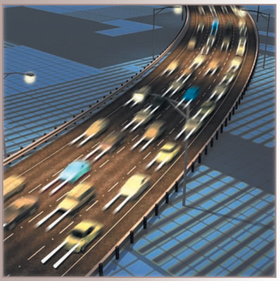

Editors: M. Brian Blake • *mb7@georgetown.edu*
Michael N. Huhns • *huhns@sc.edu*

Incorporating Events into Cross-Organizational Business Processes

Because Web-scale processes are inherently cross-organizational, they require the robust enactment of interactions among autonomous parties. However, specifying the processes involved is difficult. To overcome this obstacle, the authors use a business protocol that lets the applicable events and responses vary based on where the process is deployed and the infrastructure and IT applications installed therein. Treating events and business logic as separate concerns also yields clearer models and improves reusability. The authors describe the architecture and tools and outline a methodology by which each participant in a process can define, detect, and respond to events.

Payal Chakravarty
IBM Tivoli

Munindar P. Singh
North Carolina State University

Web-based business workflows and processes aren't just distributed — they're also inherently cross-organizational. Even a simple Web process such as shipping a product involves more than one autonomous party. Moreover, Web-scale distribution makes such processes vulnerable to the vagaries of the real world. So how can we implement Web-scale processes for robust enactment?

Event-driven architecture (EDA) offers hope by helping IT systems identify and respond to exceptions and opportunities.[1] However, conventional process models complicate event incorporation — events of interest often depend on specific configurations of physical sensors and effectors, and responses to events often depend on the participants' business goals. A delayed shipment of medications to a warehouse might be acceptable, for example, but a delayed shipment to a disaster area might not.

In our EDA, we address the engineering challenge of incorporating events in a Web process in a reusable manner — that is, without hard coding events into a process model. Specifically, we capture process models as protocols, which express an interaction's business logic in terms of commitments among the agents representing the various participants. Protocols provide a framework for capturing each agent's local policies by expressing responses to events in a particular usage setting.[2,3] In this article, we introduce our EDA as well as a methodology and tools for specifying events that naturally reflect business agreements among participants.

Motivating Example

For motivation, consider the familiar situation in which a sender hires a shipper to ship a package to a receiver. This situation can be realized via a business process, with three parties exchanging messages by using agreed-on formats and meanings. In a typical enactment, the three parties would forge a deal, and the shipper would perform accordingly, culminating in its delivery of the package to the receiver. However, several exceptions (that is, failures) can occur — for example, the package could be lost or damaged on the way. The only way to ensure robust shipping is to monitor the relevant events and, if and when they occur, respond appropriately. To monitor events presumes the necessary infrastructure is in place — the shipper might install RFID sensors at one or more checkpoints, which would make the enactment visible to the shipper's IT system. The checkpoints would generate events that help the shipper track progress. Should the package fail to materialize at a checkpoint as expected, the shipper can take corrective action, which might take the form of the shipper notifying the sender and possibly paying a penalty. The sender in turn might supply another package to deliver to the receiver or offer the receiver a refund or some other form of compensation.

Business Protocols and Commitments

An orchestration — as in the Business Process Execution Language (BPEL; http://docs.oasis-open.org/wsbpel/2.0/) — specifies a process as a procedure (consisting of tasks with control and data flows among them) to be executed by a central engine. A choreography — as in ebBP (the ebXML Business Process schema; http://docs.oasis-open.org/ebxml-bp/2.0.4/HTML/) or the Choreography Description Language (www.w3.org/TR/ws-cdl-10/) — specifies a process in terms of messages among participants. In contrast, a business protocol describes interactions (realized as messages) in terms of how they create and manipulate commitments among participants.[2,3] Physical interactions such as shipments are included in protocols on par with messages because they're important for discharging commitments.

In our EDA, we model an autonomous business partner as an *agent*; a *commitment* is a directed conditional obligation from a `debtor` to a `creditor` agent. In our approach, each commitment is stated in terms of agents `debtor` and `creditor` and formulas `precondition` and `condition`.[4] The term `commitment(debtor, creditor, precondition, condition)` means that `debtor` commits to `creditor` that if `precondition` becomes true, `debtor` will enact `condition`. When `precondition` holds, `debtor` becomes unconditionally committed. Furthermore, the agents involved might manipulate a commitment — the debtor by delegating it to a new debtor, and the creditor by assigning it to a new creditor. Here, `precondition` and `condition` resemble the preconditions and effects of services (as in conventional markup for services; www.w3.org/Submission/OWL-S/) but carry contractual weight, which traditional approaches tend to disregard.

A business protocol specifies messages exchanged among two or more interacting roles. For each role, the protocol yields a skeleton consisting of rules that capture applicable temporal constraints and meanings. The business meanings of messages naturally correspond to the conditions they bring about as well as how they create, discharge, or manipulate commitments. To create an agent that plays a role, we flesh out its skeleton via additional rules to capture the necessary decision-making policies.

Viewed as a protocol, our shipping scenario involves three roles: receiver, sender, and shipper.[2] The top of Figure 1 shows the important (asynchronous) messages involved in shipping — specifically, the quote from shipper to sender reflects a commitment that if the sender pays the specified charges, the shipper will deliver the packages. The sender's acceptance commits it to paying and the shipper to delivering when the payment is received. Each party's policies determine whether to enter into commitments and how to carry them out.

Events

Events can be normal or exceptional. An example of a normal event is the timely delivery of a package. Exceptions can be anticipated or unanticipated — an anticipated exception, such as a delayed shipment, is one that business analysts have considered but that the IT system

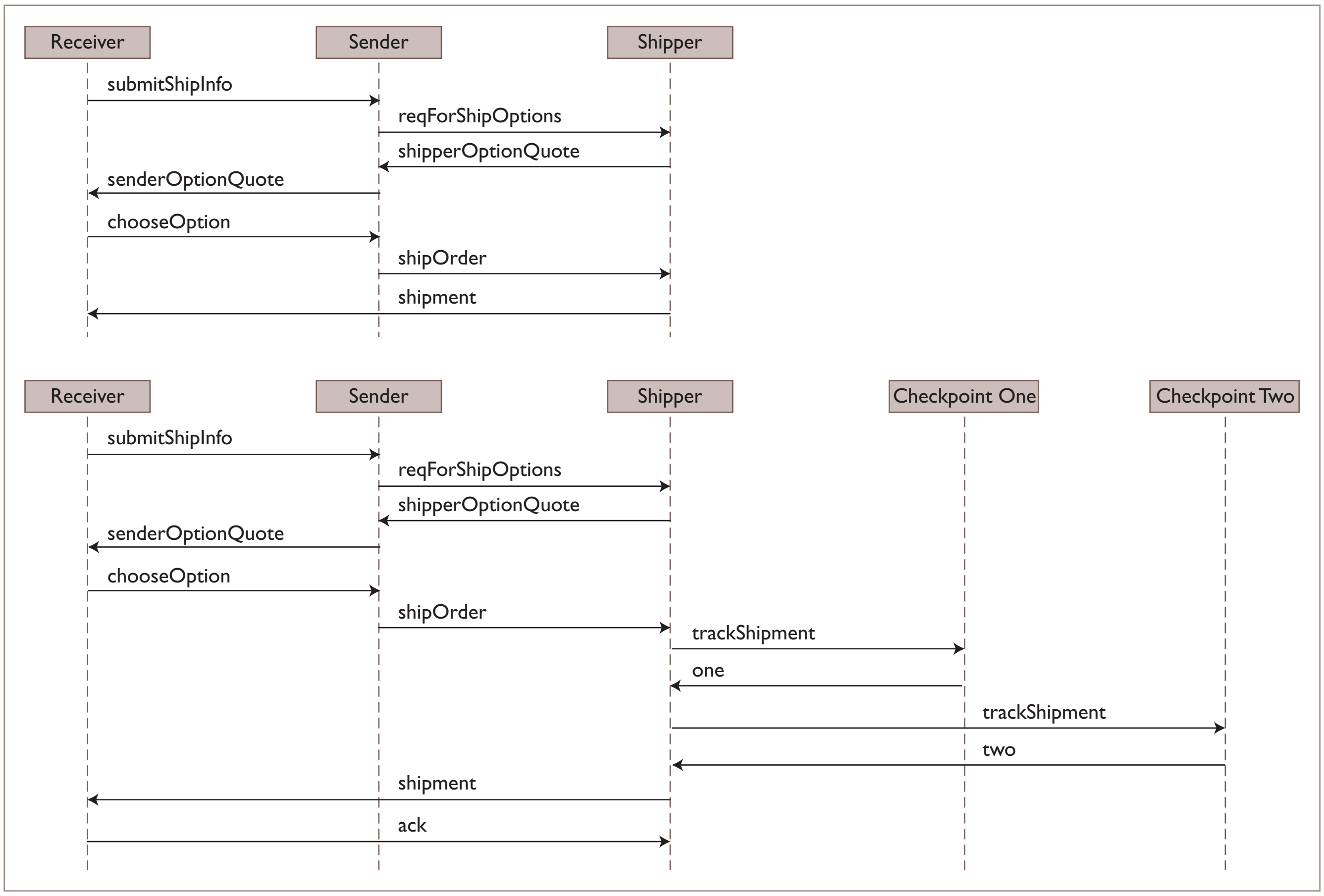

Figure 1. Shipping scenario. The rule-based specification is definitive and is configured by using checkpoints with sensors. The horizontal lines with arrows represent messages.

doesn't properly handle, whereas an unanticipated exception is one that business analysts have failed to model. Our approach shows how to address anticipated exceptions in modeling or during configuration.

An EDA provides a way of organizing systems that sense, analyze, and respond to events. For business processes, *sensing* involves receiving events from multiple sources (sensors, software applications, and such), *analyzing* involves deciding a response (perhaps by aggregating such events), and *responding* involves updating expectations and modifying executions. For simplicity, we assume that the IT infrastructure is robust: messages aren't lost, and sensors don't fail, but shipments might be delayed or damaged.

This infrastructure produces a *simple* event — in the bottom of Figure 1, for example, checkpoints (equipped with RFID or other sensors) can detect packages and produce messages informing participants of significant events. Here, `one` means that the shipper receives an acknowledgment from Checkpoint One before a specified timeout. Similarly, `two` and `ack` arrive from Checkpoint Two and the receiver, respectively. The connection between the shipment and the checkpoints is through the sensors and isn't explicit in this diagram. We specify a simple event instance through its name and parameter values[1] (typical parameters include a transaction ID, when and where the event occurred, and other domain-specific content).

A *complex* event is expressed as a pattern over simple events — for example, we can express `shipping success` as the pattern that `one`, `two`, and `ack` occur in sequence. Notice that we could alternatively model success simply as `ack`, but checkpoints are introduced precisely to enable fine-grained tracking: robustness isn't merely about succeeding but about tracking progress all the way to success. `Shipping failure` (an exception) means that at least one of the messages fails to arrive before its timeout: assuming the sensors don't fail, this means the shipment has encountered some trouble.

In business settings, events relate naturally to commitments. Because of the importance of commitments to a cross-organizational process, we want to specify events so as to monitor the creation, progress on, and discharge or violation of various commitments. A `debtor` would monitor a commitment to ensure that it isn't violated or to make amends if it is; in our running example, the shipper might track its shipment all the way to delivery. However, a `creditor` would monitor a commitment to make sure its internal plans — and its own dependent commitments to others — aren't adversely affected or to take corrective action if necessary.

Protocols, Skeletons, and Policies

Throughout this article, we use a simple language based on event-condition-action (ECA) rules for protocols, skeletons, and policies (to emphasize important concepts, we use informal notation throughout):

```
WHEN event
IF condition
THEN action
```

This type of rule is understood as follows: whenever an event occurs, the rule interpreter performs (in some arbitrary order) the actions of all matching rules whose conditions hold.

The following pseudocode snippets from Figure 1 are typical for messages in a protocol, which can constrain how messages interrelate — for example, a request should lead to a quote. The `IF` part includes zero or more gating requirements; a policy placeholder leaves some discretion with the agent playing the appropriate role (a "?" indicates a variable):

```
WHEN requestForShipOptions(?ID, ?item,
?from, ?to)
IF quoting policy (placeholder)
THEN sendShipperOptionQuote(?ID, ?item,
?from, ?to, ?charges)
```

More important, a protocol specifies each message's meaning, which includes any operations performed on commitments — for example, a quote creates a commitment that if payment is made, the item will be delivered:

```
WHEN sendShipperOptionQuote(?ID, ?item,
           ?weight, ?from, ?to,
           ?charges)
IF true
THEN CREATE(commitment
             (shipper, sender,
              shipOrder(?ID, ?charges),
              shipment(?ID, ?item, ?from,
                       ?to)))
```

Other messages might mean delegating, assigning, canceling, or releasing a commitment and would be expressed via similar rules.

In addition, rules for processing commitments are implicitly incorporated in every protocol, so, for example, if

```
shipOrder(?ID, ?charges)
```

occurs, the commitment created by this rule would become unconditional:

```
commitment(shipper, sender, true,
shipment(?ID, ?item, ?from, ?to))
```

Rules such as this one express the protocol's perspective. Agents adopt the various roles in a protocol to enact it — for example, a customer could be the receiver and a merchant the sender, but they might use other protocols to determine parameters such as item, price, and address. As mentioned earlier, a role skeleton consists of rules generated from the protocol specification to reflect the role's perspective. Specifically, a skeleton includes the rules that define the meaning of each message the role sends or receives along with constraints on message parameters and occurrence, especially with respect to other messages.[2] An agent can choose its messages, but when it sends one, its meaning is firm: a message means what the protocol specifies. Notice that commitments provide an interface between choreography and orchestration: they serve externally as contracts and internally as agent goals. The messages an agent sends and receives affect its commitments.

A process implementer specifies an agent playing a role by augmenting the role skeleton with rules to describe both the policies the protocol has left as placeholders and responses to commitments. The policies capture the business logic to determine whether and with what parameters an action must occur. If the policy fails, the agent won't perform the consequent

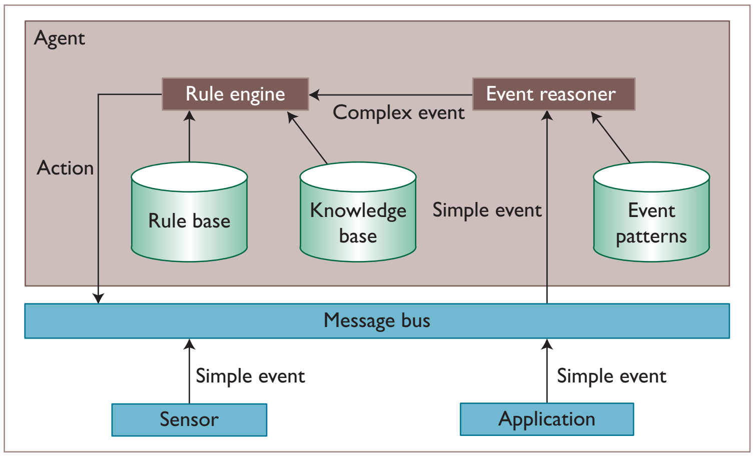

Figure 2. Our event-driven architecture. An agent representing a business partner enacts its part of a process; it receives events from sensors and applications; its actions are physical actions or communications to other agents.

action. In some cases, as in sending a quote, there are no ramifications on correctness because the protocol gives the agent discretion in this regard. In other cases, as in a commitment, an agent might violate a commitment, which would usually be considered noncompliant behavior.

An agent's policies specify how it responds to different exceptions. If a package is delayed or lost, for example, the receiver could send a reminder, the shipper could initiate a search for the package, the sender could ship a duplicate set of goods, and so on. Such responses make the business process more robust. Notice that exceptions are treated on par with other events:

```
WHEN ShippingFailure(?ID, ?item, ?from,
?to)
IF resendPolicy(?ID, ?item, ?from, ?to)
THEN resendShipment(?ID, ?item, ?from,
?to)
```

Our approach ensures that policies address normal and exceptional events. Policies in general — and responses to exceptions in particular — are specific to each agent's business goals, but they fall into common patterns from extended transaction models, such as redoing an entire transaction, retrying a subtransaction that fails, or undoing a subtransaction that succeeds.[5]

Event-Based Architecture for Process Enactment

As Figure 2 shows, our agent architecture separates but composes event reasoning with business rules. An agent primarily consists of a rule engine and an event reasoner: simple events arrive at the reasoner, which maintains (partially detected) complex events in its pattern store.

Event patterns that don't mention a simple event aren't affected by its occurrence and those that do are simplified by it. Suppose the shipper's pattern store initially contains `shipping success` and `shipping failure`, as defined earlier. When the shipper receives `one`, its stored patterns are simplified — `shipping success` reduces to the pattern of `two` preceding `ack`. Similarly, `shipping failure` reduces to the pattern of either `two` or `ack` failing to occur before their corresponding timeout occurs. Now, if `two` fails to occur before its timeout, `shipping success` reduces to false and `shipping failure` to true.

Patterns remain in the store until they reduce to true (occurred) or false (impossible). As soon as a pattern reduces to true, the event reasoner notifies the rule engine about the corresponding complex event's occurrence, and the engine applies the matching business rules, executing any actions as appropriate. An action is treated as a simple event and might appear in event patterns. Thus, we achieve process enactment (by each partner) in two parts: detecting event patterns and applying rules to respond to them.

Our event representation and reasoning is based on temporal logic,[6] which naturally expresses complex events involving temporal sequencing, conjunctions, and disjunctions. Typical business rule languages (such as Jess[7]) support only conjunctions, so mapping an event pattern into a rule language would cause a blow up in the number of rules (for disjunction) and intermediate conditions (for sequencing), and would make the resulting rules difficult to maintain.

Methodology

Our EDA requires three information sources for each agent: a knowledge base, a rule base, and an event pattern store. The knowledge base contains domain models (not in our scope) as well as facts (true conditions) that change as the enactment proceeds. Agent rules are based on the skeletons of the roles an agent plays and on its policies.

Although our approach leaves the policy specification open, it provides a simple structure for policies based on the protocol enacted and a systematic examination of normal and exceptional events. Of course, what a par-

Table 1. The main steps for incorporating events into business process models.

#	Description	Output artifacts	Approach
1	Jointly with business partners, identify a protocol to be enacted and the role each partner's agent plays	A protocol specified in terms of roles, message meanings, and constraints on messages	Selected from a repository or composed[2]
2	Model business entity as an agent playing a role in the protocol and examine the commitments in which it participates as debtor or creditor	A set of commitments	Assisted by our tool
3	Identify simple events; describe normal, exceptional, and irrelevant events informally	A set of simple events and descriptions of normal, exceptional, and irrelevant events	Based on domain knowledge and runtime configuration
4	Specify normal, exceptional, and irrelevant events; generate refined normal and exceptional patterns	Patterns that formalize normal and exceptional events, refined with respect to irrelevant patterns	Assisted by our event pattern generator tool
5	Write a policy for each skeletal rule derived from the protocol and any exceptions	Policy specifications in pseudocode that capture agent's decisions in response to normal and exceptional events	Based on business goals

ticipant considers normal or exceptional depends on its business goals. Table 1 outlines a methodology by which to specify normal and exception patterns and the concomitant policies.

Coming up with exceptions is often difficult for humans. Our event-based approach provides a natural way to overcome this challenge. A designer can specify various exception patterns to be monitored directly, but we provide a way to reduce the designer's effort. Some event patterns might be useless for monitoring because they're prevented through some system property — a designer with such knowledge can specify the irrelevant event patterns. Not specifying the irrelevant patterns is harmless, but work is wasted by monitoring events that won't produce useful information. As a simple example, the pattern in which `two` precedes `one` would be impossible, assuming reliable sensing and fast communications. Alternatively, the designer might assume that if `two` does occur before `one`, then it isn't an exception because it tells us where the shipment is. Similarly, we might consider `ack` preceding `two` or `ack` preceding `one` as impossible.

We refine normal and exceptional patterns by conjoining them with the complement of each irrelevant pattern, thus eliminating irrelevant possibilities from consideration. Because refined patterns don't consider executions that can't occur, they resolve more quickly at runtime. The event pattern generator helps designers build such refined patterns, which are often complex and thus difficult to build by hand.

Application and Evaluation

In Web-scale processes, business partners can view the "same" business occurrences differently — for example, the shipper and receiver focus on a shipment's initial and final time points, respectively. Events can help reconcile these perspectives during process enactment. A shipper that tracks a shipment through delivery can thus incorporate the receiver's perspective and thereby offer improved value to its customers.

Because our contribution is in terms of the naturalness of configuring business processes for robust enactment, we focused our evaluation on some specific cases.

Responding to an Exception

Let's start by applying our methodology on a variant of our running example. Here, the shipment's temperature must be controlled within a specified range (as for food or medicine).

Identify protocol and role. In this case, the protocol specifies the shipping protocol. Let's assume our agent is the shipper.

Examine commitments. The shipper's commitments include delivering the goods in a manner that meets the stated temperature requirements.

Identify events of interest. Specifically, we're

Related Work in Exception Handling

The benefits of separating exceptions from normal business logic are well known,[1] but doing so with interactions is novel to our approach. Holger Brocks and his colleagues[2] propose a rule-based approach for exception handling, but their setting isn't open: their exceptions aren't based on events arising during interaction among business partners. By contrast, we incorporate complex events, formalized in temporal logic, into event-condition-action rules. We also show how to reuse business logic across configurations.

Existing agent-based approaches for business process and workflow management consider coordination at a messaging (lower) level of abstraction[3] or negotiation about business goals at a deeper reasoning (higher) level of abstraction.[4] John Thangarajah and his colleagues[5] develop inference rules in which an agent can soundly abort tasks and drop intentions: such rules can potentially underlie the policies described in the main text. Business protocols occupy the trade-off space in the middle, where richer meanings are encoded but the reasoning about them is performed by designers (equipped with tools) not automated agents. By separating commitments from internal decision making, business protocols facilitate compositionality and reuse.

Chris Dellarocas and Mark Klein[6] classify exceptions in terms of process types: such knowledge could potentially be used to inform the exception patterns formulated in our approach.

Our tools and enactment engine are implemented in Java Platform, Enterprise Edition — specifically, we use Jess for the rule engine, rule base, and knowledge base; Java Message Service for messaging; and Java Regular Expressions for event reasoning. Our architecture and methodology would apply even if another event language was used.

References

1. L. Zeng et al., "Policy-Driven Exception-Management for Composite Web Services," *Proc. 7th IEEE Int'l Conf. E-Commerce Technology*, IEEE CS Press, 2005, pp. 355–363.
2. H. Brocks et al., "Flexible Exception Handling in a Multi-Agent Enactment Model for Knowledge-Intensive Processes," *Proc. IEEE/WIC/ACM Int'l Conf. Intelligent Agent Technology*, IEEE CS Press, 2005, pp. 479–482.
3. M.B. Blake, "Coordinating Multiple Agents for Workflow-Oriented Process Orchestration," *Information Systems & E-Business Management*, vol. 1, no. 4, 2003, pp. 387–405.
4. J. Dang and M.N. Huhns, "Concurrent Multiple-Issue Negotiation for Internet-Based Services," *IEEE Internet Computing*, vol. 10, no. 6, 2006, pp. 42–49.
5. J. Thangarajah et al., "Aborting Goals and Plans in BDI Agents," *Proc. 6th Int'l Joint Conf. Autonomous Agents and Multiagent Systems*, ACM Press, 2007, pp. 8–15.
6. C. Dellarocas and M. Klein, "A Knowledge-Based Approach for Handling Exceptions in Business Processes," *Information Technology & Management*, vol. 1, no. 2, 2000, pp. 155–169.

interested in simple events, normal events, exceptions, and irrelevant patterns:

- Sensors placed in two containers tell us a series of simple events of three types: t_i = temperature of container i falls outside allowed range; f_i = freezer of container i fails self-test; and c_i = the package is in container i.
- For a normal event, each container's temperature remains within range and doesn't fail the self-test.
- Exceptions include the following: the shipment container's temperature falling outside the allowed range (call this "too hot"), and the container's refrigeration unit failing before use (call this "dead fridge").
- An irrelevant pattern would place the package in the second container before the temperature of the first container is out of range or the first container fails the self-test before its temperature goes out of range.

Next, we need to drill down even further.

Specify normal events, exceptions, and irrelevant patterns. We then determine that

- In the normal pattern, neither t_1 nor t_2 occur; that is, the container's temperature doesn't go out of range.
- Exceptions include the following: for "too hot," at least one container's temperature going out of range, and for "dead fridge," the first container's freezer failing before the package is in the container.
- In the irrelevant pattern, c_2 occurs before t_1 or f_1 before t_1.
- I, the complement of the irrelevant pattern, is complicated and not of expository value.
- For the refined normal pattern, the normal pattern is conjoined with I.
- Refined exceptions include all the patterns conjoined with I.

Notice that such formulas can be unwieldy if expanded. In general, it would be quite difficult for a designer to produce accurate formulas directly, but our methodology simplifies this specification by helping factor out the problem's key aspects.

Write a policy for each event and exception. The "too hot" exception corresponds to detecting a violation of a commitment, but only after that violation has already occurred; a corrective response could be to resend the goods to ensure discharging the commitment. A response to the "dead fridge" exception might be to replan the shipping, even if it delays the shipment but doesn't spoil the goods. Of course, a container's refrigeration unit appearing to work prior to shipping doesn't guarantee that it would continue to work during shipping. Thus, the "too hot" exception might occur anyway and would remain to be handled as appropriate.

Exploiting an Opportunity: Combining Shipments

Monitoring business events helps facilitate the correct response to opportunities. Consider a case in which a shipper receives multiple orders for shipments to the same receiver and wants to combine them over a certain period of, say, one day or a half-day. Here, each of the one or more protocol instances enacted between the same parties creates a commitment on the shipper to deliver the goods. From the interaction's standpoint, there is no change to the business logic, but the shipper can reduce its costs by consolidating shipments (and still continue to discharge its commitments).

The normal event here is that zero or one order of a certain service level and for a given destination has been received at a designated time point. The "exception" in this case is really an opportunity: the shipper receives two or more orders of a certain service level and for a given destination at the designated time point. Because no irrelevant patterns exist, there's no need to refine the patterns. A possible policy is expressed via a rule that causes all the shipping orders (for one destination and of one quality of service) pending at the designated time point to ship out together in one shot. Sending a consolidated shipment thus discharges the commitments corresponding to each shipment.

Our agent-based EDA provides the key aspect of enactment to our research program of modeling and managing business processes in terms of interactions.[2] Separating a process's core business logic from events in a particular instantiation enables each participant's local policies to be much more naturally authored and applied. Importantly, complex events can differ across instantiations, even if the core business logic stays the same. An important challenge, which we defer to future work, is how to specify policies for an agent in a manner that provides guaranteed coverage to the commitments of the business partner it represents.

Acknowledgments

This research was partially supported by the US National Science Foundation under grant IIS-0139037. We substantially revised and extended this article from a previous version that appears in *Proc. 6th Int'l Joint Conf. Autonomous Agents and Multiagent Systems*, ACM Press, 2007, pp. 1261–1263. We thank Brian Blake, Mike Huhns, James Lester, Xiaosong Ma, and Ashok Mallya for comments on previous versions, and Rob Kremer, James Harland, and John Thangarajah for helpful discussions.

References

1. D.C. Luckham, *The Power of Events*, Addison-Wesley, 2002.
2. N. Desai et al., "Interaction Protocols as Design Abstractions for Business Processes," *IEEE Trans. Software Eng.*, vol. 31, no. 12, 2005, pp. 1015–1027.
3. M. Winikoff, W. Liu, and J. Harland, "Enhancing Commitment Machines," *Proc. 2nd Int'l Workshop Declarative Agent Languages & Technologies*, LNAI 3476, Springer, 2005, pp. 198–220.
4. N. Desai et al., "Representing and Reasoning about Commitments in Business Processes," *Proc. 22nd AAAI Conf. Artificial Intelligence*, AAAI Press, 2007, pp. 1328–1333.
5. A.K. Elmagarmid, ed., *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.
6. M.P. Singh, "Distributed Enactment of Multiagent Workflows: Temporal Logic for Service Composition," *Proc. 2nd Int'l Joint Conf. Autonomous Agents and Multiagent Systems*, ACM Press, 2003, pp. 907–914.
7. E.J. Friedman-Hill, *Jess in Action*, Manning, 2003.

Payal Chakravarty is a software engineer with IBM Tivoli. Her interests include event processing. Contact her at chakravp@us.ibm.com.

Munindar P. Singh is a professor of computer science at North Carolina State University. His interests include interaction-oriented abstractions for business process management. Contact him at singh@ncsu.edu.