EVIE - AN EVENT BROKERING LANGUAGE FOR THE COMPOSITION OF COLLABORATIVE BUSINESS PROCESSES

Tony O'Hagan[1], Shazia Sadiq[1]and Wasim Sadiq[2]
[1]*School of Information Technology and Electrical Engineering*
The University of Queensland, St Lucia, QLD 4072
Brisbane, Australia

[2]*SAP Research Centre*
133 Mary Street, QLD 4000
Brisbane, Australia

Keywords: Business Process Management, Enterprise Application Integration, Service Oriented Computing.

Abstract: Technologies that facilitate the management of collaborative processes are high on the agenda for enterprise software developers. One of the greatest difficulties in this respect is achieving a streamlined pipeline from business modelling to execution infrastructures. In this paper we present Evie - an approach for rapid design and deployment of event driven collaborative processes based on significant language extensions to Java that are characterized by abstract and succinct constructs. The new language is positioned within an overall framework that provides a bridge between a high level modelling tool and the underlying deployment environment.

1 INTRODUCTION

Process enablement is firmly grounded as a key objective in enterprise systems. However, with current business trends towards outsourcing and virtual alliances, the importance of business process integration has strongly emerged. Business process integration (BPI), understood as the controlled sharing of data and applications within and across an enterprise boundary, is considered to be one of the main strategies of many organizations. BPI offers new business opportunities, benefits of maximizing operational productivity, improved business resource utilization, and supports businesses in gaining competitive advantages through customer and supplier satisfaction.

Process enactment systems traditionally rely on the control flow defined within the process model to drive the process. This approach has been highly successful in coordinative processes. However, this approach becomes arguable for Collaborative Business Processes (CBPs) that are characterized by asynchronous and highly dynamic business activity. In collaborative processes, it is expected that independent specialized application components both within and across organizational boundaries will be capable of detecting and responding to the events that dictate subsequent process flow. These events can be many, can arise at any time during the overall process and their (time of) occurrence cannot be anticipated by dependent components.

Modelling a collaborative process through the exchange of event data rather than through a rigid control flow between its activities is a significantly different albeit more natural way of capturing the logic behind collaborative processes. Thus, business activity takes place within application components, however the context for the business activity is provided by the event data. How the business activity deals with the data is not the question, instead capturing which business activity may need to be informed about a particular event, and when, is the question at hand.

The critical factor is that the process enforcement system be empowered with sufficient intelligence so that the appropriate action can be taken when a particular event notification arrives. This action basically consists of communicating the relevant data to the right process participant such as, an application component, a business activity

O'Hagan T., Sadiqand S. and Sadiq W. (2007).
EVIE - AN EVENT BROKERING LANGUAGE FOR THE COMPOSITION OF COLLABORATIVE BUSINESS PROCESSES.
In *Proceedings of the Ninth International Conference on Enterprise Information Systems - DISI*, pages 372-377
DOI: 10.5220/0002363403720377

performer, or a workflow management system, at the right time.

In this paper, we present an approach that attempts to capture the dynamics of CBPs and the underlying event dependencies through a scripting language. Difficulties in providing visual models and toolkits for business analysts to capture CBPs are well known. The premise of our programming approach is that CBP setups are mostly undertaken by technical teams often software engineers, where high level models of limited or tedious functionality may prove unproductive. Without compromising on the importance of a model-driven approach, the Java language extensions are intended to provide the power of a programming style language, but at a sufficiently high level of abstraction. The developed program is intended to serve two objectives: to serve as a source for setting up an execution environment; and to serve as a target for a high level model (if available).

We first discuss the motivation and background architecture for Evie. We then present features of the Evie language with the help of an example. Section 4 elaborates on the position of this work in current technology developments. Conclusions and main contributions are summarized in section 5.

2 EVIE FRAMEWORK

The Evie language is motivated by the need for rapid but reliable development of services that act as message brokers or gateways providing routing, transport and encoding mappings between disparate legacy and business partner servers.

There is significant evidence that such infrastructures are featuring prominently in current enterprise systems (see SAP Exchange Infrastructure, IBM WebSphere, BEA AquaLogic, Oracle Integration and Microsoft Biztalk). Consequently, the need to provide tools for rapid development, testing and deployment for broker and gateway services has increased manifold. The Evie language targets this aspect by delivering an abstract and succinct means of expressing broker business logic. Compiled Evie programs are deployed within an execution framework API and user interface tools that support rapid systems integration development and simulation testing.

The overall process for design and deployment of Evie applications can be summarized as: (1) A high level collaboration process is prepared by a business analyst using a model design tool that creates a set of graphical design artefacts. (2) A model compiler translates these into a skeleton Evie rule script that is augmented by a software engineer into a complete set of executable rules. (3) Finally, an Evie compiler translates Evie rules into Java components to be deployed and executed inside the Evie Execution Framework.

2.1 Execution Framework

The Evie execution framework is a Java API class library from which a standalone broker or gateway service is constructed. Its architecture is composed of four tiers that control external communications, event routing, rule execution, and data persistence..

CBP partner organizations each agree that they will expose services that can be characterized by specific event behaviour. The behavioural contract between these services will include agreement on ***event message types***, and ***rules*** governing message exchange. The behavioural contract of a service is referred to as its ***service type*** and typically corresponds to a business role. A ***service instance*** is any partner service that exhibits the corresponding service type (role) behaviour. An Evie framework server can implement *multiple* service instances for *multiple* partners.

The **first tier** implements communications to external services. Evie developers can defer until deployment decisions such as service end point addresses, channel multiplexing of service instances, transport protocols and event message encoding.

In the **second tier**, input and output events are routed between external channels and internal service instances. This isolation ensures that developers describe interactions with abstract event and service types and thus focus on conceptual business logic rather than communications wiring.

The core **third tier** manages the Evie scripts consisting of ECA rules containing event conditions and corresponding actions (discussed in the next section). Compiled event conditions subscribe to specific input events. As each input event arrives it is matched to a set of ***persistent event subscriptions*** corresponding to active rule instances. Each matching subscription activates a compiler generated ***event procedure*** corresponding to either the rule action or a partial evaluation of a complex event condition. Evie supports ***persistent threads*** containing Java context variables. When an event procedure is fired, it is passed a reference to the persistent thread that created the original event subscription thus restoring the execution context of a rule instance. Context variables are used to compose and send output messages and compose and activate new rules.

The **fourth tier** employs a transactional object-relational mapping engine to efficiently persist and

query the state of service instances, events, event subscriptions and persistent threads.

3 EVIE LANGUAGE

The Evie language defines a notation for event processing based on event-condition-action (ECA) rules (Dayal et al., 1988).

Events: Evie defines an event as a CBP state change message observable at a given point in time. Event are communicated between partner services through messaging infrastructures. For collaborative business processes, they provide the impetus for process progression. *Event types* identify primitive CBP events.

Event Conditions: An Evie event condition is a rule precondition that defines when a rule action executes. *Simple* event conditions observe a single event type. *Complex* conditions use logical and temporal operators to express conditions requiring multiple events occurring over a period of time.

Conditions may reference attributes of the event type (Payment) or sender's service type (Customer):

```
receive Payment p from Customer c
where c.status = 'approved'
AND p.amount < = c.maxPayment
AND NOT delay(+days(3))
```

Actions: Evie Actions can compose and send output messages, read or update context variables and, interestingly, dynamically compose and activate new rule instances. They may also execute any other arbitrary Java statements.

3.1 A Request for Quote Example

Consider the following Request for Quote (RFQ) CPB example, with three participant roles: RFQ Manager, Requester and Supplier. The Requester issues a RFQ document requesting quotes for supply of a given product from a set of Suppliers. The RFQ Manager supervises the tender process..Fig. 1 below summarizes the event flow between participant roles.

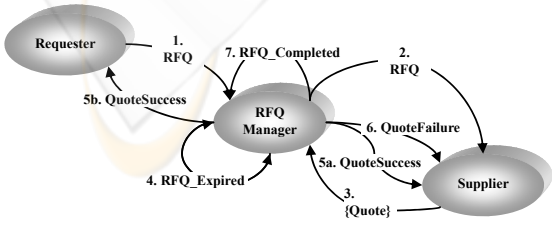

Figure 1: The Request for Quote Event Flows.

We commence the design process by identifying the service and message types in the collaboration (Example 1a)

The rules that represent the behaviour of a participant role are then coded against the corresponding service type definition. Rules for the RFQ Manager broker are illustrated in Example 1b.

Rules for Requester and Supplier can also be implemented in Evie in order to simulate the behaviour of these external services. Simulation rules allow a developer to prototype interaction with external service types, and can also be used to assist in acceptance testing of an Evie broker or its application services.

We observe that an RFQ process can be decomposed into a sequence of phases that exhibit different response behaviour. Rule instances bound to the initial RFQ event are dynamically activated and deactivated as these behavioural phases progress.

```
A.1  package evie.rfq;
A.2  message RFQ {
A.3    string (50) description;
A.4    string (50) product;
A.5  }
A.6  message RFQ_Expired  { }
A.7  message RFQ_Complete { }
A.8  message Quote   { money bidPrice; }
A.9  message QuoteSuccess { Quote quote; }
A.10 message QuoteFailure {
A.11   string (80) reason;
A.12 }
A.13 abstract service Organization {
A.14   string (50) companyName;
A.15   accepts QuoteSuccess from RFQ_Manager;
A.16 }
A.17 service Requester extends Organization{}
A.18 service Supplier extends Organization {
A.19   money maxPrice;
A.20   int   maxDelay;
A.21   accepts RFQ from RFQ_Manager;
A.22   accepts QuoteFailure from RFQ_Manager;
A.23 }
A.24 service RFQ_Manager {
A.25   accepts RFQ from Requester;
A.26   accepts Quote from Supplier;
A.27   accepts RFQ_Expired, RFQ_Completed
A.28      from self;
A.29 }
```

Example 1a: Service types and Event message types.

Phase I: The Requester initiates a new thread by sending a RFQ event to the RFQ_Manager (B.2 in Example 1b). The RFQ specifies the single product required and a Quote submission deadline. Future events must be correlated (B.6) with the RFQ event to indicate that they are part of this process. The RFQ Manager then forwards the RFQ to all Suppliers (B.9) and waits for response Quotes (B.13). The lowest bid Quote is recorded (B.15-18).

All state changes resulting from arrival of an RFQ message will be atomically persisted.

Phase II: The RFQ Manager schedules an abstracted RFQ_Expired event to be sent to itself when the RFQ deadline expires (B.10-12). Once this event occurs (B.20), the RFQ process enters a new phase and consequently the RFQ Manager behavior and rules change. The `then when` construct (B.20) causes threads and active rule instances in the previous phase (B.8 – B.19) to be terminated before executing actions in the new phase (B.21 – B.38).

Phase III: The RFQ Manager then notifies the successful Supplier and Requester (B.24-25) and with a QuoteSuccess event that contains an embedded Quote event. Late Quotes are now rejected with a QuoteFailure reply (B.33-38).

In order to clean up the remaining event Quote subscription, after a 30 day period the final phase terminates with an RFQ_Completed event (B.31, B.39) (final phase) after which time all RFQ correlated rule instances are terminated. Any input events that are not matched to an active rule instance will then be bounced back to their sender by the framework as an exception.

3.2 Additional Language Features

The example above highlights some of the features of Evie which provide an effective means to satisfy particular requirements of event driven CBPs. However, the Evie language supports several other features and characteristics which are briefly summarized below:

Type Inheritance and Aggregation. Service and event types support **type inheritance**. Event types may be imported from other collaborations so that inter-collaboration scenarios are supported and industry standard types (e.g. UBL at www.oasis-open.org) can be reused.

Event types can contain references to other events (A.9) or service instances. Events may also contain event and service instance collections (lists, sets and maps).

A service type can contain attributes used to query a set of service instances or configure service rule behaviour.

Event correlation. Evie `receive` event conditions and `send` statements can apply an arbitrary message **correlation constraint** on input events conditions or set a correlation property on output events (B.37). It can be an arbitrary computed string value or (more commonly) an event instance.

The `correlate` statement (B.4) sets a default correlation value for all `receive` and `delay` conditions and `send` statements within its scope.

A correlation scope effectively partitions all input and output events into groups of execution threads related to an initiating event instance (e.g. RFQ event B.6). This partitioning is similar to a workflow process instance. However, unlike workflow processes, nested event conditions (`receive`) and event compositions (`send`) can elect to regroup events under a different correlation value or event. This allows us to perform event grouping and batch operations on related events.

Aggregate Conditions. An Evie `receive` event condition can group events into a collection and specify aggregate conditions on that collection. The `into` clause in a `receive` condition collects a set of messages of the same type. For example the following condition collects the next three events of type `A` with field `A.x = 1` into the list `aList`.

```
when (receive A a into List<A> aList
      where a.x = 1 AND count(aList) >= 3) { …
}
```

This feature is used in conjunction with either an `AND` or `SEQ` operators. When used with the `AND` operator, the `receive` condition is evaluated as immediately `true`, however, it continues to collect messages until the value of the event condition $cond_1$ is known.

```
receive A into List<A> aList AND (cond1)
```

Similarly, when used with the `SEQ` operator, the `receive` condition is again immediately evaluated as `true` but in this case only collects messages until the first message arrives that is used by $cond_1$.

```
receive A into List<A> aList SEQ (cond1)
```

Nested Rule Overriding. Event subscriptions resulting from nested child rules may override the consumption of events by parent rule instances. This supports a common business case where we need to initiate a new execution thread (using a `repeat` rule) whenever an input message of a known type arrives containing a previously unobserved key value. The outer rule creates new threads, while an inner nested rule can consume messages of the same type and key as a prior initiating message. Consumption overriding can also occur within the evaluation of complex conditions involving multiple events.

```
repeat when (receive A firstA) {
  repeat when (receive A a correlate firstA) {
    // Process A events correlated with firstA.
    // that arrives within 30 minutes.
  } then when (delay(mins(30)) {
    // terminate
  }
}
```

Mutually Exclusive Rules. A list of Evie rules that are activated together can be identified as mutually exclusive. When one of these rules fires, others in the same persistent thread are automatically

deactivated. If more than one fires concurrently then the first rule has precedence.

A `repeat` exclusive rule can be used when a CBP cyclically toggles between mutually exclusive phases.

Multidimensional and Dynamic Conversations. Complex CBPs may be multidimensional involving any number of service types and instances. New service instances may be dynamically registered and deregistered within an active CBP.

```
B.1   package evie.rfq;
B.2
B.3   service input rules RFQ_Manager {
B.4    repeat when (receive RFQ rfq   // Phase I
B.5                 from Requester requester) {
B.6      correlate (rfq) {
B.7        Quote bestQuote = null;
B.8        when (true) {
B.9          send rfq to role Supplier;
B.10         when (delay(rfq.deadline)) {
B.11           send new RFQ_Expired() to self;
B.12         } // when
B.13         repeat when (receive Quote quote
B.14                      from Supplier) {
B.15           if (bestQuote == null ||
B.16               bestQuote.bidPrice >
       quote.bidPrice) {
B.17             bestQuote = quote;
B.18           } // if
B.19         } // when
B.20       } then when (receive RFQ_Expired from
       self) {                        // Phase II
B.21         log.info("Expired RFQ: "
B.22                  + rfq.description);
B.23         if (bestQuote != null) {
B.24           send new QuoteSuccess(
                        quote := bestQuote)
B.25             to requester, bestQuote.sender;
B.26         } else {
B.27           send new QuoteFailure(reason :=
B.28            "No Quotes received before
       deadline") to requester;
B.29         }
B.30         send new RFQ_Completed()
B.31           to self delay +days(30);
B.32         repeat when (receive Quote quote
B.33                      from Supplier) {
B.34           send new QuoteFailure(reason :=
B.35             "Quote not received before
       deadline")
                to quote.sender
                correlate quote;
B.36         } // repeat when
B.37       }then when (receive RFQ_Completed
       from self){                  // Phase III
B.38         log.info("Completed RFQ: "
B.39                  + rfq.description);
B.40       }
B.41     }
B.42   }
B.43  }
B.44 }
```

Example 1b: RFQ Manager Rules.

Fine-grained Dynamic Access Control. Evie rules not only describe *what* the event must contain but also and *when* the event may be sent or received and *who* may send or receive it. Rules that govern these constraint dimensions can be dynamically composed and correlated based on past observed events.

Persistent Context Variables. An Evie rule binds a condition to an action. When an event condition matches an input event sequence, these events (and related sender services) are bound to **persistent context variables** declared as part of the condition. The rule action or nested rules may then reference these newly bound context variables.

Threads and Transactional Memory. In Example 1b, a repeating rule captures an RFQ event (B.4) and creates a new persistent thread containing the context variable `bestQuote` (B.7).

Child threads spawned by a nested repeat rule (B.13), read and update data from shared parent threads (B.17). The framework employs optimistic locking to detect and retry transactions when concurrent update conflicts occur with persistent threads. The Evie compiler detects reads (B.15-16) and updates (B.17) to shared variables and emits code to ensure that the optimistic locking occurs.

The implied transactional memory model (Shavit et al, 1995) significantly alters the semantics of context variables. It delivers a simple, transparent and scalable solution. It also significantly reduces the code complexity normally associated with concurrent state management.

4 RELATED TECHNOLOGIES

During the past several years, as enterprise software has evolved, there has been extensive research on enterprise architectures in pursuit of the evasive business-IT alignment.

From the technology perspective, the most significant development in the recent past impacting on enterprise architectures has been through service oriented architectures or SOA (Alonso et al., 2004). Even though an essential stepping stone for *service enablement* of enterprise applications, web services standards do not provide the complete solution for CBPs.

Achieving communication between disparate enterprise applications through messaging is well established in message oriented middleware (middleware.org), with recent trends towards solutions that can scale beyond the traditional hub-and-spoke message broker. The extended functionality of the Enterprise Service Bus (ESB)

(Chappell, 2004) is currently a dominant approach in this respect, providing the ability to store messages and establishing streamlined *service communication*.

Recent developments from business software vendors have identified the need for solutions that go beyond *service enablement* and *communication* capability. These provide a development environment that allows multiple services both within and across enterprise systems to be collated into value added composite applications (see ESA & CAF from sap.com).

We observe that a critical aspect of current enterprise architectures based on the above approaches is the management of the rules for *service interaction* (serviceinterationpatterns.com). This functionality would naturally reside in middleware components and is the main driver for the approach presented in this paper. While there have been significant developments within the first two phases of service enablement and communication, the last phase of managing service interaction still holds many challenges.

Difficulties in modelling service interactions through typical control flow constructs as found in workflow modelling languages (workflowpatterns. com) are known to be ineffective in the CBP scenario due to the scale of options. Instead, approaches that utilize event processing have emerged as a more promising alternative (Luckham, 2002). Some operators and related event algebras can be found in: HiPAC (Dayal et al., 1988), Compose (Gehani et al., 1992), Snoop (Charavarthy et al., 1994), RAPIDE (Luckham, 2002), TriGS (Retschitzegger, 1998), (Cao et al. 2006).

5 CONCLUSIONS

The primary purpose of the Evie approach is to inter-connect the high level business models with underlying execution infrastructures within the context of event based CBPs.

In this paper we have presented an approach that provides the capability to setup an executable environment for event based CBPs through a rather slim specification. The Evie framework is well aligned with current trends towards event based architectures for large scale integration systems. However, the proposed approach is distinguished in three respects:

- providing simple and uniform language constructs that allow the specification of diverse service interaction patterns
- ability to provide a level of abstraction from the execution details due to the compilation phase that generates the requisite objects and code for execution
- utilization of an execution model based on event subscription, that provides the ability to cater for high volume and long duration processes with minimal impact on system performance and response latency

An important aspect of this approach is the ability to generate an Evie program from a high level modelling tool. This aspect has not been considered in this paper, but is part of our future work.

REFERENCES

Alonso, G., Casati, F., Kuno, H., Machiraju, V (2004) Web Services Concepts, Architectures and Applications. Springer Verlag

Chakravarthy, S., Krishnaprasad, V., Anwar, E., & Kim, S.-K. (1994). Composite Events for Active Databases: Semantics, Contexts and Detection. Paper presented at the Proceedings of 20th International Conference on Very Large Data Bases (VLDB' 94), Santiago, Chile.

Chappell, D. A. (2004). Enterprise Service Bus (1st ed.). Sebastopol, California: O'Reilly Media, Inc.

DatMa Cao, Maria E. Orlowska, Shazia W. Sadiq. (2006) Formal Considerations of Rule-Based Messaging for Business Process Integration, Special Issue of Cybernetics and Systems: An International Journal, Vol 37/2 (Feb/March 2006).

Dayal, U., Blaustein, B. T., Buchmann, A. P., Chakravarthy, U. S., Hsu, M., Ladin, R., et al. (1988). The HiPAC Project: Combining Active Databases and Timing Constraints. ACM's Special Interest Group on Management Of Data (SIGMOD), 17(1), 51-70.

Gehani, N. H., Jagadish, H. V., & Shmueli, O. (1992). Event specification in an active object-oriented database. Proceedings of the 1992 ACM Special Interest Group on Management Of Data international conference on Management of Data (SIGMOD'92), San Diego, California, United States.

Luckham, D. C. (2002). The power of events: an introduction to complex event processing in distributed enterprise systems. Boston, USA: Addison-Wesley.

Retschitzegger, W. (1998). Composite Event Management in TriGS - Concepts and Implementation. Paper presented at the Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA '98), Vienna, Austria.

Nir Shavit and Dan Touitou. Software Transactional Memory. Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pp.204–213. August 1995.