SOA Best Practices

Building an SOA using Process Governance

If SOA is to become the de facto enterprise standard, SOA scalability needs to be addressed. This can be achieved by using Process Governance to manage the SOA development life cycle.

Background

Although the mere mention of Service Orientated Architecture (SOA) is sufficient to attract the attention of most IT managers this is still an early stage market. For SOA to continue up the adoption curve, SOA implementations need to be delivered faster, and managing complex processes over loosely coupled, fragmented systems needs to be simplified - SOA needs to become scalable. Until SOA scalability can be demonstrated, adoption by the late-majority users is likely to remain contained.

Difficulties with SOA scalability are being experienced in a number of areas with customers and practitioners typically raising concerns relating to:

- the time and resource required to generate a system specification
- ambiguously defined system specifications resulting in a high occurrence of implementation errors
- the poor visibility of the overall implementation resulting in complexities in on-going system maintenance
- the inability to ensure conformance of the executing process against the originating specifications
- the requirement to assess the impact of an implementation change prior to the alteration of code

The generation of a system specification is problematic. Currently the business analysts document the interactions between co-operating services using a standard graphics package with explanatory notes being attached typically generated in MS-Word. As business processes become more complex, maintaining control and visibility of the required message exchanges using these tools becomes more difficult. As complexity increases, ambiguity in the system specification compounds, resulting in implementation errors.

The containment of implementation errors is being addressed by continuous testing at each stage of the build, with numerous adjustments being made until the required system outputs are achieved. If SOA is to scale, this bespoke process of fashioning, fitting and refashioning needs to be industrialised. To achieve this three enhancements to current practices are required:

- the introduction of a formal link between the design time and run-time to ensure the delivered implementation is aligned with the originating requirements
- continuous validation of the executing events to ensure the implementation remains aligned with the specification
- provision of overall visibility of the inter-dependencies between co-operating services to assess the impact of an implementation change before such change is introduced

The current approach to delivering an SOA system specification lacks formal validation that the system design will deliver the required business result. This ambiguity is compounded by the absence of a formal link between the design and the implementation phases, with the potential result that the implemented system may not deliver the required process.

Design-Time Process Governance

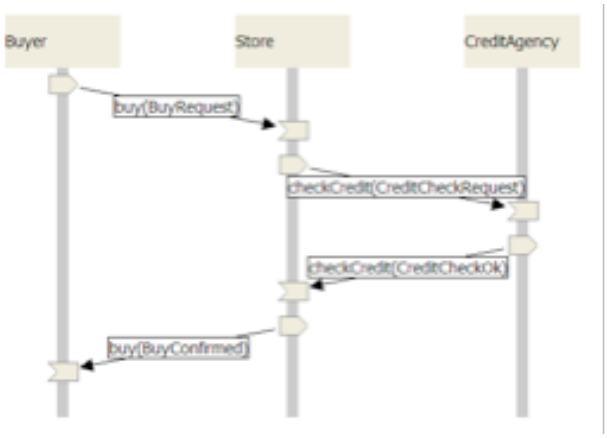
The concept of Design-Time Process Governance represents the ability to provide formal links across the various phases of the SOA development life cycle in a manner that enables each phase to be validated against the originating requirements.

Gather Business Requirements

The first phase is to document the requirements. To ensure these requirements may be validated against the implementation, these need to be described in a machine-processable manner. To achieve this, the requirements are gathered in two categories: example business messages illustrating the interactions to be passed around the system, and example scenarios detailing the alternative paths the process may follow.

By way of example, one scenario may represent a customer completing a transaction, following a successful credit check: whilst the alternative path may be an unsuccessful transaction resulting from a credit check fail. These alternative execution paths, or "scenarios" are documented using enhanced sequence diagrams, the enhancement being the attachment of example messages to each interaction that are later used to validate the implementation. A simple purchasing example illustrating alternative execution paths is shown in Diagram 1 below:

Scenario 1: Successful transaction



Scenario 2: Unsuccessful transaction

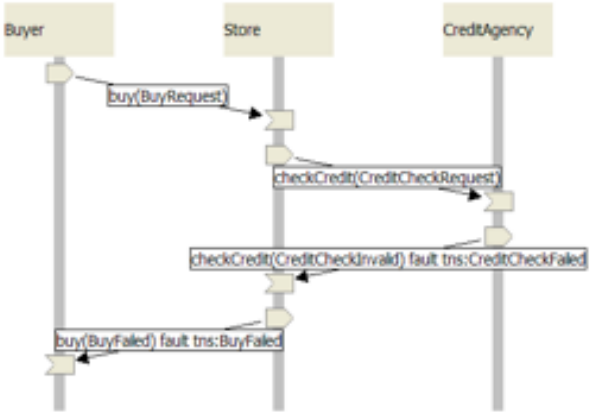


Diagram 1: Example enhanced scenario diagrams showing alternate possible execution paths with example messages attached the each interaction

Process Design

The next step is to build the Process Design by creating or re-using the type definitions for the example messages and the dynamic behaviour defined in the scenarios. If the example messages are XML based, the message type can be defined using an XML schema. To describe the dynamic behaviour, the type definition can be represented using a choreography description language, for example CDL.

For the purpose of clarity, the term "choreography" describes the interactions between a set of participating services from a neutral or "global" perspective. This is not to be confused with the concept of orchestration, which provides a description of behaviour only from a service specific perspective.

An example of a choreography description of the Process Design, illustrating the two scenarios detailed in Diagram 1 is provided below:

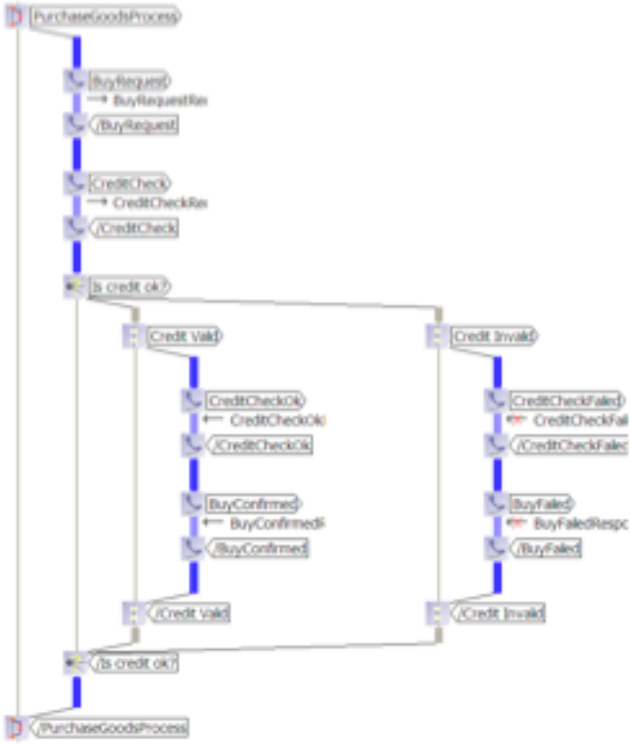


Diagram 2: Global description of the required message sequencing for the two example scenarios

Once the type definitions have been defined, the example messages are used to validate the scenarios. This ensures the type definitions correctly represent the originating requirements. This is an important step, as the type definitions will be subsequently used to implement the system. This provides the validation link between the scenarios described in the requirements gathering phase which in turn represent the business requirements. In circumstances where XML based example messages are used, validating parsers may be used to ensure that the example messages conform to the XML schema.

The choreography description is validated by simulating the execution paths using the example messages assigned in the scenarios. Below is an example of a scenario displaying a validation error - in this case *buy(BuyConfirmed)* following a *checkCredit(CreditCheckInvalid)* message response:

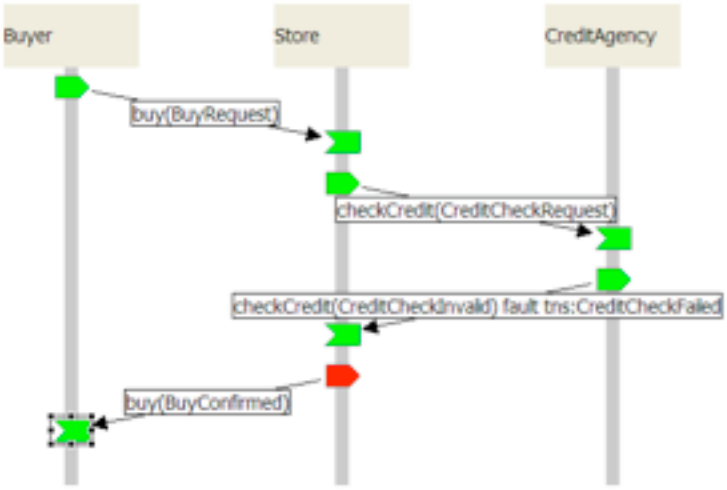


Diagram 3: Scenario displaying an error as a result of an incorrectly described type definition - in this case an incorrect behavioural sequencing

Where errors are detected, the system architect needs to determine whether the choreography description or the scenario is incorrect. If the scenario has been signed off by the business, as representing a valid use case, the system architect needs to revalidate the scenario with the business to ensure it has been correctly described. If this is the case, the choreography description requires adjustment to reflect the required use case.

Once the choreography description of the Process Design has been validated as correct, it may be exported to a variety of formats including BPMN, UML activity/state diagrams and HTML for review and sign off.

Evaluating Existing Services

Once the Process Design has been validated, the system architect needs to identify the services that are available for re-use, those that may be re-used but require modification and those services that need to be constructed. To achieve this the system architect needs to understand the specific behaviour required of each service in the context of the Process Design. This is obtained from the Process Design using a technique called "endpoint projection". This generates the *endpoint behavioural description* for each service.

In the above example, the *endpoint behavioural description* of the Store service is represented as follows:

```
Receive BuyRequest from Buyer;
Send CreditCheckRequest to CreditAgency;
choice {
Receive CreditCheckOk from CreditAgency;
Send BuyConfirmed to Buyer;
} or {
Receive CreditCheckFailed from CreditAgency;
Send BuyFailed to Buyer;
}
```

The *endpoint behavioural description* is used to interrogate the Service Repository and identify the availability of services that either match, or partially match the required behaviour. In cases where there is only a partial match, a gap analysis needs to be performed to determine whether it is appropriate to modify the existing service or build a new version.

Service Design

For services that need to be developed, the *endpoint behavioural description* can be used to generate a template of the Service Design using BPMN, UML state diagrams, and other representations. This Service Design may then be elaborated by a designer, to provide further detail regarding the internal implementation of the service. The following example displays the BPMN template for the Store service in the above example:

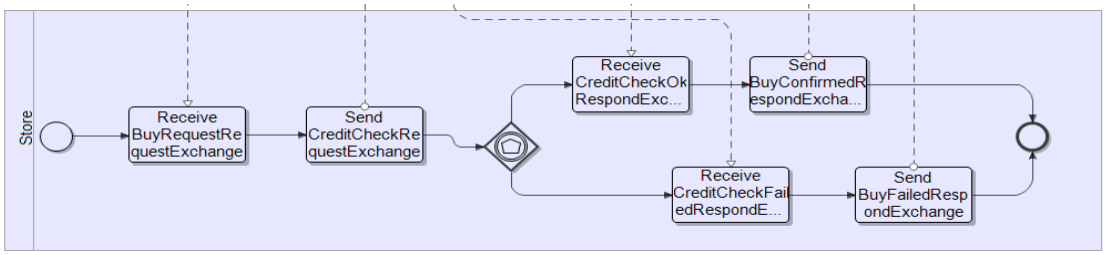


Diagram 4: BPMN template for Store service

Dependent upon the notation used to document the Service Design, it may be possible to validate the Service Design against the service's *endpoint behavioural description* derived from the Process Design, to ensure that the design conforms to the expected behaviour - although this is not currently possible using BPMN or UML unless strict conventions are followed by the designer, to enable the communication behaviour to be derived.

Service Implementation

When a service is ready to be implemented, skeletal code for the implementation is able to be generated from either the *endpoint behavioural description* or the Service Design.

An example of WS-BPEL code generated from the *endpoint behavioural description* of the Store service would be:

```
<process name="Store"... >
    ...
    <sequence>
        <receive createInstance="yes" operation="buy" partnerLink="Store"
                                portType="tns:Store"/>
        <scope>
            <faultHandlers>
                <catch faultName=”tns:CreditCheckFailed” >
                    <reply faultName="tns:BuyFailed" operation="buy"
                          partnerLink="Store" portType="tns:Store"/>
                </catch>
            </faultHandlers>
            <sequence>
                <invoke operation="checkCredit" partnerLink="CreditAgency"
                                portType="tns:CreditAgency"/>
                <reply operation="buy" partnerLink="Store" portType="tns:Store"/>
            </sequence>
        </scope>
    </sequence>
</process>
```

As the service implementation evolves, the service's *endpoint behavioural description* may be used to ensure conformance to the behaviour specified by the Process Design.

As the Process Design has been validated against the originating requirements, thus provides full behaviour traceability from requirements through to implementation, although this may only be achieved where the implementation language enables the communication structure to be derived (as in WS-BPEL, jPDL, and JBossESB conversational actions).

Service Testing

Once the services have been implemented, modified or existing services located in the Service Repository, the scenarios are used to perform Service Unit Testing.

Where the scenario shows a message being sent to a service, a service unit test harness is used to deliver the example message specified in the scenario, to the service. Where the scenario shows a message being sent by a service, the message can be intercepted by the service unit test harness and compared against the expected example message as specified in the scenario. This ensures the *endpoint behavioural description* of the service has been correctly described.

As the internal behavioural conformance of a service has been validated as it is being developed, this ensures the service implementation conforms to the originating requirements. Using the example messages specified in the scenarios re-enforces this conformance as an incorrect implementation would result in the service attempting to send and receive messages at variance with the example messages.

Service Deployment

Once the *endpoint behavioural description* for each service has been validated, the service may be deployed and the *endpoint behavioural description* recorded at the Service Repository. Once recorded this can be used either in subsequent Process Designs, to locate a service for re-use, or as part of the runtime discovery of a service based on the required behaviour.

Summary - Design Time Process Governance

Design Time Process Governance consists of six phases:

- Gather Requirements
- Process Design
- Service Design
- Service Implementation
- Service Testing
- Service Deployment

The following diagram illustrates these six phases together with the associated inter-locking conformance checking and generation cycles that ensure the deployed services are aligned with the originating requirements. This shows the use of Process Governance to generate the service related artifacts and to ensure their continuing conformance with the originating requirements as the system evolves through to deployment.

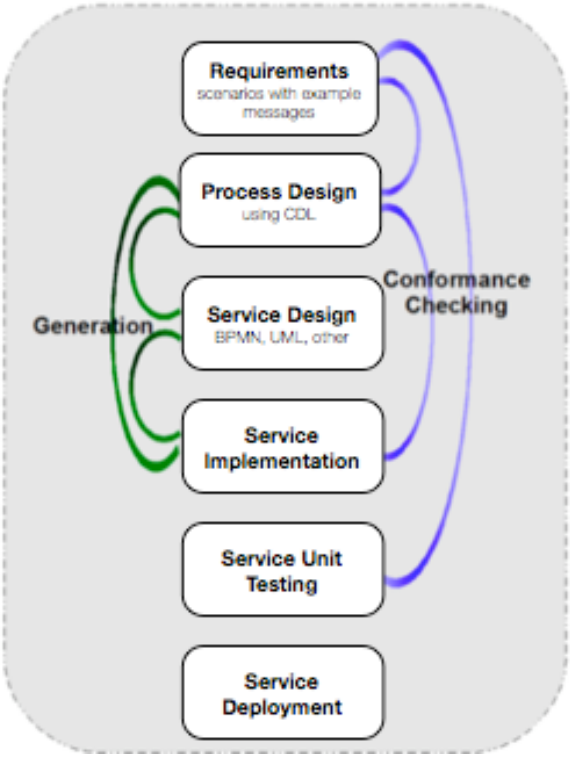


Diagram 5: Illustration of the interlocking conformance checking and generation cycles

Runtime Process Governance

Design Time Process Governance enables incompatible behaviour to be detected prior to service deployment, which in turn contains the risk of runtime processing errors. Runtime Process Governance is required to ensure ongoing validation of the process against the originating requirements. This contains the risk of "implementation drift" as the services are modified and new services introduced.

This is achieved by the use of Service Validators deployed to each service. The validators report all service interactions to the Process Correlator which reconstructs the global representation of the transaction using the Process Design. This is illustrated in the following diagram:

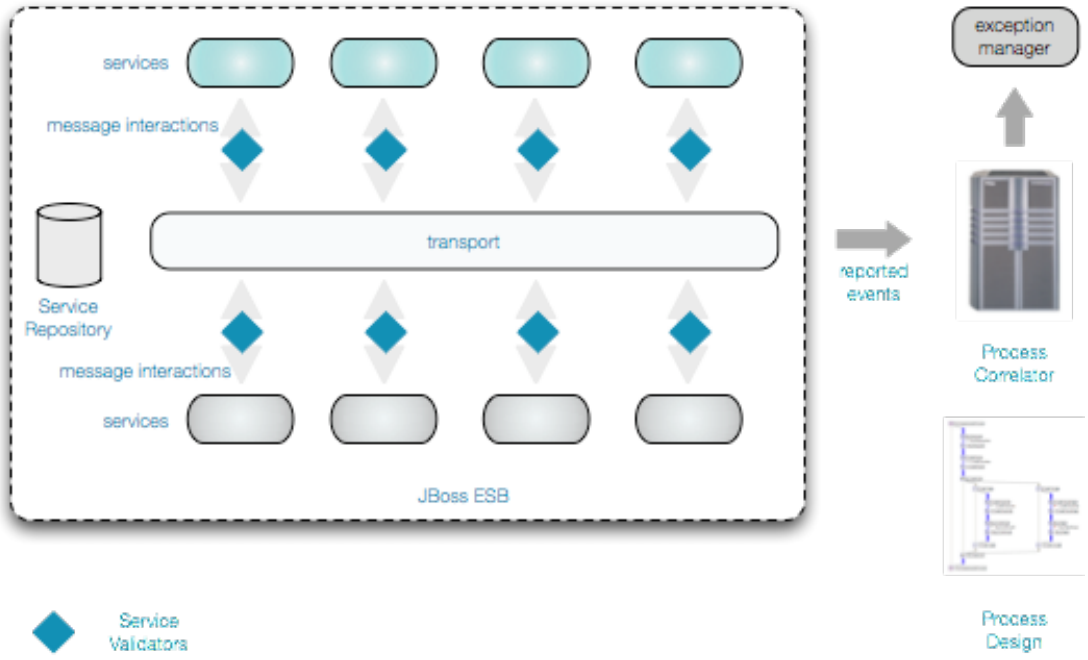


Diagram 6: Block diagram illustrating the location of Service Validators monitoring service interactions, the Process Correlator comparing these interactions against the Process Design, and the reporting of exceptions

Service Behaviour Validation

The Service Validators monitor the inbound and outbound messages for each service and compares these with the expected service *endpoint behavioural description*. In "active" mode, the Service Validators will block out-of-sequence or unexpected interactions thus protecting the downstream process: in "passive" mode, messaging events, whether compliant or not with the Process Design are allowed to execute. All interactions, including non-compliant events, are reported to the Process Correlator.

Process Correlation

The Process Correlator is a centralised correlation engine. It acts as a repository for the send, receive and error notifications as reported by the Service Validators and reports variances where the correlated interactions fail to comply with the Process Design.

This acts as a conformance validation cycle to ensure the overall SOA implementation is correctly executing according to the originating requirements. As all messaging events are reported to the Process Correlator, fine grained, granular visibility of the system is achieved. This includes the collection of messaging time stamps, used to identify and monitor relevant process performance metrics.

The Process Correlator differs from the conventional approach of monitoring each service individually by determining end-to-end processing conformance and ensuring compliance with the higher level Process Design. This addresses the shortcomings of specific service monitoring that does not guarantee the detection of non-compliance or missing interactions that may impact downstream processing.

This is illustrated by the recent problems at the opening of Heathrow Airport Terminal 5, London. In this SOA implementation, whilst each sub-system was functioning as designed, the overall process was not executing as required. In this case, the baggage handling sub-system was activated with a filter in place preventing it from communicating with inter-related systems. This prevented the baggage handling system from accessing the delivery destinations for the checked-in luggage. It returned the luggage to the terminal, resulting in aircraft departures without the passengers baggage having been loaded. This resulting chaos led to losses estimated at £16 million.

Process Governance contains this risk by ensuring the *overall* runtime behaviour is executing in accordance with the "global" description of the originating requirements.

Process Maintenance

Process Governance performs an important function both during the Design Time and Runtime phases of an SOA project. It also performs an important function in the Maintenance Phase. Maintaining robust control over implementation changes across a highly distributed infrastructure is a complex problem and is critical to maintaining overall system integrity. The following methodology describes how this is achieved using Process Governance.

Update Requirements and Process Design

Implementation changes may be specified using the originally generated scenarios or by generating new scenarios to illustrate the required change. Once the scenarios have been updated the Process Design is modified to reflect the required changes and validated against the scenarios.

Implementation Impact Analysis

Implementation Impact Analysis is a static check to identify the potential impact of the proposed changes on the deployed system. This enables incompatibility between the deployed implementation and the modified implementation to be identified thus reducing the resource required on system testing and the potential for inadvertently introducing runtime errors. An impact analysis needs to be performed in relation to all modified and introduced services to determine the effect the proposed changes will have on clients of the service. This ensures any updated services are still able to interact with other services in a manner that is consistent with their *endpoint behavioural description*. As services may be shared across multiple systems, any reference to the service being changed needs to be validated.

It is important to ensure the Service Repository contains information on all the inter-dependencies between clients and services otherwise only a partial Implementation Impact Analysis is possible.

The benefit of using Process Governance, when introducing implementation change, is that it allows clients of a new or modified service to be analysed in order to determine whether the client's requirements continue to be met by the modified behaviour. This is achieved by using conformance checking between the client's required *endpoint behavioural description* of the service and the actual behaviour provided by the service.

Implement and Deploy the Service Changes

Once the Implementation Impact Analysis is understood, it will be possible to plan the implementation and deployment of any new and modified services. If the required change is complementary to existing client usage, then the service deployment may simply be upgraded.

In situations where the change is incompatible with existing client usage, and it is not possible to simultaneously update the affected clients, then a modified version of the service will need to be deployed along side the existing version. This should not present any difficulties if service details are "hard-coded" into the relevant clients, to ensure they access the correct version of the deployed services.

However, if dynamic service discovery is used, then Process Governance will be required to ensure the impacted clients are furnished with the correct reference to the appropriate version of the service. This is achieved by the client supplying the required *endpoint behavioural description* when performing the service lookup. The client supplied description may be compared against the service description recorded in the Service Repository to ensure conformance.

Summary

This paper describes three enhancements to the delivery of SOA. These enhancements eliminate system specification ambiguity and enables SOA adoption to be industrialised across a broad spectrum of early to late-majority adopters.

The introduction of Process Governance delivers the formal linkages currently absent across the multiple stages of the SOA development and maintenance life cycles. These formal linkages ensure the resulting system is aligned and remains aligned with the originating requirements.

Process Governance may be introduced at any stage of an SOA project. It is possible to use agile software development techniques to implement an SOA and then to reverse engineer the Process Design to act as a reference model for testing and verification in order to obtain traceability back to originating requirements "after the fact". This approach can also be used to document existing SOA implementations in order to obtain ongoing benefits related to implementation change.