Specifying and Monitoring Interactions and Commitments in Open Business Processes

William N. Robinson, Georgia State University
Sandeep Purao, Penn State University

Recent developments in Web services and associated standards offer the possibility of making composite, service-based systems a reality for many enterprises. As a recent Software Engineering Institute report states,

*We have crossed a threshold, where most of our large software systems can no longer be constructed as monoliths. ... They are now constructed as groups of interoperating systems ... made to interoperate through various forms of interfaces.*

Building and deploying such systems requires proactive behavior monitoring so that individual services and collaborations among groups of services can evolve in response to changes in the environment.

An important prerequisite for such proactive monitoring is the specification of business processes that describe detailed business requirements. In current process-specification techniques, such as workflow nets and process grammars (see the related sidebar), specification occurs either a priori or at runtime. We suggest a middle way that derives participant commitments from an a priori specification of a metalevel process.

Our research builds on that of Nirmi Desai and his colleagues, who argue for process specifications with commitment semantics. Specifying commitments for roles that participate in a business process provides “a basis for modeling the state of the interaction, thus allowing a variety of possible executions.” Specifying constraints over commitments is more flexible than constraints over concrete protocols because we can specify important relationships while deferring operational details.

This view is compatible with research on goal-based requirements engineering. Goals represent prescriptive statements of intent whose satisfaction requires the cooperation of agents (or active components) in the software and its environment. ... [Goals range] from high-level, strategic objectives to fine-grained, technical prescriptions that can be assigned as responsibilities of single agents.

So, we can conceptualize the agents participating in a business process as having goals. To satisfy these goals, agents commit to perform activities.
Specifying a business process with agent commitments, then, lets us partition the larger business process into interactions among agents, thereby improving reuse, semantics, and evolvability. We've derived an approach for specifying and monitoring these interactions and commitments. Toward that end, we've extended prior research to support

1. commitment- and message-based property specifications extended with communicative acts and temporal properties,
2. specification of inherited agent properties, and
3. event acquisition from the Common Base Event (CBE), a common open, heterogeneous, enterprise-monitoring framework.

Such an approach supports service and process evolution.

**Interaction Protocols**

An interaction protocol serves as the “rules of engagement” among interacting participants. It can include rules regarding possible actions that each participant may perform and the order in which these actions must be performed. It specifies an operational design for communication among two or more roles. An example is UML message sequence diagrams that define a protocol specification as messages, with their parameters and ordering (with references to data types) further described in UML classes.

Protocol design involves three main issues. The first is the protocol’s specificity, which involves the spectrum of possibilities along the abstraction-specificity dimension. A domain-independent protocol specification, for example, might involve patterns that represent interactions among services (for example, race conditions) that don’t require domain-specific information. On the other hand, a shipment protocol specification might be domain dependent.

The second issue is semantic content, which we can specify in terms of the nature of interactions. For example, a protocol specification might use only the basic request-reply mechanism, relying on the message content (for example, payment information) to provide the semantic content. Conversely, other mechanisms such as communicative acts can capture communications semantics. (We discuss this in more detail later in the article.)

Finally, protocol composability deals with how a designer specifies the overall business process by composing operational interaction protocols. It includes concerns such as mapping abstract roles and interactions to specific agents, and protocol overlaps and conflicts.

**Protocol Specification**

We illustrate protocol specification with a common purchasing process drawn from the Secure Electronic Transactions (SET) standard. This process involves collaboration among four main roles:

- The Customer wants to purchase items.
- The Merchant sells items.
- The Shipper transports the physical items to the Customer.
- The Payment Gateway authorizes payments.

Services fulfill the four roles, engaging in pair-wise conversations within the protocol specifications’ constraints.

The conversation between the Merchant and the Payment Gateway might involve these steps:

1. The Merchant Requests an authorization from the Payment Gateway.
2. The Payment Gateway Responds (or Refuses).
3. The Merchant Requests the Payment Gateway to capture the payment.
4. The Payment Gateway Informs the Merchant that it has captured the payment.

(Request, Respond, Refuse, and Inform are types of communicative acts.)

**Workflow Nets and Process Grammars**

Workflow nets allow specification of grammatical rules that enable runtime definition of the workflow (within specified rules). Process grammars provide a plan-based perspective in which the workflow is specified a priori. The tension between the two can be traced to

- the desire to specify a metalevel process to which the instances must adhere for correctness and
- the desire to let each process instance proceed in response to a unique context.

Process grammars are more suited to collaborative applications; workflow nets are exemplified in the Web services domain, with proposed standards such as BPEL (Business Process Execution Language). Although such standards are functionally correct, they’re inadequate from the perspective of open service environments.

**Reference**

Although we can characterize protocols as concrete operational descriptions (from the SET Purchase Process description), a more useful characterization is abstract protocols. An abstract protocol specifies common interactions within a family of protocols. Like an object-oriented class, an abstract protocol can be specialized through refinement. For example, Customer-Merchant interactions could be a specialization of an Order protocol between the Buyer and Seller. Interactions among the Customer, Merchant, and Payment Gateway could be a specialization of a Payment protocol among a Payer, Payee, and Payment Gateway. Such mapping can effectively lead to multiple inheritance for specific roles. For example, the Customer role might inherit interaction responsibilities as a Buyer from the Order protocol and as a Payer from the Payment protocol. Figure 1 presents role-based UML message sequence diagrams that specify abstract protocols for the SET Purchase Process.

### Protocol Composition
To specify the SET Purchase Process, a designer can compose these abstract protocols:

**Shipping protocol + Payment protocol + Order protocol**

An associated role mapping can then specify how the roles correspond between the SET Purchase Process and the abstract protocols. Here’s part of that mapping, where $x/Y$ means that role $x$ is substituted for the more abstract role $Y$:

- $Order.Buyer/Purchase.Customer$
- $Order.Seller/Purchase.Merchant$
- $Payment.Payer/Purchase.Customer$
- $Payment.Payee/Purchase.Merchant$
- $Payment.Gateway/Purchase.PaymentGateway$
- $Order.goods/Shipment.shipmentProp$

A composed specification such as the Purchase Process might still be nondeterministic. For example, an important concern in the SET Purchase Process is the ordering among multiple service requests as a role interacts with other roles as part of multiple protocol enactments. Without explicit rules, this remains nondeterministic; that is, a programmer can implement any ordering, including one that’s concurrent.

### Policies and Commitments
A protocol policy specifies role-based participants’ desired goals and commitments. We can express it in terms of protocol messages. For example, consider this Payment protocol goal: “After the Payer sends paymentInfo, eventually the Payer receives a receipt.” This goal seeks a certain response relationship among sent and received messages.

Commitments provide another means to specify policies. We can conceptualize protocols as fulfilling role commitments. That is, we can interpret interactions in a protocol as manipulating commitments such as creation, cancelation, or fulfillment (discharge). For example, the Payment protocol can lead to these commitments: “After the Payer commits to an Order, the Payer pays for the Order (discharging the commitment).” Desai

---

**Figure 1. Abstract protocols for (a) Payment, (b) Order, and (c) Shipping.** These Secure Electronic Transactions (SET) Purchase Process elements may be specialized for a particular application.
his colleagues view commitments as “a variety of contractual relationships … [that] enable manipulations such as delegation and assignment, which are essential for open systems.” Desai and his colleagues define commitment as follows:

A commitment $C(x, y, p)$ denotes that agent $x$ is obliged to agent $y$ for bringing about condition $p$. Three

Desai and his colleagues extend the definition to include a conditional commitment: $CC(x, y, p, q)$, which means that $x$ is committed to $y$ to bring about $p$ if $q$ holds. They elaborate operations on commitments, such as create, cancel, release, assign, delegate, and discharge.

Their perspective, therefore, requires annotating the protocols in Figure 1 with commitments. For example, the Order protocol shows that the "quote" message commits the Seller to providing the Buyer goods on the condition that the Buyer pays. Conversely, the Order protocol shows that the "acceptQuote" message commits the Buyer to paying the Seller on the condition that the Seller delivers the goods. Consistent with Desai and his colleague’s nondeterministic stance on protocol enactments, the Order protocol doesn’t specify an ordering on the pay and goods-delivery events. It simply indicates that the participants are mutually committed on the basis of two dependent conditions (pay and goods). This leaves the resolution to the designer or the programmer but ensures that mutual commitments are satisfied.

### Monitoring Policies and Commitments

Protocol policies thus specify desired outcomes on the basis of interactions among software systems (for example, Web services). Seen in this manner, protocol policies represent software requirements; that is, they present the opportunity for compliance monitoring as the software components execute.

### Runtime Requirements Monitoring

Our approach thus considers goals assigned to agents for fulfillment as requirements. Requirements monitoring continually evaluates requirements satisfaction by a target system. Consider the Purchase Process example: the Payee is responsible for the Payment protocol requirement, “After the Payer sends paymentInfo, eventually the Payer receives a receipt.” This requirement can be evaluated by a monitor. We can observe that after a single transaction, the requirement is satisfied. After monitoring a number of transactions, we can observe that the requirement is satisfied 90 percent of the time. Requirements monitoring thus lets us view how well a target system complies with its stated requirements.

A requirements monitor is a software system that analyzes the behavior of another (target) system and determines the satisfaction of the target system’s requirements by observing a stream of inputs ($IN_{mon}$). We can characterize a requirements monitor as a function that processes its input data stream to derive the status of requirements satisfaction:

$$MON(IN_{mon}) \rightarrow \text{Sat}(REQ)$$

In practice, $IN_{mon}$ comprises complex objects, such as the XML objects produced by log4J or the CBE. We express requirements as specified relationships among objects, over time. So, a requirements-monitoring language needs

- matching and filtering expressions for selecting relevant events from $IN_{mon}$;
- expressions for representing requirements following object-oriented concepts similar to UML;
- relational, temporal, and real-time expressions for precisely describing object models; and
- metarequirements for representing expressions about property satisfaction.

A variety of projects support some form of requirements monitoring, including those that address some aspect of Web service policy monitoring. We build on this research by providing OCLTM, an expressive requirements-monitoring language that satisfies the four needs and employs the familiar notation of UML and its Object Constraint Language (OCL). We describe OCLTM in more detail later.

### Requirements-Monitoring Activities

SerMON (Service Monitor), our extension of the ReqMON requirements-monitoring framework, supports the specification and monitoring of protocol policies. SerMON is engaged through four main activities.

First, an analyst defines protocol policies as goal models and assigns goals to agents (for example, Web services).

Second, the analyst applies tools to analyze, compile, and deploy requirements monitors (using an integrated development environment such as Eclipse with plug-ins) for the goals. A compiler
The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:

context SoftwareObject
def: m1 : OclMessage = receivedMessage(a())
def: m2 : OclMessage = receivedMessage(b())
inv: bRespondsToA: response@0d:0h:0m:3s(m2,m1)

It specifies that after a SoftwareObject object receives a message b(), it must receive a subsequent message a() in response within the specified time-out of 3 seconds. Otherwise, the invariant is violated.

**Purchasing Policies**

To provide runtime monitoring of protocol policies, we first translate the protocols and their associated policies into OCLTM. Then, as the business processes execute, the monitor system continuously evaluates the requirements.

The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:

context SoftwareObject
def: m1 : OclMessage = receivedMessage(a())
def: m2 : OclMessage = receivedMessage(b())
inv: bRespondsToA: response@0d:0h:0m:3s(m2,m1)

It specifies that after a SoftwareObject object receives a message b(), it must receive a subsequent message a() in response within the specified time-out of 3 seconds. Otherwise, the invariant is violated.

**Purchasing Policies**

To provide runtime monitoring of protocol policies, we first translate the protocols and their associated policies into OCLTM. Then, as the business processes execute, the monitor system continuously evaluates the requirements.

The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:

context SoftwareObject
def: m1 : OclMessage = receivedMessage(a())
def: m2 : OclMessage = receivedMessage(b())
inv: bRespondsToA: response@0d:0h:0m:3s(m2,m1)

It specifies that after a SoftwareObject object receives a message b(), it must receive a subsequent message a() in response within the specified time-out of 3 seconds. Otherwise, the invariant is violated.

**Purchasing Policies**

To provide runtime monitoring of protocol policies, we first translate the protocols and their associated policies into OCLTM. Then, as the business processes execute, the monitor system continuously evaluates the requirements.

The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:

context SoftwareObject
def: m1 : OclMessage = receivedMessage(a())
def: m2 : OclMessage = receivedMessage(b())
inv: bRespondsToA: response@0d:0h:0m:3s(m2,m1)

It specifies that after a SoftwareObject object receives a message b(), it must receive a subsequent message a() in response within the specified time-out of 3 seconds. Otherwise, the invariant is violated.

**Purchasing Policies**

To provide runtime monitoring of protocol policies, we first translate the protocols and their associated policies into OCLTM. Then, as the business processes execute, the monitor system continuously evaluates the requirements.

The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:

context SoftwareObject
def: m1 : OclMessage = receivedMessage(a())
def: m2 : OclMessage = receivedMessage(b())
inv: bRespondsToA: response@0d:0h:0m:3s(m2,m1)

It specifies that after a SoftwareObject object receives a message b(), it must receive a subsequent message a() in response within the specified time-out of 3 seconds. Otherwise, the invariant is violated.

**Purchasing Policies**

To provide runtime monitoring of protocol policies, we first translate the protocols and their associated policies into OCLTM. Then, as the business processes execute, the monitor system continuously evaluates the requirements.

The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:

context SoftwareObject
def: m1 : OclMessage = receivedMessage(a())
def: m2 : OclMessage = receivedMessage(b())
inv: bRespondsToA: response@0d:0h:0m:3s(m2,m1)

It specifies that after a SoftwareObject object receives a message b(), it must receive a subsequent message a() in response within the specified time-out of 3 seconds. Otherwise, the invariant is violated.

**Purchasing Policies**

To provide runtime monitoring of protocol policies, we first translate the protocols and their associated policies into OCLTM. Then, as the business processes execute, the monitor system continuously evaluates the requirements.

The commitments of the protocol policies that Desai and his colleagues specified (see Figure 1) associate with messages in role-based sequence diagrams. We can translate each commitment to one OCLTM invariant on the class receiving (or sending) the message. In general, we can represent a protocol commitment policy as one or more OCLTM invariants over the classes related through the policy.

Consider a requirements-monitoring perspective on the purchasing policies. We can specify purchasing business-process requirements in UML, using sequence, state, and class models with OCLTM annotations. SerMon can then monitor the OCLTM invariants.

For example, consider the Payee’s OCLTM context (see Figure 3a), in which the payfine invariant:
The first OCL\textsubscript{TM} invariant is a message invariant. It specifies that after the Payer receives a message indicating that its payment wasn’t properly authorized (\textit{authNOK}), the Payer will pay a fine to the Payee.

The second and third invariants in Figure 3a provide an alternate representation that explicitly asserts commitments. The \textit{commitToFine} invariant specifies that after a Payer receives an \textit{authNOK} message for a payment, the Payer is committed to paying the fine. The \textit{committedFine} invariant asserts the committed relation when the fine is paid, thereby satisfying the \textit{commitToFine} invariant. (Commit is asserted with the OCL\textsubscript{TM} extension \texttt{new}, which is necessary for such metalevel assertions about OCL-level messages.)

Message invariants and commitment invariants are both reasonable representations. The former is efficient, whereas the latter abstracts messages as commitments, which simplifies specification and runtime commitment reasoning and increases flexibility. For example, commitments can be delegated. So, the Payer can delegate the payment commitment to another agent to fulfill its responsibility. The monitor can follow such commitment chains when such relationships are specified.

### Extending Protocol Policies with Communicative Acts

To enrich the semantics of abstract protocols (which can then lead to simplification of requirements expressions), we turn to communicative acts (also called speech acts), similar to those suggested elsewhere.\textsuperscript{15} A communicative act conceptualizes speaking or communicating as action taking. We can classify it with such types as Request, Inform, and Refuse, as we mentioned in the section “Protocol Specification.”

Consider Figure 3b, which represents a portion of the Buyer agent from the SET Purchasing Process. After receiving a quote, the Buyer must respond with an Accept or Reject message, as specified by the \textit{respondToQuote} invariant. This specific requirement fulfills the more abstract requirements on agent behavior (not shown).

Consider the second and third invariants in Figure 3b, which simply assert that a communicative act occurs—a \textit{Quote} is a \textit{Propose} act, whereas a \textit{QuoteAccept} is a \textit{Commit} act. The communicative acts are also referenced in the constraints of the Buyer superclass Agent. An Agent class invariant specifies that a Commit or Disagree act must follow a Propose act. We can check the Buyer behavior against this invariant by analyzing the communicative-act assertions generated by the Buyer’s second and third invariants. So, we can apply requirements associated with abstract classes to their specialized subclasses—a monitor

---

**Figure 3.** Object Constraint Language constraints for the (a) Payer and (b) Buyer. The constraints for the Payer indicate fine assessment requirements. The Buyer class is a subclass of Agent.
can check the compliance of both abstract and specialized requirements.

This view of interactions among participants incorporates message semantics in addition to the semantics embedded in the message content. For example, in addition to indicating that the message envelope contains the quote information, the communicative act specifies the message itself as one proposing a course of action—receiving a quote implies the Propose act, as specified by the receiveProposal invariant. Our agent constraints rely on a comprehensive set of communicative acts, derived from analysis of prior research including the use of speech acts for e-commerce transactions and for agent interactions. The set includes Direct, Propose, and Query.

Superclass requirements apply to their subclasses through shared abstractions. In the example from Figure 3b, the Agent class invariants constrain the communicative-act assertions of subclasses; the Buyer subclass asserts these as appropriate. So, the Agent invariants apply to the Buyer subclass. Invariants can also constrain class operations (referenced in the sendMessage and receiveMessage clauses). Subclasses can specify their operations as refinements of their superclass operations, thereby receiving the benefit of the superclass operation invariants.

Analysts benefit from invariant requirements on abstract classes because they can check subclasses for requirements compliance with little additional effort. Of course, subclasses can also add specialized requirements, including overriding those specified in the abstract classes, in which case both the abstract and specialized requirements must be checked. More generally, the OCL properties (specifying important requirements) in the class hierarchy provide a means to describe business processes and their protocols at multiple abstraction levels. Hierarchical requirements specified in this manner support the definition, refinement, and analysis of protocols necessary for their evolution.

### Using Monitors

To illustrate the runtime evaluation of requirements with monitors, we return to the example of the Payer’s fine-assessment requirements. Evaluation of the Payer requirement, payFine, involves four activities:

1. The target system containing the Payer class executes.
2. The monitor acquires IN_mon, including updates about Payer objects.
3. The monitor analyzes IN_mon.
4. The monitor evaluates requirements satisfaction.

The monitor acquires IN_mon using available instrumentation techniques from the executing software. Events are generated at monitoring points placed in the target program or in its context (for example, the Web server). SERMON defines listeners for common instrumentation frameworks (log4j, CBE), which are used for monitoring. With each input, the monitor analyzes and updates requirements evaluations, using an algorithm that evaluates primitive (nested) properties and then propagates the values for composite properties.

Table 1 shows the results; it illustrates property evaluation for an input sequence that satisfies the message-based requirement payFine. IN_mon consists of two significant events in sequence. The first occurs when the Payer object receives an authNOK message. The monitor processes the event, asserting that the authNOK property, \( P_1 \), is satisfied, which causes the after scope, \( S_A \), to become open. The second event, payFine, satisfies the eventually clause. Given that the after scope is still open, \( P_2 \) is satisfied, and with it the entire payFine property, \( P_3 \), is satisfied. For the commitment-based requirement committed-Fine, when \( P_3 \) is satisfied, the Payer’s fulfillment of its commitment is asserted.

Practitioners might find value in our monitoring approach, which lets them

- Monitor invariants at runtime, using commonly available instrumentation;
- Generalize message-based properties to commitment-based properties, which increases specification and monitoring flexibility with temporal properties and enriches semantics through communicative acts; and
- Aggregate monitoring of pair-wise agent interactions to determine multiagent protocol-policy satisfaction.
Using our Eclipse-based tools, they can apply these techniques to existing Web service systems.

We’ve applied ReqMon to assess the satisfaction of clinical goals for a small group of emailing patients.16 We’ve also applied our SerMon extensions to Web service systems, including the SET Purchase Process described in this article. In these cases, the use of OCLTM, property inheritance, and commitment-based properties has simplified monitoring.

To judge our approach’s usefulness, consider the absence of protocol policies and monitors. With high-volume transactions such as the Purchase Process or electronic funds transfer, recovery from execution failures can be demanding. Monitoring participant commitment can provide a more effective approach to recovery by revealing local adaptations necessary to participant commitments, without jeopardizing the eventual outcome.

Our extensions in the form of temporal properties and communicative acts can significantly help to identify these adaptations. Ongoing long-term monitoring can also reveal fundamental policy failures. This, in turn, can provide insight into protocol evolution through mechanisms such as varying the temporal dependencies and allowing or restricting message semantics. Such mechanisms remain on our agenda. Our approach is also consistent with research that seeks to automate runtime monitoring and negotiation of service-level agreements.

References