# Detecting Exceptions in Commitment Protocols: Discovering Hidden States

Özgür Kafalı
Department of Computer Engineering
Boğaziçi University
TR-34342, Bebek, İstanbul, Turkey
e-mail: ozgurkafali@gmail.com

Pınar Yolum
Department of Computer Engineering
Boğaziçi University
TR-34342, Bebek, İstanbul, Turkey
e-mail: pinar.yolum@boun.edu.tr

***Abstract*—Open multiagent systems consist of autonomous agents that are built by different vendors. In principle, open multiagent systems cannot provide any guarantees about the behaviors of their agents. This means that when agents are working together, such as carrying out a business protocol, one agent's misbehavior may potentially create an exception for another agent and obstruct its proper working. Faced with such an exception, an agent should be able to identify the problem by verifying the compliance of other agents.**

**Previous work on verification of protocols unrealistically assume that participants have full knowledge of a protocol. However, when multiple agents enact a protocol, each agent has access to its part of the protocol and not more. This will require agents to check verification by querying others and more importantly by discovering the contracts between them. Here, we propose a commitment-based framework for detecting exceptions in which an agent augments its part of the protocol with its knowledge to construct states that are previously hidden to the agent by generating possible commitments between other agents. The agent then queries others to confirm those states. Our framework is formalized using C+ and is tested using a realistic business scenario.**

## I. Introduction

In open multiagent systems, it is possible for agents to interact with others that they have no previous knowledge of. Carrying out interactions, such as business dealings, with such others is difficult since there is no guarantee about how the other agents will act. If others do not follow their parts of the interactions, the entire business may be jeopardized. This requires an agent participating in such a situation to be able to verify that others are acting by the rules.

Verification is especially important in the face of *exceptions*. Here, we deal with high-level exceptions that pertain to the workings of the underlying protocol. For example, if a buyer does not receive a merchandise that was scheduled for delivery, it can conclude that there must have been an exception in the workings of the entire protocol. When such an exception occurs, the agent facing the exception needs to identify the problem behind it. This is a two-phase procedure; first detecting the exception, and then taking proper action recover from the exceptional situation. In this paper, we focus on the first phase. That is, we propose an algorithm for finding the source of exceptions (i.e., caused by which parties and why). In addition, if the source of the exception is identified correctly, then it is a forward step in the recovery process, because the agent facing the exception has a means of proof for the cause of it. This proof can then be used to consult other authorities, which can resolve inconsistencies between parties.

Realistic business affairs consist of multiple parties that carry out different tasks. Multiparty interactions have two inherent properties; (1) interactions between different parties are regulated by different contracts (i.e., a seller may exercise different rules when dealing with an individual versus a corporation), (2) rules of interaction between different parties are private and not revealed to the outside world (i.e., a contract between a seller and a carrier may never be revealed to buyers publicly). While these properties are essential for multiparty protocols, they pose important challenges for verification, which brings the question of how an agent can verify others' compliance when it has only partial information about their activities. We use the scenario in Example 1 throughout the paper as our running example.

**Example 1.** Consider the simple purchase-and-delivery protocol that includes three business roles. The roles in the protocol are *customer*, *bookstore*, and *deliverer*. In a normal execution, the customer buys a book from the bookstore and the deliverer delivers the book to the customer. However, certain exceptions may occur during the enactment of this protocol. For example, consider the case where the customer pays for the book and expects delivery in three days. In addition, suppose that the bookstore sends books to be delivered to the deliverer in large groups. If at the time the customer buys the book, the number of books pending for delivery at the bookstore is not enough, the book will not be delivered causing an exception for the customer. However, since the customer does not know the details of the contract between the bookstore and the deliverer, the source of the exception is not immediately clear to the customer. One option for the customer is to simply ask the bookstore about the cause of exception. However, this may not be possible in some situations (i.e., the bookstore is not willing to share information regarding its contracts with other parties, or the exception is caused by a party beyond the knowledge of the bookstore). Then, the customer has to use its knowledge first to predict possible causes, and query corresponding agents

to determine which one is the actual cause of the exception.

In order to study verification rigorously, we capture agents' interactions through commitments [1], and adopt C+ as a language to formalize those interactions [2], [3]. In contrast to previous work on verification, we propose a realistic exception discovery framework in which; (1) multiple roles exist in the business, (2) business scenarios are distributed (each role has its own view of the protocol), and (3) each agent deals with an exception by discovering contracts of other agents. With this proposed approach, an agent only finds out the necessary details to continue its operation in tracing down the incompliant agents.

The rest of this paper is organized as follows. Section II gives necessary background on protocols, commitments and C+. Section III describes the running example and defines the problem formally. Section IV introduces our solution to deal with exceptions in distributed scenarios, and Section V explains its details. Section VI presents a discussion of our work with comparisons to the literature and provides directions for further research.

## II. Technical Background

In this section, we first describe formally what we mean by a business protocol, then we review the necessary concepts related to specifying commitments, and realizing them in a formal description language.

### A. Protocols & Runs

**Definition II.1.** A protocol $\mathcal{P}$ is a 6-tuple $\langle \mathbb{S}, \mathbb{A}, \mathbb{C}, \mathbb{R}, \mathcal{S}_\mathcal{I}, \mathcal{S}_\mathcal{F} \rangle$, such that $\mathbb{S}$ is a finite set of states, $\mathbb{A}$ is a finite set of actions, $\mathbb{C}$ is a finite set of conditions, $\mathbb{R}$ is a finite set of roles, $\mathcal{S}_\mathcal{I}$ is the set of initial states ($\mathcal{S}_\mathcal{I} \subset \mathbb{S}$), and $\mathcal{S}_\mathcal{F}$ is the set of final states ($\mathcal{S}_\mathcal{F} \subset \mathbb{S}$). Intermediate (middle) states $\mathcal{S}_\mathcal{M}$ are states that are not in $\mathcal{S}_\mathcal{I}$ or in $\mathcal{S}_\mathcal{F}$.

**Definition II.2.** A state is a set of conditions and commitments that hold in it.

**Definition II.3.** A run $\mathcal{R}$ of a protocol $\mathcal{P}$ is simply a sequence of states $\langle S_0, ..., S_n \rangle$ starting from an initial state ($S_0 \in \mathcal{S}_\mathcal{I}$). For now, we consider only finite runs.

**Definition II.4.** A desirable run is the one that ends in a final state ($S_n \in \mathcal{S}_\mathcal{F}$).

**Definition II.5.** An exceptional run is the one that ends in an intermediate state ($S_n \in \mathcal{S}_\mathcal{M}$), and thus does not reach a final state.

Desirable runs are preferred by agents since they lead them to reach their goals, whereas exceptional runs are unexpected by agents and proper action (i.e., exception handling routines) has to be taken in order for the protocol to evolve from those states.

**Definition II.6.** An agent-centric sub-protocol $\mathcal{P}'\langle \mathbb{S}', \mathbb{A}', \mathbb{C}', \mathbb{R}', \mathcal{S}'_\mathcal{I}, \mathcal{S}'_\mathcal{F} \rangle$ is a subset of the main protocol $\mathcal{P}\langle \mathbb{S}, \mathbb{A}, \mathbb{C}, \mathbb{R}, \mathcal{S}_\mathcal{I}, \mathcal{S}_\mathcal{F} \rangle$ in which; (1) $\forall s' \in \mathbb{S}', \exists s \in \mathbb{S} : s' \subseteq s$, (2) $\mathbb{A}' \subseteq \mathbb{A}$, (3) $\mathbb{C}' \subseteq \mathbb{C}$, (4) $\mathbb{R}' \subseteq \mathbb{R}$, (5) $\forall s' \in \mathcal{S}'_\mathcal{I}, \exists s \in \mathcal{S}_\mathcal{I} : s' \subseteq s$, (6) $\forall s' \in \mathcal{S}'_\mathcal{F}, \exists s \in \mathcal{S}_\mathcal{F} : s' \subseteq s$.

### B. Commitments

Commitments are formed between two agents and roughly correspond to obligations [1]. The debtor of a commitment is the agent that is committed to bring about a condition. The creditor benefits from the commitment. Commitments are created and discharged by the interactions of the agents. There are two types of commitments:

**c(x, y, p):** This is a base-level commitment between debtor *x* and creditor *y* with proposition *p*. When this commitment is in charge, debtor *x* becomes committed to creditor *y* for satisfying *p*.

**cc(x, y, p, q):** This is a conditional commitment between debtor *x* and creditor *y* with condition *p* and proposition *q*. When this commitment is in charge, if *p* is satisfied (by *y*), *x* will become committed to *y* for satisfying *q*.

The following four operations describe how commitments are manipulated throughout a protocol. We assume that each protocol action initiates a commitment operation (i.e., altering a contract between agents). Thus, commitment operations describe the semantics of protocol actions.

**create(x, c(x, y, p)):** This operation initiates the creation of the base-level commitment *c*. It is performed by *x*, the debtor of the commitment. Since this operation creates a new commitment which does not hold previously, it causes a state transition ($S_i \xrightarrow{create(x,c(x,y,p))} S_i \cup \{c(x,y,p)\}$).

**ccreate(x, cc(x, y, p, q)):** This operation initiates the creation of the conditional commitment *cc*. It is performed by *x*, the debtor of the commitment. This operation also causes a state transition ($S_i \xrightarrow{ccreate(x,cc(x,y,p,q))} S_i \cup \{cc(x,y,p,q)\}$).

**discharge(x, c(x, y, p):** This operation resolves the base-level commitment *c*. It is performed by *x*, the debtor of the commitment, and the commitment *c* is terminated afterward. A base-level commitment is resolved when the proposition *p* of the commitment becomes true. This operation causes a state transition since a previously holding commitment disappears ($S_i \xrightarrow{discharge(x,c(x,y,p))} S_i - \{c(x,y,p)\} \cup \{p\}$).

**cdischarge(x, cc(x, y, p, q)):** This operation resolves the conditional commitment *cc*. It is performed by *x*, the debtor of the commitment, and the conditional commitment *cc* is terminated afterward. If the proposition *q* of a conditional commitment *cc* becomes true, then *cc* is discharged immediately causing a state transition ($S_i \xrightarrow{cdischarge(x,cc(x,y,p,q))} S_i - \{cc(x,y,p,q)\} \cup \{q\}$). If the condition *p* of *cc* is brought about, then *cc* is discharged, and a new base-level commitment is created with the proposition *q* of *cc* causing another state transition ($S_i \xrightarrow{cdischarge(x,cc(x,y,p,q))} S_i - \{cc(x,y,p,q)\} \cup \{c(x,y,q)\} \cup \{p\}$).

### C. Commitment Protocols

In this section, we integrate commitments into protocols. Definitions II.7, II.8, and II.9 provide useful properties re-

garding states with respect to commitments.

**Definition II.7.** A state is inconsistent if it is one of the following; (1) state $S_i = \{cc(x, y, p, q), p\}$ is inconsistent since the conditional commitment cannot coexist with its condition, (2) state $S_j = \{p, \neg p\}$ is inconsistent since a condition cannot coexist with its negation.

**Definition II.8.** Two states are equivalent with respect to an agent if they share the same conditions and commitments regarding that agent.

**Example 2.** Let $S_i = \{cc(x, y, p, q), r\}$ and $S_j = \{cc(x, y, p, q), cc(y, z, v, w), r, u\}$ be two states, and assume $r$ is a condition that agent $x$ can bring about (but does not affect agent $y$'s working) and $u$ is a condition that agent $y$ can bring about (but does not affect agent $x$'s working). Then, $S_i$ and $S_j$ are equivalent states for agent $x$ (since the commitment $cc(y, z, v, w)$ is irrelevant to agent $x$), but not equivalent states for agent $y$ (since the commitment $cc(y, z, v, w)$ that is related to agent $y$ does not hold in state $S_i$, but holds in state $S_j$).

**Definition II.9.** The distance between two states $S_i$ and $S_j$ is the number of commitment operations that are required to bring the protocol from state $S_i$ to state $S_j$.

**Example 3.** Let $S_i = \{cc(x, y, p, q)\}$ and $S_j = \{c(x, y, q), cc(y, z, q, r), p\}$ be two states. Then, the distance between states $S_i$ and $S_j$ is 2 since it takes two commitment operations to go from state $S_i$ to $S_j$; a *ccreate* operation to create *cc(y, z, q, r)*, and a *cdischarge* operation to resolve *cc(x, y, p , q)* into *c(x, y, q)*.

### *D. The Action Description Language C+*

We realize the business scenarios to be described using commitment protocols specified in the action description language, C+ [2], [3]. A protocol in C+ is composed of a set of states and transitions between states (i.e., a transition system). A state may contain several fluents that hold in that state (true propositions). A fluent's value is changed as the consequence of an action that is performed by an agent. An inertial fluent is the one whose value is not changed until an action makes it change. Our use of C+ for formalizing commitments and their operations are based on that of Chopra and Singh [3].

Listing 1 shows how commitment operations are realized in C+. This is a basis for other protocol specifications that utilize commitments. Through lines 10-14, commitments and conditional commitments are modeled as inertial fluents. Commitment operations shown through lines 17-22 are modeled as auxiliary (i.e., simple) actions. An auxiliary action has to be initiated by a protocol action and cannot be performed independently. The causation rules associated with those operations are shown through lines 25-32.

## III. Business Scenario

In order to show how commitments are utilized in real business environments, we describe in detail our running example that represents concrete business interactions. Figure 1 describes the purchase protocol introduced in Section I. There are four states ($S_0$, $S_1$, $S_2$, and $S_3$), three actions that enable the transitions between the states (*sendPayment*, *sellBook*, and *deliverBook*), and three conditions corresponding to the outcomes of the actions in the protocol (*payc*, *bookc*, and *deliverc*). There is a single initial state ($S_0$), a single final state ($S_3$), and two intermediate states ($S_1$ and $S_2$).

```
:- sorts                                                                    1
   role;                                                                    2
   condition.                                                               3

:- variables                                                                5
   x, y, z ::role;                                                          6
   p, q ::condition.                                                        7

% Declaration of commitments                                               9
:- constants                                                                10
   commitment(role,role,condition)                                          11
    ::inertialFluent;                                                       12
   ccommitment(role,role,condition,condition)                               13
    ::inertialFluent;                                                       14

% Commitment operations                                                     16
   create(role,role,condition) ::action;                                    17
   discharge(role,role,condition) ::action;                                 18
   ccreate(role,role,condition,condition)                                   19
    ::action;                                                               20
   cdischarge(role,role,condition,condition)                                21
    ::action;                                                               22

% Commitment rules                                                          24
   create(x,y,p) causes commitment(x,y,p)                                   25
    where x<>y.                                                             26
   discharge(x,y,p) causes -commitment(x,y,p)                               27
    where x<>y.                                                             28
   ccreate(x,y,p,q) causes ccommitment(x,y,p,q)                             29
    where x<>y & p<>q.                                                      30
   cdischarge(x,y,p,q) causes -ccommitment(x,y,p,q)                         31
    & commitment(x,y,q) where x<>y & p<>q.                                  32
```

Listing 1. Commitment Operations in C+

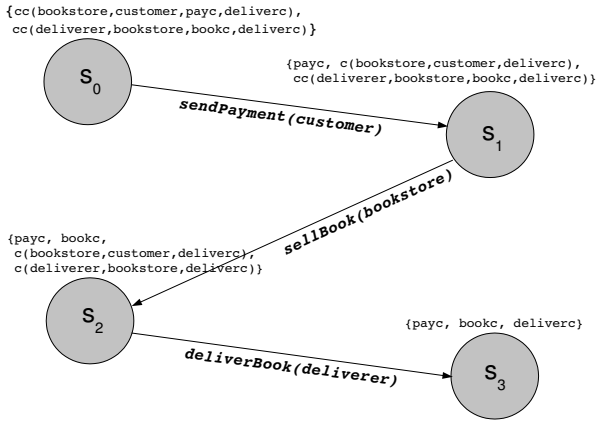


Fig. 1. Purchase & Delivery Protocol

No conditions initially hold in $S_0$, but two conditional commitments are present. The first commitment *cc(bookstore, customer, payc, deliverc)* means that the bookstore is committed to make sure that the book is delivered if the customer pays for it. The second commitment *cc(deliverer, bookstore,*

*bookc, deliverc)* means that the deliverer is committed to deliver the book to the customer if the bookstore sends it. Since the customer's goal is to get the book delivered, it performs the *sendPayment* action. This brings the protocol to state $S_1$ where condition *payc* holds as the outcome of the *sendPayment* action. Also, the conditional commitment *cc(bookstore, customer, payc, deliverc)* is discharged to the base-level commitment *c(bookstore, customer, deliverc)*. Next, the bookstore performs the *sellBook* action which brings the protocol to state $S_2$. Accordingly, condition *bookc* holds and the conditional commitment *c(deliverer, bookstore, bookc, deliverc)* is discharged to the base-level commitment *c(deliverer, bookstore, deliverc)*. Finally, the deliverer performs the *deliverBook* action which brings the protocol to state $S_3$. Both commitments in $S_2$ are discharged and condition *deliverc* holds in $S_3$. State $S_3$ is the final state for the protocol since all three conditions hold at the same time. Thus, a desirable run for the protocol is $\langle S_0, S_1, S_2, S_3 \rangle$.

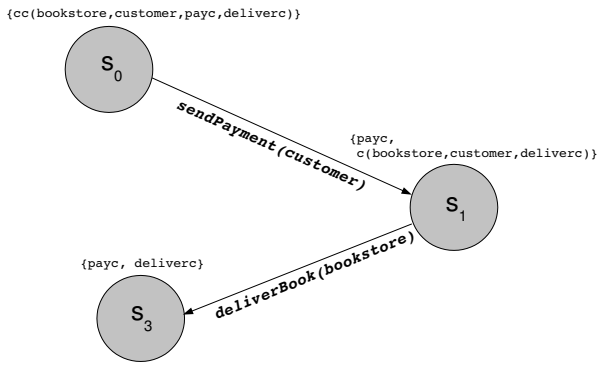


Fig. 2. $\mathcal{P}_{Customer}$ Sub-Protocol

Note that states $S_1$ and $S_2$ are equivalent states for the customer, because condition *payc* and commitment *c(bookstore, customer, deliverc)* hold in both states. In addition, condition *bookc* and the discharged commitment *cc(deliverer, bookstore, bookc, deliverc)* are irrelevant to the customer. Thus, the customer's sub-protocol $\mathcal{P}_{Customer}$ includes states $S_0$, $S_1$, and $S_3$ as shown in Figure 2. $S_0$ is the initial state, $S_3$ is the final state, and $S_1$ is the only intermediate state for this sub-protocol. There are two actions (*sendPayment* and *deliverBook*) and two conditions (*payc* and *deliverc*).

Listing 2 describes part of the customer's protocol in C+. Line 2 includes the commitment operations as introduced in Listing 1. Lines 4-6 define the roles and conditions that are involved in the protocol. Lines 9-11 define the fluents representing the messages that hold in certain states of the protocol. For example, the message *pay(customer,bookstore)* has the meaning that the customer has paid the bookstore for the book. The fluents in line 12 define the initial and final conditions for the protocol. The protocol actions are defined through lines 15-17. The *initiate* action is performed by the role *super* to initialize the conditional commitments between the parties (i.e., *super* can be considered as a protocol designer). Certain actions cannot be performed by some agents. As line 20 suggests, the *sendPayment* action cannot be performed by the bookstore.

```
% Include the commitment operations                                  1
:-include 'com-spec'.                                               2

:- objects                                                          4
  super, customer, bookstore, deliverer ::role;                     5
  payc, bookc, deliverc ::condition.                                6

% Fluents that define the states of the protocol                    8
:- constants                                                        9
  init(role), pay(role,role), book(role,role),                      10
   deliver(role,role) ::inertialFluent;                             11
  initial,final ::sdFluent.                                         12

% Protocol actions                                                  14
  initiate(role), sendPayment(role),                                15
   sellBook(role), deliverBook(role)                                16
   ::exogenousAction;                                               17

% Certain actions are done by specific roles only                  19
  nonexecutable sendPayment(bookstore).                             20
  ...                                                               21

% Protocol action sendPayment is visible to the                     23
% customer agent                                                    24
  sendPayment(customer) causes                                      25
   pay(customer,bookstore) if                                       26
   ccommitment(bookstore,customer,payc,deliverc).                   27
  sendPayment(customer) causes                                      28
   discharge(customer,bookstore,payc)                               29
   if commitment(customer,bookstore,payc).                          30
  sendPayment(customer) causes                                      31
   cdischarge(bookstore,customer,payc,deliverc) if                  32
   ccommitment(bookstore,customer,payc,deliverc).                   33
  nonexecutable sendPayment(customer)                               34
   if pay(customer,bookstore) ++ -init(super).                      35

% Other protocol actions are not visible to the                     37
% customer agent                                                    38
  ...                                                               39

% Causation relations for initial and final states                  41
  caused initial if initial.                                        42
  caused -initial if pay(x,y).                                      43
  ...                                                               44
  caused -initial if ccommitment(x,y,p,q).                          45
  caused -final if -final.                                          46

% In final state, if pay, book, and deliver holds                   48
  caused final if pay(customer,bookstore) &                         49
   book(bookstore,deliverer) &                                      50
   deliver(deliverer,customer).                                     51
  ...                                                               52
```

Listing 2. Customer's Protocol Described in C+

The rules for the protocol action *sendPayment* are given through lines 25-35. The first rule tells that the fluent *pay(customer, bookstore)* will start to hold as a result of the protocol action *sendPayment(customer)* if the conditional commitment *cc(bookstore, customer, payc, deliverc)* exists prior to it (lines 25-27). The next two rules through lines 28-33 describe how existing commitments are resolved and new commitments are created as a result of the same action. The

last rule ensures that the action *sendPayment(customer)* is not performed if the payment is already made by the customer or the protocol is not initialized yet by *super* (lines 34-35). Since the scenario is distributed, other protocol actions, such as *sellBook* or *deliverBook*, are not accessible (i.e., hidden) by the customer agent. The protocol starts with the state *initial* and is expected to terminate in state *final* (lines 42-46), if the required fluents hold (lines 49-51).

Now, let us study what can go wrong in a given protocol run and what exceptions can take place. If an expected action is not performed by an agent that is responsible for performing it, an exception occurs. Two such exceptional runs for this protocol are $\langle S_0, S_1 \rangle$ and $\langle S_0, S_1, S_2 \rangle$. The former run gets stuck at state $S_1$, because the bookstore does not send the book to the deliverer. The latter run gets stuck at state $S_2$, because the deliverer does not deliver the book to the customer. When one of these exceptions occur, the customer agent cannot find its cause immediately (i.e., in which of the main protocol states the run gets stuck) since states $S_1$ and $S_2$ are equivalent for it. However, in order for the customer to deal with the exception, it is crucial that it learns about which agent is causing the exception. Next, we look at the general idea behind our proposed solution, and then explain the details of our approach.

## IV. Proposed Solution

When faced with an exception, an agent tries to figure out what might have gone wrong. Figure 3 summarizes the approach that agents utilize when detecting exceptions. First, the agent reasons using its own knowledge-base. In many cases, this would not be enough to identify the exception. However, in many settings, as time evolves, new information about the environment becomes available (step 1). Based on the new information, the agent again tries to predict possible contracts between other agents so that it can figure out what has been violated to cause an exception (step 2). Once the agent has possible ideas about what might have gone wrong, it queries other agents that are related to the possible cause of the exception and asks them to confirm one of the possibilities (step 3).

For the above example, this would work as follows: At the beginning, note that the customer is not aware of the existence of a deliverer since its sub-protocol does not include such a role. Thus, its knowledge base includes only the bookstore other than itself. In addition, the conditions initially known by the agent are limited to *payc* and its goal condition *deliverc*. With this information only, it is not possible to construct state $S_2$ since it involves a commitment between the bookstore and the deliverer. However, even if its knowledge base does not contain that information, the customer agent may become aware of other roles, and extend its sub-protocol with new information revealed by other agents. For example, if the bookstore announces that the book is sent to the deliverer, then the customer will be aware of the existence of a deliverer role and the condition *bookc*. Information exposure is a simple task that is often performed in real-life delivery scenarios. Now, the

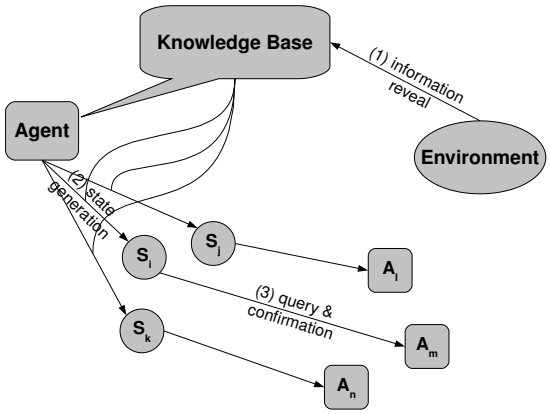


Fig. 3. General Approach

agent has enough knowledge to generate other possible states of the protocol. Once the states are generated, they need to be verified to find out whether they have caused the exception. Accordingly, the agent directs the query about each generated state to one of the agents related to that state (i.e., involved in a commitment within that state).

## V. Identifying Exception Sources

When an exception takes place, it is necessary for the agent to identify who caused the exception so that it can deal with the exception accordingly. As seen in the previous section, this is not easy since an agent may view a number of states identical when indeed they are different for other agents. The question then is how can an agent construct possible real states of the world? If the agent can generate such possibilities, then it can query the involved agents and ask them to confirm one of these states. Next, we present such an algorithm. Without loss of generality, we assume that the algorithm is used by the *customer* agent.

### A. State Prediction Algorithm

In this section, we propose an algorithm for the agents to use for constructing the hidden states (i.e., unknown states prior to exception) that might be the cause of exceptions. In order to construct a state, the agent has to generate the possible conditions and commitments that hold in that state. Recall that each agent is only aware of the commitments it is involved in. So, the agent has to predict the possible commitments among other participants to fill the definition of a hidden state.

Algorithm 1 describes how the agent predicts the hidden states for detecting exceptions. The requirements for the algorithm to execute properly are; current state of the agent in its sub-protocol, its goal condition, commitments it is involved in, conditions and roles it is aware of, and a maximum allowed distance parameter for selecting states to query. The algorithm consists of two stages; state generation and state selection, that we describe next.

**State Generation Stage:** This stage starts with creating a set

**Algorithm 1** Predicting Hidden States

**Require:** $S_c$ {current state}
**Require:** $\mathcal{C}_{goal}$ {goal condition}
**Require:** $commitments$ {initial commitments}
**Require:** $\mathbb{C}$ {conditions whose existence are known}
**Require:** $\mathbb{R}$ {roles whose existence are known}
**Require:** $dist$ {maximum allowed distance}
{I. State Generation Stage}
1: $\mathbb{S} \leftarrow \{S_c\}$ {add current state to the generated states}
2: **for all** $commitment_i$ in $commitments$ **do**
3: $S \leftarrow S_c$ {create a new state from current state}
4: $cc \leftarrow cc(GoalRole, CondRole, Cond, Goal)$
5: $Goal \leftarrow \mathcal{C}_{goal}$ {replace goal condition}
6: $GoalRole \leftarrow select(\mathbb{R})$ {pick a role}
7: generate $Cond$ and $CondRole$ using $commitment_i$
8: $cond \leftarrow select(\mathbb{C})$ {pick a set of conditions}
9: apply commitment operations on $cc$ assuming conditions in $cond$ holds
10: $S \leftarrow S \cup cc$ {add the commitments to the state}
11: $S \leftarrow S \cup cond$ {add the set of conditions to the state}
12: $\mathbb{S} \leftarrow \mathbb{S} \cup \{S\}$ {add the generated state to the result}
13: **end for**
{II. State Selection Stage}
14: **for all** $S_i$ in $\mathbb{S}$ **do**
15: **if** distance($S_i$,$S_c$) $>$ dist **then**
16: $\mathbb{S} \leftarrow \mathbb{S} - \{S_i\}$ {remove the state from the result}
17: **end if**
18: **end for**
19: **return** $\mathbb{S}$

for storing generated states and the current state of the agent is added to this set (line 1). A generated state is not constructed from scratch, but rather extended from the current state of the agent (line 3). In order to fill the state definition, the agent generates the hidden commitments between other parties starting with a conditional commitment template with two roles and two conditions (line 4). The goal condition for the commitment ($Goal$) is the agent's goal (line 5), and the business party that can bring about that condition ($GoalRole$) is picked from the set of roles the agent is aware of (line 6). In order to fill the middle parts of the commitment ($Cond$ and $CondRole$), the agent traces through its own commitments and finds which parties it has a commitment with. For each commitment $cc(x, Role, Cond, p)$ or $cc(Role, x, p, Cond)$, where $Role$ is the agent's role and $Cond$ is one of the conditions that the agent can bring about, it replaces $CondRole$ and $Cond$ of the template commitment using all possible pairs of $x$ and $p$ as line 7 suggests. The agent then searches for conditions to put into the state definition (line 8). Those conditions are also used in applying commitment operations on the generated commitments. Since the generated commitments are the initial versions of contracts between other parties, they might have been changed during the execution of the protocol. Line 9 of the algorithm provides this commitment manipulation process. Note that no inconsistent states are generated at this stage of the algorithm, because this process resolves necessary commitments with conditions whenever is possible. The state is then ready to be extended with the generated commitments and conditions (lines 10-11). Finally, the state is added to the set of generated states (line 12). This stage continues until no new states are generated.

**State Selection Stage:** This stage eliminates states generated by the first stage of the algorithm which are at a distance from the current state of the agent's sub-protocol. We apply the state distance property to compute the distance value. The maximum allowed distance for selection is a configurable parameter of the algorithm controlled by the *dist* value in line 15. The number of states selected out of this stage is expected to decrease if we select this parameter to be low. However, it increases the chance that the actual exceptional state is also eliminated by this process.

**Example 4.** Let us now depict the algorithm using our scenario. Recall that we've considered two exceptional situations; one gets stuck at state $S_1$, and the other gets stuck at state $S_2$. However, since $S_2$ is a hidden state for the customer agent, both $S_1$ and $S_2$ converge to state $S_1$ of the customer agent's sub-protocol. At this point, the customer agent thinks that the exception is caused by the bookstore since the delivery will be done by the bookstore according to its sub-protocol. But, suppose that the bookstore agent informs the customer agent on the delivery process. That is, it tells that the book is given to the deliverer agent. Now, the customer agent has extra knowledge with which it can extend its sub-protocol. Now, the customer agent can initiate the state generation process. The goal of the agent is to successfully generate state $S_2$ and query agents related to that state (i.e., deliverer) to see whether the main protocol is actually in that state. If so, the exception is caused by the deliverer agent, otherwise the bookstore agent is the cause of the exception. Now, suppose the agent has generated several different states among which one of them is the state $S_2$. To learn whether the main protocol is in state $S_2$, the customer agent queries the deliverer agent to confirm the existence of state $S_2$.

### *B. Implementation & Evaluation*

In order to implement our approach, we used C+ to describe the scenario formally as shown partly in Listings 1 and 2, then implemented the state prediction algorithm in Java. In the trivial cases where the initial commitments between the parties are in force, the protocols terminate as desired for the customer agent. However, since our aim is to observe exceptional situations, we disrupt the C+ descriptions of the scenarios (i.e., remove certain commitments) to enable the occurrence of such exceptions. Once certain parts of the scenario descriptions are extracted, the prediction algorithm is run to generate the possible missing states. Finally, one or more generations complete the scenario descriptions as they should be, leading to a desirable run for the customer agent.

The algorithm can be extended to support sequential protocols that involve more than one agent between the initiator (i.e., customer) and the terminator of the protocol (i.e., deliverer). For example, consider an extension to our scenario where books are packaged before they are sent for delivery. This packaging process needs another role, the *packager*, to be present in the protocol. Thus, the algorithm has to generate two conditional commitments instead of one for each state it will generate, involving the contracts between; (1) the bookstore and the packager, and (2) the packager and the deliverer. Our current system supports these extensions.

**Correctness of the Algorithm:** Here, we discuss the two stages of the algorithm (state generation and state selection) in order to argue on the correctness of our algorithm. That is, the state causing the exception has to be generated in the state generation stage, and it has to be selected as a candidate for querying in the state selection stage. Next, we consider each stage separately:

**State Generation Stage:** The number of states generated is limited to the knowledge of the agent about the protocol (i.e., the roles and conditions).

**Lemma V.1.** *Let $S_e$ be a state in protocol $P$, and let $c$ be a customer agent executing in $P$. Assume $c$ is currently in state $S_c$ of its sub-protocol $P_c$ ($P_c \subseteq P$), and assume state $S_e$ differs from state $S_c$ in terms of the set of conditions $E_{cnd}$ and the set of commitments $E_{cmm}$. Let the commitments in $E_{cmm}$ include the set of conditions $E_{con}$ and the set of roles $E_{role}$, and let $E_{cond} = E_{cnd} \cup E_{con}$. Now, if $c$ faces an exception caused at state $S_e$, and if $c$ knows about the conditions in $E_{cond}$ and the roles in $E_{role}$, then state $S_e$ will be generated by agent $c$.*

*Proof:* Recall that agent $c$ generates a state by filling its definition with conditions and commitments. Agent $c$ tries all possible combinations of conditions and commitments it can generate using its knowledge about $P_c$. Note that $S_e = S_c \cup E_{cnd} \cup E_{cmm}$, thus agent $c$ needs to generate the conditions in $E_{cnd}$ and the commitments in $E_{cmm}$. Using its knowledge about $E_{cond}$, agent $c$ can generate the conditions in $E_{cnd}$, since $E_{cnd} \subseteq E_{cond}$. Using its knowledge about $E_{cond}$ and $E_{role}$, agent $c$ can generate the commitments in $E_{cmm}$, since those commitments are composed of the conditions in $E_{con}$ and the roles in $E_{role}$ which are known by agent $c$ ($E_{con} \subseteq E_{cond}$). Thus, agent $c$ generates state $S_e$. ■

**State Selection Stage:** The chance of the exceptional state being selected is related to how distant it is from the current state of the agent's sub-protocol, and the choice of the maximum allowed distance parameter used for deciding whether two states are distant.

## VI. Discussion

Commitment protocols have been used before to formalize business scenarios [4], [5]. Chopra and Singh explain how transformation specifications are used in order to extend protocols to cover new situations [3]. Their formalization of commitment protocols in C+ form the basis of our work. However, Chopra and Singh do not provide mechanisms for distributed verification as we have done here.

Mallya and Singh divide exceptions into two categories [6]; (1) expected exceptions which are handled at design-time using preferences over protocol runs, and (2) unexpected exceptions which occur at run-time and are handled via splicing exception handlers with base protocols. Their work helps protocol designers for handling exceptions. However, handling unexpected exceptions with such generic handlers is costly. The work of Venkatraman and Singh resembles our work since each agent checks compliance on its own [7]. The process is distributed in a sense that each agent has access to its own set of messages during execution, but their business scenario does not fully simulate a distributed environment. Our work differs from theirs since an agent in our scenario needs extra information when resolving an exception.

Our work can also be considered in the multiagent plan execution context for identifying failures. Jonge *et al.* [8] classify the diagnosis of plan failures into two categories; (1) primary plan diagnosis simply points out the failed action, (2) secondary plan diagnosis identifies the actual cause of the failure as we also focus in our work. Although the agents in their plan execution system have partial observations over the system, they still have a major knowledge about their environment. Thus, our work differs form theirs in terms of the distributed protocol execution.

Expectations have also been used to formalize business protocols as described in the SCIFF framework [9]. SCIFF is based on abductive logic, and it does not only specify a business protocol, but also helps verify agent interactions in open systems. Compliance verification has been considered in other domains; in composite Web services [10], or in agent communication languages (ACLs) [11]. An ACL is part of an agent communication framework. The proposed verification process in Guerin and Pitt's work [11] may require access to agent's internal process, whereas our idea of verification depends only on interaction.

Our approach is based on constructing possible hidden states and querying other agents for confirming those states (i.e., identifying which one of them caused the exception in the main protocol). Intuitively, it is reasonable for the agent to query other agents which are committed to it (i.e., the bookstore agent in our example). At this point, we assume that the agent receives honest responses from others. In a real-life scenario, this querying process will continue as a delegation among the agents regarding their commitments (i.e., the bookstore agent redirects the query of the customer agent to the deliverer agent for inspecting the exception further). This delegation is also important in more complicated scenarios since the exception facing agent may not be in contact with all other agents in the protocol. In addition, trust is another important issue when considering multiple agents that enact a distributed protocol. It is more probable that an agent will respond to the agents it is committed to rather than other agents that it has no previous contact with. We aim to

investigate the application of trust strategies when considering such complicated scenarios.

## ACKNOWLEDGEMENT

This research has been supported by Boğaziçi University Research Fund under grant BAP09A106P, The Scientific and Technological Research Council of Turkey by a CAREER Award under grant 105E073, and the Turkish State Planning Organization (DPT) under the TAM Project, number 2007K120610. We thank the anonymous referees for their comments on this paper.

## REFERENCES

[1] M. P. Singh, "An ontology for commitments in multiagent systems: Toward a unification of normative concepts," *Artificial Intelligence and Law*, vol. 7, pp. 97–113, 1999.

[2] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner, "Nonmonotonic causal theories," *Artificial Intelligence*, vol. 153, no. 1-2, pp. 49–104, 2004.

[3] A. K. Chopra and M. P. Singh, "Contextualizing commitment protocols," in *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2006, pp. 1345–1352.

[4] P. Yolum and M. P. Singh, "Flexible protocol specification and execution: applying event calculus planning using commitments," in *AAMAS '02: Proceedings of the first international joint conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2002, pp. 527–534.

[5] N. Desai, A. K. Chopra, M. Arrott, B. Specht, and M. P. Singh, "Engineering foreign exchange processes via commitment protocols," in *International Conference on Services Computing (IEEE SCC)*, 2007, pp. 514–521.

[6] A. U. Mallya and M. P. Singh, "Modeling exceptions via commitment protocols," in *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2005, pp. 122–129.

[7] M. Venkatraman and M. P. Singh, "Verifying compliance with commitment protocols," *Autonomous Agents and Multiagent Systems*, vol. 2, no. 3, pp. 217–236, 1999.

[8] F. D. Jonge, N. Roos, and C. Witteveen, "Diagnosis of multi-agent plan execution," in *In Multiagent System Technologies: MATES 2006, LNCS 4196*. Springer, 2006, pp. 86–97.

[9] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni, "Verifiable agent interaction in abductive logic programming: The sciff framework," *ACM Transactions on Computational Logic*, vol. 9, no. 4, pp. 1–43, 2008.

[10] A. Lomuscio, H. Qu, and M. Solanki, "Towards verifying compliance in agent-based web service compositions," in *Proceedings of 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008, pp. 265–272.

[11] F. Guerin and J. Pitt, "Agent communication frameworks and verification," in *AAMAS 2002 Workshop on Agent Communication Languages*, 2002.