

Modeling Exceptions via Commitment Protocols *

Ashok U. Mallya
Department of Computer Science
North Carolina State University
Raleigh NC 27695 USA
aumallya@ncsu.edu

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh NC 27695 USA
singh@ncsu.edu

ABSTRACT

This paper develops a model for exceptions and an approach for incorporating them in commitment protocols among autonomous agents. Modeling and handling exceptions is critical for successful applications of multiagent systems. Protocols help build multiagent systems, but traditional representations (such as finite state machines or Petri nets) inadequately model complex interactions and exceptions therein. Emerging commitment-based representations are promising, because they declaratively reflect the semantics of an interaction. However, current approaches lack a strong treatment of exceptions.

This paper treats both expected and unexpected exceptions. A commitment protocol is modeled as a set of computations, each representing an allowed interaction and showing the evolving commitments of the participants. Exceptions are modeled via preference structures induced on these sets of computations. The preference structures statically show how expected exceptions are handled whereas the structures must be enhanced dynamically to handle unexpected exceptions. Our approach includes operators for composing protocols and exception handlers, whereby appropriate exception handlers can be dynamically introduced into a protocol as needed.

The main contributions of this paper are (1) a framework for modeling and handling exceptions intelligently in commitment protocols and (2) a demonstration of the benefits of commitment protocols over traditional formalisms in handling exceptions.

Categories and Subject Descriptors

I.2.11 [Computing Methodologies]: Artificial Intelligence Distributed Artificial Intelligence [Multiagent Systems]; D.2 [Software Engineering]:

General Terms

Reliability

*The first author is a full-time student. Both authors would like to thank Amit K. Chopra, Nirmal Desai, and the anonymous reviewers for their comments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.
Copyright 2005 ACM 1-59593-094-9/05/0007 ...\$5.00.

Keywords

Agents, Multiagent Systems, Commitments, Exception Handling

1. INTRODUCTION

Open business processes such as those conducted among independent business partners (or interactions between autonomous agents in general) face challenges quite different from those faced by processes in closed systems. First, open systems require flexible processes that can adapt as the environment evolves. Second, flexible processes require complex models to capture interactions that are rich in semantics. Such models prove notoriously intractable for human process designers. Further, verifying the compliance of participants with such models is difficult since what would be considered a valid detour from the normal enactment of a process in one situation may be considered non-compliant behavior in another.

Simply put, the key features of open systems lead to the possibility of exceptions. Traditional ways to model processes become intolerably complex in the face of exceptions. Traditional ways to enact processes simply fail in the face of exceptions or throw the entire reasoning burden on humans.

There has been an increasing recognition of the importance of agent-based, semantically rich approaches for modelling processes in open systems. However, existing approaches prove to be limited in that they do not provide a principled means for modeling exceptions ahead of time or handling exceptions that have not been explicitly modeled ahead of time. The basic insight that all agree upon is the use of a suitable rule-based mechanism and sometimes of organizational structure, but there are key aspects of the rules and the organizations that are not considered by current approaches.

Conventionally, processes are understood in a monolithic manner. However, an emerging body of work understands processes in terms of *protocols*. A protocol is a specification of an interaction between autonomous agents. Protocols provide a natural abstraction to model and handle exceptions in a reusable manner. This abstraction centers around the concept of commitment protocols, which are a promising approach for modeling interactions among autonomous agents, such as are involved in business processes. This paper is about design-time (modeling) and runtime (handling) abstractions for exceptions.

Running Example. Consider a process used to book a hotel room. It consists of two roles, a customer and a hotel. The normal execution of the process begins with the customer asking the hotel for a vacancy and price check for a room for a certain date. The hotel provides the prices for rooms available, after which the customer asks the hotel to book the room. The hotel confirms that the room has been reserved, and the customer pays for it. The room

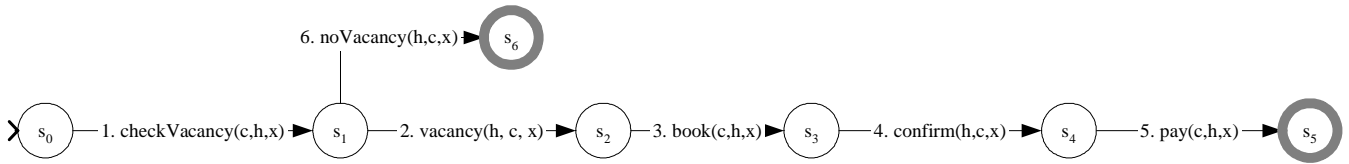


Figure 1: The room reservation process using traditional systems

reservation process is similar to the room reservation workflow example used by Casati *et al.* [2], but is more general in that there is a confirmation step before a room is reserved for the customer. Figure 1 summarizes this process.

Exceptions. Informally, exceptions are abnormal conditions that arise during the execution of a process. The importance of exceptions stems from the simple fact that they are an essential feature of real-life processes. Businesses, for example, entertain exceptional requests from customers in the interest of better customer service. Conversely, exceptions that occur in a process may lead to poor user satisfaction. Therefore, businesses must accommodate exceptions in their underlying systems and their interactions with other businesses. For concreteness, let us review a classification of exceptions proposed by Eder and Liebhart [5]:

- *Basic failures*, which are system-level failures such as network failures.
- *Application failures*, such as database transaction failures.
- *Expected exceptions*, which are deviations from the normal flow that occur infrequently but often enough to be incorporated into the process model.
- *Unexpected exceptions*, which are not modeled and hence require a change in the design of the process when they are discovered.

An alternate classification of exceptions distinguishes among system level exceptions, programming language exceptions, and pragmatic exceptions. Among these, pragmatic exceptions are the most acute and the most difficult to handle. We concentrate on expected and unexpected exceptions that are high-level (pragmatic) exceptions.

As explained above, we are concerned with protocols, which can be used to compose processes. It is therefore natural that exceptions be reflected in protocols. An exception requires special processing so that participants of the protocol can minimize the losses they might incur due to the exception. For expected exceptions, process automation efforts yield diminishing returns as increasing effort is expended to encode deviations from the normal execution of the process. For unexpected exceptions, automated processes have to interrupt human operators to rectify the process enactment

Opportunities are akin to exceptions but viewed more positively. From the perspective of modeling and enactment, opportunities are analogous to exceptions, so we treat them as conceptually similar. For example, the customer in the room reservation process might consider delegating the payment for the room to another agent an opportunity whereas the hotel, which is expecting to receive the payment from the customer, might consider this an exception.

Challenges. The major challenges to exception handling in flexible processes include the following.

- *Handling exceptions requires reasoning about domain knowledge.* The risks taken by the participants in a protocol are not the same across domains. For example, a delayed payment might be acceptable to the hotel in the room reservation process, whereas a delayed shipment of life-saving drugs might not be acceptable in a drug-purchase process. Therefore, exceptions cannot be handled in the same manner across different domains and situations. That is, an abstraction for factoring out domain-independent relationships between participants is required.
- *Handling exceptions requires changing the process model.* Traditional process frameworks such as workflows cannot handle unexpected exceptions since the process model cannot be changed at runtime. That is, an abstraction for reasoning about pragmatic exceptions is required.

Commitment Protocols. Commitment protocols have been proposed as a general mechanism for capturing the semantic aspects of interactions among autonomous agents. In this respect, commitment protocols can yield superior flexibility and a more sophisticated notion of compliance than traditional formalisms such as finite state machines and Petri nets. Mallya *et al.* [12] show how commitment protocols can be composed. Desai *et al.* [4] show how such protocols can be applied to business processes. In their work, a protocol specifies the public interactions of the participants, leaving room for their private *policies* to dictate specifics, e.g., pricing and payment or delivery deadlines. In other words, private policies together with public protocol specifications form a process. This paper extends the above works.

As noted above, exceptions in processes need to be handled in a domain-specific manner. However, a significant portion of the structure of a process can be extracted based on the applicable protocols. Since the protocols that compose a process reflect the main goals of the process participants, they provide a natural basis for modeling and handling exceptions. Refined, context-specific versions of protocols can be used to model expected exceptions and agent policies can be used to apply dynamic handlers for unforeseen exceptions. In other words, using protocols to compose processes simplifies exception handling because protocols provide the bounds for the scope of an exception.

Contributions. We propose an approach that gives semantically richer abstractions for processes based on the concept of commitments. We show how these abstractions enable effective exception modeling and handling while allowing flexibility in execution. The main contributions of this paper are

- A framework for modeling exceptions generically across domains using commitment protocols.
- A demonstration of the benefits of commitment protocols over traditional formalisms in handling unforeseen exceptions by changing the process model at runtime.

Organization. The rest of this paper is organized as follows. Section 2 describes the technical framework that we employ, beginning with commitments in Section 2.1 and operations on commitments in Section 2.2. Sections 2.3 and 2.4 formalize runs and protocols, respectively. Section 3 develops our theory of exceptions. Exception modeling is described in Section 3.1 and runtime exception handling in Section 3.2. Section 4 concludes the paper with a summary of our contributions in light of related research and describes some avenues of further research.

2. TECHNICAL FRAMEWORK

Commitment protocols have been in development for some years now. In practice, commitment protocols are specified declaratively, for example, using the OWL-P language proposed by Desai *et al.* [4]. The specification identifies roles that participate in the protocol, the messages that are exchanged (with the meanings of the messages in terms of commitments that are created), and a set of rules that constrain the set of runs of the protocol by defining ordering, data flow, and other constraints. For a formal treatment, however, we represent protocols as transition systems similar in spirit to commitment machines [18]. These protocols generate computations or *runs*, which are sequences of *states* that a valid protocol computation (execution) goes through. States are labeled by *propositions* that hold true. Propositions represent facts about the universe of discourse of the protocol such as commitments that are active and messages that have been sent. State changes are caused by *messages* that the participants send to each other. Consider the room reservation process. The steps taken in a normal execution of this process were described in Section 1. Table 1 is a snippet of the specification of this protocol. The *policy(x,y)* term checks if *x* wishes to send the message *y*, and *start* is a special term indicating the start state. The derivation of local flows for each role and their binding with the policies of each participant is described by Desai *et al.* [4]. Figure 2 shows this protocol as a set of runs. Each run begins at the start state s_0 and ends at a thick circle, which represents a terminating state. Arrows with shaded text show exceptions and filled circles are exception states.

2.1 Commitments

A commitment $C(x, y, p)$ denotes that the agent *x* is responsible to the agent *y* for bringing about the condition *p*. Here *x* is called the *debtor*, *y* the *creditor*, and *p* the *condition* of the commitment. The condition is expressed in a suitable formal language. Commitments can also be *conditional*, denoted by $CC(x, y, p, q)$, meaning that *x* is committed to *y* to bring about *p* if *q* holds. For example, the commitment $CC(h, c, confirm(h, c, x), pay(c, h, x))$ denotes the commitment by the hotel to the customer to confirm a room booking if the customer pays for the room.

2.2 Commitment Operations

Commitments are created, satisfied, and transformed in certain ways. The following operations are conventionally defined for commitments.

1. $CREATE(x, C)$ establishes the commitment *C*. This can only be performed by *C*'s debtor *x*.
2. $CANCEL(x, C)$ cancels the commitment *C*. This can only be performed by *C*'s debtor *x*. Generally, cancellation is compensated by making another commitment.
3. $RELEASE(y, C)$ releases *C*'s debtor *x* from commitment *C*. This only can be performed by the creditor *y*.

| |
|--|
| Role 1: Customer, <i>c</i> |
| Role 2: Hotel, <i>h</i> |
| Rule 1: $start \Rightarrow policy(c, checkVacancy) \ \&\& \ policy(h, vacancy)$ |
| Rule 2: $vacancy \Rightarrow policy(c, book)$ |
| ... |
| Message 1: $checkVacancy(c, h, x)$ <i>c</i> asks <i>h</i> if there is a vacancy for a certain room for certain dates, encoded by <i>x</i> . This message does not create any commitments. |
| Message 2: $vacancy(h, c, x)$ <i>h</i> informs <i>c</i> that the room-date-price combination <i>x</i> is available. Now, <i>h</i> is committed to confirming the room reservation if <i>c</i> pays the price quoted. This message creates the commitment $CC(h, c, confirm(h, c, x), pay(c, h, x))$. |
| Message 3: $book(c, h, x)$ <i>c</i> asks <i>h</i> to book the room for the given dates. By sending this message, <i>c</i> commits to paying <i>h</i> if the room is confirmed. This message creates the commitment $CC(c, h, pay(c, h, x), confirm(h, c, x))$. |
| Message 4: $confirm(h, c, x)$ <i>h</i> confirms that the room has been booked. <i>h</i> is now committed to <i>c</i> to allow the use of the room as indicated earlier in the vacancy dates. This message creates the commitment $C(h, c, use(x))$. |
| Message 5: $pay(c, h, x)$ <i>c</i> pays the amount specified in <i>x</i> to <i>h</i> . This message does not create any commitments. If <i>c</i> is committed to <i>h</i> to pay for <i>x</i> , this message fulfills that commitment. |
| Message 6: $timeout(h, c, pay)$ <i>h</i> detects that <i>c</i> has not sent the $pay(c, h, x)$ message within the specified deadline. This message does not create any commitments; it is an indication of the breach of any commitment that <i>c</i> might have made to <i>h</i> to pay for <i>x</i> . |
| Message 7: $cancel(h, c, confirm)$ <i>h</i> cancels its commitment to <i>c</i> to allow the use of the room. This message cancels the commitment $C(h, c, use(x))$. |
| ... |

Table 1: Room reservation protocol in OWL-P

4. $ASSIGN(y, z, C)$ replaces *y* with *z* as *C*'s creditor.
5. $DELEGATE(x, z, C)$ replaces *x* with *z* as the *C*'s debtor.
6. $DISCHARGE(x, C)$ *C*'s debtor *x* fulfills the commitment.

A commitment is said to be *active* if it has been created, but not yet been operated upon by a *discharge*, *delegate*, *assign*, *cancel*, or *release*. A commitment is *satisfied* when its condition becomes true. A conditional commitment such as $CC(c, h, pay(c, h, x), confirm(h, c, x))$ becomes an unconditional commitment $CC(c, h, pay(c, h, x))$ when its condition $confirm(h, c, x)$ holds. A commitment is *breached* when it is not possible that the commitment will be satisfied. Realistic settings assign deadlines to commitments to detect their breach or satisfaction [13]. Conditional commitments can also be satisfied without a transformation into an unconditional commitment. For example, $CC(c, h, pay(c, h, x), confirm(h, c, x))$ is satisfied when $pay(c, h, x)$ is true, regardless of $confirm(h, c, x)$.

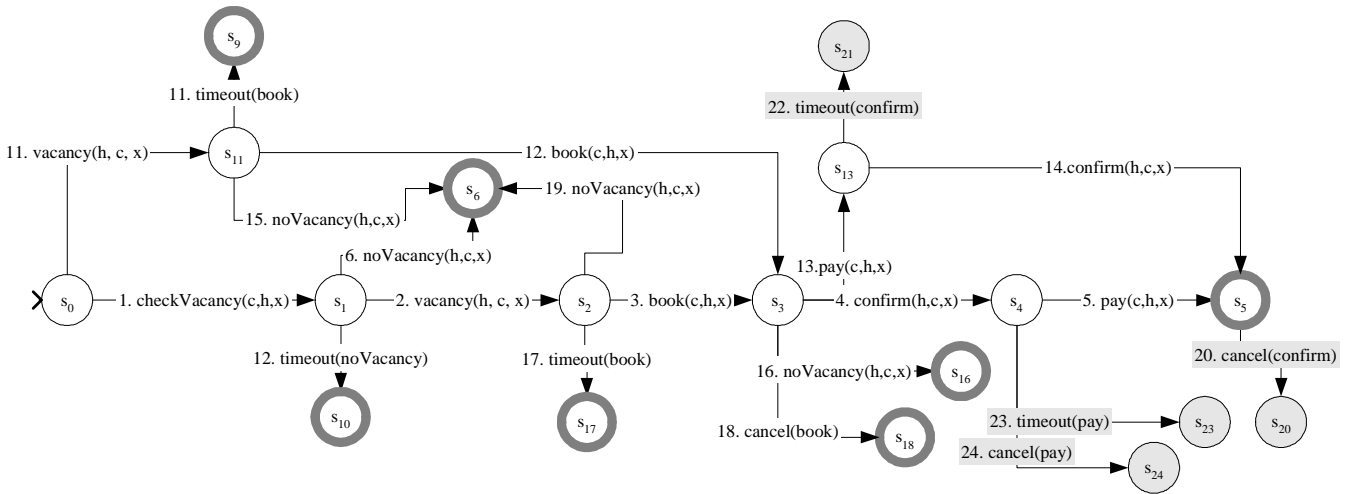


Figure 2: Room reservation protocol. Filled circles are exceptions; thick circles are terminating states

Commitment Life Cycle. Because commitment protocols are declarative, they are more flexible than state-based specifications. During enactment of such protocols, commitment operations allow the participants to choose from multiple paths available. For example, a commitment might be discharged after it is created, or it might be delegated before it is discharged. For this reason, it is instructive to see how commitments can be modified. Figure 3 shows the life cycle of a commitment, based on prior work by the authors [13], which gives a semantics to the commitment life cycle using a variant of Computational Tree Logic (CTL). In the figure, propositions are used to capture facts about commitments and commitment operations. States are represented by circles and some of the propositions that label a state are written in adjoining grey rectangles. Transitions between states occur on commitment operations. Note that in practice, a commitment operation might not be

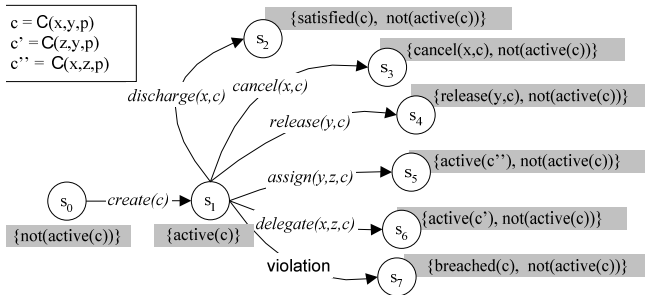


Figure 3: The commitment life-cycle. States are labeled by propositions. To avoid clutter, only pertinent propositions are shown.

atomic, but might involve a series of message exchanges all within the scope of a transaction. For instance, the delegation of a commitment involves three agents, the creditor and the debtor of the commitment being delegated, and the new debtor. A commitment-based execution framework might require that all three parties agree if the delegate operation is to succeed. Such reasoning is described in more detail by Venkataraman and Singh [16].

As shown in Figure 3, a commitment is created before any other operation is performed on it. Before it is created, a commitment is not active. Once created, the commitment stays active until some operation is performed on it. Propositions record operations on

commitments. Operations on active commitments are persistent, i.e., once an active commitment is discharged, cancelled, released, assigned, or delegated that operation is recorded by a proposition and that commitment cannot be operated upon again. This scheme assumes that commitments have unique identifiers. A commitment that is discharged is said to be *satisfied*. A commitment that is violated is said to be *breached*. To avoid clutter, the figure does not show the persistence of propositions in the state labels. Also, violation is not a commitment operation. Commitment violation can be detected in many ways, including through the expiration of the deadline of a commitment.

2.3 Runs

A run τ is a sequence of states $\{s_0 \dots s_{|\tau|}\}$, for which $[\tau]^0$ represents the first state s_0 . We consider only nonempty runs, i.e., a run must contain an initial state. Likewise, $[\tau]^\top$ represents the last state of a run, defined only for finite runs. The operator \prec_τ orders states temporally with respect to a run τ , so that $s_i \prec_\tau s_j$ implies that s_i occurs before s_j in the run τ .

State Similarity. A *state-similarity function* f is a mapping from a state to a set of states, i.e., $f : \mathbb{S} \mapsto 2^{\mathbb{S}}$. A state s_i is *similar* to a state s_j under the state-similarity function f if and only if $s_j \in f(s_i)$. State-similarity under the state-similarity function f is denoted by the operator $\llbracket f \rrbracket$. That is, $s_i \llbracket f \rrbracket s_j \iff s_j \in f(s_i)$. Since states are labeled by propositions, state-similarity functions help us reason about protocols via propositions that denote commitments.

Consider σ , a state-similarity function, defined as $\sigma(s_i) = \{s_j \mid s_j \text{ can be reached by finite number of } \textit{delegate}(\cdot, \cdot) \textit{ actions from } s_i\}$. σ treats a state s_i as being similar to a state s_j if in the two states all the participants of the protocol have the same commitments being made towards them, regardless of which participant makes it. In practical settings, σ can be used to indicate, for example, that a commitment to pay can be delegated to a bank while still honoring the protocol.

Subsumption of Runs. Let $\llbracket f \rrbracket$ denote a *subsumption* operator over runs. The operator $\llbracket f \rrbracket$ is an order-preserving mapping from one run to another, and depends on the state-similarity function f . A run τ_j *subsumes* a run τ_i under the state-similarity function f if and only if, for every state s_i that occurs in τ_i , there occurs a state

s_j in τ_j that is similar under f , and s_j has the same temporal order relative to other states in τ_j as s_i does with states τ_i . Formally, $\tau_j \llbracket f \rrbracket \tau_i$ if and only if $\forall s_i \in \tau_i, \exists s_j \in \tau_j$ such that $s_j \in f(s_i)$ and $\forall s'_i \in \tau_i, \exists s'_j \in \tau_j$ such that $s'_j \in f(s'_i) \Rightarrow (s_i \preceq_{\tau_i} s'_i \Rightarrow s_j \preceq_{\tau_j} s'_j)$. Longer runs subsume shorter ones, provided they have similar states and in the same order.

2.4 Protocols

Formally, a protocol is modelled as a tuple, $\langle \mathbb{A}, \mathbb{S}, \mathbb{S}_0, \Delta, \mathbb{F}, \mathbb{R} \rangle$, where \mathbb{A} is a set of messages, \mathbb{S} is a set of states \mathbb{S}_0 is the set of start states, $\mathbb{S}_0 \subseteq \mathbb{S}$, Δ is a set of transitions, $\Delta \subseteq \mathbb{S} \times \mathbb{A} \times \mathbb{S}$, \mathbb{F} is a set of final states, $\mathbb{F} \subseteq \mathbb{S}$ and \mathbb{R} is a set of roles in the protocol. This tuple is derived from a protocol specification such as that shown in Table 1.

Δ contains transitions of the form $\langle s_i, a, s_j \rangle$, where $s_i, s_j \in \mathbb{S}$ represent the source and the destination of the transition, $a \in \mathbb{A}$ is the message that triggers the transition. Such a transition advances a run from s_i to s_j based on a . A run can be generated from a protocol by the successive concatenation of transitions beginning from the initial state of the protocol. The concatenation of a transition to a run appends the destination of transition to the run if the source of the transition matches the last state of the run. The set of runs generated by a protocol is denoted by $\llbracket P \rrbracket$.

Every protocol P is considered to belong to a *frame* with enough propositions in it to label all states that can occur in the runs generated by $\llbracket P \rrbracket$. Frames serve as a common ontology for the propositions used by different protocols and demarcate the universe of discourse of a protocol. In this paper, we assume that ontologies match wherever required.

Protocol Subsumption. A protocol P_j *subsumes* a protocol P_i under the state-similarity function f if and only if, for every run τ_i that P_i can generate, P_j can generate a run τ_j that is subsumed by τ_i under f . Formally, $P_j \llbracket f \rrbracket P_i$ if and only if $\forall \tau_i \in \llbracket P_i \rrbracket, \exists \tau_j \in \llbracket P_j \rrbracket, \tau_i \llbracket f \rrbracket \tau_j$. In other words, if P_j is a protocol that allows numerous runs and P_i is a protocol that allows only a few runs, P_j subsumes P_i as long as each of P_j 's runs is subsumed by one of P_i 's runs. Since long runs subsume shorter ones, protocols that specify a few short runs subsume protocols that specify a large number of short runs. This follows from our intuition that fewer constraints make for more flexible protocols.

Protocol Algebra. We use the protocol algebra developed by Mallya *et al.*, consisting of two operators (*merge* and *choice*), and an ordering relationship (subsumption, as defined above). For brevity, we only describe only the components that are used in this paper. The merge operator, denoted by \otimes_f , splices two protocols under a state-similarity function f so that refined protocols can be created from existing ones. A merge of two protocols is a meshing of the runs of the protocols. Formally, $P \otimes_f Q = R$ such that $\llbracket R \rrbracket = \{r \mid \forall r_p \in \llbracket P \rrbracket, \forall r_q \in \llbracket Q \rrbracket, r \llbracket f \rrbracket r_p \text{ and } r \llbracket f \rrbracket r_q\}$. The merge operator refines the protocols being merged, i.e., each of the protocols being merged subsume the merged protocol. Merge is idempotent, commutative and associative.

3. EXCEPTION HANDLING

We propose an approach to exception handling based on commitments that exist between the participants. Our approach has the following features:

- For expected exceptions, we define a preference structure over runs, i.e., a notion that some runs are more desirable than others, which can be used to model expected exceptions.

This structure and its application is described in Section 3.1.

- For unexpected exceptions, we introduce protocol splicing using the merge operator to handle unexpected exceptions in business processes. Essentially, exception handlers are formalized as sets of runs, formally on par with protocols. Where appropriate, these handlers are spliced into protocols, as explained in Section 3.2.

In the room reservation example, the challenges to exception handling that we outlined earlier manifest themselves as follows.

1. *Domain-independent reasoning.* The normal process dictates confirmation of the hotel room before payment. A customer who sends in the payment before the confirmation is in violation of the protocol under such a model, even though practical processes should allow for it, since the essence of the process (reserving a room) is not violated.
2. *Unforeseen exceptions.* The hotel might be forced to revoke the confirmation for the room because of a fire that destroys the room. In traditional process enactment mechanisms such as workflow engines, such exceptions cannot be handled because they require a change in the workflow model. Realistically, the customer should be compensated by some means such as a refund or an offer for an alternative room or hotel.

3.1 Expected Exceptions

A protocol allows multiple runs, which is what gives its participants flexibility in enacting the protocol. We propose that each protocol specify a hierarchy of preferred runs. The preference relation between runs need not be complete. That is, given two runs of a protocol, the protocol might not state whether one of those runs is more or less preferred than the other. Such preferences prove valuable in reasoning about why a particular state is an exception state. For example, the process model shown in Figure 1 can be enhanced by stating that run with no *noVacancy*(h, c, \cdot) states is preferred over the other.

Preferences between runs can be used to define exceptions based on the social environment in which the protocol is enacted. Given a protocol P and set of runs r_1 generated by it, which are preferred over a set of runs r_2 , also generated by P , one can define all runs in r_2 to be exceptions. When P is used in another context, the preference structure can be changed, to define new exceptions for the same protocol.

Consider the room reservation protocol. The protocol might specify that runs in which the hotel does not recant its *vacancy*(h, c, x) message (by sending a *noVacancy*(h, c, x) message) are preferred over runs that do, and the latter set of runs signal an exception. This protocol will consider the runs $\langle s_0 s_1 s_2 s_3 s_1 s_6 \rangle$, $\langle s_0 s_1 s_6 \rangle$, and $\langle s_0 s_1 s_2 s_6 \rangle$ as exception-causing runs (see Figure 2). The same protocol without such a preference structure and exception definition, would allow the the runs listed above and thus be better for the hotel. Notice that commitments enable the reasoning that $s_1 s_6$ is a valid state, i.e., the hotel can send the *noVacancy*(h, c, x) message even after it receives the *book*(c, h, x) message, since no unconditional commitments exist at s_3 .

Run preferences may be based on different metrics. For example, short runs might be preferred over longer ones and runs that do not involve the delegation or cancellation of a particular commitment may be preferred over runs that do.

Usage. We envision that published protocol specifications (such as OWL-P) will contain several different preference structures and exception definitions of the kind explained above. Participants of a

protocol negotiate the particular preference structure and exception definitions that they wish to use for a particular enactment of a protocol. The study of such negotiation schemes is beyond the scope of this paper and an interesting direction for our research.

Benefits. The benefit of inducing a preference structure over runs are

1. A preference structure relates an exception to the protocol as a whole rather than modeling an exception as a trigger for specific exception handler code.
2. Participants can reason about what kind of preference structures and related exception definitions they want to entertain. For example, customers might like to choose a room reservation protocol with a preference structure that terms the $noVacancy(h, c, x)$ sent after a $vacancy(h, c, x)$ message an exception.
3. Preference structures decouple the specification of a protocol from the environment in which it is enacted. A situation that is normal in a protocol might be deemed an exception in the same protocol when it is being executed in a different context. For example, the room reservation protocol allows the hotel to send a $vacancy(h, c, x)$ message to the customer to start the protocol. Starting the protocol in this manner is equivalent to the hotel advertising the availability of a room. In some situations, however, such an unsolicited advertisement might be deemed illegal, as in the case of spam.

3.2 Unexpected Exceptions

Unexpected exceptions are detected at runtime but are not part of the process model in traditional systems. In a commitment protocol, however, a model is a declarative specification of the meanings of messages, which, combined with the semantics of commitment operations, drives the protocol. This section shows how our protocol algebra can be used to combine exception handlers with protocols at runtime so that new situations can be handled when they arise.

Splicing Protocols. Protocols can be *spliced* using the merge operator, with additional constraints. For example, if a complicated payment protocol (where a payer transfers funds to a payee via a bank) were available, that payment protocol could be spliced into the room reservation protocol between states s_4 and s_5 or between states s_3 and s_{13} . The new protocol thus obtained would be a *refinement* of the room reservation protocol, since it dictates a finer grained protocol to its participants than the plain room reservation protocol. This concept of protocol splicing and refinement has been developed by Mallya and Singh [12].

We treat exception handlers as sets of runs, just as we treat protocols. This enables handlers to be spliced into protocols. As an example, consider the exception state s_{24} in Figure 2. This exception is caused by the customer cancelling its commitment to pay for the room after the hotel has confirmed it. Normally, this would be a violation of a commitment. The customer might wish to cancel the commitment to pay because of financial hardship. Consider now an exception handler as shown in Figure 4, at the top. This handler can be spliced into the room reservation protocol after appropriately binding the role a to customer, role b to hotel, and role d to the customer's bank. Splicing the handler enables the customer keep the hotel room while not having to pay the hotel. States s_{100} and s_{102} of the exception handler correspond to states s_4 and s_5 of

the room reservation protocol, respectively. The splicing is shown in Figure 4, in the middle.

The same handler can also be used to delegate commitments when the need arises to cancel them. For example, this handler can be used by the hotel to delegate the reservation of a room to another hotel in case of fire. In this case, the handler would be spliced at state s_5 of the room reservation protocol, with the role a corresponding to the hotel, role b to the customer, and role d to another hotel which can provide the room in lieu of the former hotel. State s_{100} now corresponds to s_5 and states s_{101} and s_{102} are added to the protocol, as shown at the bottom of Figure 4.

3.3 Preferences and Splicing: Tradeoffs

Splicing exception handlers during execution and inducing a preference structure over runs during the design of a protocol are both means to handle exceptions. Using one method over the other involves some tradeoffs as we describe next.

Splicing exception handlers at runtime requires a library of handlers and a search through this library. This search can be computationally demanding. However, a protocol designer can choose to handle all exceptions at runtime. The advantage of this approach is that the designer's job is simplified, since she uses an existing library of handlers developed by someone else. Since the handlers are not designed and developed by the protocol designers, this approach is applied to unexpected exceptions, meaning the exceptions are not anticipated by the protocol designer. On the other hand, inducing a preference structure over the runs of a protocol requires the protocol designer to have a good understanding of the protocol and the possible exception conditions that may arise. This makes for considerable design-time effort and extensive domain specific knowledge. However, design-time preference specification is computationally faster and speeds up execution of an interaction protocol.

Advantages. Both splicing and preference specification are novel methods for exception handling, and address drawbacks of traditional process models. Process models such as those used for web service interactions, namely the Business Process Execution Language (BPEL) [1] and the Web Services Choreography Description Language (WS-CDL) [9] do not capture the consequences of autonomy of their participants, since they do not have a notion of commitments. For example, the room reservation process cannot be specified independently of the exceptions that occur during the process. There is also no construct to specify that a certain set of exceptions is less desirable than another set of exceptions. Since web services aim to ease interactions between autonomous components, our approach applies naturally to service-oriented architectures. Another advantage of our approach is the treatment of protocols as first-class entities. Such a global protocol view enables protocol reuse in contrast with local views of processes generated using OWL-S (OWL for Services) [14], which is an ontology for semantic markup of services that aids automatic composition of services to create processes.

4. DISCUSSION AND FUTURE WORK

Exceptions are numerous in real-life systems since the universe of discourse of a business process has ill-defined boundaries. Real-world processes need pragmatic exception handling, i.e., reasoning about meaning in context. We have proposed a methodology that utilizes commitments among participants to handle exceptions in a protocol. Our approach induces a preference structure over the possible execution sequences of a protocol, which helps decouple real-world preferences from protocol specifications. We have shown

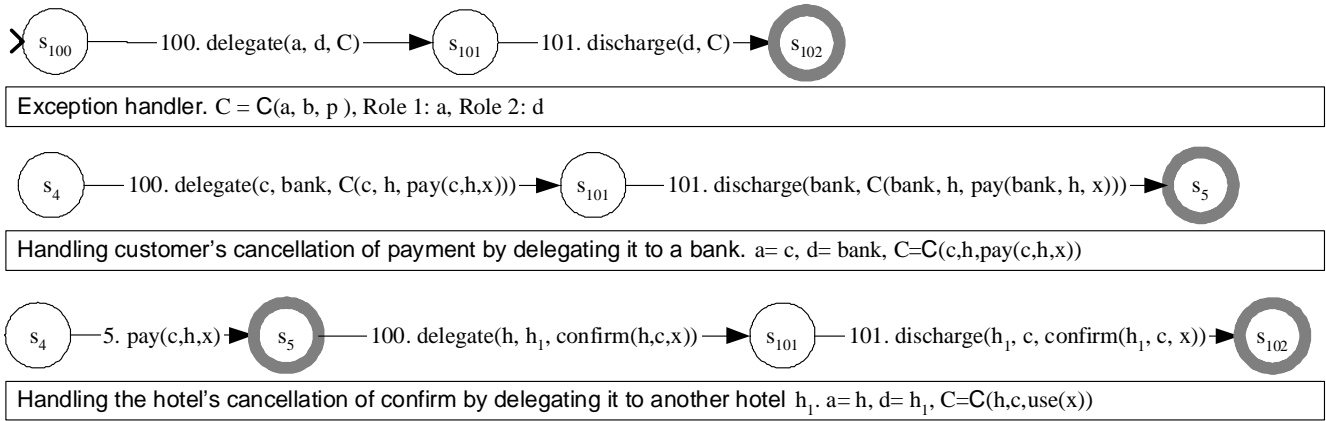


Figure 4: An exception handler and its usage by splicing

how this approach can be used to model exceptions. Further, we have also demonstrated how an algebra of protocols can be employed to handle exceptions that are not originally in the process model. With this approach, we have demonstrated how commitment protocols used in specific contexts can create agile processes. These contributions are a significant step forward from the closed, rigid process models provided by traditional formalisms.

Next, we discuss literature relevant to our work and chart out interesting directions that we intend to pursue.

4.1 Related Literature

Exception handling has been studied for programming languages, multiagent systems, and workflows, among others.

Programming. Miller and Tripathi [15] identify four different reasons for exceptions in object-oriented systems: *errors*, which are illegal conditions, valid *deviations*, *notifications* that invalidate certain assumptions, and *idioms*, which are legal conditions that are rare, such an end-of-file encounter. They study errors in detail and deviations and notifications to a limited extent whereas our research deals mainly with deviations, notifications, and idioms.

Multiagent Systems. Klein *et al.* [11] study exception handling in multiagent systems in detail. We agree with their premise that open systems require fundamentally different exception handling techniques than those required by closed systems. Klein *et al.* develop an architectural framework for multiagent systems that uses a directory of agents to keep track of the agent population so that situations such as the death of an agent can be handled with minimal resource wastage. In another work, Klein [10] develops a library of generic exception handlers and proposes the use of specialized agents that handle exceptions. This work could be combined with our approach to enable the specialized agents to reason about exceptions.

Fornara and Colombetti [6] propose an alternative life-cycle for commitments which differs from ours in having a commitment proposal state before the commitment is created. They describe the construction of interaction protocols using commitments. Yolum [19] motivates definitions of correctness and consistency of commitment protocols, thus formalizing commitment protocol design. Our work used in conjunction with the above would strengthen a commitment protocol model since we propose runtime exception handling mechanism whereas Yolum develops a design time verification framework.

Workflows. Kamath and Ramamritham [8] develop a rich workflow model that enables the *opportunistic* rollback of tasks that failed, so that the effect of the failure is contained. Their work represents a significant improvement over traditional workflow engines. Chiu *et al.* [3] describe several exception resolution techniques in ADOME-WFMS, a workflow system. They identify tasks in the workflow as critical, optional, replaceable, or repeatable. Exceptions are handled by modifying task assignments, or skipping a task. Casati *et al.* [2] develop a library of exception patterns for workflow systems.

Finally, Wegner [17] argues that interaction-oriented programming models the openness of the real-world better than an algorithmic approach. The interaction-centric view, which we have adopted, will help develop powerful abstractions for multiagent systems.

4.2 Research Directions

Some important research paths remain to be explored to develop a complete theory of exception handling in commitment protocols.

As a first step, the development of a language for specifying sets of runs would enable the specification of a preference lattice over sets of runs of a protocol.

The introduction of spheres of commitment (SoComs) [7] into protocol specifications and a methodology to assign agents to SoComs will help us develop a theory for limiting the range of propagation of the effects of exceptions. For example, when protocols are merged, an exception in the merged protocol can be handled within the SoCom formed by the roles defined by the protocols that were merged.

The incorporation of transactions, whose boundaries are derived from the commitments that exist in a protocol would ease the automation of exception handling, i.e., the choosing of an exception handler.

The development and incorporation of patterns of exceptions that commonly occur would enhance the protocol library that designers can use to compose robust protocols.

5. REFERENCES

- [1] T. Andrews, F. Curbera, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services, (BPEL 1.1), May 2003. www-106.ibm.com/developerworks/webservices/library/ws-bpel.

- [2] F. Casati, M. Fugini, and I. Mirbel. An environment for designing exceptions in workflows. *Information Systems*, 24(3):255–273, 1999.
- [3] D. K. W. Chiu, Q. Li, and K. Karlapalem. ADOME-WFMS: towards cooperative handling of workflow exceptions. *Advances in exception handling techniques*, pages 271–288, 2001.
- [4] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Processes = Protocols + Policies: A methodology for business process development. Technical report, North Carolina State University, 2004. TR2004-34.
- [5] J. Eder and W. Leibhart. The workflow activity model WAMO. In S. Laufmann, S. Spaccapietra, and T. Yokoi, editors, *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS-95)*, pages 87–98. University of Toronto Press, May 1995.
- [6] N. Fornara and 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)*, pages 535–542. ACM Press, July 2002.
- [7] A. K. Jain, M. Aparicio IV, and M. P. Singh. Agents for process coherence in virtual enterprises. *Communications of the ACM*, 42(3):62–69, Mar. 1999.
- [8] M. Kamath and K. Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 334–341, 1998.
- [9] N. Kavantzias, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web service choreography description language (WS-CDL 1.0), December 2004. <http://www.w3.org/TR/ws-cdl-10/>.
- [10] M. Klein. Exception handling in agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*, 1999.
- [11] M. Klein, J. A. Rodriguez-Augilar, and C. Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Journal of Autonomous Agents and Multiagent Systems*, 7(1/2), 2003.
- [12] A. U. Mallya and M. P. Singh. A semantic approach for designing commitment protocols. In R. V. Eijk, editor, *Developments in Agent Communication*, volume 3396 of *Lecture Notes in Artificial Intelligence*, pages 37–51. Springer, Berlin, 2005.
- [13] A. U. Mallya, P. Yolum, and M. P. Singh. Resolving commitments among autonomous agents. In F. Dignum, editor, *Advances in Agent Communication*, volume 2922 of *Lecture Notes in Artificial Intelligence*, pages 166–182, Berlin, 2003. Springer.
- [14] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. W3C Member Submission, November 2004. <http://www.w3.org/Submission/OWL-S>.
- [15] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103, 1997.
- [16] M. Venkatraman and M. P. Singh. Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, Sept. 1999.
- [17] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.
- [18] M. Winikoff, W. Liu, and J. Harland. Enhancing commitment machines. In *Proceedings of the AAMAS-04 Workshop on Declarative Agent Languages and Technologies*, 2004.
- [19] P. Yolum. Towards design tools for protocol development. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2005.