has the choice of discharging. The protocol should allow an agent to terminate its commitment by providing at least one appropriate action. It is then up to the agent to either terminate it or delegate it as.

Definition 8. A protocol $\mathscr{P}$ is *effectively progressive* if and only if and only if (1) $\mathscr{P}$ is progressive and (2) $\mathscr{P}$ does not have infinitely repeating cycles.

Theorem 1. *$\mathscr{P}$ is an effectively progressive protocol if for any commitment $c \in CS$ either* (1) discharge $\in O(c)$ *or* cancel $\in O(c)$ *or* release $\in O(c)$ *or* (2) *by applying finite number of operations a commitment c' is reached for which* discharge $\in O(c')$ *or* cancel $\in O(c')$ *or* release $\in O(c')$.

Proof. In both cases, for all commitments in $\mathscr{P}$, there is at least one operation defined. Hence, by Lemma 1, $\mathscr{P}$ is progressive. Assume that $\mathscr{P}$ has an infinite cycle. By Lemma 3, there has to be a commitment c'' holding in some state on the cycle for which none of the operations lead to a state with discharge, cancel, or release operators. Since $\mathscr{P}$ does not allow such a state, $\mathscr{P}$ does not contain an infinitely repeating cycles. □

Example 5. Let us study if the CNP specification given in Example 2 is effectively progressive. According to Theorem 1, each commitment in CS should have at least one of discharge, cancel, or release operation. Otherwise, it should be delegated or assigned to a different commitment that defines one of these operations. For the CNP specification, the CS set contains C(P, M, result) and C(M, P, (accepted ∨ rejected)). The first commitment can be discharged, canceled and delegated by the participant. The second commitment can be discharged be the manager. Hence, the CNP specification given in Example 2 is effectively progressive. An algorithm that automatically checks for an effectively progressive protocol is given in Section 6.

5. Protocol consistency

In Section 4 we have defined the requirements to guarantee that a protocol can effectively progress. However, in addition to effective progress, a protocol should always preserve a consistent computation. In other words, a protocol that functions correctly does not allow creation of conflicting information. Following the CNP example, a participant should not be able to both refuse to send a proposal and send a proposal during the same execution of the protocol. That is, the available information that is created by the protocol should be consistent at every time point of the protocol. To explain the consistency requirements for a commitment protocol, we again start with studying individual states. Since each state is defined in terms of holding commitments and propositions, we start by defining when the commitments and propositions are inconsistent.

Definition 9. Let p and r be two propositions such that $p, r \in PS$. If p entails the negation of r, that is, $\mathsf{false} \leftarrow HoldsAt(p,t) \wedge HoldsAt(r,t)$ then p and r are *conflicting*. A protocol state $s(t)$ is *consistent* if $s(t) \not\equiv \mathsf{false}$.

Obviously, the protocol should never enter an inconsistent state. The set of operations defined for a commitment should ensure that only consistent states are realized. Notice that we allow two base-level commitments to exist together even if the propositions that need to be brought out by these commitments are conflicting. That is, a state could contain two commitments $C(x,y,p)$ and $C(x,y,r)$ such that p and r are conflicting. Obviously, both commitments cannot be satisfied simultaneously. Hence, discharging one commitment restricts the discharging of the second commitment.

Definition 10. A protocol $\mathscr{P}$ is *consistent* if and only if $\mathscr{P}$ is progressive and each possible state in the protocol is consistent.

Lemma 4. *Let $\mathscr{P}$ be a commitment protocol and c and c′ be two base-level commitments in CS such that c and c′ have conflicting propositions. If $O(c) = O(c') = \{\mathsf{discharge}\}$, then $\mathscr{P}$ is not consistent.*

Proof. Let $C(x,y,p)$ and $C(x,y,r)$ be any two commitments in CS with conflicting propositions. If either of them is not discharged, then the protocol state will contain a base-level commitment. By Definition 3, it will not be a proper end-state. If both of them discharge, the protocol will move to the false state. Thus, by Definition 10, it will not be consistent. □

Definition 11. Extended commitment set ECS contains all base-level commitments that can be derived by possible operations on the commitments in CS.

Theorem 2. *Let $\mathscr{P}$ be an effectively progressive commitment protocol, and c and c′ be two base-level commitments with conflicting propositions in ECS. If either* release $\in O(c\prime)$ *or* cancel $\in O(c\prime)$ *then $\mathscr{P}$ is consistent.*

Proof. If discharge$\notin O(c)$, then $\mathscr{P}$ can never move into the false state and hence will be consistent. If discharge $\in O(c)$, by Lemma 4, c' needs to define an operation other that discharge to avoid the false state. By Theorem 1, commitment c' should define at least one of discharge, release, or cancel. Since discharge is eliminated by Lemma 4, at least release, or cancel should be defined. □

Example 6. Assume that a participant commits to send a proposal and at the same time refuses to send a proposal (commits not to send a proposal). Then the participant will not be able to discharge both of its commitments. On the other hand, if the participant can cancel one of its commitment or if the manager releases the participant from one of them, then the protocol can continue consistently.

6. Algorithms

The results of the previous sections can be implemented in a design tool. This section provides algorithms to compute the derived effectiveness and consistency requirements of Theorems 1 and 2.

A commitment graph $G = (V, E)$ consists of a set of nodes V and a set of edges E. Each node denotes a single possible base-level commitment in a given protocol. A directed edge between node u to v denotes an operation applied on the commitment at node u, yielding node v. A commitment graph contains three designated nodes, namely RC, D and S. These nodes do not contain any commitments. RC is used as a sink node for all commitments for which a release or a cancel operation is defined. In other words, if a node u is connected to node RC then the operation on edge (u, RC) could only be a release or a cancel operation (since these operations resolve the commitment, and do not create other commitments). Similarly, node D is a sink node for commitments for which discharge is defined. If a node u is connected to node D then the operation on edge (u, D) could only be a discharge. If there is an edge (u, v) such that v is not the RC or the D node, then the operation associated with the edge is either a delegate or an assign. S node does not have any incoming edges, but only outgoing edges. The only edges allowed out of S are create operations that lead to nodes for newly created commitments. Fig. 4 shows an example commitment graph for the contract net protocol that is explained in Examples 1 and 2.

Algorithm 1 takes as input the base-level commitment set CS and operations set O and builds a commitment graph. The algorithm starts by creating the RC and the D nodes. Then, the algorithm iterates over the set of possible commitments that can be created by the protocols (ECS) and adds a new node for each commitment. After adding a node for a commitment, it goes through the operations set of the commitment and adds an edge between the node and the RC state for cancel and release operations and an edge between the node and the D state for discharge operation. If there is an assign or a delegate operation, the algorithm applies the operation on the commitment and creates a new node with the resulting commitment. The resulting commitment corresponds to the initiated commitment as explained in Section 2.2. The new commitment is added to the set of possible commitments.

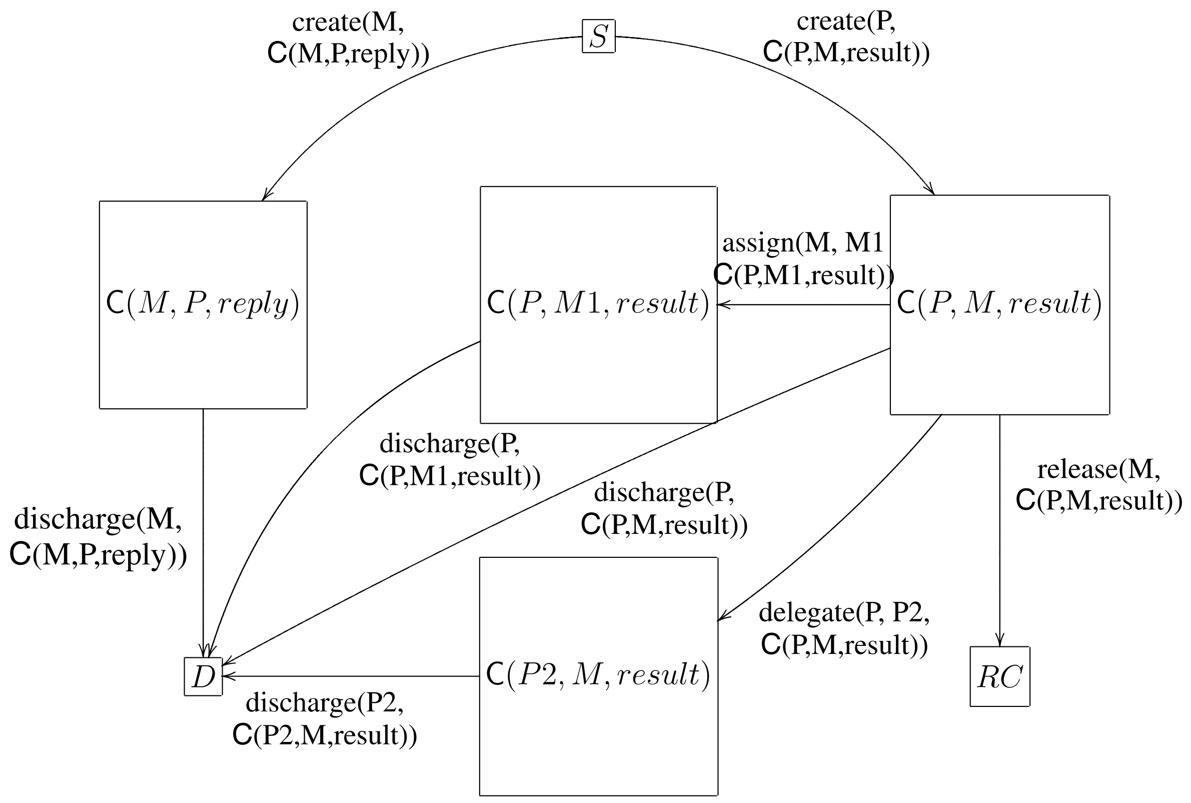
create(M, C(M,P,reply))
S
create(P, C(P,M,result))
C(M, P, reply)
C(P, M1, result)
assign(M, M1 C(P,M1,result))
C(P, M, result)
discharge(P, C(P,M1,result))
discharge(P, C(P,M,result))
release(M, C(P,M,result))
discharge(M, C(M,P,reply))
delegate(P, P2, C(P,M,result))
D
discharge(P2, C(P2,M,result))
C(P2, M, result)
RC

Fig. 4. An example commitment graph for CNP.

Algorithm 1. Build-commitment-graph(CS: Set of base-level commitments; O: Set of operations on base-level commitments)

```
1: Create a new node $RC$ {$RC$ stands for a sink node for release and cancel}
2: Create a new node $D$ {$D$ stands for a sink node for discharge}
3: ECS = $CS$
4: while (ECS !=$\emptyset$)
5:   Remove a commitment $c$
6:   Add a new node $c$ to $V$
7:   for $i = 1$ to $|O(c)|$ do
8:     if $(O(c)[i] ==$ delegate) then
9:       Add a new node $c.delegate$ to $V$
10:      Add $(c, c.delegate)$ to $E$
11:      Add $c.delegate$ to ECS
12:    else if $(O(c)[i] ==$ assign) then
13:      Add a new node $c.assign$ to $V$
14:      Add $(c, c.assign)$ to $E$
15:      Add $c.assign$ to ECS
16:    else if $(O(c)[i] ==$ release$)\|(O(c)[i] ==$ cancel) then
17:      Add $(c, RC)$ to $E$
18:    else if $(O(c)[i] ==$ discharge) then
19:      Add $(c, D)$ to $E$
20:    end if
21:  end for
22: end while
```

We assume that the graph contains a standard adjacency matrix that can determine if a node has an edge to another node. In the commitment graph, this shows whether applying a single action can transform the commitment either to another commitment or lead it to one of the discharge, cancel, or release states. The *adjacentTo* method serves this purpose. If a commitment node has at least one outgoing edge, then the commitment is said to have a neighbor (i.e., *hasNeighbors()* method is true).

Algorithm 2. Color-graph(G:Commitment Graph)

```
1: $visited = \emptyset$
2: $whiteList = \emptyset$
3: $blackList = \emptyset$
4: for $i = 1$ to $|V|$ do
5:   if $(V(i) \notin visited)$ then
6:     visit (V(i))
7:   end if
8: end for
```

Algorithm 2 checks if all the commitments in the commitment graph can be resolved. To do this, it functions like a search algorithm. Algorithm 2 takes as input a commitment graph and visits each node (with Algorithm 3) to color each node. If a node satisfies the properties in Theorem 1, then it is colored white, if not black. The algorithm terminates when all nodes are colored.

Algorithm 3 takes as input the node u that will be visited, goes through the nodes as in depth first search (DFS), and assigns a color. White nodes are stored in the *whiteList* and the black nodes are stored in the *blackList*. All visited nodes are stored in the *visited* set. Initially, nodes do not have any color. The node u is first added to the *visited* set.

If u does not have any outgoing edges, then it is a singleton in the graph and is not connected to the rest of the graph. Hence, the commitment has no operations defined and thus cannot be resolved. Such nodes are labeled as black and put into *blackList*. If the commitment at node u has one of the discharge, cancel, or release operations defined (there is an edge between u and RC or u and D), then the color of the node u becomes white. This means that the protocol allows commitment node u to be resolved. Otherwise, the neighbors of the node u are analyzed.

Algorithm 3. visit(u: node)

```
1: Add $u$ to $visited$
2: if ($u$.adjacentTo($D$ OR $CR$)) then
3:    Add $u$ to $whiteList$
4: else if ($u$.hasNeighbors()) then
5:    while ($u \notin whiteList$) AND ($\exists E(u,v)$: $v \notin visited$) do
6:       if ($v \notin visited$) then
7:          visit($v$)
8:       end if
9:       if ($v \in whitelist$) then
10:         Add $u$ to $whiteList$
11:      else
12:         Add $u$ to $blackList$
13:      end if
14:   end while
15: else
16:   Add $u$ to $blackList$
17: end if
```

If any one neighbor node v is already white, then u is also labeled as white. The intuition is that if the commitment at v can be resolved and if the commitment at u can be transformed (by delegate or assign) to v, then v can be resolved, too. If no neighbor node is already white, then the algorithm visits neighbor nodes that are not already visited. The aim is to find a directed path from the current node to a white node. When a white node is found, then all nodes on the path become white and are inserted into *whiteList*. If a white node cannot be reached by a directed path, then all nodes on the path become black and are added to the *blackList*. Algorithms 2 and 3 are a variant of DFS and thus computes the set of unresolvable commitments in $O(|E|)$ [7]. The protocol designer can modify the protocol until the *blackList* computed by this algorithm is empty.

Algorithm 4. Check-consistency(G: Commitment Graph)

```
1: $inconsistentList = \emptyset$
2: for $i = 1$ to $|V| - 1$ do
3:    for $j = i + 1$ to $|V|$ do
4:       Determine if $V(i)$ and $V(j)$ are conflicting
5:       if conflicting($V(i)$ and $V(j)$) then
6:          if ($\nexists E(V(i), RC)$) AND ($\nexists E(V(j), RC)$) then
7:             Add $V(i)$ and $V(j)$ to $inconsistentList$
8:          end if
9:       end if
10:   end for
11: end for
```

Algorithm 4 checks the protocol consistency (Theorem 2). The algorithm compares all commitments to each other to see if they have conflicting propositions. If so, the algorithm checks if either of the commitments can be released or canceled.

The *inconsistentList* keeps the pairs of commitments that fail the test. Algorithm 4 computes the set of inconsistent commitments in $O(|V|^2)$. After this set is computed, a protocol designer can modify the protocol until the set of inconsistent commitments is empty.

The constructions developed here can be implemented by a design tool that takes in a description of the protocol, which contains the actions and the commitment operation each action corresponds to as specified in Section 3. The commitment and operation set of the protocol can then be easily formed and fed into Algorithm 1 for creating a commitment graph. Once, there is a commitment graph both Algorithms 2 and 4 can be applied to check effectiveness and consistency, respectively.

7. Robustness considerations

When applied in a design tool, the algorithms in Section 6 can signal when a protocol in inconsistent or ineffective. After deciding that a protocol does not suffer from inconsistency or ineffectiveness, the next step is to analyze the protocol's applicability in different contexts. To decide which protocol is more appropriate in a particular setting, robustness of the protocols need to be considered. For example, if a protocol suffers from a single point of failure, then the protocol may not be appropriate for mission-critical applications or conversely, a protocol that provides multiple options for fulfilling its tasks may be costly for everyday applications. To understand the robustness of protocols, we study the protocols in terms of the failures that may arise in a protocol.

Traditionally, protocol failures are attributed to message losses, or message errors. While such failures are important, here our focus is not on the message losses, but on the agents that generate the messages. More specifically, we are concerned that an agent participating in a protocol may not fulfill its tasks by invoking relevant commitment operations. Since the agents are autonomous, there are no guarantees that the agents fulfill their commitments. But, this may have the consequence of the protocol not being successfully completed. For example, a commitment on which the only operation that can be invoked is the discharge operation. If the debtor of the commitment cannot discharge the commitment, then the protocol will get stuck. The reason for not being able to discharge the commitment may vary from not having the know-how to discharge to potential errors in the agent code. Similarly, if an agent is supposed to delegate a commitment, but its business contracts no longer allow a delegation, then the protocol will again be obstructed. We call these *protocol failures*, an agent's disability to fulfill its task affects other participants of the protocol. Overall, this failure is a jeopardy on the successful realization of the protocol.

We identify two classes of protocols based on how well they handle errors and show how these two classes are related.

7.1. Recoverable protocols

Definition 12. A protocol is *recoverable* if at least one agent can monitor a possible failure and take an action to put the protocol back in track.

For example, agent x delegates a commitment to agent z, which fails to discharge the commitment. If agent x can monitor this failure ands can delegate the task to another agent w, then the protocol recovers from the failure. Fig. 5 depicts this scenario.

The failures mentioned above are related to the agents that take place in the protocol. More specifically, the debtors of the commitments are being examined. To compute whether a protocol is recoverable, we again study the protocol's commitment graph.

Given the commitment graph G, we need to decide if the protocol represented by this graph is recoverable. Based on the definition of recoverable protocols, at least one agent should be able to monitor the commitments originating from it. The monitoring agents will be those that initially create the commitments. For example, in Fig. 4, for commitment C(P, M, result), the creator of the commitment P is responsible for making sure that the commitment gets done. If it delegates the commitment to a different participant $P2$ and if $P2$ fails, then P can follow an alternative path; i.e., discharge the commitment itself. This means that each commitment in G has two outgoing edges so that if one fails, the other edge can be used. This is equivalent to checking that

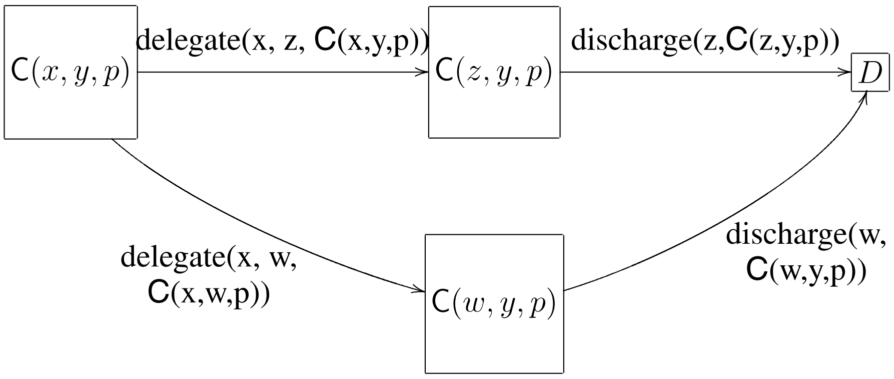

Fig. 5. An example recoverable protocol graph.

the graph does not have a cut-vertex; i.e., a vertex whose removal will increase the number of components in the graph.

7.2. Fault-tolerant protocols

Definition 13. A protocol is *fault tolerant* if any agent in the protocol can monitor a failure and correct the failure.

This is useful when each agent monitors its own commitments. Following the previous example, agent x delegates a commitment to agent z, which fails to discharge the commitment. Agent z can take an action for the commitment to be, say, delegated to another agent w (see Fig. 6).

Recoverable protocols are satisfactory in organizations where tasks are done in a hierarchy. The agents in the hierarchy constantly monitor, which tasks are done lower in the hierarchy. However, for more decentralized organizations, fault-tolerant protocols are more appropriate since each agent is responsible for correcting its own fault.

For a fault-tolerant protocol, each agent should be able to adjust its behavior. Hence, if one commitment operation fails, the agent should be able to perform a second operation; that is, alternatives should always exist. At first sight, it might seem like ensuring that each commitment has at least two operations would guarantee that the alternatives exist. However, consider the example in Fig. 7. Each commitment node has two

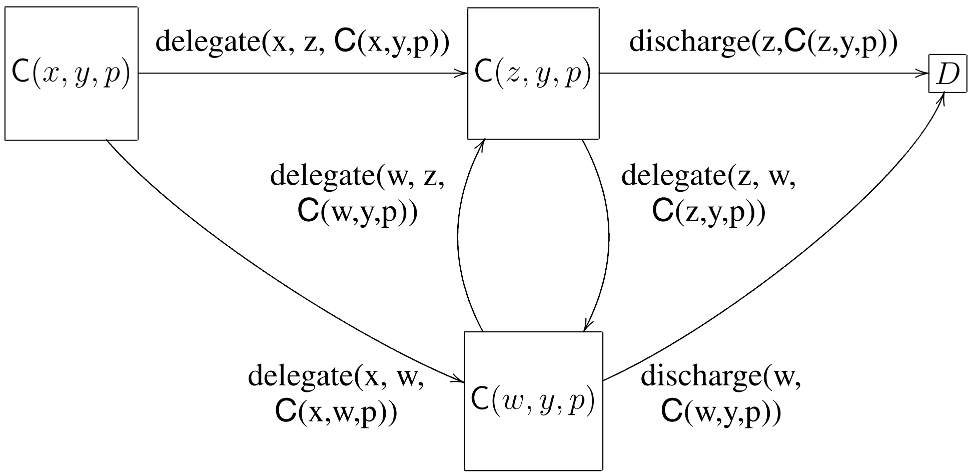

Fig. 6. An example commitment graph for a fault-tolerant protocol.

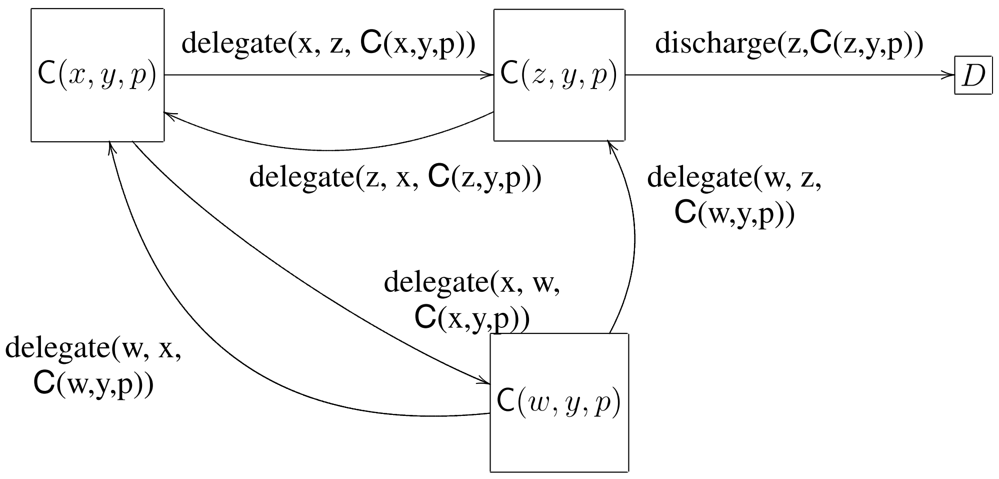

Fig. 7. An example 2-connected commitment graph that is not fault-tolerant.

outgoing edges; two operations defined on them. However, if agent *z* fails to discharge the commitment, then there is no alternative way to satisfy the existing commitment. Hence, it is not enough to check if a node has two outgoing edges, because even then removing one node (e.g., C(z,y,p)) can disconnect the graph. This yields us to refine our understanding of alternatives. Between any commitment node and the discharge node, removing one edge should not be enough to disconnect the graph; we should need at least two edges. By Menger's theorem [23], this corresponds to having at least two internally disjoint paths between any node and the discharge node.

Example 7. The CNP specification given in Example 2 is not fault-tolerant. Again, the set *CS* contains C(P, M, result) and C(M, P, (accepted $\vee$ rejected)). The first commitment can be discharged, canceled and delegated by the original creator as well as those to whom the commitment is delegated. Hence, the first commitment does not pose any problems for fault-tolerance. However, the second protocol does pose a problem. Only one operation (discharge) has been defined for the second commitment. This means that if the manager cannot fulfill the commitment, the protocol will not be able to proceed as planned.

Lemma 5. *If a protocol is fault-tolerant, then it is also recoverable.*

Proof. If a protocol is fault-tolerant, then all its commitment nodes have two internally disjoint paths to node *D*. These nodes include the commitment nodes that are linked from the start node *S*. By definition of internally disjoint paths, such a graph cannot have a cut-vertex, hence it is recoverable. □

8. Relevant literature

We review the recent literature with respect to our work. Fornara and Colombetti develop a method for agent communication, where the meanings of messages denote commitments [12]. In addition to base-level and conditional commitments, Fornara and Colombetti use precommitments to represent a request for a commitment from a second party. They model the life cycle of commitments in the system through update rules. However, they do not provide design requirements on effectiveness or consistency as we have done here. The requirements and algorithms developed here can easily be applied to their framework.

Artikis et al. develop a framework to specify and animate computational societies [2]. The specification of a society defines the social constraints, social roles, and social states. Social constraints define types of actions and the enforcement policies for these actions. A social state denotes the global state of a society based on the state of the environment, observable states of the involved agents, and states of the institutions. Our definition of a protocol state is similar to the global state of Artikis et al. The framework of Artikis et al. does not specify any design rules to establish the effectiveness of the executed societies. It would be interesting to apply the design ideas here to their setting where in addition there are social constraints.

Bauer, Müller, and Odell extend Unified Modeling Language (UML) into Agent UML to account for agent interactions [3]. Agent UML models agent interaction protocols as communication patterns and allows description of allowed sequences of messages in the protocol. Agent UML provides an intuitive pictorial representation for nested and interleaved protocols. However, Agent UML does not study design requirements of interaction protocols as we have done here.

Dignum et al. formalize interaction protocols within an organization through contracts [8]. They develop a language for specifying contracts that can capture various contracts and their deadlines. They use the interaction protocols to realize the objectives of the organization that the agents are situated in. However, they do not provide a methodology for analyzing contracts defined in that language as we have done here.

Alberti et al. specify interaction protocols using social integrity constraints [1]. Given a partial set of events that have happened, each agent computes a set of expectations based on the social integrity constraints; e.g., events that are expected to happen based on the given constraints. If an agent executes an event that does not respect an expectation, then it is assumed to have violated one of the social integrity constraints. We have studied the violating of commitments in richer time structure elsewhere [18]. Alberti et al. does not provide any design rules to ensure the effectiveness of their interaction protocols. Since the commitments and their operations are more flexible than the expectations defined by Alberti et al., our requirements can also be applied to their framework.

Endriss et al. study protocol conformance for interaction protocols that are defined as deterministic finite automaton (DFA) [9]. The set of transitions of a DFA are known *a priori*. If an agent always follows the transitions of the protocol, then it is compliant to the given protocol. Hence, the compliance checking can be viewed as verifying that the transitions of the protocol are followed correctly.

Flores and Kremer develop a commitment-based approach for designing conversation protocols [10]. The design phase includes steps for identifying agent roles, agent actions, and so on. However, the approach does not contain an automatic analysis phase where the effectiveness and consistency of the designed protocol can be checked. Flores and Kremer's treatment of protocol flexibility is in some respects similar to our understanding of robustness. In their work, flexibility is calculated in terms of unique message sequences that can be generated by a protocol. In our treatment, we consider unique paths to discharge commitments. Their flexibility metric can also be incorporated in a design tool as mentioned above.

Benatallah, Casati, and Toumani analyze Web service protocols from the standpoint of compatibility [4]. Their analysis is based on identifying similar, intersecting, and replaceable protocols. Similar to our approach, their analysis can be used at design-time. However, our focus in this paper has been analyzing protocols individually rather than comparing them with other protocols as Benatallah, Casati, and Toumani have done. In this respect, our analysis is complementary.

McBurney and Parsons propose posit spaces protocol to handle e-commerce transactions of agents [19]. The protocol consists of five locutions: propose, accept, delete, suggest_revoke, and ratify_revoke. The usage of propose and accept locution resembles the conditional commitments in commitment protocols. The delete locution corresponds to the release, or discharge operation. Suggest_revoke and ratify_revoke enable canceling of posits. McBurney and Parsons do not provide any design rules to develop posit space protocols as we have done here. The analysis constructed in this paper may be applied in the posit space framework.

9. Conclusion

This work derives some important design time requirements for commitment protocols. A commitment protocol should be effective in that it should make it possible for agents to reach desired end states by following the protocol. Based on this work, the effectiveness of a commitment protocol can be decided at design time, thereby allowing protocol designers to modify the protocol if necessary. Further, the actions that the agents take to follow the protocol should not carry the protocol to an inconsistent state. The consistency of protocols is closely related to possibility of actions generating conflicting information. The analysis of the commitments show that if all commitments can be released or canceled, then the protocol will always have a chance to stay at a consistent state. Finally, it is important to guarantee that the protocol is robust, such that if an agent's action fails, the protocol can still end correctly. To decide whether a protocol is robust or not, the alternative executions of a protocol need to be considered.

In our future work, we plan to study the relations between protocols with emphasis on similarity and equivalence. Such relations can be converted into metrics that can aid protocol designers in choosing alternative protocols for a given task.

Acknowledgement

This paper extends an earlier version that appeared in the 5th Joint International Conference on Autonomous Agents and Multiagent Systems (AAMAS-05) [24]. This research has been supported by Boğaziçi University Research Fund under grant BAP06A103 and The Scientific and Technological Research Council of Turkey by a CAREER Award under grant 105E073.

References

[1] M. Alberti, D. Daolio, P. Torroni, Specification and verification of agent interaction protocols in a logic-based system, in: Proceedings of the ACM Symposium on Applied Computing (SAC), ACM Press, 2004, pp. 72–78.
[2] A. Artikis, J. Pitt, M. Sergot, Animated specifications of computational societies, in: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), ACM Press, 2002, pp. 1053–1061, July.
[3] B. Bauer, J.P. Müller, J. Odell, Agent UML: a formalism for specifying multiagent interaction, in: P. Ciancarini, M. Wooldridge (Eds.), Agent-Oriented Software Engineering, Springer, Berlin, 2001, pp. 91–103.
[4] B. Benatallah, F. Casati, F. Toumani, Analysis and management of web service protocols, in: Conceptual Modeling – ER 2004, Lecture Notes in Computer Science, vol. 3288, 2004, pp. 524–541.
[5] J. Bentahar, B. Moulin, J.-J.C. Meyer, B. Chaib-draa, A logical model for commitment and argument network for agent communication, in: Proceedings of the 3rd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), ACM Press, 2004, pp. 792–799, July.
[6] C. Castelfranchi, Commitments: from individual intentions to groups and organizations, in: Proceedings of the International Conference on Multiagent Systems, 1995, pp. 41–48.
[7] T.H. Cormen, C.E. Leiserson, R. Rivest, Design and Analysis of Algorithms, MIT Press, 1990.
[8] V. Dignum, J.-J. Meyer, F. Dignum, H. Weigand, Formal specification of interaction in agent societies, in: 2nd Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS), Maryland, October, 2002.
[9] U. Endriss, N. Maudet, F. Sadri, F. Toni, Protocol conformance for logic-based agents, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann Publishers, 2003, pp. 679–684.
[10] R.A. Flores, R.C. Kremer, A principled modular approach to construct flexible conversation protocols, in: Canadian Conference on AI, Lecture Notes in Computer Science, vol. 3060, 2004, pp. 1–15.
[11] F. for Intelligent Physical Agents (FIPA), Contract net interaction protocol specification, 2002, Number 00029.
[12] N. Fornara, M. Colombetti, Operational specification of a commitment-based agent communication language, in: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), ACM Press, 2002, pp. 535–542.
[13] M.G. Gouda, Protocol verification made simple: a tutorial, Computer Networks and ISDN Systems 25 (1993) 969–980.
[14] G.J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, New Jersey, 1991.
[15] M.-P. Huget, J.-L. Koning, Requirement analysis for interaction protocols, Proceedings of the Central and Eastern European Conference on Multiagent Systems (CEEMAS), vol. LNAI 2691, Springer, Berlin, 2003, pp. 404–412.
[16] N.R. Jennings, On agent-based software engineering, Artificial Intelligence 177 (2) (2000) 277–296.
[17] R. Kowalski, M.J. Sergot, A logic-based calculus of events, New Generation Computing 4 (1) (1986) 67–95.
[18] A.U. Mallya, P. Yolum, M.P. Singh, Resolving commitments among autonomous agents, in: M.-P. Huget, F. Dignum (Eds.), Proceedings of the AAMAS Workshop on Agent Communication Languages and Conversation Policies, LNAI, vol. 2922, Springer, Berlin, 2003, pp. 166–182.
[19] P. McBurney, S. Parsons, Posit spaces: a performative model of e-commerce, in: Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), ACM Press, 2003, pp. 624–631.
[20] M. Shanahan, Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia, MIT Press, Cambridge, 1997.
[21] M.P. Singh, An ontology for commitments in multiagent systems: toward a unification of normative concepts, Artificial Intelligence and Law 7 (1999) 97–113.
[22] M. Venkatraman, M.P. Singh, Verifying compliance with commitment protocols: enabling open Web-based multiagent systems, Autonomous Agents and Multi-Agent Systems 2 (3) (1999) 217–236.
[23] D.B. West, Introduction to Graph Theory, second ed., Prentice-Hall, Upper Saddle River, NJ, 2001.
[24] P. Yolum, Towards design tools for protocol development, in: Proceedings of the 4th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), ACM Press, 2005, pp. 99–105.

[25] P. Yolum, M.P. Singh, Flexible protocol specification and execution: applying event calculus planning using commitments, in: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), ACM Press, 2002, pp. 527–534.
[26] P. Yolum, M.P. Singh, Reasoning about commitments in the event calculus: an approach for specifying and executing protocols, Annals of Mathematics and Artificial Intelligence 42 (1–3) (2004) 227–253.

Pınar Yolum is an assistant professor at Department of Computer Engineering in Boğaziçi University, Istanbul. She received her Ph.D. and M.S. in computer science from North Carolina State University in 2003 and 2000, respectively, and her B.Sc. in computer engineering from Marmara University, Istanbul in 1998. She worked as a post-doctoral researcher in the Free University of Amsterdam. Her research interests include multiagent systems and service-oriented computing.