SOA, EDA, BPM and CEP are all Complementary
by
David Luckham

Part I

Approaches to designing and managing information systems have proliferated over the past 15 years, so much so that the space of technical concepts has become quite confusing. There is the **SOA** arena (service oriented architectures), the **BPM** arena (business process management), and more recent arrivals in the area of event processing (**EP)** including event driven architectures (**EDA**) and complex event processing (**CEP**). Some of these technical movements, particularly SOA and EDA, have often been viewed as competing or conflicting, and religious wars have threatened to break out from time to time. [1]

However, the truth is that at the conceptual level they are complementary, and they all have a role to play in design and management of IT systems.

First, a word about SOA. If one goes to a resource that is independent of the vendors such as Wikipedia for a definition, one finds:

"There is no widely-agreed upon definition of **service-oriented architecture** other than its literal translation that it is an architecture that relies on service-orientation as its fundamental design principle."

Why is this? I've always understood that SOA has two top level defining concepts, *modularity* (in the computer science sense[2]) and *remote access*. Simple enough, one might think. But SOA has become commercial big time. There are competing standards and proposed standards to design, implement and deliver SOA. Every vendor has products that claim to do SOA or make SOA easy. Secondary ideas on design process and implementation are often hyped as conceptual level principles. The two top level principles have become

---

[1] Google "SOA 2.0" for a recent view of this.

[2] Wikipedia: A **module** is a software entity that groups a set of subprograms and data structures. Modules provide a separation between interface and implementation. A module interface expresses the elements that are provided and required by the module. The elements defined in the interface are visible to other modules. The implementation contains the working code that corresponds to the elements declared in the interface.

"fuzzed up" with other ideas in these commercial battles about how to do SOA.[3] That's what happens when a simple idea gets to be big business! So, just remember, at the conceptual level there are only two main principles, *modularity* and *remote access*.

What is a *service*? Think of it as something you want to use, a weather report or a credit rating. Send the weather service your zip code and you get back a report for your local area. In its purest and oldest form, a service is a function — you call it with data and you get an answer.

SOA (service oriented architecture) focuses on the user's view of a system. At a conceptual level, SOA is really an extension of earlier object-oriented programming ideas. The *modularity* principle deals with the organization of related services into a single server module – an abstract interface and its implementation. So, sets of related services must be grouped into server modules. A server consists of two parts: (1) An interface that specifies the services that are provided and contains meta data defining how they behave. (2) A separate implementation of those services. An implementation (or module body) can be anything from, say, pure Java code, to an adaptor that maps the services to the facilities of some middleware product.

An interface tells a user what the services do and how to use them. The interface is the users' view of the services. On the other hand, the separate implementation is hidden from users and can be changed without the users knowing. So, one of the goals of SOA is that users of services don't have to understand how they work, which is pretty much the way drivers of cars operate these days. Of course, the implementation has to be truthful to the interface metadata – i.e., when you step on the brake pedal the car should stop. Consistency of interface and implementation is something that is glossed over in most SOA presentations. Inconsistencies can be a source of surprises for the user – but that's nothing new in software, is it?

Actually, a modern interface design contains other information. It will also specify which services are *required* by the server module itself – i.e., the services that may be used to implement the services that are provided. The reason for required service specifications has to do with architectural design issues that are beyond the scope of our article. Think of them as a privacy statement – "when

---

[3] Google "service oriented architecture" and see offerings like "SOA made easy" or "What is SOA?", etc.

you use one of our provided services we will only send your data to these required services."[4]

In olden days a SOA module interface was an object written in a programming language like Java, but these days it is likely to be the user interface of a website. For example, services related to using stock market feeds, such as computing various averages and statistics over time windows, e.g., VWAP, are provided by a market feed website. See for example www.traderbot.com, a website that provides these services.

The *remote access* principle of SOA requires that services should operate in a distributed computing environment. SOA must therefore provide users with an ability to access services remotely. The remote access principle has different paradigms. Access in traditional SOA[5] was by remote procedure call (RPC) also called Request/Reply (R/R). Figure 1 is a schematic view of a traditional SOA containing interfaces for emarketplace services, users (buyers and sellers) and service access by RPC.

---

[4] The motivation for required service specifications is to enable implementations to be changed without introducing inconsistencies, so called plug -and-play – see "The Power of Events", Section 4.6.1.

[5] By traditional SOA we refer to the state of SOA thinking about the time of the initial release of CORBA in 1991.

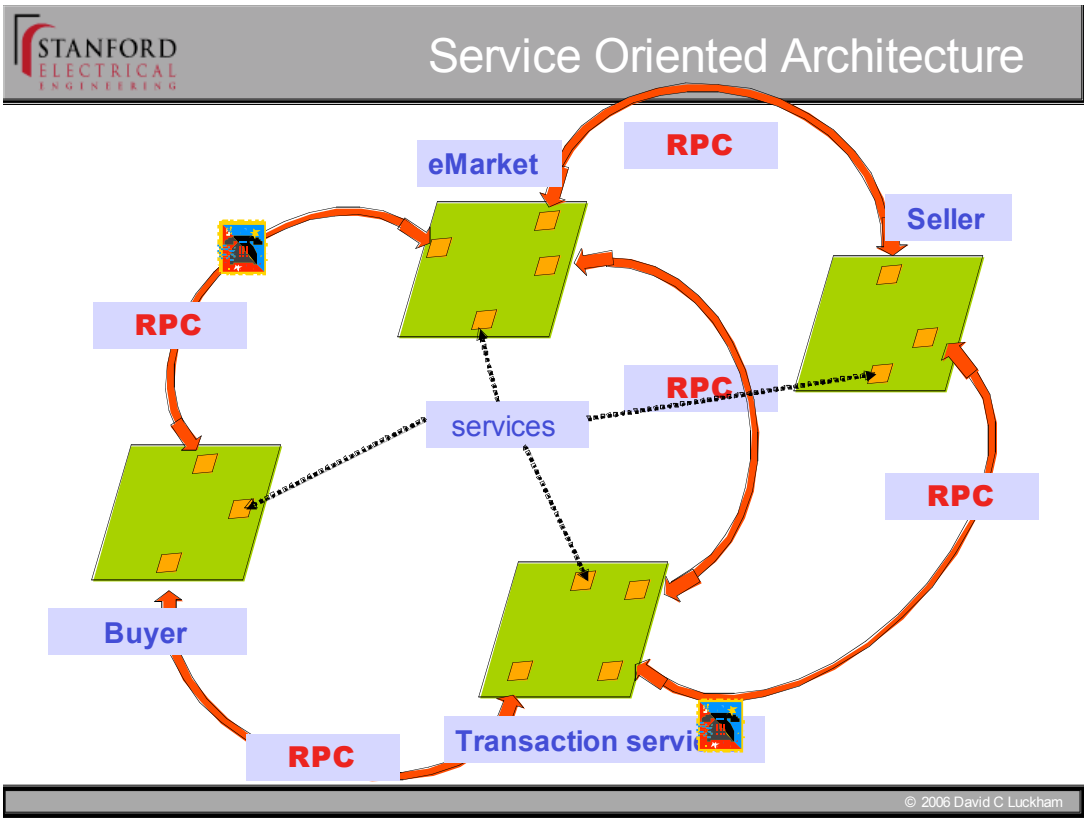

Figure 1: Traditional SOA with server interfaces and remote procedure call

RPC involves a synchronizing handshake between user and service. A user sends data to a service (e.g., the remote call to a function) and blocks, waiting for a reply, (e.g., the value of the function on that data). The user is not free to do anything else until the reply. An everyday example of RPC is the telephone call. This may be a little more conversational that a straight request/reply call, but it is a remote access, and requires synchronization. And as we all know, phone calls can often waste a lot of the caller's time. We've all had the experience of phoning an airline for a service such as booking a flight and first having to deal with an automated menu of services we don't want before reaching a human who can provide the service (or answer the question) we do want. A lot of that time is wasted time for the user. Nice for the airline who can employ fewer humans, at the expense of the user!

Event processing (**EP**) at the level of business events now enters the picture with a new conceptual paradigm for remote access. A user no longer needs to access a service by RPC. Instead a user can access services by sending and receiving events, asynchronously.

The up side of event driven service access is that it is much more versatile and efficient than RPC. It allows all actors – users and services – to multitask. Instead of a phone call you send an event, lets say over the Internet. In an event driven SOA a call to an airline is executed by the user sending a request event, “Need Service”. A protocol reply is “Here is the menu of services”. The user chooses to send the event answer, “Human contact needed, call back number N”. Response event , “your wait time will be exactly 15 minutes; make sure your call back number is free in 15 minutes, you will receive only one call back event”. Now the user is free to use the 15 minutes wait time to do other things instead of hanging on the end of a phone line.

Perhaps it is true that communications have been driven by events since the earliest days of networks. First, there were network protocols like TCP. Then the level of events in communications gradually rose with the advent of pub/sub middleware, message queues etc. Nowadays the communication between users and services in SOA can always be by means of events. This gives us event-driven SOA (ED-SOA). I’ve shown this in Figure 2.

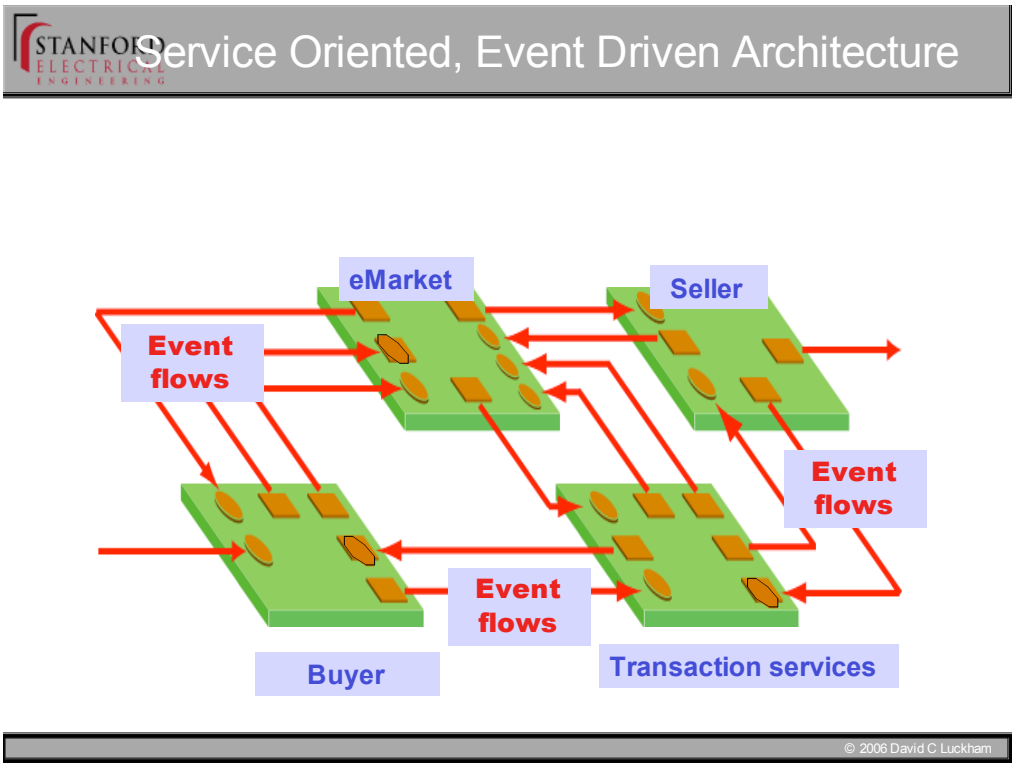

Figure 2: SOA with event-driven communication.

Event driven communication decouples the handshake by using events and protocols. So event-driven SOA is an evolution of traditional SOA whereby the communication between users and services is by events instead of RPC. Services are triggered by, and react to events instead of procedure calls. Usually the implementations of services use communication standards such as WSDL, and SOAP for remote access.

There is always a downside, of course. The increased versatility has magnified opportunities for everyone – users, services and, of course, crooks. Scalability has become another issue. We're talking about real time operations with, in some cases, very large numbers of events – e.g., 200,000 business events/sec in stock market operations. So now the management and business intelligence issues are becoming more and more challenging. This is where BPM and CEP come into play – more about this in Part 2.

Now, beware! When you read further you may be told that "service component architectures (SCA) are part of SOA", or "a SOA must use XML and WSDL and SOAP" or "SOA must contain load balancing and central service catalogues" or … Indeed, these are all good and useful things that can play their part in various implementation approaches to building SOAs.[6] But, at the conceptual level, there are only two principles in SOA.

Event driven architecture **(EDA)** and SOA are related in two ways. First, event driven access to services is an *implementation paradigm* for SOA – not a competing technology. ED-SOA is a very good way to organize distributed services over any kind of IT infrastructure. And we're talking real-time use. Pretty much any modern enterprise's IT operations are organized as an ED-SOA, especially if they use the Internet or commercial networks such as SWIFT. This includes financial services, stock markets, cellular service providers, automated supply chains and inventory systems, on-demand manufacturing, SCADA control systems (e.g., for power stations, dams), banking and retailing - the list is endless. In fact, I have a difficult time thinking of anything that isn't event-driven these days. Of course, there's always legacy in most IT infrastructures, so in practice what you see is a mix of event driven and request/reply.

And the EDA field is expanding. New event-driven communication protocols are being developed all the time to support new applications such as Mobile Adhoc Wireless Networking to enable wireless Mesh computing (e.g., the OLPC project). These technology developments are all event-driven.

Secondly, SOA is a *design paradigm* for event driven applications. In software systems we've seen this happening, first in the middleware products, and more recently in the enterprise service bus offerings. But SOA designs of event driven systems have been around for nearly 50 years in hardware designs – long before SOA in fact! That is why I have used a hardware diagram in figure 2. Typically all the components in a hardware architecture have interfaces that present services, i.e., input events that request a service and output events that deliver the service. And their implementations are hidden. An adder requires two input events and then delivers sum and carry output events (i.e., its service). Registers, ALUs etc. all define similar services with interfaces presenting their services (inputs and outputs). And the communication between the interfaces of these components is by triggering events flowing down the connecting wires, timed by a controller. That is, the architecture is a wiring up of the interfaces of

---

[6] Although I have never had complete confidence in Wikipedia, the entries on SOA are as good a documentation of the state of concept chaos in the SOA implementation world as I have found.

the components. It is simple, it is SOA, it is disciplined design, and it is much less error-prone than software. Indeed, why can't software be more like hardware!

To summarize thus far, SOA and EDA are complementary. EDA is the paradigm for communications in SOAs, and SOA is the design methodology for EDAs. In the future we would like to be able to define EDA as follows:

An event driven architecture (EDA) is a SOA in which all communication is by events and all services are reactive event processes (i.e., react to input events and produce output events).

Many systems out there today cannot conform to this since many of their components are not event driven[7] and they were not designed with SOA principles in mind. Which brings up another question: Is there a test for whether or not a system is SOA or EDA? This seems to me to be about as difficult as asking if a system has a good design or not. Some systems are clearly SOA (e.g., built in Java using OO techniques and JMS). Others are clearly not (e.g., lumps of Cobol code). But there's a large space of systems in between. Perhaps someone might design a questionnaire for users: "Is the system you are using SOA/EDA?" It might contain questions like (with a scale of satisfaction from 1 to 10):

- Do you find services are organized in logical groups?
- Is their documentation easy to understand?
- Do the results from services agree with their documentation?
- Do you need to understand how services are implemented in order to use them?
- Can you find the service you're looking for easily?
- Are you happy with your service response time?
- Do you get surprising results when you request a service?

...

and so on.

Finally, the proposed definition of EDAs has to be viewed as a philosophy for the future. The spirit of the philosophy is this: "think hardware designs. Then think hardware designs in which the numbers of components and the pathways

[7] See the definition of EDA in the "Event Processing Glossary" http://complexevents.com/?p=124.

connecting them can vary at runtime. And then, think multi-layered designs." That is, dynamic, multi-layered, even driven architectures.

Acknowledgment I am indebted to Roy Schulte and Tim Bass for suggestions and comments on this article.