# An Event-Driven Approach for Agent-Based Business Process Enactment*

Payal Chakravarty†
North Carolina State University
Department of Computer Science
pchakra@ncsu.edu

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

## ABSTRACT

Agents enacting business processes in large open environments need to adaptively accommodate exceptions. Work on multiagent approaches can flexibly model business processes. This paper proposes an event-driven architecture that enriches such models with events resulting in a more robust and proactive system. Specifically, we place this architecture in a business process framework based on protocols and policies, where agents' behaviors are specified via rules. The contributions of this paper include (1) an event-driven architecture, (2) a specification language that combines event logic with rules and (3) a methodology to incorporate events into a process (such as for fine-grained monitoring), (4) a way to manage subscriptions to simple events efficiently. This approach is applied on a well-known business scenario.

## 1. INTRODUCTION AND BACKGROUND

Business process modeling and enactment in service-oriented computing environments is extremely complex due to the requirement of supporting autonomy, heterogeneity and dynamism. Multiagent approaches to modeling and enacting business processes that offer flexibility have been around for some time. This paper builds on this work but incorporates a more natural way to detect and handle exceptions by incorporating events into business rules thereby enriching the agility of existing models.

Event-Driven Architecture (EDA) handles events by managing and executing rules of the form: *WHEN reality deviates from expectations, THEN update expectations and initiate response* [1]. The primary characteristic of a system built using this architecture is to sense, analyze, and respond to events.

This paper combines event-driven architecture and a business process framework based on protocols and policies called OWL-P [2], where agents' behaviors are described via rules and commitments. Each agent is an *Event Processor* that correlates data from multiple agents to detect complex event patterns, thus capturing a fine-grained view of a high level interactions of a business protocol. On detection of such exceptions and events, appropriate action is taken by the agent.

### 1.1 Running Example

We illustrate our approach with an example interaction in which a shipper ships some goods to a customer [2]. The shipping protocol ends with a *Shipment* message from the *shipper* to the *receiver*. But this high-level interaction does not capture the fact of whether the shipment was actually delivered or not. For example, the package may have been lost or damaged in transit. Events help get details of how the high-level interaction of was actually carried out. The *shipper* now tracks the shipment through some checkpoints (e.g., RFID sensors). When all such sensors return a positive response and finally the receiver too acknowledges the receipt of the shipment, the *shipper* can confirm that the shipment did really go through.

## 2. APPROACH AND ARCHITECTURE

Our approach makes protocols event driven. A simple event occurs on the receipt of a *message* or on the creation of a *commitment*. An event source could be other participating agents in the protocol or sensors (e.g., RFID sensors). The agents are event processors who detect complex events by making sense of simple events using pattern matching. Consequently, they react to these detected complex events by triggering rules.

### 2.1 Event Processing Language

This is the specification language that combines event logic with rules. An *event type* specifies the template to represent a simple event. Events are represented as tuples of data as described in [3]. We represent an event by the tuple $e(\mathit{message}, \mathit{source})$ where message and source are *predefined public attributes*.

We use the event-driven linear temporal logic introduced by Singh [4] as the *event pattern language* [3] to formalize complex event patterns in our system. An example *complex event* or *dependency* [4] is $E = a \cdot b \vee \bar{a} \vee \bar{b}$. The *complex event* $E$ is generated from the *aggregation* of simple events $e_1$, $e_2$, and $e_3$. Here, $\wedge$ and $\cdot$ are *pattern operators* that express the relationship between the events. We wish to differentiate between the occurrence of an event and the conditions that

*The second author's research was partially supported by the National Science Foundation under grant DST-0139037.
†Student author.


*AAMAS'07,* May 14–18, 2007, Honolulu, Hawai'i, USA.




978-81-904262-7-5 (RPS) 

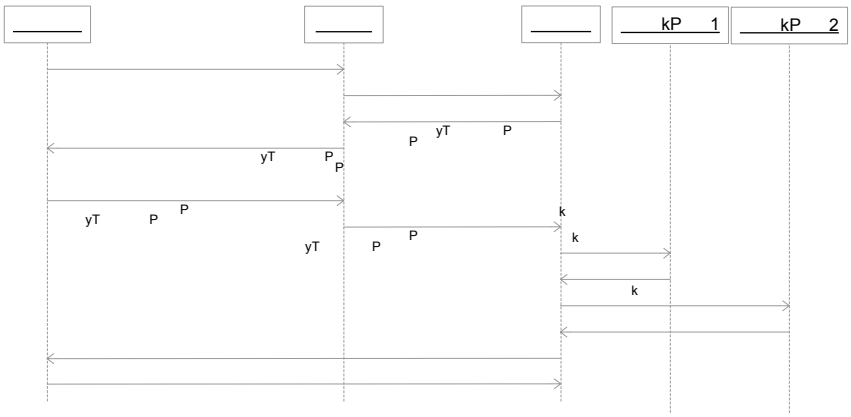


**Figure 1: Shipping Protocol refined to include tracking at two checkpoints**

would result in its non-occurrence. Hence, we create a pair of complementary dependencies, $E$ and $\overline{E}$.

In the present example, $E = a \cdot b$, and $\overline{E} = \overline{a \cdot b} = \overline{a} \vee \overline{b} \vee b \cdot a$. The second dependency represents exceptions in the system. Continuing with our running example, we could express a successful shipment by the complex event, $E = e_1(\textit{shipSuccess}(\ldots), P_1) \cdot e_2(\textit{shipSuccess}(\ldots), P_2) \cdot e_3(\textit{shipSuccess}(\ldots), \textit{Receiver})$. Or, more simply as $E = e_1 \cdot e_2 \cdot e_3$.

Business *action rules* are represented as condition ⇒ action. On incorporating events into the typical OWL-P rules [2] for the Shipping Protocol introduced in Section 1.1, the rule would look like:

```
contains(KB, shipmentSendEventProp)
  ⇒ send(receiver, shipment(itemID))
```

where *shipmentSendEvent* is the complex event formed by aggregating simple events as follows: $\textit{shipmentSendEvent} = \textit{shipOrderEvent}(\textit{shipOrderMsg}, se) \cdot P_1\textit{SuccessEvent}(\textit{shipSuccessMsg}, P_1) \cdot P_2\textit{SuccessEvent}(\textit{shipSuccessMsg}, P_2)$

## 2.2 Architecture

Figure 2 shows our event-driven agent architecture. The agent primarily consists of a rule base (RB), a knowledge base (KB) and an event processor (EP). Incoming messages from several event sources are processed by the EP. On detecting an event, the EP stores that event in the KB. The KB then activates rules in the RB.

The event processor is the heart of the agent made up of the following modules:

**Adapter.** This is the entry point for messages. It transforms incoming messages to events conforming to the event type and forwards them to the Pattern Matcher.

**Pattern Matcher.** This is responsible for aggregating simple events to detect complex event patterns stored in the *pattern repository*. It has two components. The *pattern repository* is initialized during the configuration of the EP. It maintains the current state of the event and exception patterns. The *residuator* performs residuation [4] on the events and exceptions in the pattern repository and updates its state. If any of the dependencies or event patterns evaluate to true, that event is asserted to the knowledge base. Further, the Pattern Matcher activates the subscriber.

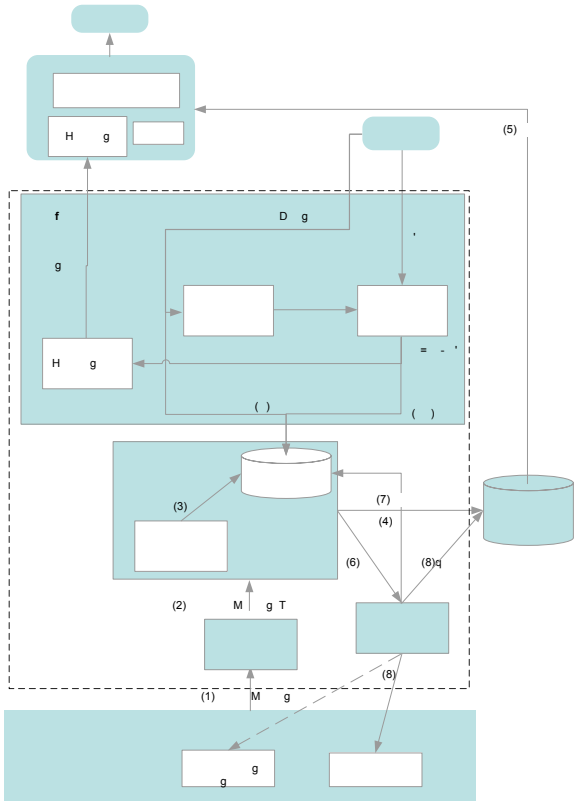


**Figure 2: Event-Driven Agent Architecture**

**Subscriber.** This consults the pattern repository to determine the simple events that might lead to an occur-

rence of a complex event, and subscribes to them.

# 3. EVENT PROCESSING

Before events can be processed, the EP undergoes an initial set up phase called Event Configuration.

## 3.1 Event Configuration

Event configuration involves initializing the Pattern Repository with complex event patterns and exception patterns derived from them.

**Deriving Exception Patterns**: The event processor derives patterns of exceptions from an event pattern $E$ in two steps. (1) Complementing the complex event $E$ to generate all possible conditions corresponding to $E$ not happening. This pattern is represented as $\overline{E}$. (2) Refining $\overline{E}$ by eliminating conditions that cannot happen or are uninteresting. This step requires designer inputs and is performed by the Exception Pattern Filter or EPF. The EPF takes a set of events $E'$ as input from the protocol designer and performs $\overline{E} - E'$ to generate $E_{ex}$.

**Generating Exception Handling Rules**: Detected exceptions are handled by triggering the agent's internal policies. The rule to handle the exception ShipException is

```
contains(KB, ShipExceptionProp) ⇒
    ShipExceptionPolicy(itemID)
```

This rule implies that if the event *ShipException* exists in the knowledge base the *ShipExceptionPolicy* of the agent should be consulted to take a decision of the necessary action. The ShipExceptionPolicy is an internal business decision of the shipper. It might decide act in several ways like resend the package and thereby reinitiate this entire process or send a message to the customer informing it of a delay or delegate the control of action to $P_1$ who reported an error, who in turn might track the shipment via another checkpoint leading to a nested shipping protocol.

## 3.2 Runtime Event Monitoring

Once the event processor has been configured at the beginning of the protocol enactment, the agent is set to start monitoring. The runtime monitoring occurs as specified in Figure 2. Continuing with our running example, let us consider the shipper agent who has committed to ship an item to the receiver, has sent out the shipment and is tracking the package. After deriving its *role* in the shipping protocol, the events ShipmentEvent ($E$) and ShipmentException ($E_{ex}$) are derived and stored in the pattern repository. The shipper needs to match the event pattern $E = e_1 \cdot e_2 \cdot e_3$ and exception pattern $E = \overline{e_1} \vee \overline{e_2} \vee \overline{e_3}$. It has subscribed to $P_1$ in order to get information about the status of the package. When $P_1$ receives it and forwards it to the next destination, it sends the message $shipSuccess(itemID)$ to the shipper. The sequence of action that occurs after the receipt of the message (1) as shown in Figure 2 is described as follows:

**Transformation**: Event transformation (2) is the process of converting a message received to a simple event conforming to the event template described in Section 2.1. For example, message *shipSuccess*(*itemID*) from $P_1$ is transformed into event $e_1(\textit{shipSuccess}(\textit{itemID}), P_1)$. The adapter then forwards this event to the pattern matcher.

**Pattern Matching**:The pattern matcher performs residuation (3) on all complex event patterns in the pattern repository with this event. The complex event patterns in the pattern repository thus reduce to:

$E/e_1 \doteq (e_1 \cdot e_2)/e_1 \wedge (e_2 \cdot e_3)/e_1$ or $E/e_1 \doteq e_2 \cdot e_3$

$e_1$ is then stored in the knowledge base (4). Also if any complex event pattern evaluates to true due to the occurrence of $e_1$, that will also be stored in the KB. Next the pattern matcher activates the subscriber (6) who will consult (7) the pattern repository, analyze event trends, and selectively subscribe (8) to different event sources.

**Action**: If the occurrence of a simple event $e_1$ or a complex event activates (5) some business rule in the rule base, the consequent action is taken.

**Exception Detection and Recovery**: Exceptions are detected as soon as any of the Exception patterns evaluate to true or event patterns evaluate to a false. Let us consider the circumstance in which after $e_1$ occurs and then $e_2$ fails due to a missing package at $P_2$, i.e., $\overline{e_2}$ occurs. On residuation of all the current patterns in the pattern repository with $e_2$, the exception $E_{ex} = e_2 \vee e_3$, will evaluate to true. Once an exception $E_{ex}$ is detected, it is stored in the KB and that in turn activates the $E_{ex}$*Policy* in the rule base.

**Selective Subscription**: While evaluating a complex event pattern, if a particular simple event that is a part of the aggregation occurs, looking at the pattern the agent can tell what to expect next and from whom to expect it. It will then subscribe to the expected event source and can unsubscribe itself from all sources that will have no further influence on its reasoning. The advantage of this is when the agent is a part of a huge distributed network where there are thousands of event sources, it does not need to waste its resources on subscribing to sources that would not assist in its decision making.

# 4. CONCLUSION

Agents involved in business process enactment in distributed environments should be able to detect opportunities and exceptions and handle them efficiently. This paper described an approach for agent-based business process enactment that takes care of these issues. Specifically, the paper describes an event-driven agent architecture, a specification language that combines event logic with rules and a method to subscribe selectively to low level event sources. Agents built using this architecture track complex patterns of events and exceptions by aggregating simple events and activate rules in order to react to exceptions automatically. The idea has been demonstrated by using the protocol based business process enactment model of OWL-P. The feasibility of this approach has been demonstrated with the help of a prototype implementation.

# 5. REFERENCES


[1] K. M. Chandy. Sense and respond systems. In *Proceedings of the 31st International Computer Measurement Group Conference*, 2005.

[2] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engg*, 2005.

[3] D. C. Luckham. *The Power of Events*. Addison-Wesley.

[4] M. P. Singh. Distributed enactment of multiagent workflows: Temporal logic for service composition. In *Proceedings of AAMAS*, 2003.