From Calls to Events: Architecting Future BPM Systems

Alejandro Buchmann, Stefan Appel, Tobias Freudenreich,
Sebastian Frischbier, and Pablo E. Guerrero

TU Darmstadt, Germany
lastname@dvs.tu-darmstadt.de

Abstract. Contemporary BPM systems fit very well with traditional architectures that are based on a pull invocation principle, such as SOA. The proliferation of sensors and streams of events has led to event driven architectures that decouple event producers and consumers. EDAs are push-based and support different control structures. Future BPM systems must therefore deal both with pull and push-based architectures. In this talk we will analyze the interplay of the different architectures, their components and the desirable and achievable correctness notions and non-functional properties.

1 Introduction

By 2020 it is predicted that a large portion of enterprises will be process driven [20]. The key driving forces that are mentioned are customer centricity, operational excellence, new regulations, new business models, and a global workforce. In this world Business Process Management (BPM) will play a deciding role. A BPM system, according to [28] is "a generic software system that is driven by explicit process designs to enact and manage operational business processes". We will refer to BPM systems in their most general form, including not only traditional workflow management (WFM) but also Business Process Analysis (BPA), Business Process Monitoring (BAM) and Process Aware Information Systems (PAIS).

The traditional BPM lifecycle consists of process design, system configuration, process enactment, and diagnosis. The result of business process diagnosis may result in process redesign. The automatic extraction of new business processes through process mining of the event log is an alternative to manual process (re)design. Emergent software [13] is a closely related topic. Researchers in this area are trying to develop software systems that can adapt to changing environments producing new behaviors from local events. The goal of Emergent Enterprise Software Systems is to facilitate the cooperation across enterprise boundaries by self-adapting the business process.

The degree of human involvement in the (re)design of business processes can vary and is a continuum ranging from completely manual design to autonomic behavior with enforcement of self-x properties, and finally the automatic extraction of new processes. This continuum is the top layer of Figure 1.

A. Barros, A. Gal, and E. Kindler (Eds.): BPM 2012, LNCS 7481, pp. 17–32, 2012.

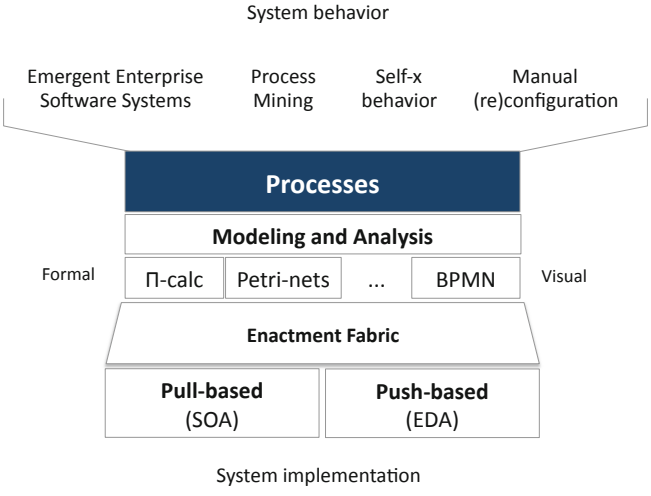

Fig. 1. Business processes from the system perspective

Processes are at the core of process aware software systems. Processes must be represented by a model in the corresponding formalism or language. The modeling formalism may range from intuitive and mostly graphic representations, such as BPMN, to more formal notations based, for example, on petri-nets or variants of process algebras, such as CCS or Pi-calculus. To automate the process design or redesign, a formal model is needed. Informal or intuitive graphical notations are good for human consumption but unsuitable for automatic process extraction. This is the middle tier in Figure 1.

Processes must be mapped to the corresponding execution environment for enactment. The by now established platform for enactment of business processes are web services. Many of the most interesting new applications, however, are monitoring and reactive applications responding to events that are sensed by a multitude of sensors or are generated through event aggregation, composition and/or derivation. These applications are not well served by web services and service oriented architectures. There are two main reasons: 1) Web services employ a request-respond invocation mechanism, i.e., they are primarily pull-based. Monitoring and reactive applications are better served by push-based dissemination (of streams) of events and their associated data. 2) Web services have been conceived as black boxes that hide the middleware they are running on. Only recent efforts [21] have tried to provide for vertical service composition that enables the execution of services on the best suited middleware platform. The lower portion of Figure 1 shows the two complementary paradigms and the corresponding architectures: Service Oriented Architecture (SOA) for more traditional pull-based invocation and Event Driven Architecture (EDA) for push-based invocation. It is our conviction that environments for execution of a broad range of business processes must offer both kinds of invocation. However, since SOA is well established for the processing of business processes, we will concentrate on EDA, its advantages and challenges.

The advantages of EDA consist in the loose coupling between producers and consumers of events that lead to easy extensibility of systems, and the celerity with which events are detected, propagated and reacted to. The challenges consist in establishing a common understanding of the semantics of events and the execution of triggered tasks, and in establishing understandable correctness criteria for the execution of event driven business processes.

The remainder of this paper discusses events and event composition in Section 2; Section 3 presents briefly EDA; Section 4 discusses quality of service in Event Driven Architectures; Section 5 presents the concluding discussion.

2 Events and Event Composition

There exist many interpretations of what an event is. Often these different interpretations are community-specific. Therefore, we start by introducing a classification proposed by Chandy and Schulte in their book 'Event Processing, Designing IT Systems for Agile Companies' [7] that reflects some of the ongoing debate. There are three views of what an event is, each being true from one perspective but none describing the whole truth or being equally useful in different situations.

- *An event is a happening of interest.* This is an activity-based view of events and quite intuitive. For example, a person entering a room or a container leaving a warehouse. While easy to understand and useful for describing a situation it is not helpful computationally and must be translated into measurable quantities.
- *An event is a (meaningful) change of state.* This definition considers any change in the model of reality that is observed as a potential event. Such a change could be the detection of an RFID tag on a container at the warehouse gate. This approach is very useful in implementing event based systems but often requires deriving the more abstract event (e.g. the container leaving the warehouse) from one or more observations. This definition depends on detecting change and makes it difficult to handle observations that may be of interest even though no change occurred.
- *An event is a detectable condition that can trigger a notification.* This is a reporting-based view of events. This definition is somewhat more general than the change of state view in that it also considers the absence of change as a detectable condition. However, it depends on a reporting capability in the form of notifications, defined as an event-triggered signal sent to a run-time recipient. This view of events is quite useful to detect all kinds of events, even observations that do not depend on a change, but anything that is either not observed or not reported is not considered an event.

In [14] we expanded the second definition above, by including time as a basic dimension, thereby allowing us to consider two observations taken at different times to be considered as two events even if none of the other dimensions describing the state have changed. Notifications are the natural way of notifying parties interested in an event, usually via a publish/subscribe notification service.

2.1 Important Concepts in Event Driven Architectures

Event types and event instances: The event type determines the attributes of an event and its structure. An event instance is a particular materialization of an event type and may be identified either through an event-identifier or through a unique combination of attribute values.

Event objects, event representations and notifications: An event carries a timestamp and descriptive parameters and is typically represented as a tuple of values. This representation is called the event object. Event objects can have other representations, for example, an XML document. When the event object is packaged into a message, we talk about an event notification.

Temporal events are first class citizens. We distinguish between absolute temporal events and relative temporal events. Absolute temporal events conceptually consist only of a timestamp and the source and have a simple representation. The timestamp has a given granularity that is often encoded in the format of the timestamp and an identifier of the clock that may be omitted in centralized systems. Relative temporal events are time offsets relative to a base event. The base event can be either an absolute temporal event or it could be any other event. The granularity of the offset can be specified as a function of the capability of the clock and the requirements of the application.

State events are sometimes identified as a special kind of event to distinguish them from change events, i.e. the change of state. State events are observations in which the observed quantity may not have changed but only time has advanced. Given the framework proposed earlier to include time as an integral part of the definition of state, there is no need to make a distinction for state events. Subscribers to state events must provide the frequency at which they need the observation events.

Change events refer to any change of state. We must restrict ourselves to meaningful state changes. What is meaningful is determined by an event consumer who subscribes to events. The producer may generate events at a certain rate, for example, every second, but one consumer of those events is only interested in every tenth event while another consumer needs that type of event only every minute, i.e. every 60th event.

Event filtering is applied to eliminate those events that were observed but for which no interest exists. Event filtering can occur in principle anywhere along the path between producer and consumer of a certain event type. Placing the filter near the event source minimizes the notification costs and the processing cost for the event consumer.

Simple events are basic events that are not the result of some composition.

Complex events are all the events that are the result of an event processing step that combines several events or enriches the events with context information. We distinguish between composite and derived events.

Composite events are the result of combining multiple events, typically through the operations of an event algebra. Although this is not always accepted, the glossary of the Event Processing Technical Society (EPTS) [19] defines that composite events always must carry all the constituting events. A special case of composite events are aggregate events which are formed through applications of the standard aggregation operators, such as average, sum, max, min, count, or top-k. Aggregate events rarely carry the full set of simple events that were aggregated. Particularly in wireless sensor networks it is the goal to reduce the volume of transferred events and only the aggregation is propagated.

Derived events are events of a higher level of abstraction. For example, if we observe 3 failed login attempts we might consider this to be an attempt to penetrate the system, or 10 temperature readings that increase monotonically imply a failure of the air conditioning system. Derived events are the events one typically expects when talking about complex event processing. They can be quite abstract and may involve correlation of events with external information. For example, the sale of stock by a manager of a pharmaceutical company two days before it is publicly known that a new drug will not be approved by the FDA might be an insider trading event. The combination of events with additional information, either from external sources or from databases, is generally referred to as event enrichment or event contextualization.

Complex event processing encompasses event filtering, aggregation, composition and derivation, as well as event contextualization. Complex event processing can be accomplished with several mechanisms. Most common is the processing of streams of event objects on which windows are defined [5]. Stream processing systems apply the operators of an algebra, for example those of relational algebra, to the instances of event objects in a window. Windows can be defined based on a number of objects in the window or based on time intervals. They can also be sliding or tumbling. Sliding windows open a new window at predefined intervals. If the interval for opening a new window is the size of the window, we speak of tumbling windows.

An alternative (or complement) to stream processing is the correlation of events, e.g. based on time. This approach is typical in sensor fusion. More sophisticated event compositions are based on event algebras that typically contain operators for sequencing (ordered events), intersection (`AND`, two events occurring in any order), union (`OR`, any of two events occurring), negation (an event `NOT` occurring in a well-defined interval), accumulation (`ANY n` events occurring in a well-defined interval), etc. [10,6]. Complex events are then represented as expressions of the event algebra. These expressions are transformed into a tree structure in which the inner nodes are the operators of the algebra and the operands (event objects) are at the leaves. These graphs are evaluated similarly to query graphs in a database from the leaves to the root with the composite event being the result of the evaluation of the root node.

2.2 Events in BPM

Usage of events is supported in mainstream business process models such as BPMN and UML activity diagrams by means of two workflow patterns: the "deferred choice" pattern [25] and the "event-based task trigger" pattern [26]. These offer an activity-based view of events; events are sent as messages to trigger subprocesses. The producer of the event sends a single event as a direct message to activate the subprocess. In this sense it is comparable with imperative programming. The notion of streams of events, subscriptions and composition/derivation of events is not considered yet.

3 Event Driven Architecture

A basic tenet of EDA is the loose coupling between event producers and event consumers. This means that the producer of events need not be aware of who will eventually consume the produced events. It also means that producer and consumer of events should be decoupled in space and time. Spatial decoupling results in distributed systems, temporal decoupling in asynchronous systems. The main properties an EDA should fulfill as stated in [7] are:

- Reporting of current events as they happen
- Pushing notifications of events from the producer to the consumer
- Responding immediately to recognized events
- Communicating one-way without the need for acknowledgements
- Reacting to event notifications and not to commands

Events should be reported as soon as they happen rather than being stored and later forwarded or requested by the consumer. Events will be reported as discrete event objects that are packaged in a notification. A notification service is thus responsible for prompt delivery of notifications. Consumers of events, i.e. the applications, should respond immediately to relevant events. These three conditions guarantee timely response of event driven systems.

Decoupling is important since events occur independently of the reactions. Interested parties must subscribe to events in order to receive the corresponding event notifications. Subscribers can also unsubscribe and this does not affect the future detection of the events, only their notification. The event based interaction pattern does not require any answer, i.e., the event producer will not block. Because the event producer is not aware of the event consumers, it cannot request the event consumer to execute any actions. Event driven systems are, therefore, consumer controlled. Loose coupling provides the desired flexibility because components need not be active at the same time and new components can be added without affecting existing components as long as the notifications do not change.

3.1 Components of an Event Driven Architecture

An EDA minimally consists of three components: event producer, notification mechanism, and event consumer, as shown in Figure 2. Event objects are accepted by the notification mechanism and packaged into notifications.

Fig. 2. Components of an EDA

Event producers detect events and produce event objects. The event object has a structure that is defined by the event type and contains the necessary event parameters. Event parameters are instantiated by the event detection process and the event contextualization process. The event detection process typically will probe the environment. The event contextualization process will add context information, such as location of the detector and timestamp but may rely on external data sources as shown in Figure 3, although for practical reasons type and context information may be held locally.

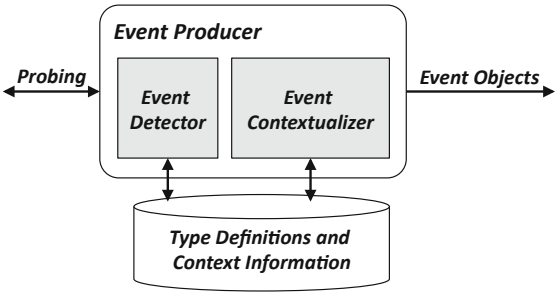

Fig. 3. Event Producer

Event consumers receive event notifications from the notification mechanism. Event consumers must unpack the event notification, extract the event object and execute an action in response to the received event. In principle, the response may be a local action, the invocation of a (remote) service or business process, a rule that must be triggered, an event composition or storage of the event for logging. Event consumers may act as event producers, for example, when they produce a composite event that is forwarded. Figure 4 shows schematically an event consumer.

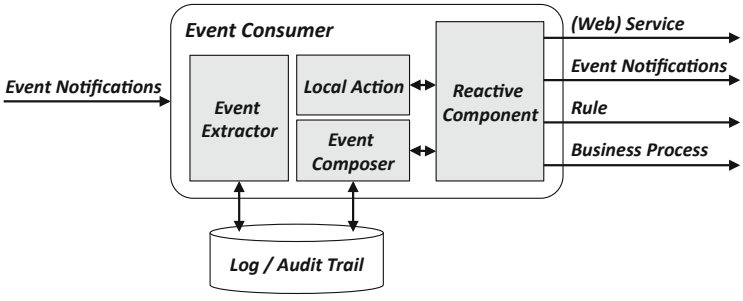

Fig. 4. Event Consumer

The notification mechanism is the most interesting component of the whole EDA. Its function is that of a communication channel that pushes events from the producers to the consumers thereby providing an end-to-end push-style communication. Because in an event driven interaction the producer does not know the consumer, the notification mechanism must mediate the communication. In its simplest form this could be a dedicated communication channel carrying all the events of a producer to the consumer, or it could be a sophisticated publish/subscribe system. We will analyze the spectrum of options. The relevant questions are:

- How are producers and consumers brought together?
- Does the channel deliver all messages or does it filter?
- If filtering is done, on what criteria and where are the filters placed?
- Are events transformed or only routed by the notification mechanism?
- If transformations are applied, where are they applied and what are they?

3.2 Channel-Based Notification Systems

Common Area Channels of two kinds are popular: blackboards and queues. In a blackboard-type channel, publishers post their detected events to a common area and consumers pick up events from there. Examples of blackboard-type common areas are tuple-spaces. An extreme form of a persistent tuple-space is a relational database. Queues are message buffering structures administered by a queue manager. Queues come in a variety of flavors: persistent vs. non-persistent, transactional vs. non-transactional. Common-area channels provide asynchrony and loose coupling but do not fulfill the requirement of end-to-end push-style notification required for EDA, since consumers must pull events from the common area. Therefore, we will not pursue common-area channels further in this discussion.

Notification Routing Channels contain one or more brokers that implement some form of routing table. Routing tables are built based on event subscriptions. Consumers declare their interest in certain (types of) events and the brokers mediate between producers and consumers. If no filtering occurs in the broker, we

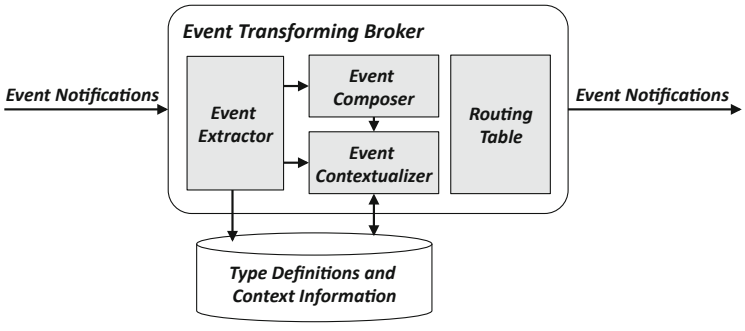

Fig. 5. Event Transforming Broker

speak about flooding. The power of notification routing channels, however, lies in the ability to filter event notifications and deliver only the relevant notifications to the consumer.

Notification Transforming Channels consist of a network of brokers, each of which can act as a consumer and producer of events. Notification transforming channels in their simplest form take in an event and change the structure of the event object and/or the format of one or more parameters. Events can also be enriched with external context data, aggregated or composed with other events. The Notification Transforming Broker receives an event notification, unpacks it, transforms the event, re-packs it into a notification and routes the new notification according to its routing table. This type of broker is shown schematically in Figure 5. Event transforming brokers are used in some advanced types of publish/subscribe systems and Enterprise Service Busses.

3.3 Publish/Subscribe Notification Systems

Publish/Subscribe is the mechanism of choice in Event Driven Architectures. Although it is possible to implement an EDA with other notification channels, Pub/Sub offers many advantages. In a Pub/Sub system, consumers subscribe to events of interest. Subscriptions are mapped to routing tables and to filters. Optionally, event producers can advertise the types of events they are prepared to produce. Depending on the filtering and the kind of routing performed by the brokers, we can distinguish different classes of publish/subscribe systems:

Channel-based Pub/Sub provides a named channel to which subscribers can subscribe. All the events of a given type are dumped into the channel. The subscriber receives all the notifications published to this channel and must apply the filters. This is the approach used in early middleware platforms, e.g. CORBA Event Service [22]. The CORBA Notification Service [23] improves the Event Service by providing filters.

Type-based Pub/Sub uses path expressions and subtype inclusion tests to select notifications. Through multiple inheritance, the subject tree can thus be

converted into a type lattice with multiple rooted paths to the same node. This approach circumvents some of the limitations of simple subject hierarchies [24].

Topic-based Pub/Sub is the approach associated with the Java Messaging Service [27]. Topic-based pub/sub uses channels that include filters. Filtering is done through Boolean predicates defined on the envelope of the notification. While the filters that can be expressed are quite flexible their expressiveness is limited by the fact that they can only be defined over the metadata contained in the header.

Content-based Pub/Sub extends matching to the content of the body of the message rather than limiting it to the header. The expressiveness of this approach is determined to a large extent by the data model used for the content and the corresponding formalism for expressing the filter predicates. Systems have been proposed based on simple template matching [9], filter expressions on name/value pairs [1] and XPath expressions on XML documents [11]. A tacit assumption for content-based pub/sub to work is the existence of a common name space and context used by publishers and subscribers. This is often the case in small systems but rarely in large, heterogeneous environments.

Concept-based Pub/Sub [8,12] addresses the problem of implicitly assumed semantics and makes the semantics of advertisements, subscriptions and notifications explicit. Concept-based pub/sub associates a context with each notification and filter. Matching compares first if the contexts of filter and notification are the same. If true, the normal content-based pub/sub approach kicks in. If the contexts of filter and notification are different, an ontology service is invoked to resolve the different semantics, for example by converting currencies or formats of dates. This approach can be used in conjunction with many of the other pub/sub approaches in large, heterogeneous environments.

3.4 Hybrid Architectures

Real-life systems rarely conform to the pure reference architecture. Therefore, it is necessary to combine different invocation styles in the form of hybrid architectures. Referring to Figure 4, the event consumer may trigger a request-driven interaction to a business process or a service in a SOA, resulting in an event driven SOA. Likewise, an event may trigger a request to a queue to pick up something posted there. Both architectural styles must coexist.

4 Quality of Service in Event Driven Architectures

End to end Quality of Service (QoS) in an EDA can be affected by various components. Large event driven systems are often distributed, so the network characteristics play a big role. For example, messages may be delayed, lost or arrive out of order. This must be considered when looking at the achievable QoS of the components of an EDA. We will consider four aspects in this section: the QoS of the stream processing mechanism, the QoS of the algebra-based

event composition mechanism, the QoS provided by the notification mechanism, and the effect on transactional behavior of business (sub)processes if these are triggered by an event.

4.1 QoS of Stream Processing

Stream processing is used to detect patterns and raise alarms or trigger rules or actions when certain patterns are observed. Stream processing is highly dependent on the nature of the streams. While the operators used in stream processing resemble the operators of relational algebra but applied to windows (when the event objects are tuples), the nature of streams is quite different from the nature of data in a database. Event objects in a stream arrive continuously and are usually processed in arrival order. They are generated by external sources that are outside the control of the stream processing system. Therefore, the input characteristics can't be controlled and event objects may be lost or corrupted. The volume of incoming streams may vary widely depending on the subscription pattern of the application, and the structure of the event objects may range from simple tuples to XML documents or even unstructured data, such as images.

Stream processing requires continuous processing of incoming event objects. Since applications depending on stream processing typically have timing constraints to meet, the timeliness of a response is one of the most relevant QoS requirements. Many applications that depend on stream processing can tolerate approximate results. Therefore, it is common to trade-off accuracy for timeliness. A closely related QoS metric is the achievable throughput. Throughput is less sensitive to load than response time.

Stream processing systems attempt to optimize continuous query execution to maximize throughput. However, load shedding is often the only practical approach, resulting in a trade-off of accuracy for timeliness. Application processes must be aware of this and specifications of expected timeliness and tolerable accuracy are required during business process design.

Another issue that is application dependent is the tolerance to events being processed out of order. Research on how to deal with out of order events in streams [4] has been incorporated into some of the industrial offerings.

4.2 QoS of Event Composition

The achievable QoS in event composition depends largely on the possibility to establish an ordering between events. While operators such as intersection and union do not require ordering, the sequence operator, which is part of most event algebras, requires an ordering of events. The natural ordering is done on time. This is perfectly fine if there is only one central clock and at most one event can occur per clock tick. As soon as multiple events can occur simultaneously and are time-stamped by different clocks it becomes impossible to establish a total order.

The granularity of time is also important when trying to establish an ordering. Two events with distinguishable order with timestamps of fine granularity (e.g.

milliseconds) may not be distinguishable with coarser timestamps (e.g. seconds). Some applications are not affected, while for others it is essential. For example, if a tagged container passes two RFID readers, the proper sequence determines the derived event that the container has either entered or left the warehouse. It is often the case that application semantics or additional information sources must be brought in to resolve ambiguities. However, in case of ambiguity, the underlying middleware should never arbitrarily choose a solution and pretend there exists an unambiguous ordering.

The delay or loss of messages, especially in wide area networks, is another source of potential ambiguity. To evaluate the negation operator, i.e. to determine whether an event did not occur in a given interval, one must be able to establish that the message with the notification is neither lost nor delayed. In networks with bounded delay, the 2g precedence model is adequate [15]. It establishes that anything outside an interval formed by one maximum delay before to one maximum delay after a given point in time can be known with certainty. For unbounded networks, such as the Internet, an approach based on sweeper events has been proposed [16]. It does not assume ordering, but requires only that two events in the same channel do not overtake each other. By injecting the heartbeat events from an outside time service, the recipient knows that everything coming over that channel after the heartbeat must be younger. Delivery in publishing order can be ensured by the messaging middleware. The past before the heartbeat thus becomes certain while the past between the heartbeat's timestamp and the present is still uncertain.

The last issue impacting the QoS of the event composition is the order in which events are consumed. Event expressions are written based on event types. Expressions are instantiated by the arrival of instances of events that are part of an expression. If we do not specify in what order the events should be consumed, we can't have clear semantics. For example, the expression `E AND C` with an event stream `e1`, `e2`, `c1` would consume `e1 AND c1` under chronological consumption, but `e2 AND c1` if we use the most current instances of an event type. A good solution to this problem was given in [6], but the domain expert must decide what semantics fit the application. It is equally important that the event composition software offers the right choices.

4.3 QoS of the Notification Mechanism

The notification mechanism is essential to disseminate event notifications in an asynchronous and decoupled way. Event driven business process components subscribe to events and a single event notification can trigger or change the execution of a business process. It is thus necessary to make business process components aware of the QoS of the underlying notification mechanisms. Components that rely on events should therefore be able to express QoS demands. Different QoS properties are adopted in current notification middleware, e.g., in JMS brokers.

- *Persistence*: The middleware takes extra care to ensure that no event notifications are lost in case of a server crash by buffering them on persistent storage.
- *Delivery Mode*: The delivery mode determines whether events are delivered at least once, at most once, or exactly once.
- *Durability*: With non-durable subscriptions a subscriber will only receive notifications that are published while he is active. With durable subscriptions notifications are buffered in case subscribers temporarily disconnect.
- *Transactions*: A notification session can be transactional or non-transactional. A transaction is a set of notification operations that is executed as an atomic unit of work, e.g., send all or discard all notifications in a session.
- *Order*: When order of event notifications is guaranteed, the middleware ensures that notifications arrive in the order they were published.
- *Performance*: The number of event notifications that can be handled by the middleware in time (throughput and latency).

A more detailed discussion of quality of service in Publish/Subscribe systems is presented in [2].

4.4 QoS of Transaction Management

Business processes often require transactional behavior. However, transactions come in many different flavors. Database transactions are tightly coupled and guarantee full ACID properties (atomicity, consistency, isolation and durability). This is possible because the DBMS has full control over (synchronous) communication, execution, storage, and release of results. In object transactions, the components communicate directly with each other 1:1 and communication is reference based, i.e., each component knows its counterpart and how to address it directly. Interaction requires communicating components to be present at the same time and the requestor blocks while the other component answers. This ought to be compared to a mediated communication based on publish/subscribe, where n producers communicate with m consumers, the addressing is not reference based but logical, e.g. content-based, and asynchronous. If transactions are to be executed successfully when producers and consumers of notifications are completely decoupled by the middleware, the middleware must be incorporated into the transaction.

This is the approach originally purposed by Middleware Mediated Transactions (MMT) [18,17]. The key to this proposal is to incorporate the sending and receiving of notifications into the transactional boundaries of the producer and/or the consumer of the notification. This, together with a controlled delivery mode by the messaging middleware as described above, defines a very flexible and powerful transaction model for event driven systems.

The key properties of MMTs are grouped by so-called coupling modes that reflect the visibility rules, commit and abort dependencies of complex transaction models [3].

- *Visibility* refers to when a notification is sent to consumers relative to the completion of the producer's transaction: with immediate visibility, notifications are sent to the middleware and on to consumers before the producer's transaction commits. On commit (abort), notifications are sent out only after a commit (abort) of the producing transaction. Deferred visibility means that notifications are propagated when the producer begins the commit process.
- *Context* allows the recipient of a notification to join the same transaction context as the producer (shared context) or the middleware or the consumer may establish a separate context for the recipient (separate context).
- *Forward dependency* limits the freedom of the consumer of a notification to commit. A forward commit dependency means that the consumer of a notification may only commit if the producer commits. Likewise, a forward abort dependency states that the consumer can only commit if the producer aborts.
- *Backward dependency* limits the freedom of the producer of an event notification to commit. If vitally coupled, the producer may only commit if the consumer transaction committed. A marked-rollback producer may complete but may be rolled back on request of the consumer.
- *Production* of a notification in transactional mode limits the delivery of the event notification to the mediating middleware to after the commit of the producer. An independent production policy leaves the decision to the producing transaction how a failure in delivery should be handled.
- *Consumption* refers to when the event notification is considered to have been delivered. It could be either on delivery, or when the recipient returns from executing its reaction, or when the consumer begins commit.

QoS of event driven systems is still a wide open area. We do not advocate a specific solution for stream processing, event composition, notification or a transaction model for EDA. Instead, we raise awareness of the issues that must be addressed jointly by researchers, product vendors and domain experts.

5 Summary and Conclusions

An increasing number of sensors and other sources are generating streams of valuable information that business processes should exploit. To support this, process models and the formalisms used for their description need to be expanded to represent more powerful notions of events and their integration as first class citizens in the specification and design of business processes.

We made a case for hybrid architectures combining a Service Oriented Architecture with an Event Driven Architecture. Both architectural styles are needed to satisfy the requirements of modern process oriented enterprises.

Domain experts should exploit the benefits of event processing to improve timeliness and agility. At the same time they must be aware of limitations and potential pitfalls.

End to end QoS is important for business processes. We identified four aspects that can affect the end to end QoS in an Event Driven Architecture. Additional elements are required in the modeling formalisms for the specification of quality of service expected by business processes and the acceptable trade-offs.

Acknowledgements. We wish to acknowledge many interesting discussions with Mani Chandy, Dimitrios Georgakopoulos, Annika Hinze and Christoph Liebig. Research underlying the present position paper was sponsored by German Federal Ministry of Education and Research (BMBF) under research grants ADiWa (01IS09020D) and Software-Cluster EMERGENT (01IC10S01) as well as by Hessen Ministry of Higher Education, Research and the Arts under research grant LOEWE Dynamo PLV. The authors assume responsibility for the content.

References

1. Antollini, J., Antollini, M., Guerrero, P., Cilia, M.: Extending Rebeca to Support Concept-Based Addressing. In: ASIS 2004 (2004)
2. Behnel, S., Fiege, L., Mühl, G.: On Quality-of-Service and Publish/Subscribe. In: DEBS 2006 (2006)
3. Buchmann, A.P., Ozsu, M.T., Hornick, M., Georgakopoulos, D., Manola, F.A.: A Transaction Model for Active Distributed Object Systems. In: Database Transaction Models for Advanced Applications, pp. 123–158. Morgan-Kaufmann (1992)
4. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams - A New Class of Data Management Applications. In: VLDB 2002. Morgan Kaufmann (2002)
5. Chakravarthy, S., Jiang, Q.: Stream Data Processing: A Quality of Service Perspective. Springer (2009)
6. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.-K.: Composite Events for Active Databases: Semantics, Contexts and Detection. In: VLDB 1994 (1994)
7. Chandy, K.M., Schulte, W.R.: Event Processing: Designing IT Systems for Agile Companies. McGraw-Hill, Inc. (2010)
8. Cilia, M.A., Bornhövd, C., Buchmann, A.: CREAM: An Infrastructure for Distributed, Heterogeneous Event-Based Applications. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) CoopIS/DOA/ODBASE 2003. LNCS, vol. 2888, pp. 482–502. Springer, Heidelberg (2003)
9. Cugola, G., Di Nitto, E., Fuggetta, A.: The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. IEEE Transactions on Software Engineering (TSE) 27, 827–850 (2001)
10. Dayal, U., Buchmann, A.P., McCarthy, D.R.: Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In: Dittrich, K.R. (ed.) OODBS 1988. LNCS, vol. 334, pp. 129–143. Springer, Heidelberg (1988)
11. Diao, Y., Franklin, M.J.: XML Publish/Subscribe. In: Encyclopedia of Database Systems, pp. 3608–3613 (2009)
12. Freudenreich, T., Appel, S., Frischbier, S., Buchmann, A.: ACTrESS - Automatic Context Transformation in Event-based Software Systems. In: DEBS 2012 (2012)
13. Frischbier, S., Gesmann, M., Mayer, D., Roth, A., Webel, C.: Emergence as Competitive Advantage - Engineering Tomorrow's Enterprise Software Systems. In: ICEIS 2012 (2012)

14. Hinze, A., Sachs, K., Buchmann, A.: Event-Based Applications and Enabling Technologies. In: DEBS 2009 (2009)
15. Kopetz, H.: Sparse Time versus Dense Time in Distributed Real-Time Systems. In: ICDCS 1992 (1992)
16. Liebig, C., Cilia, M., Buchmann, A.: Event Composition in Time-dependent Distributed Systems. In: CoopIS 1999 (1999)
17. Liebig, C., Malva, M., Buchman, A.: Integrating Notifications and Transactions: Concepts and X^2TS Prototype. In: Emmerich, W., Tai, S. (eds.) EDO 2000. LNCS, vol. 1999, pp. 194–214. Springer, Heidelberg (2001)
18. Liebig, C., Tai, S.: Middleware mediated transactions. In: Blair, G., Schmidt, D., Takizawa, M. (eds.) DOA 2001. IEEE Computer Society (September 2001)
19. Luckham, D., Schulte, R., Adkins, J., Bizarro, P., Jacobsen, H.-A., Mavashev, A., Michelson, B.M., Niblett, P., Tucker, D.: Event processing glossary (2011)
20. Mann, S.: ebizQ (2012), http://www.ebizq.net/topics/int_sbp/features/13366.html
21. Mietzner, R., Fehling, C., Karastoyanova, D., Leymann, F.: Combining Horizontal and Vertical Composition of Services. In: SOCA 2010 (2010)
22. OMG. CORBA Event Service (2004), http://www.omg.org/spec/EVNT/1.2/PDF/
23. OMG. CORBA Notification Service (2004), http://www.omg.org/spec/NOT/1.1/
24. Pietzuch, P., Bacon, J.: Hermes: A distributed event-based middleware architecture. In: ICDCSW 2002 (2002)
25. Russell, N., ter Hofstede, A.H.M., Mulyar, N.: Workflow ControlFlow Patterns: A Revised View. Technical report (2006)
26. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow Data Patterns: Identification, Representation and Tool Support. In: Delcambre, L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 353–368. Springer, Heidelberg (2005)
27. Sun Microsystems, Inc. Java Message Service (JMS) Specification - Ver. 1.1 (2002)
28. Weske, M., van der Aalst, W.M.P., Verbeek, H.M.W.: Advances in Business Process Management. Data Knowl. Eng. 50(1), 1–8 (2004)