Engineering Commitment-Based Multiagent Systems: A Temporal Logic Approach*

Jie Xing
IBM Corporation
jiexing@us.ibm.com

Munindar P. Singh
North Carolina State University
singh@ncsu.edu

ABSTRACT

Commitments model important aspects of agent interactions, especially those arising in e-business. A small number of patterns of commitments accommodate a variety of realistic interactions among agents. We represent these patterns and agent behavior models formally and show how certain behavior models can be formally proved to be sound for certain patterns. Thus a designer may use a library of patterns and behaviors to engineer systems that are guaranteed to work correctly.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

General Terms

Verification

Keywords

Commitments; operational semantics; behavior models, statecharts.

1. INTRODUCTION

Multiagent systems are ideally suited for applications that involve autonomous, heterogeneous components, which can be naturally represented by agents. Interactions of autonomous agents in a multiagent system cannot be adequately captured merely in terms of the sequences of their actions or messages. Commitments among agents provide a conceptually well-founded basis for modeling the interactions of agents while respecting their autonomy.

Motivation. To ensure flexibility in handling opportunities and exploiting opportunities, the agents should be able to deviate from rigid scripts. However, to ensure correctness, the deviations must be principled. Representing the commitments that the agents have to one another and specifying constraints on their interactions in terms of commitments provides a solid basis for agent interactions. But if commitments were set arbitrarily, the multiagent systems would be unduly complex and it would be difficult to ensure that the member agents were behaving properly.

In simple terms, designing a multiagent system consists of designing its roles and their mutual interactions, and designing the agents to play the given roles. For our purposes, a role can be modeled by a set of commitments. For example, a seller in an online market may be expected to commit to its price quotes and a buyer may be expected to commit to paying for goods received. In general, the designers of the individual agents would be different from the designers of the roles in the multiagent system. For example, an online market would only specify the roles of buyer, seller, and broker and their associated commitments, but individual participants may, and would often prefer to, use differently implemented agents to play these roles.

Obviously, each application would impose its own requirements. Further, each competing implementation of a role would involve different algorithms and heuristics. However, we can abstract out a small number of *patterns* of commitments and *behavior models*, capturing the essential structure of agents' interactions and their internal designs, respectively. The internal design of an agent is characterized only to the extent that it enables the agent to interact in the manner specified.

Contributions. Software engineering has two major components: (1) methodologies and (2) formal reasoning about models. This paper is a contribution to formal aspects of agent-oriented software engineering with a view to enabling rigorous methodologies for designing software systems based on agents. Our objective is to enable a designer to reason about whether a given behavior model can correctly play a given role, i.e., satisfy stated patterns involving that role. Further, our objective is to provide a library of commitment patterns and a library of behavior models along with formally proved assertions of which behavior models satisfy which patterns. If the commitments required for the roles are chosen from our library of patterns, then the corresponding behavior models can be readily found without the designer having to explicitly apply any formal reasoning during the design process. In this manner, the formal results can support sound engineering without making significant demands on the designer. Further, to ensure that our work embeds easily into traditional software engineering, we use statecharts and temporal logic as our representational frameworks.

Organization. Section 2 presents important background regarding commitments and commitment patterns. Section 3 formalizes commitment patterns. Section 4 introduces and proves soundness of the operational semantics for agent behavior models. Section 5 discusses the literature and future directions.

*This work was supported by IBM and by the National Science Foundation under grants IIS-9624425 and DST-0139037. We are indebted to Feng Wan for useful discussions. A previous version of this paper appears in the Symposium on Applied Computing, 2001, but much of the technical development is new here.

AAMAS'03, July 14–18, 2003, Melbourne, Australia.

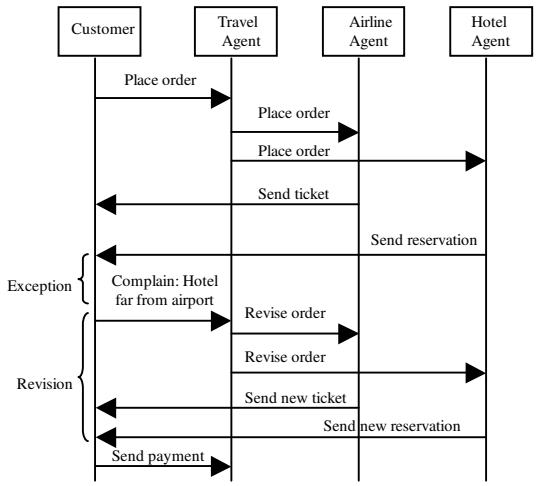

Figure 1: An e-commerce scenario

2. COMMITMENTS AND PATTERNS

This section provides some background on commitments applied to business process interoperation [11].

EXAMPLE 1. *A customer contacts her travel agent to book a trip to a city with multiple hotels and airports. The travel agent requests airline and hotel clerks to make appropriate reservations and send confirmations to the traveler. The customer may have requirements that are not initially disclosed. For example, the hotel location is important if the flight arrives late in the day. If the customer's requirements are not met, she complains to her travel agent. If the travel agent works to satisfy his customer, he would then make a revised request to the clerks, say, for an earlier flight.*

Although small, Example 1 is not trivial. Four autonomous parties are involved, and communication among them does not follow a simple nesting of requests and responses. As shown in Figure 1, even a task that executes successfully may need to be revisited. Current approaches offer no conceptual or operational support for exceptions, leading programmers to employ ad hoc techniques that lead to reduced productivity and ineffective solutions.

Concepts. The following are some key concepts. A *task* identifies a definite piece of work. Tasks may be atomic or compound, but their structure is not directly relevant for this paper. A *commitment* relates a debtor, a creditor, and a condition [7]. This means that the debtor is obliged to the creditor to satisfying the given condition. The condition involves the relevant predicates from the domain. In our example, the airline can commit to the given traveler that she is confirmed on a particular flight. A *role* encapsulates a set of commitments. Any agent playing a given role is expected to satisfy the commitments associated with that role.

Importantly, commitments are flexible, and can be revoked or modified. For example, even a confirmed flight booking can be canceled. However, the revocation or modification of commitments is constrained through further commitments. In our example, the airline can commit to the traveler that if her booking (a commitment) is canceled, the airline will create a new booking (another commitment) for her. Thus, commitments can capture the structure or qualitative aspects of service agreements.

Commitment patterns. *Commitment patterns*, inspired from design patterns, apply to interactions among agents. The following patterns involve two roles: a consumer and a producer. An agent may play different roles in different patterns. Predicates with arguments (i.e., data values) represent the information exchanged. Although each individual commitment is a directed obligation, they usually occur in sets involving multiple roles—thus multiparty scenarios can be captured. The following patterns are generic and a reasonable first approximation for business process scenarios. We imagine other patterns will be developed later.

1. *Entertain request*. This means that a producer will accept requests from another agent and act according to the tasks requested. The producer may refuse the request and may accept a retraction from the other agent. For example, a travel agent entertains a call for a trip from a customer and performs the requested task.
2. *Notify*. This comes into effect when a producer has just completed its tasks for the first time. It informs its consumer of the computed data values, and becomes committed to the specified predicates. For example, a travel agent, who reserved a trip package for a customer, notifies the customer and commits to the package.
3. *Entertain update*. This comes into effect when a producer accepts requests to correct some data values that another agent supplies. Further, the producer will perform the task to entertain the new data values. For example, a customer may request a later flight on the same date. The travel agent performs the task again with the updated data values.
4. *Renotify*. This comes into effect when a producer has just completed its tasks for the second or a later iteration, and some of its existing commitments are violated by the recently completed task. The violation would typically occur because the predicates to which the agent had committed have been falsified by the results just obtained. The producer will send the new results to its consumer. For example, the travel agent informs the customer of a new schedule, because the previously booked flight was canceled.
5. *Retry*. This comes into effect when a consumer is dissatisfied with the results. It sends a reject request to its producer. The producer performs its task again in order to satisfy the consumer. The practical ramifications of this pattern are much greater, however, because it enables an agent to send a complaint upstream and to demand that a property it desires be satisfied. For example, the customer sends reject information to the travel agent to reject the trip package. The travel agent is released from the associated commitment and performs the task again to find an alternative package to the customer.
6. *Resign*. A producer may cancel its commitment by sending a message to the other agent. For example, the travel agent cancels to provide a later airline ticket if it is unable to make the revised reservation for customer.

EXAMPLE 2. *We now describe our commitment patterns as applied on Example 1.*

- *After the travel agent receives a request from the customer, it performs the make order task [entertain request].*
- *After the travel agent finishes its task, it notifies the hotel and airline agents about the customer's order information (date, place, and price) [notify].*
- *When the hotel and airline agents receive the request from the travel agent, they perform book room and book flight, respectively [entertain request].*
- *After the airline and hotel agents finish their tasks, they send their results to the customer [notify].*

- *If the customer is dissatisfied with the schedule and suggests a different date or destination city, she renotifies the travel agent about the revisions [retry].*
- *When the travel agent receives the revised requests, it applies the updates. If the effects of the revisions are significant and lead to the violation of a prior commitment by the travel agent, it may propagate the effect of revisions to the hotel or airline agent as appropriate [entertain update].*
- *When the hotel or airline agents receive a revision, they re-execute book room or book flight [entertain update].*
- *After the hotel or airline agent finishes its tasks, it sends the updated information to the customer if any of the prior commitments do not hold any more [renotify].*
- *Sometimes the customer's requirements simply cannot be satisfied by the airline agent [resign]. The travel agent tries another airline agent [notify], or fails for its customer [resign]. It depends on the customer's decision about whether to abort the order or renotify the travel agent of a new requirement.*

3. FORMALIZATION

Formalization can improve the clarity of the specifications to guide implementers and can increase one's confidence in the correctness of software. Clarity and assurance are significant here, because the interacting agents will, in general, be supplied by different vendors. For our purposes, a formalization should have both declarative and operational components. A declarative semantics describes *what* rather than *how*. Therefore, it can be applied to a variety of settings, not just those that satisfy some low-level operational criteria. However, operational considerations are essential for rigorously producing valid computations.

A commitment $\mathsf{C}(x, y, \pi)$ relates a debtor x, a creditor y, and a condition π [7]. The condition may involve predicates or commitments, allowing the commitments to be nested or conditional.

DEFINITION 1. *A commitment can be viewed as an abstract data type. The following operations are important—a commitment once created holds until canceled or released. Here, x and y denote roles, and c denotes commitments of the form $\mathsf{C}(x, y, \pi)$.*

- *Create(x, c) establishes the commitment c. The create operation can only be performed by the debtor of the commitment.*
- *Cancel(x, c) cancels the commitment c. Generally, cancellation of a commitment is followed by the creation of another commitment to compensate for the former one.*
- *Release(y, c) releases the debtor from the commitment c. It can be performed by the creditor and means that the debtor is no longer obliged to carry out his commitment.* ∎

3.1 Computation Tree Logic

The formal declarative semantics for the commitment patterns is given in Computation Tree Logic (CTL), a well-known branching-time temporal logic [2]. CTL formulas are given a semantics in terms of a CTL structure, which is a finite rooted directed graph whose paths correspond to different computations. One can imagine the graph being unraveled into an infinite tree.

DEFINITION 2. *A CTL structure M is a four-tuple $(S, R, \mathbf{I}, s_0)$, where S is a finite set of points; R is a binary relation on S, which gives the possible transitions between points; $\mathbf{I} : S \mapsto \wp(\Phi)$ is a labeling that assigns to each point the set of atomic propositions that are* true *at that point, where Φ is a set of atomic propositions; and s_0 is the root of S. That is, $\mathbf{I}(s_0) = \emptyset$. Further, R must be serial; i.e., $(\forall x : x \in S \Rightarrow (\exists y : y \in S \text{ and } (x, y) \in R))$.* ∎

The following Backus-Naur Form (BNF) grammar with start symbol L gives the syntax of our language. Below, *slant* typeface indicates nonterminals; $\longrightarrow$ and | are metasymbols of BNF specification; $\ll$ and $\gg$ delimit comments; the remaining symbols are terminals. Φ describes the basic elements of the logic, including commitments, operations on them, and communications.

- $L \longrightarrow$ *Prop* $\ll$atomic propositions and predicates$\gg$
- $L \longrightarrow \neg\ L \ll$negation$\gg$
- $L \longrightarrow L \wedge L \ll$conjunction$\gg$
- $L \longrightarrow \mathsf{A}\ P \ll$universal quantification over paths$\gg$
- $L \longrightarrow \mathsf{E}\ P \ll$existential quantification over paths$\gg$
- $P \longrightarrow L\ \mathsf{U}\ L \ll$until: operator over a single path$\gg$

The meanings of formulas generated from L are given relative to a model and a state in the model. The meanings of formulas generated from P are given relative to a path and a state on the path. The boolean operators are standard. Useful abbreviations include $\mathsf{false} \equiv (p \wedge \neg p)$, for any $p \in \Phi$, $\mathsf{true} \equiv \neg\mathsf{false}$, $p \vee q \equiv \neg(\neg p \wedge \neg q)$ and $p \rightarrow q \equiv \neg p \vee q$. The temporal operators A and E are quantifiers over paths. Informally, $p\mathsf{U}q$ means that on a given path from the given state, q will eventually hold and p will hold until q holds. $\mathsf{F}q$ means "eventually q" and abbreviates $\mathsf{true}\mathsf{U}q$. $\mathsf{G}q$ means "always q" and abbreviates $\neg\mathsf{F}\neg q$. Therefore, $\mathsf{E}p\mathsf{U}q$ means that on some future path from the given state, q will eventually hold and p will hold until q holds.

$M \models_s p$ means "M satisfies p at s" and $M \models_{P,s} p$ means "M satisfies p at s along path P." Below, for $s \in S$, $\mathbf{P}_s$ is the set of paths emanating from s.

- $M \models_s \psi$ iff $\psi \in \mathbf{I}(s)$, where $\psi \in \Phi$
- $M \models_s p \wedge q$ iff $M \models_s p$ and $M \models_s q$
- $M \models_s \neg p$ iff $M \not\models_s p$
- $M \models_s \mathsf{A}p$ iff $(\forall P : P \in \mathbf{P}_s \Rightarrow M \models_{P,s} p)$
- $M \models_s \mathsf{E}p$ iff $(\exists P : P \in \mathbf{P}_s \text{ and } M \models_{P,s} p)$
- $M \models_{P,s} p\mathsf{U}q$ iff $(\exists s' : s \leq s' \text{ and } M \models_{P,s'} q \text{ and } (\forall s'' : s \leq s'' \leq s' \Rightarrow M \models_{P,s''} p))$

Our formal language involves the usual boolean operations plus some temporal operations. $\mathsf{AG}p$ holds in s iff p is true at all future points on all paths through s. $\mathsf{AF}p$ holds at s iff p holds eventually on each path through s.

3.2 Communication Primitives

We consider the following communicative actions, each taking three parameters: sender x, receiver y, and content $\pi(\vec{v})$ (a predicate with a vector of domain arguments). Preconditions and postconditions are not specified for most actions, because these are determined from specific models, based on the commitments that apply in those models.

- *Inform*(x, y, π): x tells y that π is *true* and commits to y about the fact. This is the simplest case for a conversation.

  $\forall\ x,\ y,\ \pi,\ \vec{v}$: AG[*inform*$(x, y, \pi(\vec{v})) \rightarrow \mathsf{AF}$[*create*$(x, \mathsf{C}(x, y, \pi(\vec{v})))$]].
- *Request*(x, y, π): x asks y to do a task described by π. But there is no guarantee that y will accept.
- *Correct*(x, y, π): x requests y to repeat a task described by π. This enables x to send an updated request to y.
- *Retract*(x, y, π): x cancels a request π made to y.

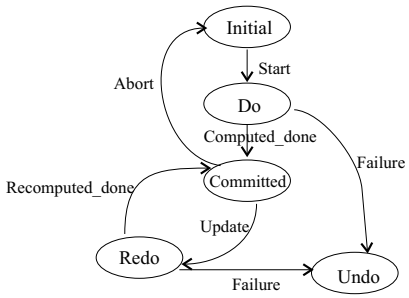

Figure 2: Task-related events and operations

- *Withdraw*(x, y, π): x applies after an *inform* and takes away a previous commitment to (π).

  $\forall\ x,\ y,\ \pi,\ \vec{v}$: AG[*withdraw*$(x, y, \pi(\vec{v}))$ $\rightarrow$ AF[*cancel*$(x,$ C$(x, y, \pi(\vec{v})))$]].

- *Refuse*(x, y, π): x would not accept y's request.

  $\forall\ x,\ y,\ \pi,\ \vec{v}$: AG[*refuse*$(x, y, \pi(\vec{v}))$ $\rightarrow$ AF[*release*$(x,$ C$(y,$ $x,\ \pi(\vec{v})))$]].

  When x refuses $\pi(\vec{v})$, y is released from its commitment to $\pi(\vec{v})$. That is, the commitment C$(y, x, \pi(\vec{v}))$ holds no more.

- *Reject*(x, y, π): x rejects y's information.

EXAMPLE 3. *Following Example 1, the travel agent notifies the customer about the flight, hotel and price information by using an inform. If the customer's requirements cannot be satisfied by the travel agent, it sends a refuse. Or if the flight schedule has been changed, the airline agent sends a withdraw.*

3.3 Elements of Behavior Models

Agent communications represent their external activities. The internal reasoning of agents is represented via tasks. The operation *do*, *redo*, and *undo* describe agents' internal reasoning. Events *start*, *update computed_done*, *recomputed_done*, *abort* and *failure* represent agents' task status, and may trigger communications.

The following domain-independent task-oriented propositions correspond to transitions in an agent that performs a given task. Intuitively, t would bring about $\pi(\vec{v})$. Figure 2 represents these as a statechart.

- *start*(x, t): x starts its task t execution for the first time, based on its own reasoning.
- *update*(x, t): x starts its task t execution for the second or later time, based on its own reasoning.
- *Computed_done*(x, t): x finishes task t execution successfully for the first time—intuitively, *computed_done* brings about $\pi(\vec{v})$.
- *Recomputed_done*(x, t): x finishes its task t execution successfully for the second or later time—intuitively, *recomputed_done* brings about $\pi(\vec{v})$.
- *Abort*(x, t): x aborts its task, causing it to resign from bringing about $\pi(\vec{v})$.
- *Failure*(x, t): x's task t fails because of an exception.
- *Do*(x, t): x performs task t (first try).
- *Redo*(x, t): x performs task t (second or later try).
- *Undo*(x, t): x compensates task t after the task t fails.

EXAMPLE 4. *Following Example 1, consider how the travel agent (TA) performs the operations of task makeOrder. When TA finishes makeOrder (do(TA,makeOrder)) for the first time, computed_done(TA, makeOrder) becomes true, which triggers inform primitive to notify the customer (CA) about the order. The CA may be dissatisfied with the order and send a correct. TA can redo makeOrder (redo(TA, makeOrder)). After TA finishes its task again, recomputed_done(TA, makeOrder) becomes true. Or if TA cannot finish its task, a failure(TA, makeOrder) event is generated and triggers TA to undo makeOrder via undo(TA, makeOrder).*

3.4 Formalizing Commitment Patterns

We now apply the specifications to formalize our commitment patterns. Here t is a task implementing predicate π, and *holds* indicates that the given assertion holds. $\underline{\triangle}$ represents *is defined as*.

- *Notify.* When x finishes task t successfully, which means *computed_done*(x, t) and $\pi(\vec{v})$ is true, then x informs y about $\pi(\vec{v})$.

  notify$(x,\ y,\ \pi)$ $\underline{\triangle}$ $\forall\ \vec{v}, t.$ AG[*computed_done*$(x,\ t)$ $\wedge$ *holds* $(\pi(\vec{v}))$ $\rightarrow$ AF*inform*$(x, y, \pi(\vec{v}))$]

- *Renotify.* When x finishes its tasks for the second or later time successfully, *recomputed_done*(x, t) becomes true. Then x checks if its current commitments hold with new results. If it is false, x informs y of the predicate associated with the new values and withdraws the previous commitment.

  renotify(x, y, π) $\underline{\triangle}$ $\forall\ \vec{v_1},\ \vec{v_2},\ t.$ AG[*recomputed_done* $(x,$ $t)$ $\wedge$ C$(x, y, \pi(\vec{v_1}))$ $\wedge$ $\neg$*holds*$(\pi(\vec{v_1}))$ $\wedge$ *holds*$(\pi(\vec{v_2}))$ $\rightarrow$ AF[*inform*$(x, y, \pi(\vec{v_2}))$ $\wedge$ *withdraw*$(x, y, \pi(\vec{v_1}))$]]

- *Entertain request.* When y sends x $\pi(\vec{v})$ as a request, this triggers x to reason about the request.

  entertain-request(x, y, π) $\underline{\triangle}$ $\forall\ \vec{v}$,$t.$ AG[*request*$(y,\ x,\ \pi(\vec{v}))$ $\rightarrow$ AF[*do*(x, t)]]

  When task t is completed, x may send out a notification, but that notification would be in the scope of another pattern.

- *Entertain update.* When y sends x a correct request, x performs its task about that again.

  entertain-update$(x,\ y,\ \pi)$ $\underline{\triangle}$ $\forall\ \vec{v_1},\ \vec{v_2},\ t.$ AG[*correct*$(y,\ x,$ $\pi(\vec{v_2}))$ $\wedge$ C$(x, y, \pi(\vec{v_1}))$ $\rightarrow$ AF[*redo*(x, t)]]

  Finally x uses *renotify* patterns to inform the new results, which would be in another scope of another pattern.

- *Resign.* When x's task t fails, or x's previous commitment doesn't hold and cannot be restored, x needs to withdraw the fact it committed to y.

  resign(x, y, π) $\underline{\triangle}$ $\forall\ \vec{v}$, t. AG[(*failure*(x, t) $\vee$ *abort*(x, t))$\wedge$ $\neg$ *holds*$(\pi(\vec{v}))$ $\wedge$ C$(x, y, \pi(\vec{v}))$ $\rightarrow$ AF[*withdraw*$(x, y, \pi(\vec{v}))$ $\wedge$ *undo*(x, t)]]

- *Retry.* If y sends x a reject, x redoes its task again, and tries to satisfy its consumer.

  retry(x, y, π) $\underline{\triangle}$ $\forall\ \vec{v}, t.$ AG[*reject*$(y, x, \pi(\vec{v}))$ $\wedge$ C$(x, y, \pi(\vec{v}))$ $\rightarrow$ AF [*redo*(x, t)]]

4. ESTABLISHING CORRECTNESS

To support our operational semantics for agent interaction, we require that the agents follow some generic behavior models. Specific behavior models are expressed as statecharts. Establishing the soundness of operational semantics is important to assure correctness of the system operation. The above CTL formulations of the

patterns are declarative. Statecharts provide operational semantics of agent behavior.

Statecharts are a well-known means in software engineering to specify concurrent computations [3]. A statechart is composed of *states* (OR-state, AND-state, and basic state) and *transitions*. Transitions are labeled by an expression of the form $E[C]/A$. Intuitively, event E triggers the transition if condition C is true when E occurs. As a result, action A is performed. Each of E, C, and A is optional. The states in our statecharts are abstract and correspond to sets of physical states of the underlying computation.

4.1 Composition of Agent Behavior

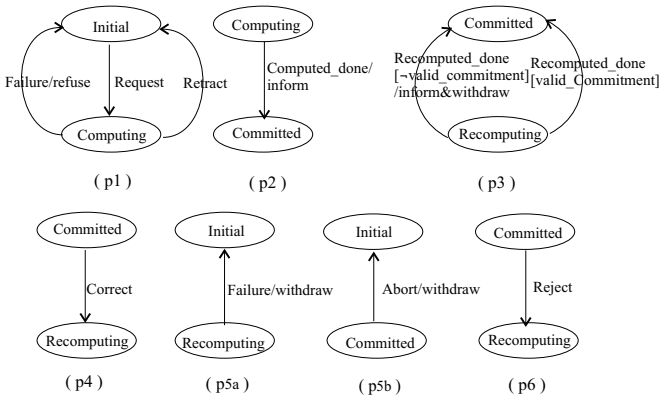

Figure 3: Commitment patterns: (p1) entertain request, (p2) notify consumer, (p3) renotify consumer, (p4) entertain update, (p5) resign, (p6) retry

Our patterns have two crucial properties.

- *Minimality*. Each pattern imposes the fewest reasonable restrictions to maximize design flexibility. Each pattern captures one agent interaction property, which is considered as minimal granularity for representing agent behaviors. Thus each pattern represents one agent property.
- *Composability*. The patterns may be assigned in any desirable combination by a modeler. Composition is important for engineering.

The following procedure is used to generate our agent behavior models from a set of commitment patterns.

1. Annotating the states and transitions for each pattern.
2. Translating each pattern into a statechart.
3. Merging multiple statecharts into a statechart to represent agent behavior.

4.1.1 Statechart Representation

First we need to carefully define states and transitions of each commitment pattern. A state in a statechart may have incorporated some activities. Figure 3 shows the statecharts we generate from our commitment patterns.

1. *Notify.* The transition *computed_done* signifies an exit from the *computing* state. The rule is fired when *computed_done* is received. Results are sent to the given consumer and the associated commitment is created and stored, which is performed by *inform* action. The *computing* state includes *do* activity. (Figure 3, p_2)

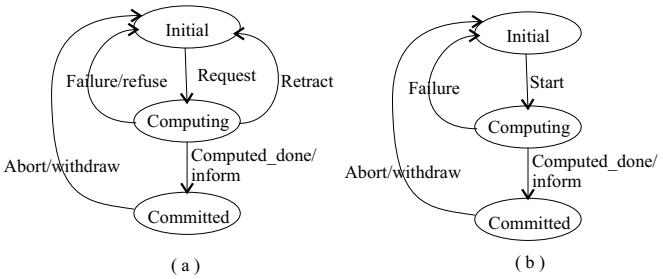

Figure 4: Transaction agent model, (a) pull mode (b) push mode

2. *Renotify.* The transition *recomputed_done* occurs when the system exits from *recomputing*, and the new fact holds. If a previous commitment to a consumer isn't valid, the previous fact doesn't hold, then it must send the updated information to the consumer and store the new commitment. The *inform* and *withdraw* actions need to be performed. Otherwise the previous commitment and the associated fact still holds. (Figure 3, p_3)
3. *Entertain request.* This consists of the *initial*, and *computing* states, and the *request*, *failure*, *refuse*, and *retract* transitions. The *computing* state includes the *do* activity. (Figure 3,p_1)
4. *Entertain update.* This consists of the *committed*, and *computing* states, and the *correct* transitions. The *recomputing* state includes *redo* activity. (Figure 3, p_4)
5. *Retry.* This consists of *committed*, and *recomputing* states and the *reject* transition. The *recomputing* state includes *redo* activity. (Figure 3, p_6)
6. *Resign.* This comes into effect when something happens to previous commitment and producer cancels the previous commitment. There are two cases. The first case is that the producer fails to perform its task again. This pattern consists of *recomputing*, *initial* states. When *failure* occurs, *withdraw* is sent to the given consumer and associated commitment is canceled. The second case is that the previous commitment doesn't hold. This pattern consists of *committed*, *initial* states. When an *abort* occurs, *withdraw* is sent to the given consumer and associated commitment is canceled. *undo* activity are in *initial* state. (Figure 3, p_{5a} and p_{5b})

The above patterns deal with some interactions between producer and consumer. In general, internal events may trigger a producer to perform its task; i.e., it may start a task by itself. This motivates us to build two further patterns p_7 and p_8. The *start* event signifies an exit from the *initial* to the *computing* state. This transition and related states build a pattern p_7. The *update* event signifies an exit from *committed* state to *recomputing* state. This transition and related states yield a pattern p_8. We lack the space to elaborate these here.

4.1.2 Merging

In order to merge a set of statecharts into a statechart, we introduce some concepts for the composition.

- *Identical state*. If the names of two states in different statecharts are the same, we take the two states to be identical. This is justified because all the patterns are based on the same agent behavior model. For example, *computing* state in the *notify* pattern statechart is the same as *computing* state in the *entertain-request* pattern statechart.

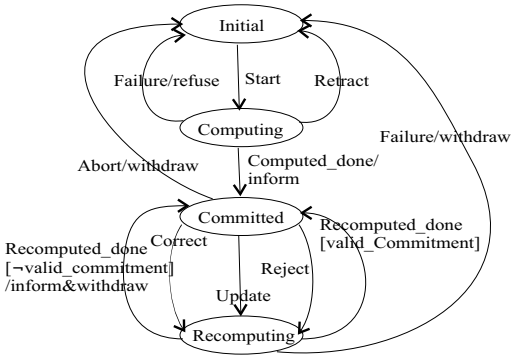

Figure 5: Basic agent behavior model for push mode (S_{a1})

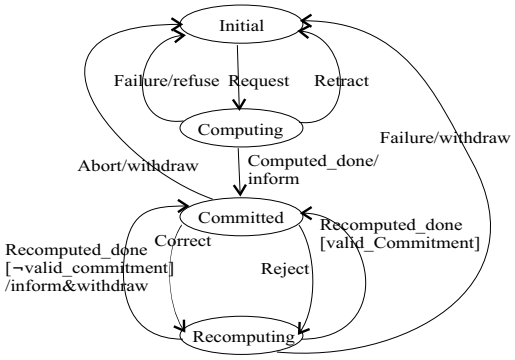

Figure 6: Basic agent behavior model for pull mode (S_{a2})

- *Identical transition*. Two transitions are identical if they have the same label, are triggered from the identical state, or lead into the identical state. This is useful for introducing a hierarchy.
- *Initial state*. The *initial* state should be defined.
- *Merging of two statecharts*. Union of two statecharts.

We apply the above approach to composing statecharts based on commitment patterns. First, we compose patterns p_1, p_2, and p_{5b} into a special agent behavior model, which acts as a simple transaction task agent, whose initial state is *initial* (Figure 4(a)). This is the pull mode. We compose patterns p_7, p_2, and p_{5b} to create the push mode. Both modes allow producers to cancel their commitments to consumers. Second, we compose patterns p_1, p_2, p_3, p_4, p_{5a}, p_{5b}, and p_6. The resulting statechart can be used as a pull mode of the basic agent behavior model (Figure 6). Third, we merge p_7, p_2, p_3, p_8, p_{5a}, p_{5b}, and p_6 to obtain the push mode of the basic agent behavior model (Figure 5).

Figure 5 and Figure 6 show the basic behavioral model expressed as statecharts. Agents who follow these behavioral models may invoke any capabilities (implemented in any manner), but the agents persist and include well-defined states in which they can reexecute a capability, and enter into commitments. The proprietary details of the capability or the agent's design are not revealed. On receiving a request or a control signal, an agent following basic behavioral model begins to perform a task. Upon completion, it sends the results to some selected consumers and commits to those results. Further events may cause the agent to reexecute its task. If the results change substantially to invalidate the agent's commitments, it announces the new results (canceling the old commitments and creating the new ones). The behavioral model just limits the agents' actions. However, the agent will have specific commitments that force it to carry out certain actions.

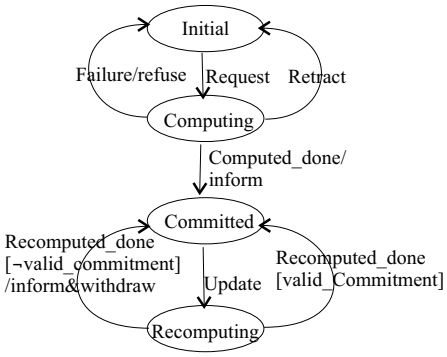

Figure 7: Monitor agent behavior model (S_{ma})

The patterns can be composed to yield a variety of behavior models. We can use the same method to compose p_1, p_2, p_3, and p_8 to obtain a monitor agent (Figure 7), which is a behavior model for an agent that periodically supplies the latest information to its consumer. The information is automatically delivered to its consumer without an explicit request each time after a request is entertained for the first time.

EXAMPLE 5. *We now apply the above behavior models on Example 1. In order to meet the design requirements, these agents may be given different behavior models. For example, the customer agent can take the transaction behavior model for push mode. The travel, hotel, and flight agents can take the transaction agent behavior model for pull mode. These models together can handle normal business scenarios. For dealing with revisions and exceptions, the customer agent can follow the basic agent behavior model for the push mode. Other agents too can follow the pull mode of the basic agent behavior model. The travel agent can adopt the monitor agent behavior model to provide a travel-related information service for the customer wherein it can send periodic updates to the subscribers.*

4.2 CTL Structure Derivation

To relate our operational semantics with the temporal logic specifications, we must produce a CTL structure from our statecharts. In general, because an external event may never occur during the execution of a statechart, an eventuality condition is always falsified if external events are required to reach the condition. To capture possible executions of a statechart, we separate out points in our CTL structure corresponding to the same basic configuration [8]. We introduce a point for each transition and add further members of the relationships into R. The following rules derive a CTL structure $M = (S, R, \mathbf{I}, s_0)$ from a statechart $S_c = (S, S_{bc}, T, c_0)$, where S is a set of states, S_{bc} is a set of basic configurations, T is a set of transitions, and c_0 is the initial basic configuration.

Starting from c_0, for each transition $t = (S_1, S_2, lbl)$ and $lbl = E[C]/A$, we create a point S' that is between S_1 and S_2, which are basic configurations of a statechart. S' has atomic propositions (denoted by $\{E,C\}$), which can be derived from E and C, we don't elaborate this here. Similarly action A generates atomic propositions (denoted by $\{A\}$) in S_2. We place the pair (S_1, S'), (S', S_2) into R. If S_2 is reached from different transitions and has different propositions, we separate the basic configuration S_2 into different points in the CTL structure. If E includes an external event, we place the (S_1, S_1) into R. The propositions in S_1 don't change.

This forces an execution from S_1 to S_1 if the external event doesn't occur. We add the name of basic configuration as propositions of a related point of M because it signifies some execution meaning. If there is a set of activities $\{acts\}$ in S_2, each activity generates an atomic proposition in state S_2. We denote the set of atomic propositions as $\{acts\} \subseteq \mathbf{I}(S_2)$. $\mathbf{I}(S_2)$=$\{A\}\cup\{$acts$\}\cup\{$name of $S_2\}$.

The following algorithm shows how to build a CTL structure from a statechart. This algorithm runs in time $O(|T||S_c|)$, where T is the set of transition and S_c is the set of basic configuration, because it visits each transition at most one time.

Algorithm 1 Build a CTL structure from an agent behavior model

Starting with statechart $S_c = (S, S_{bc}, T, c_0)$ to build a CTL structure $M = (S, R, \mathbf{I}, s_0)$.
Add s_0 to S of M, where $\mathbf{I}(s_0) = \{$name of $c_0\}$; s=c_0;
HAS_VISITED(c_0) = TRUE;
Build_CTL(s);
function Build_CTL(s)
for all $t = (s, s', lbl)$ of s, where $lbl = E[C]/A$ **do**
  if HAS_VISITED(t) **then**
    Add a binary relation $s \rightarrow t$ to R, continue;
  else
    HAS_VISITED(t) = TRUE;
    Add a new point t to S of M, where $\mathbf{I}(t) = \{E,C\}$;
    Add a binary relation $s \rightarrow t$ to R;
    if HAS_EXTERNAL_EVENT(t) **then**
      Add binary relation $s \rightarrow s$ to R;
    end if
    if HAS_VISITED(s') **then**
      HAS_SAME_PROPOSITION = FALSE;
      for all point v with $\{$name of $s'\} \subseteq \mathbf{I}(v)$ **do**
        if $\mathbf{I}(v) == \{A\}\cup\{$acts$\}\cup\{$name of $s'\}$ **then**
          Add binary relation $t \rightarrow s'$ to R.
          HAS_SAME_PROPOSITION = TRUE;
          break;
        end if
      end for
      if HAS_SAME_PROPOSITION **then**
        continue;
      end if
    end if
    Add a new point s' to S of M, where $\mathbf{I}(s')=\{A\}\cup\{$acts$\}\cup\{$name of $s'\}$;
    Add binary relation $t \rightarrow s'$ to R;
    HAS_VISITED(s') = TRUE;
    Build_CTL(s');
  end if
end for

To derive a CTL structure from the statechart of basic agent behavior model for push mode, consider the type of each transition. *Start*, *update*, *computed_done*, *recomputed_done*, *failure* and *abort* are internal events. *Reject*, *correct* are external events. The proposition *committed* and *recomputing* represent the existence of commitment in the associated points. Figure 8 shows the CTL structure derived from the statechart of Figure 5 by Algorithm 1. Figure 9 shows the CTL structure derived from the monitor agent behavior model (see Figure 7), where *request* and *retract* are external events, and *failure*, *computed_done*, *recomputed_done*, and *update* are internal events.

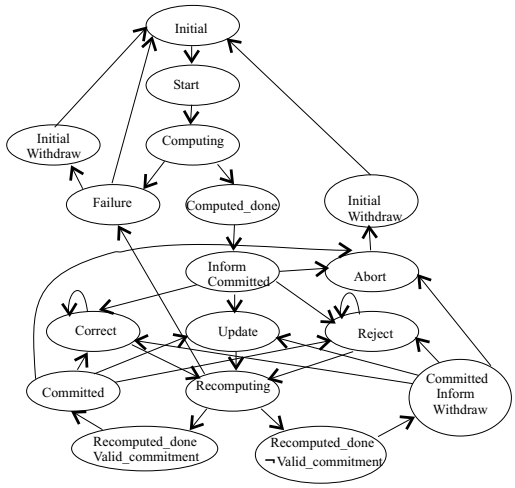

Figure 8: CTL structure for basic agent behavior model (S_{a1})

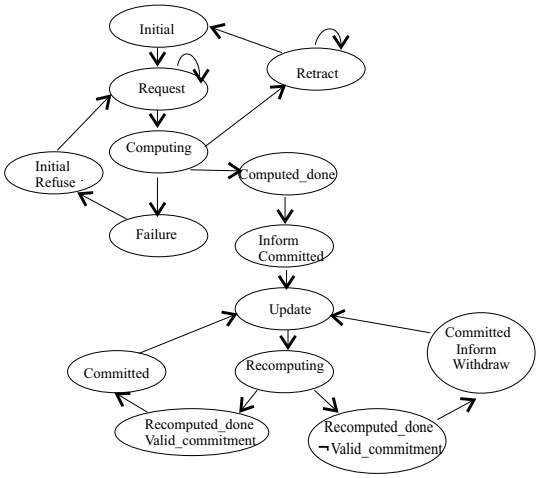

Figure 9: CTL structure derived from the statechart S_{ma}

4.3 Soundness of Operational Semantics

DEFINITION 3. *A statechart S_c is sound with respect to a formula ϕ iff ($\forall M$: (S_c generates M) $\Rightarrow$ $M, s_0 \models \phi$). A statechart S_c is sound with respect to a specification $\{\phi_1, \phi_2, \ldots, \phi_n\}$ iff S_c is sound with respect to each ϕ_i.* ∎

Now we apply the above to check the soundness of two agent behavior models. Notice that the proofs are straightforward, because the complexity is handled by the construction algorithm. Since each pattern is of the form $\mathsf{AG}[p \rightarrow \mathsf{AF}q]$, we can establish its truth in a given CTL structure simply by inspecting the given CTL structure to find the states where the antecedent (p) holds and then checking if the consequent (q) holds in all paths emanating from those states. These proofs could be readily mechanized by applying a *CTL model checker*, e.g., [2].

THEOREM 1. *Any agent that follows the monitor agent behavior model is sound with respect to any specification consisting of a subset of the patterns* {*entertain-request, entertain-update, notify, renotify*}.
PROOF SKETCH. *First construct the CTL structure corresponding to the monitor agent behavior model. Next, establish by inspection*

that each of the given patterns is satisfied in the constructed CTL structure. ■

THEOREM 2. *Any agent that follows basic agent behavior model is sound with respect to any specification consisting of the patterns* {*entertain-request, entertain-update, retry, notify, renotify, resign*}.
PROOF SKETCH. *As for the above, first construct a CTL structure corresponding to the basic agent model; next inspect it for the given patterns.* ■

Although the monitor agent satisfies p_1, p_2, p_3, p_4, it does not satisfy p_{5a}, p_{5b} and p_6. Thus the monitor agent is simpler than the basic agent in behavior. The basic agent has all of these properties of the monitor agent. Thus if ever a *reject* stimulus occurs, the agent eventually tries to satisfy the request. This is because the model satisfies the *retry* pattern. Further, the basic agent can cancel a commitment, because it implements the *resign* pattern.

EXAMPLE 6. *Theorems 2 and 1 establish that each of the behavior models assigned to the agents in Example 5 is correct with respect to the patterns given in Example 2. In this manner, we can abstract out the interactions and behaviors while leaving the designer full control of the domain-specific requirements.*

5. DISCUSSION

Conventional software techniques fall into one of two extremes, being either too rigid or too unstructured. Agents offer flexibility while dealing with autonomy, but can sometimes be nontrivial to apply. The proposed approach emphasizes the coherence engendered by commitments, while enabling agents and their interactions to be readily created from a library of patterns and behavior models. We demonstrated our approach for five typical behavior models. However, it can readily be applied to other such models. The correctness of these models with respect to patterns can be easily determined by using well-known model checking algorithms for CTL. As a result, the task of designing multiagent systems can be simplified in a rigorous manner using an extensible library of patterns and behavior models.

Decomposition, abstraction, and organization are three strategies for tackling complexity in developing agent systems agent-oriented software engineering [9]. Our approach contributes to each of these strategies. Patterns help decompose a multiagent system design problem. Patterns and behavior models provides abstraction at system and agent architectures, and help operationally relate the two layers. Patterns enable us to capture organizational requirements among the agents. Further, we address the design problem of how behavior models support various combinations of desired patterns.

Gaia is a leading methodology for agent-oriented analysis and design, especially when there is an organizational view on application scenarios [10]. Roles are key concepts with responsibilities, permission, activities, and protocols. In our approach, an agent bound to a role performs tasks and interacts with other agents. We emphasize commitments, which reflect organizational structure. Patterns are abstracted from agent interaction scenarios.

Mylopoulos et al. propose *Tropos*, a requirement- and goal-oriented methodology for agents [5]. Tropos emphasizes modeling requirements to improve flexibility and to narrow the semantic gap between system requirements and design. This complements our approach, which improves the flexibility of agent interactions by abstracting out patterns and behavior models.

Odell et al. propose a three-layer representation of Agent-Interaction Protocols (AIP) [6]. AIPs are defined as patterns representing both the message communication between agents, and the corresponding constraints on the content of such messages. Odell et al. extend UML to include richer role specifications that require modification of the UML sequence diagrams.

Klein and Dellarocas handle exception by employing a knowledge base of generic exception detection, diagnosis, and resolution expertise [4]. Our approach is complementary; special roles could be included in our approach with commitments by other roles.

Dignum and van Linder [1] propose a framework for social agents based on dynamic logic. Agents interact based on deontic relationship. Dignum and van Linder deal with the motivational attitudes such as wishes, goals, intentions, and obligations. This complements our approach. We focus more on operational semantics. The reasoning results represented by predicates with data values as the arguments trigger the interaction.

This work opens up some interesting directions. The approach is extensible in terms of commitment patterns and behavior models some of which we developed for business processes. However, other specific patterns and models are needed for other applications so that our libraries are more complete. A remaining challenge is to evaluate this approach on large software efforts.

6. REFERENCES

[1] F. Dignum & B. van Linder. Modelling social agents: Communication as action. In *Intelligent Agents III: Agent Theories, Architectures, and Languages*, pages 205–218. Springer-Verlag, 1997.

[2] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. B, pages 995–1072. North-Holland, 1990.

[3] D. Harel & E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[4] M. Klein & C. Dellarocas. Exception handling in agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 62–68, Seattle, 1999.

[5] J. Mylopoulos, M. Kolp, & J. Castro. UML for agent-oriented software development: The Tropos proposal. In *UML 2001: 4th International Conference on the Unified Modeling Language*, pages 422–441, 2001.

[6] J. Odell, H. V. D. Parunak, & B. Bauer. Representing agent interaction protocols in UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE)*, 2001.

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

[8] A. C. Uselton & S. A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *International Conference on Concurrency Theory*, pages 2–17, 1994.

[9] M. Wooldridge & P. Ciancarini. Agent-oriented software engineering. In S.-K. Chang, ed., *Handbook of Software Engineering and Knowledge Engineering*. World Scientific, 2001.

[10] M. Wooldridge, N. R. Jennings, & D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents & Multi-Agent Syst.*, 3(3):285–312, 2000.

[11] J. Xing, F. Wan, S. K. Rustogi, & M. P. Singh. A commitment-based approach for business process interoperation. *IEICE Transactions on Information and Systems*, E84-D(10):1324–1332, Oct. 2001.