On the Promises and Challenges of Event-Driven Service Oriented Architectures

Sven De Labey and Eric Steegmans
KULeuven, Department of Computer Science
200A Celestijnenlaan, B-3000 Leuven, Belgium
{svendl,eric}@cs.kuleuven.be

Abstract

Research on client-service interactions in Service-Oriented Architecture mainly focuses on request/response-style messaging. This strategy is based on explicit client-service communication because clients directly invoke operations that are exported by the public interface of the service. Explicit invocation is less suited for reacting to event occurrences. This led to the introduction of implicit invocation, a communication strategy that is extensively used in Event-Driven Architectures to signal that a particular situation has occurred. EDA and SOA complement each other because a typical service architecture needs both explicit and implicit invocation. The integration of both paradigms in a single architecture is often referred to as Event-Driven Service-Oriented Architecture.

In this paper, we provide an overview of the state of the art of EDSOA. We evaluate the key drivers behind Event-Driven Architecture and we show how concepts from EDA can be integrated into SOA. Then, we focus on a list of challenges, opportunities, and future research directions in current EDSOAs.

1. Introduction

Key technologies for building Service Oriented Architectures focus on request/response-style message passing. Web services, for instance, provide developers with means for interacting with heterogeneous services in an interoperable way based on XML message exchanges. But such explicit client-service interactions, where a client invokes operations exported by the predefined interface of a well-known target service, are only one piece of the messaging puzzle. To make this clear, consider a service architecture containing a printer service. Web Services technology would be a good choice to request *file printouts* because: (1) the invocation target is known in advance and (2) the responsibilities between the requestor and the provider are defined in a service contract (typically a WSDL document) that can be understood by both parties. But that same messaging strategy is far less suitable for indicating *event occurrences*. Examples of such events in the context of a printing environment include low ink levels, paper outages, and overpopulated job queues. In those situations, developers either don't *know* who is interested in the event, or they don't want to hardcode the event handling logic in the printer service. Indeed, doing so would increase the complexity and reduce the reusability and the maintainability of the printer software.

What we need is a way to deliver event notifications to those parties that somehow registered their interest in one or more events. Unlike Web Service interactions, (1) developers are unable to predict the target services in advance, (2) notification often comprises one-to-many communication and (3) there are no dependencies between service interfaces because the notification producer typically invokes a generic operation such as `notify` (rather than service-specific business methods). This messaging style is often referred to as *implicit invocation*. Rather than contacting target services directly (and thus depending on their interface), an implicit invocation mechanism only signals that an event has occured — it does not say what needs to be done to react to that event. So, in our example, it is *not* the responsibility of the printer to solve problems such as paper outages. The printer is just a messenger. This clearly improves its maintainability, and it eases reengineering processes. For example, altering the workflow for replacing empty ink cartridges can be done by replacing the *event handler* for low ink events. Since event handlers are external to the printer service, this workflow modification does not require modification of the printer software.

The observation that Event-Driven Architecture complements the request/response-oriented messaging style of client-service interactions from Service-Oriented Architecture underpins the introduction of Event-Driven Service Oriented Architectures (EDSOA). Such architectures contain a healthy mix of explicit and implicit service interactions, and there are various reasons why enterprises are willing to integrate event-driven aspects in their SOAs. A company may want to be warned when dangerous situations oc-

cur. Such issues are detected by sensors (e.g., a fire alarm) or they may be found after detailed analysis of relations between various event occurrences. Also, complex event correlation can lead to the detection of opportunities yielding a competitive advantage.

In this paper, we analyze the promises and challenges of Event-Driven Service-Oriented Architecture. We start by giving a quick overview of Event-Driven Architectures in Section 2. Next, we discuss how technologies such as active databases and event stream processing contribute to strategies and techniques currently used in EDSOA in Section 3. Section 4 then examines its promises and challenges, leading to a list of concrete research goals. A conclusion is presented in Section 5

2. A World of Events

We first give a concise overview on Event-Driven Architecture based on recent research literature. First, we define the stakeholders in a typical eventing architecture. Next, we present the concepts that are key to event-driven programming, such as (1) the difference between *primitive* and *composite* events and (2) some event *notification strategies*.

2.1. Event-Driven Architecture

A typical event-driven architecture comprises four key stakeholders: a publisher, an event, a mediator, and a consumer (see also Figure 1). The *publisher* is the source of the event. It is the printer service in the example we presented above. The second stakeholder, the *event*, is typically modelled as an object containing semantic information about the situation that occured. An event signaling a low ink level, for instance, may provide an estimate of the remaining number of printable pages before ink completely runs out. The third stakeholder, the *consumer*, is the party that reacts to the event. A printer monitor, for instance, may listen to *low ink* events and send an e-mail to the technical support team, requesting them to refill the cartridge. Finally, the *mediator* or *event bus* is the fourth stakeholder that manages relations between consumers and publishers. Similar to an Enterprise Service Bus, an event bus serves as a matchmaker between requestors (event consumers) and providers (event publishers). Its main responsibilities include (1) registration managament, (2) event correlation (as discussed further) and (3) the dissemination of relevant events to interested *event consumers.* The use of an Event Bus is not a conditio sine qua non for EDA, and its use is typically motivated when the need for centralized management of the abovementioned functionality increases.

From Figure 1, it immediately follows that event-driven service interactions provide a very high degree of loose coupling between services. First, no interface dependencies exist between the *publisher* and the *consumer*, since their connections are managed by an *event bus.* This is paramount

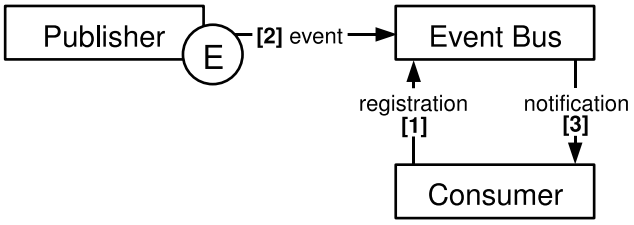

Figure 1. Overview of an Event Architecture

in such architectures: unlike dependencies between web services, publisher-consumer relations are often many-to-many relations. Also, note that this architecture is very dynamic: the targets of an event notification are unknown until the SOA is up and running and both the number as well as the characteristics of event consumers may vary over time.

2.2. Primitive and Composite Events

Primitive events are the most basic type of events. They reflect the occurence of a single, atomic issue or opportunity. In our printer architecture, events signaling that a file has been printed or that the job queue has been modified are instances of primitive events. Primitive events often reside in *inheritance hierarchies*. For example, a general `PrinterEvent` can be specialized in a `PrinterQueueEvent`, which can in turn be specialized into a `FullQueue` and an `EmptyQueue` event. Event consumers interested in all events concerning the queue of a printer, would listen to a general `PrinterQueueEvent`, whereas more specialized services would listen to technical, fine-grained events such as a `FullQueue` event.

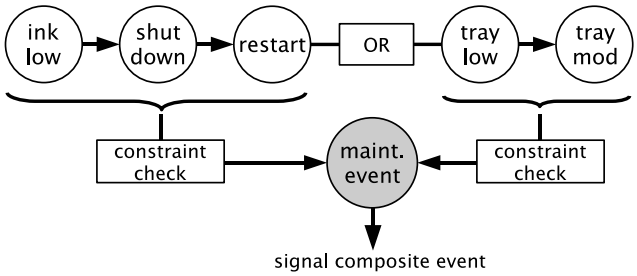

Figure 2. Composite Event as a combination of primitive events and constraints

Composite events, then, are combinations of primitive events. Unlike atomic events, the event publisher is typically *not* made responsible for detecting and signaling complex events. There are two good reasons for this: (1) complex event detection would severely complicate the code of the event publisher and (2) the introduction of new composite events would entail reengineering the code of the printer

service. Therefore, printers only signal primitive events, and the detection of complex event patterns (i.e. composite events) is done by the event architecture's mediator, the *event bus*.

Figure 2 gives an example of a complex event. It shows how a `MaintenanceEvent` is defined as the *disjunctive combination* of two *event sequences*. This means that there are two different event patterns marking the occurrence of a general maintenance operation: (1) a *low ink warning*, followed by a *shutdown* and a *restart* event, or (2) a *paper warning* event followed by a *tray modification* event, with the additional constraint that the paper amount returned to a normal level when the latter event occurs.

In summary, a *composite event* is a combination of *primitive events* using *composition operators* and optionally some *constraints* that allow for further limiting the circumstances under which an event is to be fired. When an event is detected and it satisfies the imposed constraint, it must be notified to the consumer. The next section presents two common strategies for doing that.

2.3. Notification Strategies

There are two basic notification strategies in event notification systems: *push* and *pull*. These strategies can be combined into more complex notification mechanisms, but that discussion is beyond the scope of this text.

Using *push-style* event notifications, the publisher immediately forwards the event to the consumer when it occurs, possibly with an event bus acting as a mediator between both services. This strategy ensures that events are delivered when they are most relevant. One liability is that this strategy may increase network pressure, but this depends on the expressiveness of the *Event Query Language* that is used to specify complex events (see further). With very expressive languages, consumers are able to specify *exactly* what kind of events they are interested in, as such reducing the number of irrelevant event notifications.

Pull-style notifications, on the other hand, expect the event consumer to regularly check whether an event has occurred. Network pressure is reduced when a sufficiently large polling interval is chosen because events can be retrieved in bulk. But this strategy also has two clear disadvantages. First, a larger interval may lead to the detection of events that have become irrelevant in the mean time. This is particularly problematic in the context of *warnings* or *alerts*. Late detection of an alert typically increases the cost of solving the underlying problem. For example, the longer a low ink level is ignored, the longer the job queue of the printer. Second, it also increases memory pressure on the event bus because it needs to cache events (without knowing the desired retention time).

Since we consider *event relevance* to be one of the most important properties in Event-Driven Architecture and because most existing systems implement push-style notification, the remainder of this text implicitly assumes the use of the first notification strategy.

3. The Advent of EDA and EQL

The roots of Event-Driven Architecture can be found in operating systems and distributed publish/subscribe systems. A more recent application domain where implicit invocation has seen widespread use is that of *active databases*. Those are databases that support the definition and execution of *Event-Condition-Action* (ECA) rules. If an *event* occurs (e.g., a database CRUD operation) and the specified *condition* holds, then the *action* of the ECA rule is executed. Examples of active databases and their accompanying event query language include SNOOP [3, 2], SAMOS [5, 6] and COMPOSE [7].

Next to active databases, a similar evolution can be found in *middleware systems*, such as CORBA [13] and its Event Service Specification [14] as a basic publish/subscribe mechanism. This service was later improved by the CORBA Notification Service [4, 19], which supports event filtering based on event-specific characteristics. The structured event types from CORBA-NS are also used by more sophisticated notification systems, such as READY [9, 8], which is in turn an extension of Yeast [10]. READY provides compound event detection and grouping constructs using separate communication channels and *event zones*. It also supports QoS-based event delivery. Another example of an event architecture based on CORBA is COBEA [11].

One of the first *event buses* is Oki's Information Bus [15], now developed by Tibco. This bus satisfies the need for an extensive distributed system to integrate legacy components with new systems. Integration is done using the publish/subscribe event notification paradigm. Components simply publish requests (events) on the bus, which are then answered by other components (listeners). By replacing explicit message routing by implicit invocation, this Information Bus achieves a very high degree of loose coupling.

The Open Services Gateway initiative (OSGi) [16] also ships with an event notification mechanism called the *Whiteboard* pattern [17], but currently, this pattern is only used for signaling non-functional events (such as services joining and leaving the architecture). Thus, EDA integration is currently not supported by the OSGi framework.

ESPER [12] is a framework for integrating Complex Event Processing (CEP) in Java. It allows developers to write complex event queries without forcing them to implement the bookkeeping algorithm. EJava [18] is another example of how event-driven programming can be integrated with Java.

The Staged Event Driven Architecture (SEDA) [20] takes this process one step further. SEDA shows that EDA can be used to overcome the scalability limits of *threads* in

highly concurrent internet servers. Task flows are now modelled in finite state automata rather than in separate threads, and the system is divided in several stages that can be replicated autonomously. This increases the scalability of the EDA without compromising concurrency.

3.1. Event Query Languages

Event-Driven Architecture stands or falls by the expressiveness of its Event Query Language (EQL). The unability to specify complex, composite events puts high pressure on event consumers because they will have to detect complex event patterns themselves. Also, event query languages with poor expressiveness refrain event consumers from clearly delineating their interest, leading to the dissemination of irrelevant events that satisfy the query, but not the interest of the event consumer. An expressive EQL should at least support the following concepts:

- **Basic composition operators.** These operators are supported by most event query languages. Examples include event *conjunction* (E occurs when events E_1 and E_2 have occurred), event *disjunction* (E_1 or E_2) and event *sequencing* (E_1 followed by E_2). An example of a composite event using these basic composition operators can be found in Figure 2.

- **Constraints** Every event in the composite event can have its own constraints referring to the specific characteristics of the event object. Also, the composite event as a whole can have constraints. The latter are often used for event correlation. Stating that every event must come from the same service, for instance, would be done using a correlation constraint.

- **Event non-occurrence.** Advanced query languages introduce an operator to specify that a given event should not occur in a certain interval. Such a *non-occurrence* operator can be used in combination with a timer to state that the event should not occur during the specified interval. Or it can be used in combination with a composite event, meaning that the event is not allowed to occur during the detection of the composite event. This advanced language concept requires a temporal stream model to represent non-occurrence intervals. CEDR [1] is an EQL that integrates such a model.

4. Promises and Challenges of EDA and EQL

So far, we have sketched what an EDA looks like, and we have given a quick overview on important technological evolutions that led to the rise of EDSOA. In this section, we analyze the benefits and challenges of combining explicit and implicit invocation mechanisms.

4.1. Promises

We focus on three advantages of integrating event-driven programming in Service Oriented Architectures: (1) information detection, (2) reusable event tracking, and (3) extensibility.

- *Information Detection.* The main advantage of an event bus is that it can detect events from various sources and correlate them to new, complex events. Request-response message systems are far less suited for this task because they would have to poll each service in turn. That technique would either lead to network chattiness (short polling interval) or missed event occurrences (long polling interval). Explicitly signaling events improves the *visibility* of changes that occur in the service architecture.

- *Reusable Event Tracking Logic.* The event bus serves as a centralized event detection system that can be used by every service in the architecture. The event bus can thus be seen as a reusable service that provides event correlation and registration management. With such a centralized service, every consumer would be responsible for its own event bookkeeping. This is manageable in local, object-oriented programs using (an extension of) the Observer pattern, but it is known to be a hard and error-prone process in the context of volatile, distributed service environments.

- *Extensibility.* Using event-driven programming, it is much easier to extend a services architecture. The newly introduced service does not create a need for modifying existing services, even if the new service needs to interact with already existing services. The new service simply listens to events that are emitted by those services, and reacts when a particular combination of events has occurred. This is much better than a callback system, where existing services must be modified so as to invoke the proper callback method of the new service.

4.2. Research Opportunities

In this section, we focus on the challenges of integrating EDA and SOA. First, we isolate the challenges that relate to programming with events. Then, we focus on challenges introduced by the Event-Driven Architecture itself. Finally, we also point out some difficulties that relate to Event Query Languages (EQL).

Events. The major challenge is to find the right level of abstraction for modelling events. Events can be represented at different levels of granularity, and there is no standardized way to represent composite events.

- *Event Granularity.* One important challenge is concerned with choosing the right granularity of events. Fine-grained events reflect small changes in the services architecture. This leads to the publication of very detailed information, but it also leads to the dissemination of a large number of events, thus putting a high strain on the network and on the processing power of the architecture. Such detailed events are also less interesting when consumers are interested in more general events. This can be solved by using coarser-grained event, but this leads to potential information losses. One advice might be to select the granularity according to the characteristics of the service architecture that will be deployed, but this reduces the reusability of the services in architectures with other properties.

- *Composite Event Representation.* There is no common agreed upon technique to represent composite events. In active databases, for instance, some runtime systems represent composite events using *finite state automata*. More complex approaches use *colored petrinets*, and even others model them using specialized *event graphs*. The latter were defined to satisfy the need for saving relevant *state information* for events. The main challenge here, is to integrate into the event bus a data structure that can save stateful, composite events without degrading performance and without increasing memory pressure.

- *Web Services Integration.* Web Services currently focus on request/response-style interactions. Some specifications for combining WS and EDA were proposed, such as WS-BrokeredNotification and WS-Topics. These documents propose a basic event notification system, but the expressiveness does not satisfy the complex needs of an EDSOA. Only primitive events can be signaled and no event *correlation* is supported. Compared with architectures such as SEDA [20], Web Services are severely lagging behind when it comes to implicit invocation.

Challenges EDA. The major challenge here is to deal with performance issues, and to satisfy the need for interoperability that naturally arises in heterogeneous service architectures.

- *Runtime Performance.* Research papers on Complex Event Processing (CEP) and Event Stream Processing (ESP) often refer to architectures were more than 1000 events are occurring each second. The more events arrive, the more difficult it becomes to find complex event patterns because of the combinatorial explosion. This complexity grows with the number of services running in the architecture and with the granularity of event dissemination (see above). This is different from request/response-style interactions, where correlation of client-service interactions is not supported.

- *Dynamic and Decentralized Nature.* The success of event stream processing in active databases is largely due to the nature of those databases. They are typically *centralized* and *static*. Service architectures, on the other hand, are always *decentralized*, with services running on heterogeneous and distributed computing environments. Also, they are inherently *dynamic*: services can join and leave the service architecture without publishing their intent. Such runtime issues call for a more complex bookkeeping algorithm at the Event Bus when compared to active databases technology.

- *Interoperability.* Migrating concepts from a centralized database to a distributed, heterogeneuous service architecture also introduces interoperability challenges. Services are heterogeneous because they were developed for different systems, or because they talk different protocols. One additional challenge, then, is to provide interoperability between different event processing systems. There is no standardized way to represent events and there is no agreed-upon language to define what a composite event looks like. Additionally, different programs depending on different event notification frameworks will not be able to understand each other's events. Contributions from the field of semantic services engineering will probably be helpful to tackle this challenge, but we did not find any research attempts to augment interoperability in that context.

- *Maintenance issues.* Most commercial object-oriented programming languages use garbage collection techniques to remove objects that have become unreachable in programs. It is clear that similar techniques will be needed to prevent the Event Bus from running out of memory. Making every event consumer responsible for cleaning up his event queries at the event bus brings us back to C++-style memory management, which is barely manageable in the dynamic context of a service architecture. A clear maintenance strategy is needed, for instance, if an event consumer leaves the architecture. Do we need to cache occurrences until he returns? And how long do we need to do that? Some research has been conducted to share (parts of) event graphs between different event consumers. This reduces memory pressure, but it requires a more sophisticated memory manager.

Challenges of Event Query Languages. Finally, we discuss some challenges that are generated by the query language that is used to specify *composite events*, event *constraints*, and possibly *reactions* to event occurrences.

- *Language Integration.* Event publishers and consumers are often implemented using object-oriented programming languages, but these languages provide no integrated concepts for writing event queries. Consequently, event query languages are provided by frameworks that rely on reflection or meta-object protocols. This strategy bypasses important compile-time guarantees because queries are passed as *strings*. Compilers are thus unable to check the syntactical correctness of event queries. This lack of static guarantees increases the odds for introducing bugs, and these bugs are extremely hard to detect, given the decentralized nature of an implicit invocation mechanism. The challenge here, is to evolve programming language with specialized concepts for seemlessly integrating event queries.
- *Expressiveness.* As already mentioned before, an important challenge is to maximize the expressiveness of the query language by introducing sufficient composition operators. Event *conjunction* and *disjunction* are easy to implement. Modeling non-occurrence, however, is harder as it forces the event bus to take into account timing and relevance issues. One of the hardest composition operators are *causal sequences*. Two events are causally related when a handler for event A triggers the notification of a new event B. The detection of causal relations requires intimate knowledge about the source code of event handlers, and it is currently unclear how such information can be revealed without compromising important encapsulation requirements that are paramount in decoupled SOAs.

5. Conclusion

A typical service architectures comprises both explicit and implicit service interactions. SOA and EDA focus on providing these mechanisms, respectively, and their integration leads to the concept of EDSOA. We have focused on research opportunities introduced by Event Driven Architectures and Complex Event Processing. One of the bigger challenges will be to integrate Event Query support into current programming models. Another research direction is the integration of Event-Condition-Action rules in a dynamic, decentralized, and heterogeneuous service architecture.

References

[1] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. *CoRR*, 2006. informal publication.

[2] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, 1994.

[3] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[4] R. S. S. Filho, C. R. B. de Souza, and D. F. Redmiles. The design of a configurable, extensible and dynamic notification service. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, 2003.

[5] S. Gatziu and K. Dittrich. SAMOS: An Active Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):23–26, December 1992.

[6] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *RIDE-ADS*, pages 2–9, 1994.

[7] N. Gehani, H. Jagadish, and O. Shmueli. Compose: A system for composite event specification and detection. In *Advanced Database Concepts and Research Issues.*, 1994.

[8] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[9] R. E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the READY event notification system. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for Composing Distributed Applications*, pages 195–202, 1998.

[10] B. Krishnamurthy and D. S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, 1995.

[11] C. Ma, C. Ma, and J. Bacon. COBEA: a CORBA-based event architecture. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 9–9, 1998.

[12] F. Marinescu. Esper: High volume event stream processing and correlation in java – http://www.infoq.com/news/esper–esp-cep, 2006.

[13] Object Management Group (OMG). Common object request broker architecture (corba) – http://www.corba.org/. 1991.

[14] Object Management Group (OMG). Corba event service v1.2 – http://www.omg.org/technology/documents/. 2004.

[15] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the 14th ACM symposium on Operating Systems Principles*, pages 58–68, 1993.

[16] Open Services Gateway Initiative. *http://www.osgi.org*.

[17] OSGi Alliance. Listeners Considered Harmful: The Whiteboard Pattern. A Technical Whitepaper – www.osgi.org/documents/osgi_technology/whiteboard.pdf. 2004.

[18] A. Santoro, W. Mann, N. Madhav, and D. Luckham. ejava-extending Java with causality. In *Proceedings of Tenth International Conference on Software Engineering and Knowledge Engineering*, pages 251–60, 1998.

[19] D. Schmidt and S. Vinoski. Object Interconnections – Distributed Callbacks and Decoupled Communication in CORBA. In *C++ Report*, 1996.

[20] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.