Picasso: A Service Oriented Architecture for Model-based Automation

Sharad Singhal, James Pruyne, Vijay Machiraju
Enterprise Systems and Software Laboratory
HP Laboratories Palo Alto
HPL-2007-50R1
January 23, 2008*

SOA, model-based automation, service models

The increasing costs and complexity of maintaining IT environments has resulted in a focus on both Services Oriented Architectures (SOA) as well as model-based automation. By creating IT functions as modular building-blocks and automating common interaction patterns between them, IT can be made more flexible, easier to manage, and less costly to maintain.

This paper presents Picasso, an architecture targeted at automating IT capabilities based on SOA and Model Driven Architecture (MDA) principles. Picasso structures IT capabilities as a set of services, each of which exposes a description of itself as a service model to the SOA. Service interactions are restricted to operations on the service models. Picasso thus balances a decentralized view of the world with the ability to re-use services in an automated manner. It provides service architects with a structured methodology and interaction patterns that can be used to create model based automation capabilities.

* Internal Accession Date Only

Approved for External Publication

Table of Contents

1 Introduction

Recently, the notion of a Services Oriented Architecture (SOA) has been proposed as a means of governing large-scale information technology (IT) systems to simplify the integration and management of IT. An SOA is defined[1] as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." An SOA breaks IT functionality into a series of services, each assumed to be controlled and managed independently of others. The services expose state and operations to one another, and can be composed into larger building blocks by integrating them through standardized interfaces. The premise of SOA is that by defining services that are linked to business value and business process, the governance of IT systems can be improved. By explicitly creating IT "service blueprints" that define interactions between services, SOAs formalize relationships and interactions[2] between IT systems.

While the reason for breaking up tightly coupled systems into service-oriented building blocks is novel from an IT governance point of view, the underlying distributed computing concepts behind SOAs date back at least two decades. The notion of encapsulating functionality as a "service" dates back to client-server systems. Similarly, object-oriented programming paradigms use "objects" as the abstraction to capture state encapsulation and create application programming interfaces (APIs) to allow software modularity and reusability. Microsoft's DCOM[3] and the Object Management Group's CORBA[4] standards have service-oriented architectures at the implementation level. The Object Management Group (OMG) is currently developing Model-driven-architectures[5] (MDA) to extend the notions of object-oriented paradigms in distributed systems to services oriented blocks. MDA treats service specifications as model abstractions, and places them into two categories:

- Platform Independent Models (PIM): provide vendor neutral specifications for services, and specify standard vocabularies and taxonomy that can be used to ensure that independent implementations can interoperate.
- Platform Specific Models (PSM): translate and extend PIM to specific underlying implementations. This enables the PIM to be mapped to different underlying middleware platforms, and provide additional platform specific capabilities beyond PIM.

---

[1] Reference Model for Services Oriented Architecture 1.0, OASIS SOA Reference Model Technical Committee: http://www.oasis-open.org/committees/download.php/19434/soa-rm-cs.pdf

[2] SOA Adoption Blueprint "GeneriCo", OASIS SOA Adoption Blueprints Technical committee: http://www.oasis-open.org/committees/download.php/17616/06-04-00002.000.doc

[3] COM: Component Object Management Technologies, Microsoft Corporation: http://www.microsoft.com/com/default.mspx

[4] CORBA Basics, Object Management Group: http://www.omg.org/gettingstarted/corbafaq.htm

[5] Model Driven Architecture FAQ, Object Management Group: http://www.omg.org/mda/faq_mda.htm

Within this context, MDA applies formal modeling and object oriented concepts to the development of an SOA.

While SOA methodology focuses on the taxonomy of services required for IT governance and the interactions between those services, it typically follows a functional remote procedure call (RPC)-like methodology (usually based on web services) for the service interfaces. Similarly, while MDA takes a model-based approach for integration, its focus is on the process for defining the models and the tooling required for managing those models. The Picasso architecture treats the underlying functional capability (implemented as software tools) required for automation as a set of loosely coupled services, which are orchestrated using SOA to perform automation tasks. However, it models the service descriptions and the service interactions using the modeling process described for MDA.

The Picasso architecture melds the SOA concepts with the object-oriented approach suggested by MDA for describing IT services. It is therefore a concrete instantiation of SOA and MDA principles for the purposes of IT automation.

1.1 Audience

The primary audience of this document includes system architects, designers, and implementers wishing to instantiate SOAs using model-based interaction patterns for the purposes of automating IT systems. A secondary audience includes strategic planners and product managers who wish to understand how systems management tools can expose model-based interfaces to enable interoperability and "plug-and-play" capabilities within solutions or tasks that require interaction between multiple management tools.

1.2 Assumptions

It is assumed that the reader is familiar with UML[6] notation (especially class diagrams and sequence diagrams), basic object-oriented concepts that define software patterns, as well as SOA and MDA principles.

1.3 Benefits

The Picasso Architecture offers the following benefits for IT automation:

Plug-and-play Services: By merging SOA and MDA concepts, Picasso balances a decentralized view of the world with the ability to re-use services in an automated manner.

Services Evolution: The Picasso architecture assumes that both services and service descriptions will evolve over time, and multiple versions of services will need to co-exist in the IT environment. Since version incompatibility is one of the main causes of faults within distributed systems, the Picasso architecture supports service and model version handling and validation at a low level.

Decentralized Service Models: In any IT automation environment, there will be multiple vendors or developers for the management tools that are required. This makes it difficult, if not impossible, to

[6] Unified Modeling Language, http://www.uml.org/

maintain a single model that captures all dependencies and tools within the IT environment. The Picasso architecture addresses this issue by clearly defining domains within which model coherency can be maintained, while providing freedom to each service to define its own model of interaction.

Protocol Evolution and Independence: There is a plethora of ever growing and evolving "standards" that are used within tool implementations. This poses a problem for SOA developers, since a change in interaction protocols requires changes in every service within the SOA. The Picasso architecture follows the MDA approach of defining service descriptions in a platform independent manner by describing them in models. It then uses the notion of a communication proxy service to handle the details of transforming these descriptions to actual "on-the-wire" data structures, thus freeing the service developer from these details. By localizing any protocol related dependencies within the communication proxy service, the Picasso architecture frees the service developer from the burden of protocol evolution.

Customization: Picasso is targeted at service automation. Because the scope of IT automation within the enterprise is large, the architecture assumes that any given instance of the architecture will need to be customized to the IT domain. Thus the architecture makes very few assumptions about the services, products, and protocols that are present within a domain

1.4 Document Scope

The Picasso architecture is targeted at IT automation. Thus references to service interactions within the architecture relate to management functions exposed by services, not the interactions necessary to provide end-user functionality. The document describes requirements that any instance of Picasso must meet, and provides design patterns for important interactions between services as a set of use cases. However, many architectural and design choices are left open because they necessarily depend on the specific IT functions being considered for automation. In that sense Picasso may be considered a "meta-architecture," in that any instance of this architecture may need additional architectural choices within boundaries specified in this document.

Footnotes within the document provide references, carry important clarification information and highlight currently unresolved issues within the architecture. Thus they should be read as an important part of the text.

Please note that this document is under active revision, and undergoing continuous change as the architecture is fleshed out further. The document version is maintained for this purpose, and information in a later version of the document supersedes that in earlier versions. Issues that need addressing or discussion items are marked using colored text in brackets such as <comment or issue yet to be resolved, or text yet to be written>.

1.5 Document Overview

This document is organized as follows:

Section 2 provides definitions for important terms used within the architecture. Because these terms (e.g., service, event etc.) are used widely in different contexts, they are defined precisely to avoid confusion over their use. In addition, their purpose and the concepts supported by them are given. Finally, their relationships are specified to define how these terms relate to one another within the scope

of the architecture. In case of ambiguity, the reader is referred to this section for the semantics associated with any term defined there, and no additional semantics should be assumed by the reader beyond that provided in this section.

Section 3 defines the overall architecture. Because we assume that the actual functional capabilities within any instance of the architecture will be custom to that instance, the traditional functional block diagrams are avoided, and only a very small set of services and roles that are necessary for every instance of the architecture are identified. In addition, the model exchange pattern used by all services to communicate with one another is defined.

Section 4 elaborates on the architecture to show how the roles and services defined in Section 3 accomplish lifecycle tasks such as creating a domain; starting or stopping services; invoking operations on services, and communicating asynchronous events using the communication patterns also defined in Section 3.

Some operations that are used to manipulate models are enumerated in Section 5. These operations provide guidelines that describe typical model operations that are available within the architecture. Finally, we finish with some frequently asked questions and our responses to those questions in Section 6.

2 Definitions

This section introduces concepts that underpin the Picasso architecture and recur throughout it. These concepts establish the terms (the vocabulary), notions and principles as well as how they are used within the architecture.

Picasso uses the terms defined in this section in a precise manner. Within the scope of the architecture, no additional semantics should be assumed for any definition provided in this section.

The material within this section is normative for the Picasso architecture.

2.1 Architecture Entities

A *system of entities* defines a canonical set of types within the architecture. It is preferred that the set be a minimal and complete set of types that are semantically orthogonal.

▶ **Entity** – An entity is a thing or a concept that is relevant for the operation of the system and thus needs representation inside the system.

The following entities have been identified within the Picasso architecture.

Table 1: **Primary entities in the Picasso architecture**.

Entity	Purpose	Concepts supported by entity
Domain	Defines the scope of an instance of the architecture	Ontology, scope, type system
Service	Encapsulates specific functionality, state, and behavior	State encapsulation, implementation neutrality, capability re-use, loose coupling, contract-based interactions
Service Model	External representation of a service	Introspection, interoperability, abstraction, discovery, service composition
Service Access Point	Hosting service that provides a model-exchange interface	Protocol independence, simplified interfaces, model validation, SOA services
Service Model Event	A pending change within a service model at a service access point	State coherence, distributed operations, orchestration, efficiency, asynchronous communication, scalability
Role	Mapping of function to services	Permission, delegation, orchestration

These entities provide the basic building blocks for structuring the information about things and concepts needed for the operation of any instance of the architecture. Reasoning upon entities controls

the behavior of an instance of the architecture with respect to all internal and external interactions. They are introduced in more detail in the following sections.

2.1.1 Domain

The scope of an instance of the architecture is defined by a domain.

▶ **Domain** – A *Domain* is the set of related entities that interact with one another to accomplish some purpose. Domains define the scope of discourse between related entities, and hence the types of entities necessary, the vocabulary (ontology) used to describe the entities, and the syntactic constructs that are understood by all entities within the domain. A given entity may participate in multiple domains.

Domains limit the scope and types of entities that exist in a given instance of the architecture, as well as the vocabulary used to describe them. Thus, for example, an instance of the architecture that is concerned with business process may define BPEL[7] workflows while an instance of the architecture used for deployment of computing resources may define computers and networks. Frequently, entities that participate in multiple domains create communication bridges between the domains, and may provide different views of themselves to the different domains. Domains may be nested within other domains.[8]

Within the architecture domains are used for specifying:

Ontology – Domains define the vocabulary (semantics) that is understood by communicating entities within that domain. This allows a specific implementation of the architecture to be customized and specialized for the purpose at hand.

Scope – Domains allow separation of multiple instances of the architecture that co-exist side-by-side by restricting the set of members within the domains to be disjoint, while at the same time providing the freedom to create bridges between the domains in controlled fashion through entities which participate in both instances.

Meta-model – Communicating entities within a domain use a common type system and syntactic constructs (the meta-model) to communicate. This improves interoperability and evolution of the underlying entities by ensuring that existing entities can inter-operate with new entities without requiring extensive re-work.

2.1.2 Service

A service defines the basic entity that captures some useful functionality within the architecture.

---

[7] Business Process Execution Language for Web Services 1.1, IBM, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[8] The architecture currently leaves open the exact relationship between a domain and its sub-domains. It is anticipated that sub-domains will be used within the architecture for purposes of *SOA administration* (allowing sub-domains to be administered independently of the enclosing domain), for *SOA specialization* (allowing sub-domains to extend and refine the vocabulary or entities in the enclosing domain, without affecting other entities outside the sub-domain), and for *SOA partitioning* (allowing separate vocabularies to exist in the different sub-domains, while the enclosing domain reconciles communication across them).

▶ **Service** – A *Service* is an entity that represents an encapsulation of functionality and state useful to other entities within a domain.

A service[9] is the central entity in the architecture. A service provides (in an object-oriented sense) an encapsulation of some functionality[10] that is useful to other services within a domain by exposing that functionality through operations. A service may expose part of its internal state, and provides mechanisms for other services to access and manipulate the exposed state. Services may also capture interactions with entities in the external environment (e.g., IT systems or human operators) and present the results of those interactions as exposed state to other services within a domain. Thus, from the perspective of communicating services, it is possible for the exposed state within a service to change spontaneously or autonomously.

Services address the following capabilities within the architecture:

State Encapsulation – Services encapsulate parts of the state of the external world and present that state to other services within a domain. This simplifies the management of state by allowing creation of modular services, each of which is concerned with managing only part of the global state. It also limits the scope that any given service needs to handle.

Implementation Neutrality – Services use implementation neutral interaction patterns among them. This implies that implementation specific communication patterns are prohibited in the architecture and it is possible to replace one instance of a service with another instance that provides the same capability.

Capability Reuse – By appropriately scoping the functionality provided by a service, it is possible to develop services that are re-usable in different contexts.

Loose Coupling – Services allow the overall system to be re-configured and re-purposed by changing the interactions between them.

Contract Based Interactions – Services offer both mechanisms to access and manipulate state within them, as well as explicit behavior guarantees. This permits the overall system to recover from faults (or at least isolate them) and enables automation of different tasks performed by the collection.

2.1.3 Service Model

All services within a domain interact through service models.

▶ **Service Model** – A *Service Model* is the representation of a service within the SOA. It defines the externally visible description, behavior, state, and operations available from a service to other services. Within a domain, each service defines its own service model and is responsible for exposing it to other

[9] We will use the word "service" to mean both the service type (the external definition of the service) as well as the service instance (the concrete instance of the service based on some implementation), unless the context requires distinction. In that case, "service type" is used to mean the definition of the service, while "service instance" is used to mean a running instance of a service. The term "service package" will be used to mean the software (or other) implementation that underlies the service instance.

[10] Note that this does not mean that the service implementation is object oriented, or even automated. Indeed, the service implementation may rely on human operators behind the scenes.

services within that domain.

Note that other architectures[11] define both external and internal models for services. The Picasso architecture does not concern itself with any internal models, since they are, by definition, internal to the service, and thus do not (or should not) affect any external interactions from that service. Similarly, the architecture does not require (but also does not preclude) any desired state or observed state[12] models.

Within the architecture, service models[13] are used for the purposes of:

Introspection – The requirement that all services expose models of themselves[14] allows new services to be added (or old services to be updated) within the architecture, and their semantics and interaction patterns understood by new and existing services without reprogramming.

Service Composition – By including appropriate meta-model constructs (e.g., references), services can be composed for different functions. A composed service is not required to have a service model separate from its constituent services, although the architecture permits a service instance to act as a proxy for the composite (and thus be required to expose a model of the composite).

Interoperability – The type system used within the domain is represented by a common meta-model[15], i.e., the model structure used by all entities is also explicitly represented using a model. While each service is free to define its own service model, all communicating services use the same meta-model. This allows the services to selectively access model components of interest to them, without requiring them to recognize and deal with model elements that are not of concern to them. Note that it may be possible to use more than one meta-model within a domain. However, this requires the presence of at least one service that can communicate using both meta-models and can act as an intermediary between other services that use different meta-models. While different meta-models may be unavoidable across domains, use of different meta-models is strongly discouraged within a single domain.

Discovery – The models exposed by all services within a domain allow discovery of information and service instances within the architecture. The architecture does not presume (or preclude) any specific form of discovery (e.g., through registries).

Abstraction – Service models present abstractions of the external world within a domain as state exposed by services. Thus, information that is unnecessary for interactions among services over the

---

[11] Extending Radia into a Service Delivery Controller, http://www.hpl.hp.com/techreports/2005/HPL-2005-52.pdf

[12] Ibid.

[13] Note that we use the word model to represent both the model schema (M1) and the model instance (M0) unless the context requires distinction. In that case we use "Model Type" and "Model Instance" to distinguish the two. We will also frequently omit the prefix "Service" from "Service Model" if the context is clear.

[14] Within the context of management services used for automation tasks, it is important to understand that the service is required to expose a model of *itself* for purposes of introspection, not just a model of the elements it manages, although that information is also likely to be present in the model as exposed state. Thus a deployment service exposes a model that permits other services to recognize that the service is a type of "deployment service." It is not sufficient for the service to simply expose the computers or network models that it is managing. The domain within which the service exists determines both the representation of the model as well as the type ontology (e.g. "deployment service") that are permitted inside the domain.

[15] Meta-models (M2) represent the common structure that is used to define Model Types (M1) within a domain.

SOA can be hidden from view by omitting it in the models. By properly abstracting information present within models, services can hide heterogeneity and support the domain purpose to be accomplished even when the underlying components change over time.

2.1.4 Service Access Point

Service instances within a domain communicate with one another using proxy services provided by service access points.

▶ **Service Access Point** – A *Service Access Point* is a service end-point that provides a standard interface for interaction between services using a model exchange pattern. A service access point represents a distributed proxy service that provides the ability to utilize other services through a restricted set of model operations.

The Service Access Point (SAP) hosts service models exposed by services, and provides a standardized interface for accessing those models. Because the models capture the external view of the service, service access points act as proxies for services within the SOA, and jointly provide an access service within the domain. Note that a service access point may act as a proxy for many services, and a given service may advertise itself through multiple access points[16]. The service access point differentiates the Picasso architecture from other service-oriented architectures, in that it allows service instances to delegate the responsibility for their interaction with other services to the services access point.

A service access point provides the following capabilities to the architecture:

Simplified Interfaces – In traditional web services, each service presents its own interface (e.g., WSDL[17]) to the SOA implementation. In addition, this interface is frequently tied directly to "on-the-wire" representations (e.g., XML message formats) of the information. The service access point, however, restricts its API to model exchange patterns[18], and therefore presents a much simpler API within the SOA that can be used independently of the service operations (which are captured in the models). By shifting the complexity away from the SOA APIs, Picasso simplifies the composition of services at the SOA level, while allowing the services to define more complex operations as necessary within service models.

Protocol Independence – traditional web services use an ever-evolving set of standards for on-the-wire communications. Each update or change requires all service implementations to change within the SOA. By delegating the communication responsibility to the service access point, service designers can focus on the capabilities provided by their services rather than requiring updates from every service instance in the architecture.

Model and Message Validation – Typically, message validation is required within an SOA by each service because of the loosely-coupled nature of the services. The service access point uses the meta-

[16] Even if a service instance is accessible through multiple access points, its *identity* (within a domain) at the different service access points stays the same, i.e., the service clients can query the service at any service access point and recognize that they are communicating with the same service instance. See discussion under "Identity Service" later in Section 3.2.1.2.

[17] Web Services Description Language (WSDL) 1.1, W3C: http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[18] The model exchange pattern is defined in detail in Section 3.1.

model and domain ontology defined by the domain to validate all model exchanges through it, and ensures that mal-formed messages and models are avoided.

Model Caching – In a distributed environment, SAPs may cache information about the different models that exist within the SOA, thus reducing the network and communication overhead of repeatedly transporting model elements between services.

2.1.5 Model Event

A Model Event represents a pending change within a service model hosted by a service access point.

► **Model Event** – A *Model Event* is a pending change of some service model element caused by a model operation performed by some service at a service access point. The model operation may target either the model type or the model instance at the SAP. Model operations or changes in external state that cannot cause in changes in models hosted by an SAP do not represent model events.

Model events[19] are the primary mechanism by which the service access points track service model changes, decide which changes are permitted (and which ones are not), and interact with services and service clients in order to respond to pending changes.

Within the architecture model events are used for:

State Coherence – Since model events represent pending changes (before they are made), services have the opportunity to reject the proposed change before it is committed to be a part of the service model. This reduces the likelihood that the distributed state within the models becomes inconsistent.

Distributed Operations – Model events allow operations to be distributed by providing a mechanism whereby a single operation on some model results in multiple operations on other models in a distributed environment. They provide a way of creating, customizing, and orchestrating distributed workflows implicitly within the architecture.

Efficiency – Model events avoid the need for constant polling of service models by services to detect changes. By delegating the responsibility of tracking model changes to the SAP and requesting notification when a change is pending, the services and service clients reduce the overhead of monitoring state within the environment.

Asynchronous Communication – Model events provide asynchronous communication capability between services. Since only model operations are permitted at service access points, a pending change on some service model element represents a request for one or more service actions. This permits the service access point to communicate with other services on an as-needed basis.

Model event handling is provided in the architecture through three entities: Model Event Triggers, Indications, and Model Event Filters.[20]

---

[19] We will often omit the prefix "Model" when referring to Model Events, unless the context requires it.

[20] We will often omit the prefix "Model Event" when referring to Model Event Triggers and Model Event Filters unless the context requires it.

2.1.5.1 Model Event Trigger

Not all model events require action on part of some service or a service client. A model event trigger is a model event that requires notification of some service other than the service access point itself.

▶ **Model Event Trigger** – A *Model Event Trigger* is a model event that requires the service access point to notify some service (or service client) of a pending change in some model element.

Model event triggers are intended to distinguish model events that can be handled by the service access point from those that require notification of a service by the service access point. Note that implementations may choose to use the same mechanism to implement model event triggers and other events that are internal to the service access point—that decision is implementation specific and does not affect the architecture.

Model event triggers are further refined into two[21] categories:

Approval request: An approval request is a trigger that requires approval from some listener (*the change approver*) before the pending change can be committed. In this case, the SAP waits for a response (approve/reject) from the approver before committing the model change or generating other triggers to services that actually perform the underlying operations implied by the pending model change.

Change notification: A change notification is a trigger that that simply informs the listener of a model change that has been committed at the SAP. Note that in this case, the SAP allows the pending change to complete before notifying the listener that the change has taken place. Change notifications are intended primarily as a mechanism to improve efficiency in distributed environments, because they are sent "after-the-fact" and do not require the listener to query the SAP again to check if the pending change actually succeeded or not.

2.1.5.2 Indication

Indications are model elements that define the messages that are passed to a service or a service client by service access points as a result of a model event trigger.

▶ **Indication** – An *Indication* is a model element that is passed to a service (or service client) as result of a model event trigger. Indications provide sufficient information to the recipient to allow it to obtain the model state as proposed by the change.

A service that is capable of receiving indications is an *indication listener*. Because indications are used by approval requests, the architecture recommends that indications should provide sufficient information to allow the indication listener an efficient way of accepting (or rejecting) the proposed change.

[21] Note that this does not imply that these are the *only* two types of triggers. Triggers may represent other conditions besides approval requests and change notifications. For example, a service may be notified of a pending change, and may explicitly have to take action or perform other operations on the model in order to commit the change rather than just returning an accept or reject status to the SAP. See Model Exchange Pattern later (Section 3.1) in the document.

2.1.5.3 Model Event Filter

Model event filters are model elements that allow the service access point to decide which model events should be labeled as model event triggers.

▶ **Model Event Filter** – A *Model Event Filter* is a model element that defines a query on service models held at a service access point. The query specifies model operations (create/ update/ delete/ invoke) on specified model elements, as well as references to one or more indication listeners that need to be notified if the corresponding query succeeds.

Model event filters provide mechanisms for different services to register interest in model operations performed at the service access point, and provide the service access point with a model-neutral way of recognizing when a model change requires action on part of an external entity.

2.1.6 Role

Service (or human) entities within the domain perform activities based on roles assigned to them.

▶ **Role** – A *Role* represents the expectation that a certain service (or human) entity within a domain is assigned or required to perform a given task or activity.

Note that the assignment of roles to entities may change over time; multiple entities may share a role, and entities may delegate their roles to other entities. Roles are used within a domain to determine which service instance has responsibility for which function, and which service instance is permitted to perform which operation. The visibility of service models (or model elements) may depend on the role of the requestor.

A role provides the following capabilities to the architecture:

Permissions – Roles define which services (or service instances) are permitted to perform which operations within the domain. Note that the role of a service is not the same as its identity. Roles permit permissions to be defined independently of the identity of service instances.

Delegation – Within a domain, services may perform operations on behalf of other services, and may delegate their authority to perform operations to other services. Roles provide a mechanism for allowing such delegation.

Orchestration– When services interact to accomplish some larger task, it becomes important that specific tasks be performed in some well-defined order, but it is usually not as important which service instance performs a given task. Roles allow such tasks to be specified independently of the underlying service instances that perform those tasks.

2.2 Entity Relationships

This section summarizes the relationships between the different entities defined in the architecture. For ease of representation and precision in description, we use UML notation.

As shown in Figure 1, each instance of the architecture is scoped by some domain. The domain defines one or more meta-models that are used by all services within that domain[22] to express their service models. The domain is an aggregate of one or more entities, and an entity can participate in one or more domains. Domain entities include other domains, services, models exposed by services, events and roles.

Each service within a domain exposes a model of itself, which is hosted by some service access point on behalf of that service. All services within the domain are thus associated with service access points[23]. The model exposed by the service conforms to the meta-model defined by the domain. Service access points are end points of a domain-wide proxy service that provides SOA communication capabilities based on a model-exchange pattern to other services. The model-exchange pattern is a restricted set of standard model operations (on models exposed by the services) for communication with other service access points and/or service clients.

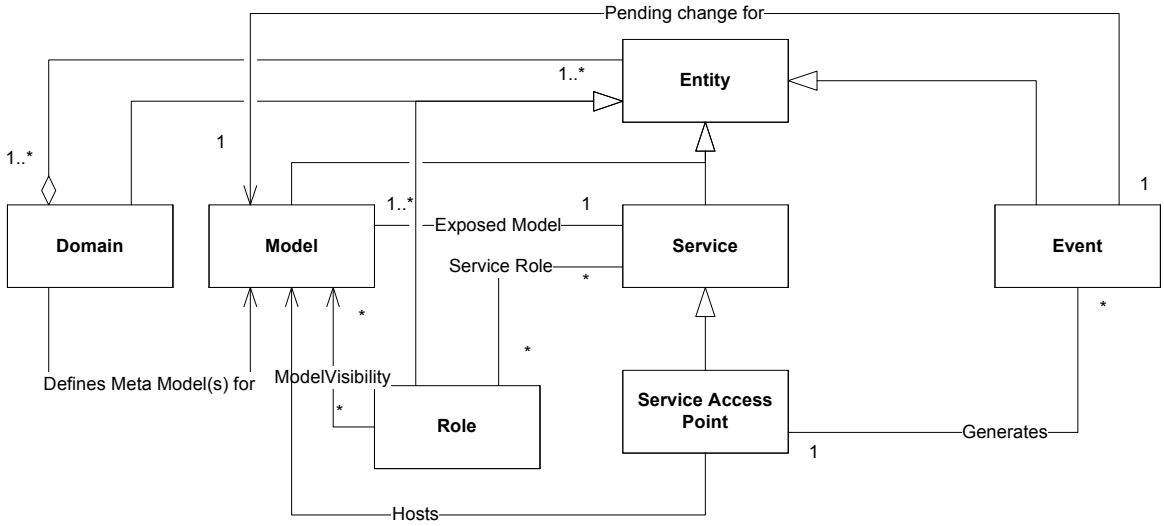


Figure 1: Entity Relationships

Services may have roles associated with them. The service role defines the visibility of models (or model elements) to that service, and hence the functionality available to that service from other services. Roles are used within a domain for defining access from one service to another.

[22] Currently, the architecture mandates a "dictionary service" for each domain that holds the authoritative meta-model(s) and the service ontology for the domain.

[23] Note that service access points provide an access service, and are thus required to expose a model representing that they are "service access points" for that service.

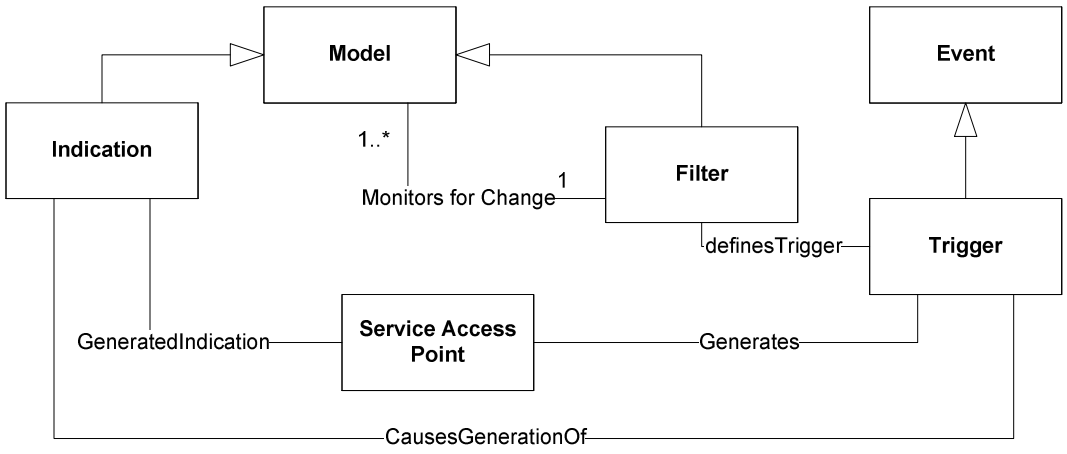


Figure 2: Event Generation

Service access points may generate model events as shown in Figure 2. Model events represent pending changes in models at a service access point. Service access points monitor model events using Model event filters, which are model elements that represent an interest in the pending change by some service. Model events selected by Model event filters are Model event triggers, which cause the service access point to generate indications to be sent to the service(s) interested in the pending change. Triggers may represent notifications or approval requests (not shown).

Within the Picasso architecture, these relationships form the basis for specification of services (in terms models), interaction between services (as changes to the models, permitted by service roles) and the means to maintain state coherency and control within the IT environment (using event triggers and indications).

3 SOA Architecture

This section defines the overall Picasso architecture using terms defined in Section 2. Because Picasso is a services-oriented architecture, at the top level it is structured simply as a set of communicating services.

Figure 3 illustrates an instance of the architecture. It consists of a number of service access points (sap_1 - sap_4). The SAPs host a number of services (S_1 – S_7) of types (T_1 – T_6) in a distributed manner. The services in turn interact with external world entities including human operators (e.g., S_7), information systems (e.g., S_2), or IT infrastructure (e.g., S_5 and S_6). Note that multiple instances of the same service type may be present in the environment, perhaps providing the same service bound to different parts of the infrastructure (e.g., S_5 and S_6), and the same service instance may present itself to other services through multiple SAPs (e.g., S_2), perhaps to provide a distributed service.

The Picasso architecture is concerned with interactions between the SAPs, interactions between the SAPs and the services, and (mediated via the SAPs) interactions between the services themselves.

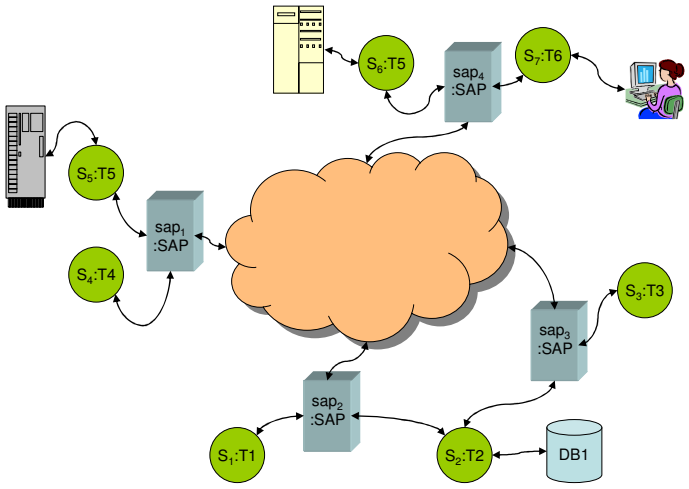


Figure 3: Illustration of Picasso Architecture

The SOA capabilities within Picasso are represented by the capabilities of foundation services (see Section 3.2) required within the architecture as well as interaction patterns that are used by services to communicate with one another (see Section 3.1).

The subsequent material within this section is normative for the Picasso architecture.

3.1 Model Exchange Pattern

Services communicate by exchanging models (or changing exposed models) using a "REST-ful" style of communication that uses operational semantics such as "GET" and "PUT" on model elements. However, the architecture enforces additional semantics on this communication because models represent "exposed" state and operations available from any service. Thus, it is important that changes within service models be controlled to ensure that invalid or undesired representations are not propagated between service entities. This is facilitated by the model exchange pattern within Picasso.

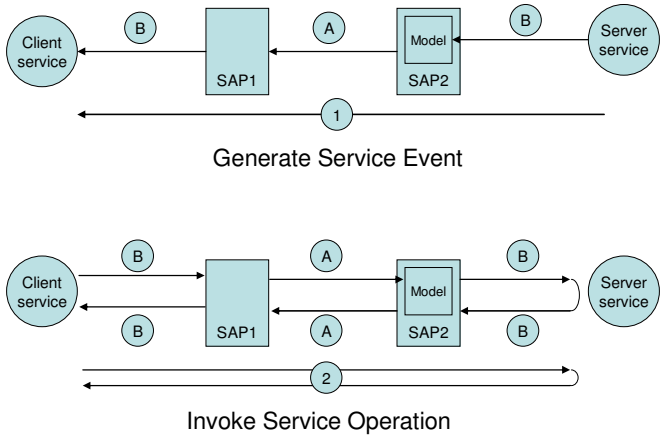


Figure 4: Communication between services

Within Picasso, three types of communications are possible:

1. *Communication between two SAPs*: Since the SAPs represent the ability for services to communicate, they are aware of "on-the-wire" protocols used within the SOA and communicate with one another directly using those protocols. However, the architecture limits the communication between SAPs to operations defined on service models[24] that are being hosted at an SAP. These communications are shown in Figure 4 by interactions labeled (A).
2. *Communication between a service and an SAP*: Services expose their operations and state to other services within the SOA by providing models to the hosting SAP. This implies the need for a service to communicate with an SAP. The API between the service and the SAP follows the same pattern of model-based operations as the SAP – SAP communication. However, the Picasso architecture leaves the actual protocols and APIs for the communications as choices made by the Service Developer and the Domain Architect[25]. These communications are shown in Figure 4 by interactions labeled (B).
3. *Communication between two services*: Within Picasso, services communicate with one another mediated by the SAPs. Two forms of communication between services are defined, depending

[24] Note that the models may represent the SAP itself, or may represent a service hosted at the SAP.

[25] In general, we expect that this API will be a "programmatic API" such as a Java or C++ API depending on implementation choices made by the Service Developer. While it is possible to also expose "on-the-wire" protocols for this purpose, the architecture does not encourage this.

on the model targeted by the communication. Note that in both cases, the communication can be broken down further into SAP – Service communication and SAP – SAP communication.

a. *A service modifies its own model*: A service may make a change to its own model at the SAP based on changes in its internal state. In this case, it is possible for other services to subscribe to changes in the service model (by defining the appropriate triggers on the service model). Here, the SAP sends appropriate indications to the subscriber to notify the remote service of the model change. This form of communication is closest to a "publish/subscribe" mechanism between services that is provided by the SAPs. This is shown in Figure 4 by interactions labeled (1).

b. *A service invokes an operation on a different service*: In this case, the service invokes an operation on the model exposed by a different service. If both services are hosted at the same SAP, that SAP can handle the operation internally. If the "client service" is located at a different SAP than the "server service", the client SAP invokes the appropriate model operation at the remote SAP on behalf of the client service[26]. This is shown in Figure 4 by interactions labeled (2).

All three types of communication target models held at an SAP. Picasso treats these communications as "model exchanges" wherein a change in some model at an SAP is requested by a service (or by an SAP acting on behalf of some service).

The basic model exchange pattern is shown[27] in Figure 5. When a change[28] in some service model is requested (Step 1) at a service access point by some service (the change requestor), the model representation is not changed immediately. Instead, the request is validated[29] (Step 2), the pending change is cached (Step 3) and Filters defined on the model are used to decide if an approval request is generated (Step 4).

If no approval requests are generated (Case A), the corresponding service operations are performed [See 4.2.2- SI02 – Invoke Service Operation], the change is committed (Step 7a) and the requestor is notified of a successful change (Step 8a). If however, an approval request[30] is generated (Case B), the "Model Owner" associated with the affected model (or model fragment) is notified (Step 5) by sending an indication to it. If the model owner approves the change (Step 6a), any service operations

---

[26] Note that we are using the terms "server" and "client" in the traditional distributed computing sense. That is, a server is the entity that is exposing some operation, and the client is the entity invoking the operation. Depending on the direction of communication, a particular service instance can act either as the server or the client. This should be distinguished from terms such as "Service Provider," "Customer," or "End User," which are roles attached to entities within the SOA.

[27] In this and subsequent diagrams, we are taking some liberties with UML notation. Specifically, rather than showing alternate cases on messages, we are showing alternatives by splitting the object lifeline. In our opinion, at the conceptual level, this makes the diagram more readable.

[28] Note that the definition of a "change" includes initial creation and final deletion of the model instance or model type.

[29] Note that "validation" in this step includes checking the request against the domain ontology and meta-model for semantic and syntactic correctness, and may include mutual authentication between the change requestor and the SAP, as well as checks against requestor roles to decide if the change is permitted. Depending on implementation, these steps may be explicitly done by generating Triggers at Step 4, or may be a part of the SAP logic.

[30] It is currently open what happens if more than one approval request is generated. <One suggestion is that only one "owner" can exist at a time. Another possibility is that the domain meta-model defines the semantics (AND/OR) followed by the SAP when multiple requests are present>

associated with the change are performed [see 4.2.2- SI02 – Invoke Service Operation], the change is committed (Step 7a), and the requestor notified of the successful change (Step 8a), and if notifications need to be generated, the appropriate indications are sent (Step 9). If however, the model owner rejects the pending change (Step 6b), the change is discarded (Step 7b) and the requestor is notified of change failure (Step 8b). Note that during this process the pending change is not visible to services (including the requestor) other than the SAP and the model owner until the end of Step 7a, when the change is committed.

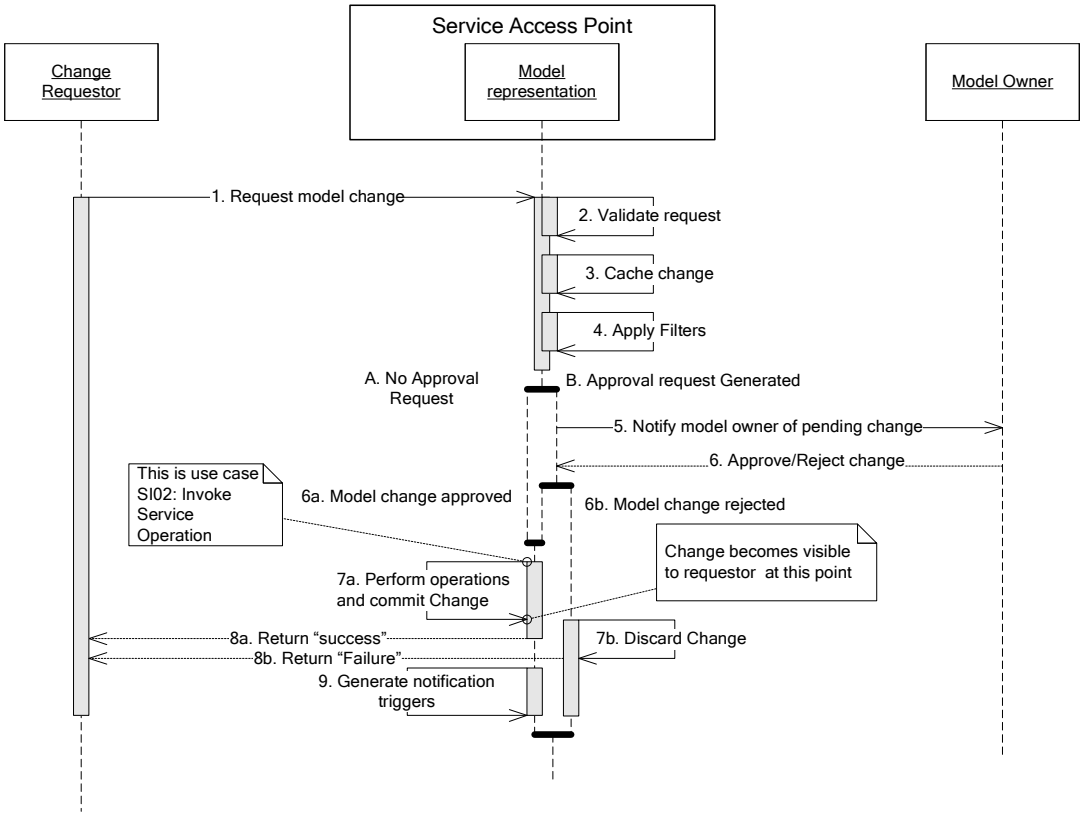


Figure 5: Model Exchange Pattern

This basic pattern forms a "handshake" that is used extensively within the Picasso architecture to maintain coherence of models at SAPs and prevents arbitrary or incorrect changes to state represented within the service models in Picasso. Variants of this handshake are used in different interactions to ensure that only valid state is presented through the models.

Because the model owner (typically the service exposing the model) stays in control of the exposed model at all times, it can decide which model changes require approval and/or notification. It can also analyze pending changes to ensure model coherence. Finally, such a handshake provides assurance to the clients that any changes they make in the models are valid changes, and receive explicit notification when the changes fail.

In the subsequent discussion, the model exchange pattern is not explicitly shown to simplify interaction diagrams. However, it should be recognized that this semantics is followed by all interactions that require model changes (including initial creation and final deletion of models) at the SAP.

3.2 SOA Foundation Services

The SOA within Picasso relies on a number of services (Foundation Services). Note that foundation services are defined like any other service within Picasso, and follow the same restrictions of model-based communications mediated by the SAPs as other services. However, they are listed separately as part of the architecture because they are core to the operation of the SOA, and are often required by any distributed architecture. This section defines the foundation services identified within Picasso.

3.2.1.1 Service Access Service

The service access points provide access to all services within the Picasso architecture using a model exchange pattern. The SAP distinguishes the Picasso architecture from other MDA or SOA instantiations. The service access points expose service descriptions and state as models to one another, and are responsible for implementing any "on-the-wire" protocols to communicate model information between them. They jointly provide a distributed access service that is responsible for all communication between services. *The Service Access Service provided by SAPs is the only foundation service that MUST be implemented within any instantiation of the Picasso Architecture.*

3.2.1.2 Identity Service

Every service instance within Picasso is required to have a globally unique identity that differentiates a given service instance from all other service instances[31]. Picasso does not mandate a specific form for the service identity, beyond the requirement that two service instances are considered to be the same instance if and only if their identity is the equal. The architecture assumes that while the semantics of "equality" may vary depending on the domain, the identity service provides operations that allow two identities to be compared.

Service identities may be managed within each (sub) domain by an "identity service," which serves as the authoritative source for assigning identities to service instances within that (sub) domain. The identity service in a (sub) domain may form a hierarchical relationship with the identity service in the enclosing domain.

---

[31] Note that the notions of an identity, a name, a role, an address, and a credential are frequently used interchangeably in many contexts. The Picasso architecture draws clear distinctions between a service "name", its "identity", its "role", its "address" and its "credentials." These distinctions are best understood by drawing parallels with how these words are used within human contexts. Each person is unique from birth and has an identity that is unique from any other person. A person may have a specific identifier (e.g., a social security number) that is unique to them for identification purposes within a domain (e.g., the domain defined by all US residents). People are referred to by their names, which serve as convenient labels for them. However, there may be many people named "Joe" and a given individual may be known by many names, e.g., "Joe" and "Joseph." A person may take on one or more roles, e.g, "Joe" could have the role of a "father", a "son", and an "employee". People may be located at addresses, which are a mechanism for associating location information with people. Finally, people carry credentials (e.g., driver's licenses or passports) that vouch for their identity.

Within Picasso, we postulate that each service instance has a unique identity for its lifetime, with an associated identifier that may be provided to it by the identity service. A service instance may take one or more convenient names (maybe registered with the name service) that act as convenient mechanisms for labeling the service. A service instance may take on one or more roles (defined in the directory service). The service instances have associated addresses (maybe registered with the name service) that allow other services to locate them. Finally, a service instance may have credentials (possibly obtained from a certificate authority service) that can be used to validate the identity of the service to other services.

Once an identity has been assigned to a service instance, it cannot be "revoked" for the duration of the lifetime of the service instance[32].

3.2.1.3 Certificate Authority Service

The certificate authority service provides cryptographic certificates to service instances and acts as a trusted mediator service to enable service instances to authenticate their identities to one another. Each (sub) domain should contain at least one instance of a certificate authority service. In case of nested domains, or when multiple certificate authority services co-exist, they may form hierarchical relationships with one another that parallel the domain hierarchy.

The certificate authority service may revoke certificates at any time.

3.2.1.4 Name Service

The name service maps one or more "friendly names" to one or more service addresses. Names may be used by service instances to refer to other service instances, and services may obtain the mapping from names to addresses (and *vice versa*) using the name service. Each (sub) domain normally contains a single instance of a name service. Like the identity service and the certificate authority service, the name service instances may also form a hierarchy when nested domains are present. The naming convention (e.g., URIs[33]) used within a domain is determined by the ontology and type system defined for that domain.

3.2.1.5 Directory Service

The directory service contains information (e.g., location, role, organization, contact information, etc.) that defines pertinent information (metadata) about a service instance within the domain. Frequently, the directory service is co-located with the identity service (and/or the name service), and is tied closely to it. The architecture recommends (but does not mandate) that each (sub) domain contain a single instance of the directory service. Note that unlike the Name Service, which simply maps names to addresses, and vice versa, the directory service supports much more complex queries about the meta-data associated with the services.

3.2.1.6 Dictionary Service

The dictionary service provides an authoritative repository for the ontology and service model types used within the domain. Note that the dictionary service provides "semantic definitions" as well as the "syntactic definitions" within the type system used for model representation. The service access points validate and check service definitions against the type system provided by the domain dictionary and reject any non-conforming definitions. The dictionary service provides a mechanism for the domain to control the taxonomy of services, the vocabulary used within the domain, and to ensure that model representations are consistent across services within the domain. If multiple dictionary services are

---

[32] It is currently open if a service instance is allowed to maintain its identity across a "restart" of the instance. <This seems a reasonable assumption assuming that the restarted service instance binds to the same model instance within the SAP, but we need to test it.>

[33] Naming and Addressing, W3C: http://www.w3.org/Addressing/

present in a given (sub) domain, it is recommended that they be linked in a hierarchy to prevent semantic conflicts between them.

3.2.1.7 Match-Making Service

A Match making service provides yellow pages or capability-based selection service. It is similar to a registry in that it allows instances to discover one-another based on queries made on capability.

3.2.1.8 Audit and Logging Service

The audit and logging service provides the capability to log (and audit) messages between SAPs within a domain. Since all[34] service interactions within Picasso happen through the SAPs, Audit and Logging services can be conveniently be added there to provide "non-repudiation" capabilities within the SOA.

3.2.1.9 Orchestration Service

The orchestration service provides a mechanism for dynamic composition of services by offering workflow management capabilities within the SOA. Note however, that implicit workflows that do not require centralized orchestration can be built by defining appropriate model event triggers.

3.2.1.10 Policy Service

The policy service provides a mechanism for creating, managing, and enforcing domain-level policies within the SOA. The architecture recommends that the policy service should be implemented so that SAPs can act as policy enforcement points.[35]

3.2.1.11 Registry Service

A registry service typically provides capabilities similar to a directory service in that service instances register themselves with the registry, thus providing "discovery" capabilities within the SOA. It also frequently provides matchmaking capabilities.

3.3 Roles

The following primary roles are defined within the Picasso architecture:

Table 2: **Roles defined within the architecture**.

Role	Purpose	Responsibilities supported by role
Domain Architect	Defines a domain. Typically a human role.	Creates the ontology, scope, type system used within the domain. Defines domain-level service taxonomy and domain policy. This is

[34] Recall that within the context of Picasso, we are concerned only with interactions that relate to management operations, not end user functionality.

[35] Terminology for Policy-Based Management, RFC 3198, IETF: http://www.ietf.org/rfc/rfc3198.txt

Role	Purpose	Responsibilities supported by role
		an extension of the Service Architect role
Domain Administrator	Manages a domain. Typically a human role.	Manages domain-related services. Grants domain-level permissions to roles within the domain. This is an extension of the Service Administrator role.
Service Architect	Defines a service model and service logic. Typically a human role	Defines model types within the domain. Defines service-level policy and interactions with other services.
Service Developer	Implements the service logic. Typically a human role.	Develops, tests, and packages the service logic. Creates the service implementation.
Service Administrator	Manages the service during operations. Maybe an automated system role or a human role.	Registers the model types with the SAP. Starts or stops service instances. Approves addition or deletion of service instances from the SAP.
Service	Encapsulation unit for of state and functionality in the system. Provides state and functionality to other services. Typically an automated system role.	Links (binds) model representations to actual state within the external environment (e.g., IT systems and user interfaces). Provides functional capabilities useful to other services within the architecture. Performs operations when models changes are made. Performs model changes to synchronize models to external environment.
Service Access Point	Hosts models on behalf of services and provides communication capabilities between services based on model-exchanges. An automated system role.	Maintains service model representations on behalf of services. Supports secure communications and distributed SOA services. Supports events and messaging between services.
Service Client	Requests service functionality and/or state from other services. May be an automated system role or a human role.	Requests model changes to perform specific tasks. Request model operations to be performed by other services.

Note that any specific implementation of Picasso is likely to extend these roles as needed. Furthermore, if automation capabilities extend up to the business level, additional roles (e.g., business manager) may be necessary. Thus this list of roles should not be considered exhaustive.

4 Use Cases

This section defines basic interaction patterns between services within a domain as a set of use cases. Note that in the use cases that follow, many interactions with foundation services are not explicitly shown to simplify the interaction diagrams. Depending on the foundation services used within a given domain, additional interactions will be necessary beyond shown in the use cases below.

4.1 Domain Creation

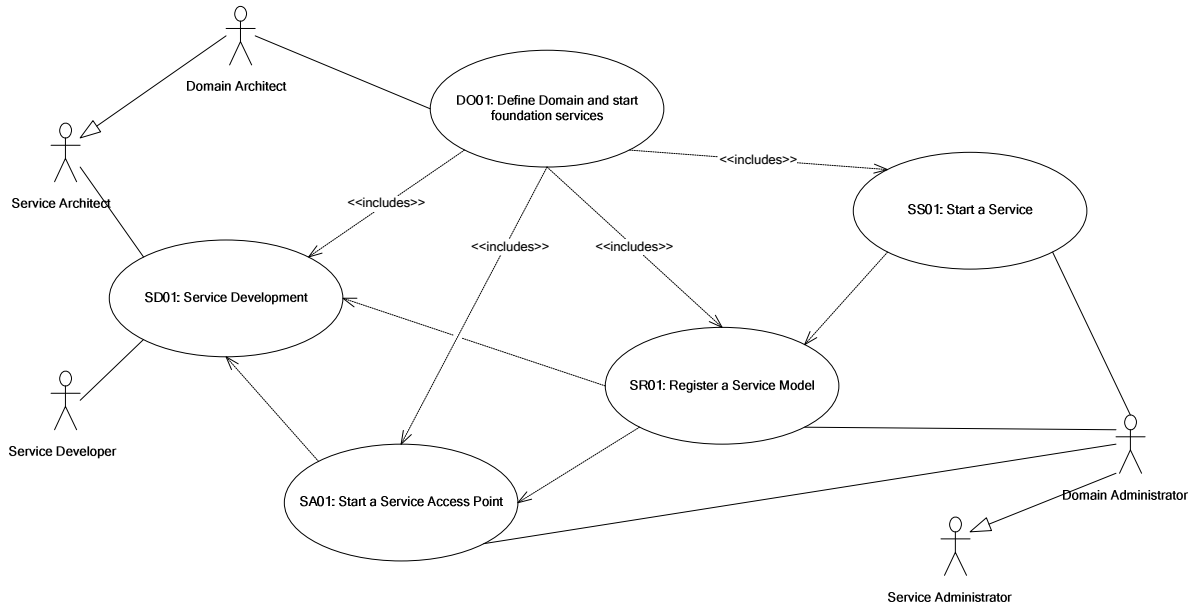


Figure 6: Use cases for creating domain SOA

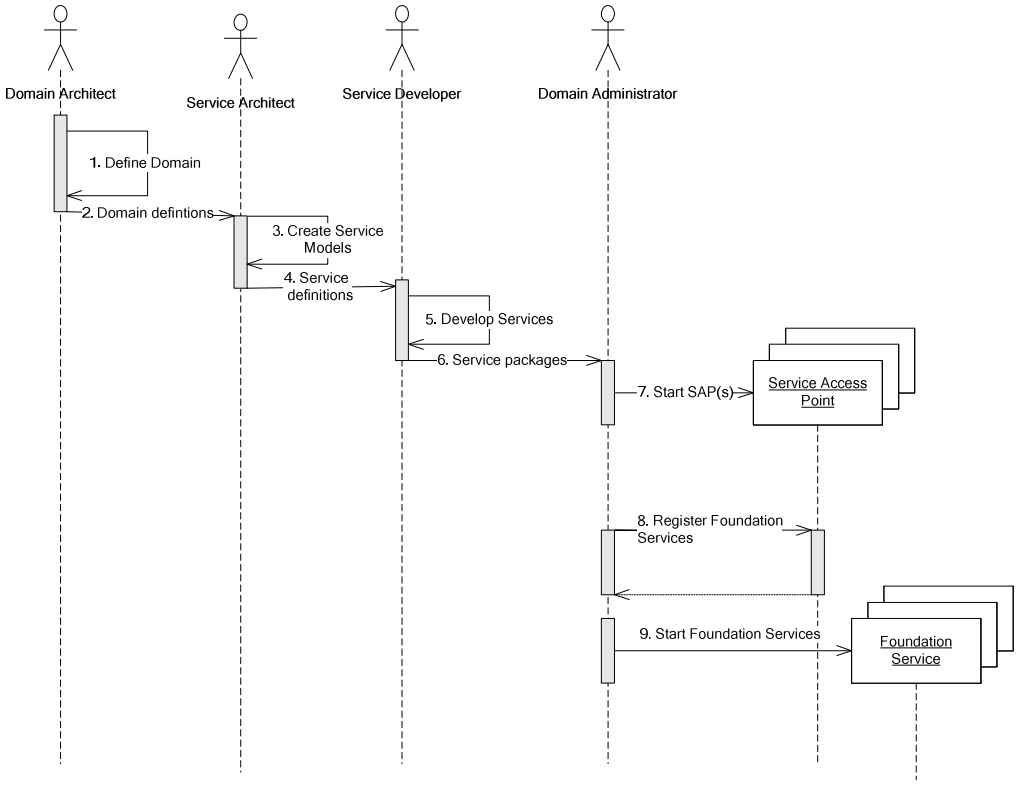
Figure 6 shows the use cases necessary to create a domain and start the foundation services[36] necessary for the SOA. Note that all use cases start with the definition of the domain provided by the Domain Architect. The SAP and each foundation service [See 3.2: SOA Foundation Services] defined within the domain needs to be developed by the Service Architect and the Service Developer. Once the

[36] Note that in case of the foundation services, the domain administrator (an extension of the service administrator role) is responsible for the services, because the privileges associated with the foundation services are likely to be different from those associated with other services that use the SOA. However, in the subsequent discussion, the service administrator role is used within the detailed discussion of the individual use cases, because that role is more generic.

services have been created, the appropriate foundation services can be started. These use cases are defined in more detail below.

4.1.1 DO01 – Create Domain and Start Foundation Services

This use case specifies how the Domain is defined and Foundation Services within it are created and started.



Actors	Domain Architect, Service Architect, Service Developer, Domain Administrator, Foundation Services, SAP
Preconditions	None
Input	None
	1. Domain Architect defines domain information (ontology, meta-model). 2. Domain Architect provides domain information to Service Architect. 3. Service Architect defines the SAP and foundation service models and determines their dependencies, as well as the service logic [See 4.1.2: SD01 – Service Development]. 4. Service Architect provides the service models to the Service

	Developer. 5. Service Developer develops service packages for the SAP and other foundation services based on information from Step 3 6. Service Developer provides the service packages to the Domain Administrator. 7. Domain Administrator starts up one or more SAPs to host foundation service models [See 4.1.3: SA01 – Start a Service Access Point]. 8. Domain Administrator registers foundation services with the SAPs [See 4.1.4: SR01 – Service Model Registration]. 9. Domain Administrator starts the foundation services [See 4.1.5: SS01 – Start a Service].
Post-conditions	**Case success:** Domain has been defined, one or more SAP(s) that host Foundation Services are up and running, Foundation Service Models are up and running. **Case Failure:** Errors in foundation service definition, initial service bring up caused failures. Foundation Services within the domain may only be partially working.
Notes	Order of starting SAPs and foundation services in Steps 8 and 9 depends on the dependencies between them. To avoid circular dependencies, the names, addresses, identities, and credentials for the name service, the identity service, the directory service, and the certificate authority service, and at least one SAP may need to be "baked in" to the foundation service models to allow resolution/binding during bring-up. Steps 3-9 provide a variant of this use case that is usable for any service within the domain (not just the foundation services).
Issues	This is a bootstrap case. Need to understand how the bring-up of domain services can be automated. Currently, the process for creation and startup of domain services is assumed to be manual. Therefore failure cases are handled on a case by case basis by interactions among the Domain Architect, the Service Developer, and the Service Administrator.

4.1.2 SD01 – Service Development

This use case specifies how a service is developed.

Actors	Service Architect, Service Developer
Preconditions	Domain must have been defined by Domain Administrator
Input	Domain information [See Steps 1, 2 in 4.1.1: DO01 – Create Domain and Start Foundation Services]. Service definitions for existing services within the SOA [e.g., see 3.2.1.6 Dictionary Service]

Actions	1. Service Architect defines the service Model Type compliant with the Domain information. The service Model Type includes: a. All externally visible attributes of the service (including models of managed elements which are part of the exposed state) b. Service operations defined on any element within the service model c. Any triggers and filter definitions required to interact with the SAP d. Dependencies on other services required for this service to operate 2. Service Architect defines the service logic necessary to implement the service. 3. Service Architect provides the service model and service logic definitions to the Service Developer 4. Service Developer writes service logic that is compliant with the service Model Type, along with logic/scripts necessary to start/stop the service. 5. Service Developer writes test cases, tests the service, and documents it. 6. Service Developer packages items from Steps 1-5 in a form that can be used by the Service Administrator to register and start the service.
Post-conditions	**Case success:** Service logic has been developed and packaged in a form ready to be registered and started. Service models have been defined and provided as part of the package. **Case Failure:** It may not be possible to develop the service logic to comply with the domain information provided.
Notes	In case of failure, the Domain Architect, the Service Architect, Service Developer and the Service Administrator may have to iteratively adjust both the service model as well as the domain information. Dependencies may also require adjustment to other services impacted by the changes.
Issues	This use case contains a very large code development and test process in Steps 2-5. It is also likely that Step 1 will require interaction between the Domain Architect and the Service Architect.

4.1.3 SA01 – Start a Service Access Point

This use case specifies how a Service Access Point is started.

Note: The Service Access Point requires access to the foundation services when it starts up. At the time the SAP is started, the following cases may apply:

1. The SAP is intended to host service models for some foundation services, and cannot assume the presence of the foundation services.

2. The SAP is intended to host service models for some service, and other foundation services necessary are already available.

This use case provides the interactions necessary for the second case above. The first case is a bootstrap case, and is described in the Notes section of the use case below.

Actors	Service Administrator, Service Access Point, Certificate Authority Service, Identity Service, Directory Service
Preconditions:	The Service Administrator has credentials registered with the Certificate Authority Service, and has authority to start a SAP. The required services are up and running and the Service Administrator can communicate with them. The service package for the SAP has been developed [4.1.2: SD01 – Service Development].
input:	Addresses of foundation services.
Actions	1. Service Administrator obtains an identity value from the identity service for the SAP. 2. Service Administrator starts SAP code using SAP service package, and passes the identity to it, along with addresses of the Directory Service. 3. Service Administrator requests that the Certificate Authority generate a certificate for the SAP using his credentials. 4. The Certificate Authority returns a token to the Service Administrator. 5. The Service Administrator provides the token and the address of the Certificate Authority to the SAP. 6. The SAP uses the token to obtain its credentials from the Certificate Authority. 7. The SAP registers itself with the Directory Service.
Post-conditions	**Case success:** Service Access point has been started, and is ready to host models for other services. **Case failure:** The Certificate Authority or the Identity service may reject the request from the Service Administrator if the Service Administrator is not authorized to start a SAP.
Notes	1. This use case assumes that the Certificate Authority and the Identity Service can be reached by the Service Administrator independent of whether the SOA services have been started. 2. If the Certificate Authority and the Identity Service have not been

	started, then the Service Administrator (or the Domain administrator) has to manually generate the identity and the credentials, and provide them to the SAP in step 2.
Issues	A variety of three-way handshakes are possible to avoid spoofing attacks in steps 1-7 depending on the level of security desired. For example, rather than returning a single token to the Service Administrator in Step 4, the Certificate Authority can send separate tokens to the Service Administrator and the SAP, and require the Service Administrator to give his token to the SAP before the SAP can request a certificate.

4.1.4 SR01 – Service Model Registration

This use case specifies how a Service Administrator registers a service model at the SAP.

Note: There are three common options for registering a service model at the SAP.

1. The service administrator manually registers the model at the SAP.

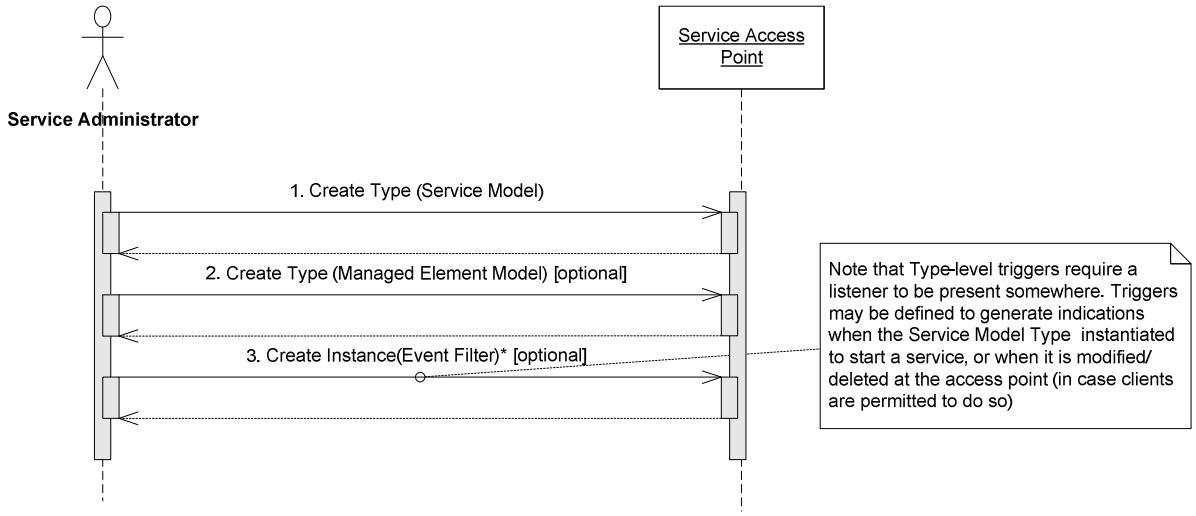
2. The Service Developer packages the service executables and includes instructions/logic that enable the SAP to register the model (and possibly start the service ala cgi-bin executables) as necessary. This presumes a "create a transient instance" capability or dynamic loading capability within the SAP.

3. The Service Developer points to a "factory service" that knows how to register the service model with the SAP.

This use case provides the interactions necessary for the first case above. Other cases[37] may be defined in a similar manner, but are omitted below for brevity.

[37] Note that in addition to the SAP, the model may also need to be registered with the dictionary service. If a dictionary service is present as a foundation service, the following additional choices are available:

1. The SAP registers the model with the dictionary service (as part of registering the model with itself). This may make use of type level triggers at the SAP that communicate the model insertion information to the dictionary service.
2. The model is registered with the dictionary service, and contains references to the SAP. Type level triggers may then be used to inform the SAP that a new model that it is required to host has been registered with the dictionary service.
3. The Service Administrator independently registers the model at the SAP and the dictionary service. This is a straightforward extension of the use case described here.
4. The model is registered with the SAP, but not the dictionary service. The SAP permits this as long as the model does not conflict with other definitions within the dictionary service. The advantage of this approach is that it permits services to expose models to one another without burdening the dictionary service with unnecessarily detailed models that are only of interest to small set of services. The disadvantage is that the definition may become invalid at a later stage as a result of something else being added to the dictionary service.

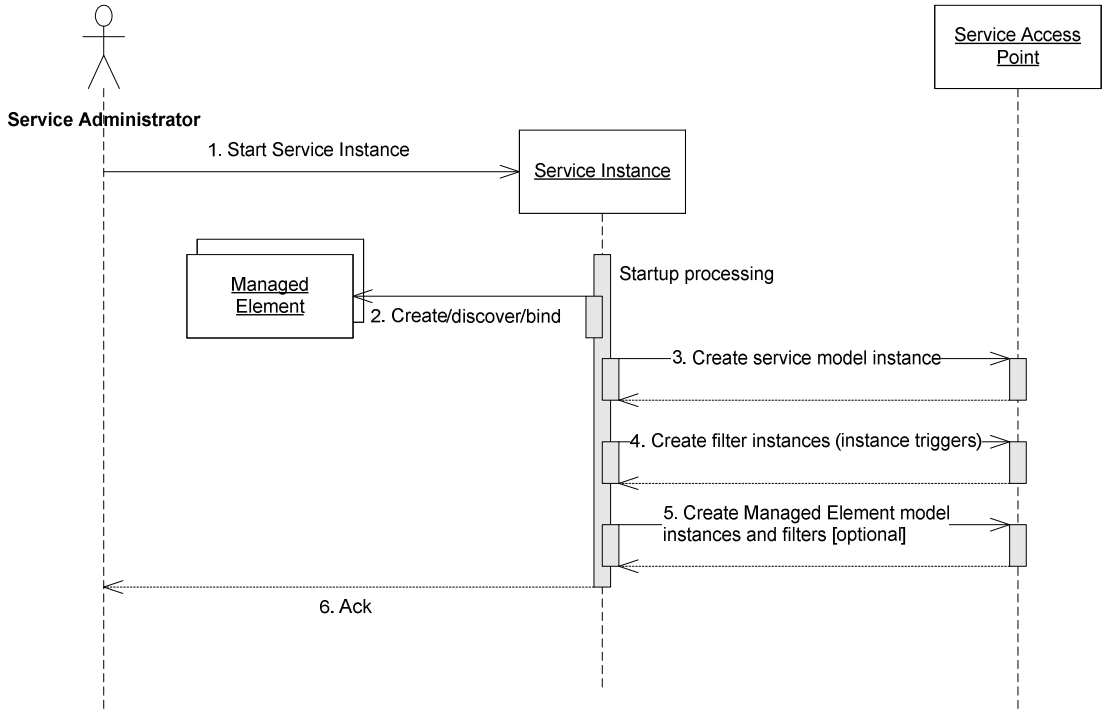


Actors	Service Administrator, Service Access Point
Preconditions:	The service has been developed [4.1.2: SD01 – Service Development] and the service access point to be used is alive [4.1.3: SA01 – Start a Service Access Point].
input:	Model Types defining the service, as well as Filter instances required for handling Triggers on Model Types.
Actions	1. Service Administrator adds Service Model Type definition to the SAP 2. Service Administrator adds Models Type definitions for any additional Managed Elements necessary for service operation (if SAP does not already contain those definitions) 3. Service Administrator adds Filter instances to register any event triggers (type level) necessary.
Post-conditions	**Case success:** Service model has been registered at the service access point. **Case failure:** Errors on service registration have been communicated back to the service administrator. The SAP has rolled back state to before registration was attempted by the service administrator[38].
Notes	If any type level triggers are desired (e.g., for modification/changes in the type, then an indication listener must be present in the eco-system somewhere and information about it must be included during the registration.
Issues	"Private" information needed for service start-up is also required at the SAP. However, since that information (by definition) is not part of the model, additional configuration information needs to be provided to the SAP if it is required to start the service.

[38] Note that this will happen implicitly as a result of rollback defined for model operations [See 4.2.2: SI02 – Invoke Service Operation] when the operation to create the model type fails.

4.1.5 SS01 – Start a Service

This use case specifies how a service administrator starts a service at an SAP.



Actors	Service Administrator, Service Access Point, Service Instance, Managed Element(s)
Preconditions	The service model has been registered at the SAP [See 4.1.4: SR01 – Service Model Registration] and the SAP to be used is alive [See 4.1.3: SA01 – Start a Service Access Point].
input:	Reference to the hosting SAP. Service package and service configuration information necessary for starting the service, and creating a model instance for the service within the SAP.
Actions	1. Service Administrator configures and starts the service instance and provides it with the SAP reference as part of the configuration. 2. The service instance discovers (or creates or binds) the managed elements, if any, that it wishes to expose as part of its model. 3. The service instance creates a model instance of itself at the SAP. 4. The service instance creates instances of filters that define triggers that the service instance should receive from the SAP. 5. The service instance optionally creates instances of models for the managed elements that it wants to expose to the domain, as well as filters that define triggers on them.

	6. The service instance acknowledges successful startup to the service Administrator.
Post-conditions	**Case success:** Service instance has been registered at the service access point, is accessible for use by clients, and service administrator has been notified of successful startup. **Case failure:** Errors on service startup have been communicated back to the service administrator.
Notes	It is also possible for the service administrator to perform steps 2-5 instead of the service instance. In that case an acknowledgement in step 6 is not needed, because the ack from steps 2-5 will go directly to the service administrator. The Filters created by the service instance in step 4 and 5 define triggers that are specific to this instance of the service. This allows multiple instances of the same service type to co-exist at the same SAP.
Issues	A "commit" step should be required at the end of this use case to ensure that unstable model instances are not left at the SAP in case the service instance fails during startup (steps 4, 5). This should not be done as part of the model (e.g., state=running) because that requires the SAP to understand the model semantics for arbitrary services. This is a more generic problem and points to the need for a "service died and left model state behind in SAP" use case.

4.2 Service Operations

The following use cases define how a client service invokes operations on another service.

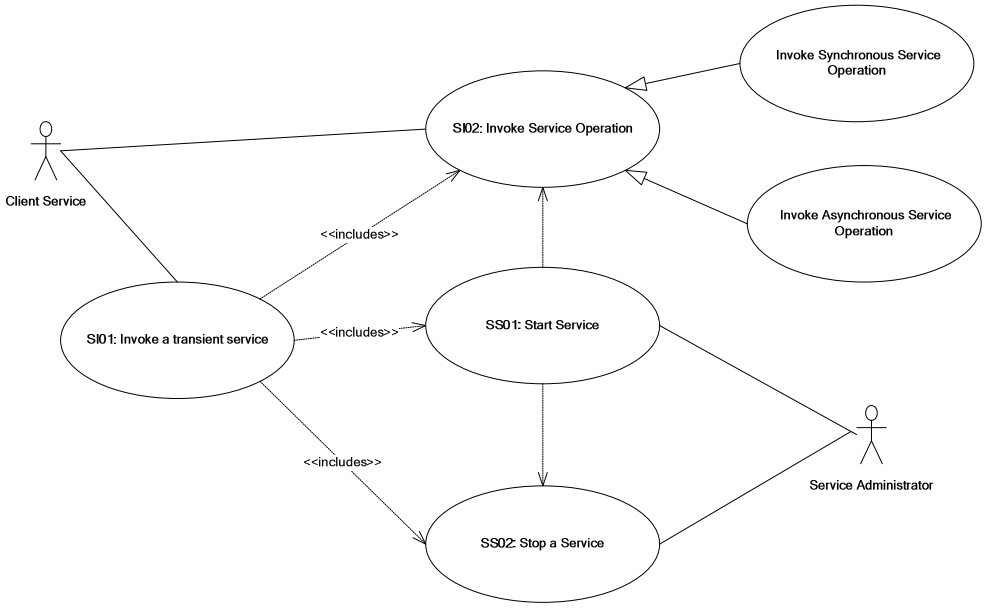


Figure 7: Use cases for service operation

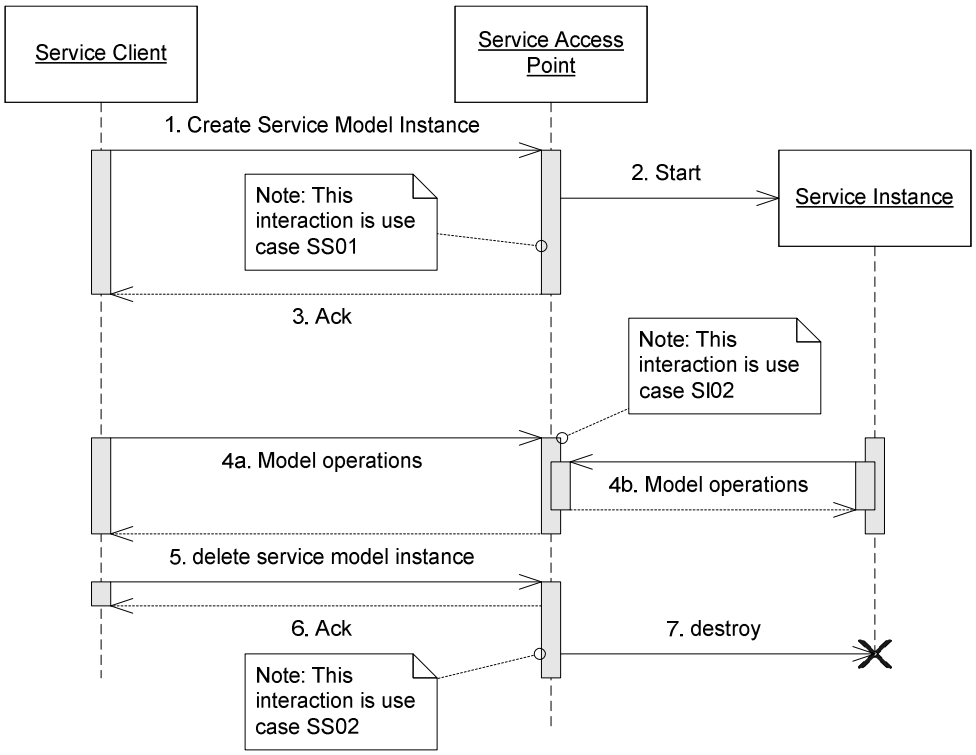
In case the service is a transient service[39], the service needs to be instantiated before it can perform operations on behalf of the client. The client may invoke one or more operations on the service, and then request that it be stopped. The operations themselves may either be synchronous (the client waits for the operation to finish) or asynchronous (the client continues to do other tasks after requesting the operation, and is notified when the operation is finished.

These use cases are described in more detail below.

4.2.1 SI01 – Invoke Transient Service

This use case specifies how a service access point instantiates a service to perform some operations, and stops the service at the request of a client.

[39] By transient service, we mean a service instance whose lifetime is determined by the client. In order to use a transient service, the client must first request that it be instantiated, and after invoking the operations necessary, the client must request that it be stopped. There are no architectural assumptions about how long the client makes use of the service.

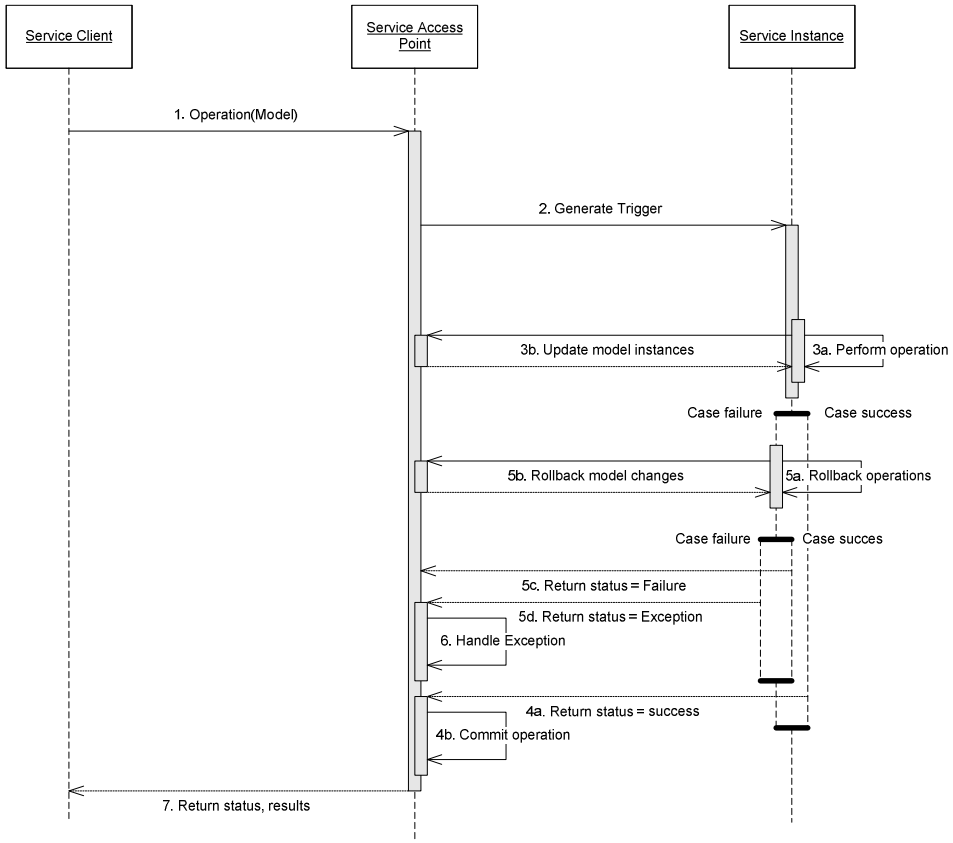


Actors	Client Service, Service Access Point, Service Instance
Preconditions:	The service access point to be used is alive [4.1.3: SA01 – Start a Service Access Point], the service has been packaged [See 4.1.2 - SD01 – Service Development] so that it can be started.
input:	model operations by requested client
Actions	1. Service client attempts to create a model instance of the service at the service access point. 2. The service access point instantiates the service and binds it to the model instance [See 4.1.5 - SS01 – Start a Service]. 3. The Service access point acknowledges model creation to the service client. 4. The service is now available for use by the client, which can perform model operations on the service model (4a). Operations on the model are tracked by the service, which performs the corresponding service operations (4b). [See 4.2.2: SI02 – Invoke Service Operation]. 5. The service client requests deletion of the service model instance. 6. The service access point acknowledges the deletion to the client. 7. The service access point stops the service instance and deletes the service model instance [See 4.2.3: SS02 – Stop a Service].

Post-conditions	**Case success:** Service was brought up, made available to the client for use, and terminated after the client was finished with it. **Case failure:** See embedded use cases.
Notes	Note that in Steps 2 (and 7), the SAP will notify the Service Administrator that the service needs to be instantiated (and stopped), or may take on the role of the Service Administrator for that purpose.
Issues	The deletion Ack to the client currently happens before the instance is actually deleted (the client rarely cares what happens to the service after it requests deletion). This Ack can also be moved to after the service has been stopped at Step 7.

4.2.2 SI02 – Invoke Service Operation

This use case specifies how a service client accesses a service capability using model operations. This use case corresponds to Step 7a in the model exchange pattern [See 3.1: Model Exchange Pattern].

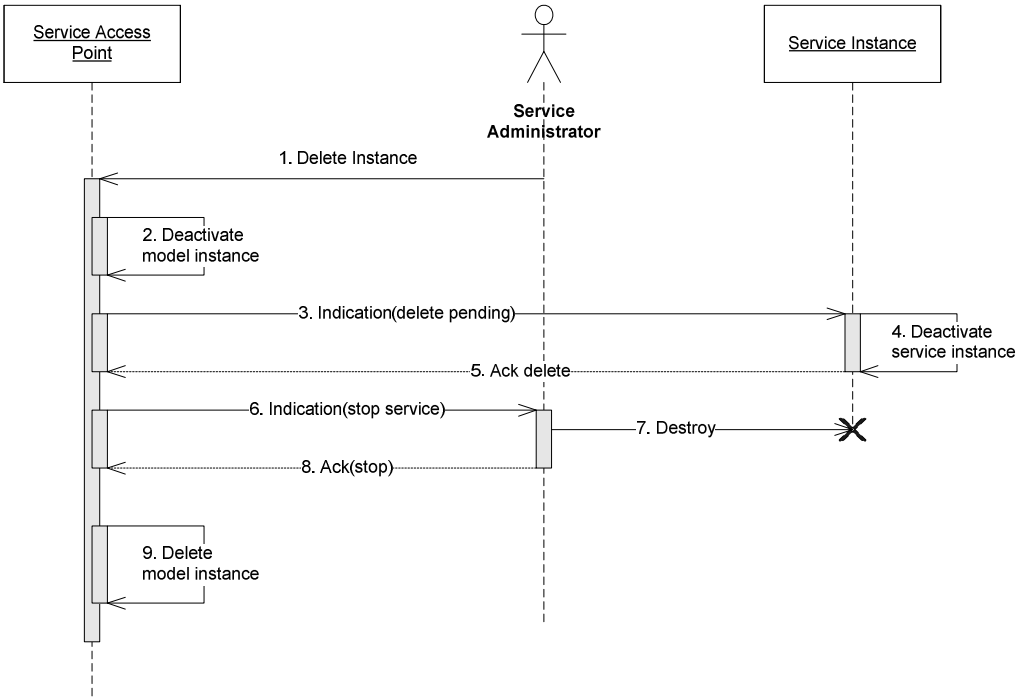


Actors	Service Client, Service Access Point, Service Instance
Preconditions:	The service instance has been started [See 4.1.5 - SS01 – Start a Service, or Steps 1-3 in 4.2.1 - SI01 – Invoke Transient Service], and the service access point to be used is alive [4.1.3: SA01 – Start a Service Access Point].

input:	Service Model operation (create/read/update/delete/invoke) desired by client along with any parameters necessary for the operation
Actions	1. Service client invokes the model operation on the model exposed from the service access point using any model parameters necessary. 2. Service access point validates the operation and obtains any necessary approvals [See Steps 1-6 in 3.1: Model Exchange Pattern] and sends an indication to the service instance containing pending operations to be performed on the model. 3. The service (a) performs any necessary operations and makes corresponding changes in the model (b) at the SAP 4. Case: The service operations succeed a. The service returns status = succeed to the SAP. b. The SAP commits the operation 5. Case: The service operations fail a. The service attempts rollback of operations performed before failure in an attempt to recover. b. The service unrolls model changes in an attempt to maintain synchrony between its state and the model c. If rollback succeeds, the service returns status = failure to the SAP d. If rollback fails, the service returns status = exception to the SAP 6. Optionally, in case of rollback failure, the SAP may handle the exception by notifying incident management service or taking other actions if the appropriate triggers are defined. 7. The SAP returns the operation status (success, failure, exception) to the service client.
Post-conditions	**Case success**: Service operation has been completed. Models exposed by the SAP reflect the new state, and all operations have been committed. **Case failure**: Service operation failed. However, all state has been recovered back to the point before the operation was invoked. **Case exception**: Service operation and recovery failed. Error handling procedures may be in progress. The model state at the SAP is unreliable and may be incorrect.
Notes	None
Issues	None

4.2.3 SS02 – Stop a Service

This use case specifies how a service administrator stops (deletes) a service at an SAP.



Actors	Service Administrator, Service Access Point, Service Instance
Preconditions:	The service access point to be used is alive [4.1.3: SA01 – Start a Service Access Point], the service has been started [See 4.1.5: SS01 – Start a Service].
input:	Model operation by service administrator
Actions	1. Service Administrator instructs the SAP to delete the service model instance. 2. The SAP deactivates the service model instance by removing it from the service model instances visible to clients at the SAP. This prevents any clients from accessing the model instance. 3. The SAP sends an indication to the service indicating the pending delete. 4. The service performs pre-delete activities. This may include putting the service in stand-by mode (or preparing for shut down) 5. The service acknowledges the pending delete 6. The SAP sends an indication to the service administrator that the service can be destroyed (or deleted). 7. The service administrator destroys the service instance.

	8. The service administrator acknowledges the service deletion to the Service access point. 9. The service access point deletes the model instance.
Post-conditions	**Case success:** All interested clients have been notified, and the service instance has been deleted from the SAP. Service Administrator has terminated the service. **Case failure:** It may not be possible to delete the service instance if permission is not granted (based on AAA constraints defined by the Service Developer at creation time, or by the administrator at service start time).
Notes	
Issues	The System administrator can terminate the service without notifying the SAP. This is a more generic problem and points to the need for a "service died and left model state behind in SAP" use case.

4.3 Service Event Triggers

This section describes how event triggers are handled within the SOA. This requires a service to register interest in some model exposed by a different service. If that model is changed for some reason, the SAP generates the appropriate triggers to propagate the change information to the appropriate listener using indications [See 3.1: Model Exchange Pattern]. These patterns are described in the following use cases.

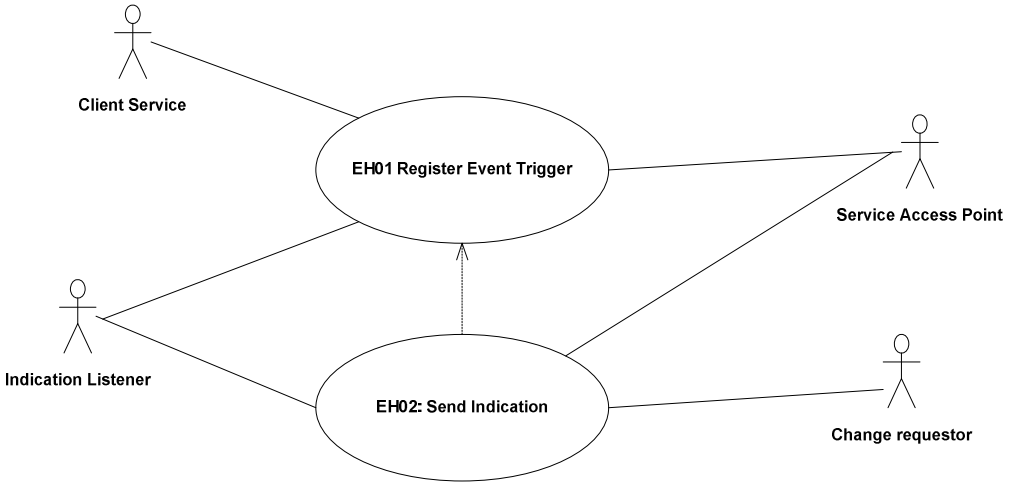
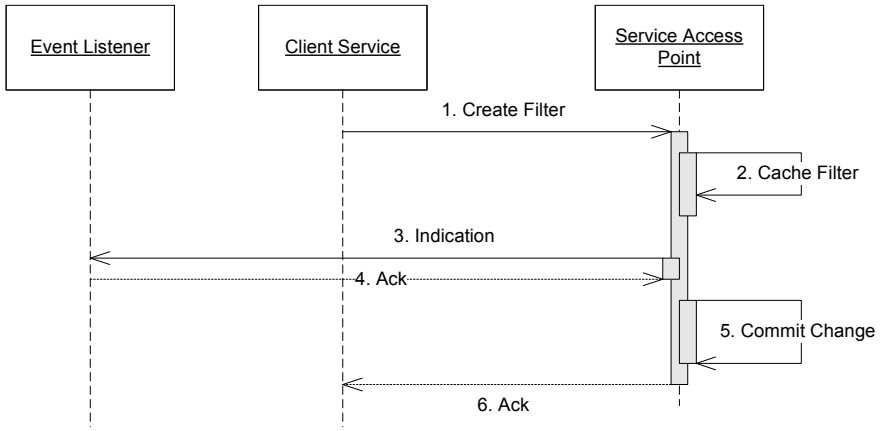


Figure 8: Event handling within the SOA

A Client Service may register for some model event at a Service Access Point by creating an instance of a model event filter at that SAP. Note that the client may delegate the responsibility for handling the resulting event triggers to a different service (the Indication Listener). When a change is made to the model by some service (the Change Requestor), the SAP sends the appropriate indications to the Indication Listener.

4.3.1 EH01 – Register Event Trigger

This use case specifies how a service client registers for an event trigger for a model change.



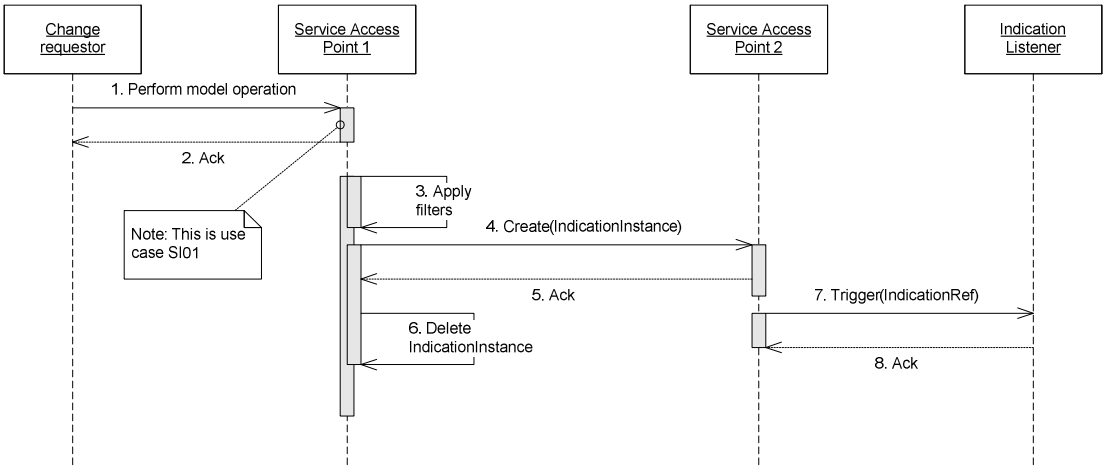
Actors	Client Service, Service Access Point, Incident Listener
Preconditions:	The indication listener[40] has been started [See 4.1.5 - SS01 – Start a Service], and the service access point to be used is alive [4.1.3: SA01 – Start a Service Access Point].
input:	Filter definition for the trigger desired by client along with reference for the indication listener
Actions	1. Client Service attempts to create a filter at the SAP. 2. The SAP caches the filter. 3. The SAP generates an approval trigger and sends an indication to the Indication Listener 4. The Indication Listener approves the addition of the filter. 5. The SAP commits addition of the filter and starts generating triggers based on it. 6. The SAP acknowledges creation of the filter to the client.
Post-conditions	**Case success:** Client has successfully registered for a model trigger at the SAP. The Indication Listener has accepted responsibility for handling any subsequent indications sent by the SAP [4.3.2: EH02 –Send Indication]. **Case failure:** Operation failed. However, all state has been rolled back to the point before the operation was invoked.
Notes	Note that this use case is a variant on the model exchange pattern because the approval trigger at Step 3 goes to someone other than the "model owner." Step 3 is explicitly necessary to ensure that a client cannot designate

[40] Note that the indication listener can be a SAP (or a service hosted at an SAP).

	indication listeners that are not prepared to handle indications generated.
Issues	Is the interaction at step 3 necessary, or is it overkill? We need to understand what action is appropriate if the event delivery fails at step 6.

4.3.2 EH02 –Send Indication

This use case specifies how an indication is sent as a result of a model change made by some change requestor to some indication listener. In this use case we assume that the indication listener exists at a different SAP than the SAP and the change results in a notification trigger.



Actors	Change requestor, Service Access Point1, Service Access Point2, Indication Listener
Preconditions:	Instances of the indication listener [4.1.5: SS01 – Start a Service] and the service access points to be used [4.1.3: SA01 – Start a Service Access Point] are alive. The listener has registered an interest in the trigger [4.3.1: EH01 – Register Event Trigger].
input:	none.
Actions	1. The change requestor invokes an operation on the model held at the service access point (SAP1). 2. The operation is performed; SAP1 commits the model change, and acknowledges the success of the operation to the change requestor. 3. SAP1 applies filters and generates a notification trigger containing information about the indication listener, and generates a local instance of the indication to the sent. 4. SAP1 creates an instance of the indication at remote service access point (SAP2). 5. Remote service access point acknowledges correct creation of indication.

	6. Local service access point deletes the local copy of the indication. 7. Remote service access point triggers the indication listener and gives it a reference to the indication.
Post-conditions	**Case success:** Indication has successfully been created at the remote service access point, deleted from the local service access point, and the indication listener has been notified of the indication. **Case failure:** Failure cases include: 1. Create fails in step 1. The Local SAP will return failure. State is as it was before indication was created. 2. Create fails in step 4. Indication is held at local SAP. TODO: need to decide if indication creator needs to be informed. Also need to decide the disposition of the indication. 3. Trigger fails at step 7. The indication is held at Remote SAP. TODO: need to decide if indication creator needs to be informed, and disposition of the indication. **case exception**: None identified so far
Notes	This use case is assumed in all other use cases described when indications are sent, and may not be explicitly shown as such.
Issues	See TODO items in Results. Can we just assume reliable message delivery?

5 Model Operations

This section provides the model operations available at SAP. These operations are listed as guidelines based on use cases defined in Section 4, and it is assumed that the domain architect will refine or augment these operations based on the actual domain definition. Thus *information in this section should be considered as advisory* within the Picasso architecture.

The SAP provides the following Type-level (M1) operations[41]:

Operation	**createType**
Parameters	Type Definition
Result	Success/Failure
Description	Creates a new model type definition (locally) within the SAP. The SAP ensures that the type definition does not conflict with any existing definitions in the domain dictionary, and that it obeys the meta-model and ontology defined for the domain.
Operation	**getType**
Parameters	Type name (and version)
Result	Type definition
Description	Returns the type definition known locally to the SAP. The SAP implementation MAY choose to query the domain dictionary for the type definition if it does not hold the definition locally.
Operation	**modifyType**
Parameters	Type Definition
Result	Success/Failure
Description	Modifies the type definition locally. Changes that are not backward compatible require the ability to maintain both the previous and the new definition within the SAP (for example, by changing version numbers on model types).

[41] All model operations follow the restrictions defined in the model exchange pattern (see Section 3.1). It is currently open how the local model type definitions are reconciled with those in the domain dictionary. The architecture requires the SAP to ensure that local definitions are not in conflict with those in the domain dictionary, but leaves open what happens if a local definition is either an extension of a definition in the dictionary, or is not present in the dictionary. A number of choices were enumerated as part of Footnote 37 earlier.

Operation	**deleteType**
Parameters	Type Name (and version)
Result	Success/Failure
Description	Deletes the local type definition for a model with the SAP. Note that this may not be permitted by the implementation if the SAP has model instances for that type, or may require the type for some other service.
Operation	**enumerateType**
Parameters	Selection Criteria
Result	Names (and versions) of model types that match selection criteria
Description	Enumerates the model types that match the selection criteria specified. Note that the SAP implementation MAY choose to also query the domain dictionary to extend the scope of the query.

The following instance level (M0) operations[42] are provided at each SAP:

Operation	**createInstance**
Parameters	Instance definition
Result	Reference to model instance (or null reference if unsuccessful)
Description	Creates a new model instance (locally) within the SAP. The corresponding type definition must already exist within the SAP. The SAP implementation MAY choose to fetch the type definition form the domain dictionary if necessary.
Operation	**getInstance**
Parameters	Reference to model instance
Result	Instance definition
Description	Returns the instance definition known locally to the SAP.
Operation	**modifyInstance**
Parameters	Reference to model instance, new definition
Result	Success/Failure
Description	Modifies the instance definition locally. Note only a fragment may be present in the new definition provided to limit the scope of the change.
Operation	**deleteInstance**
Parameters	Reference to model instance

[42] As with model types, local model instances may need to be reconciled with the domain directory service. See previous note relating to model types for how this may be handled.

Result	Success/Failure
Description	Deletes the model instance at the SAP. Note that this may not be permitted by the implementation if other services are also using the model instance
Operation	**enumerateInstances**
Parameters	Selection Criteria
Result	References to model instances that match selection criteria
Description	Enumerates the model instances that match the selection criteria specified. Note that the SAP implementation MAY choose to also query the domain directory to extend the scope of the query.

The following operation provides "pass-through" capability at the SAP for service operations.

Operation	**InvokeModelOperation**
Parameters	Model Reference, Operation parameters
Result	Operation results
Description	This operation provides a mechanism for the SAP to invoke operations defined within model instances held within it. Note that because this operation is dependent on the domain meta-model (which is defined by the domain architect), it may take a variety of forms necessary to support model operations.

It is likely that additional operations may be required based on the exact domain definition. In particular, additional operations may be needed to support "bulk" operations that can treat a number of model types or instances in an atomic operation.

6 Frequently Asked Questions

1. *It would be easy to argue here that you have just pushed the problem up a layer by saying that the interoperability problem is now one of model consistency rather than interface consistency. Why do you think models are a better way of specifying the service-specific behavior? This is perhaps THE fundamental question you need to answer in this work.*

   We agree. This is a hypothesis to be tested. We will be creating a reference implementation of this architecture to validate this hypothesis, and hopefully can show side-by-side comparison of the two ways to handle interoperability.

2. *You have not defined other concepts which are usually provided by the foundation services. That is, who provides definitions for things like identity and name in the SOA? Such definitions are critical for interoperability.*

   At the moment, only top-level entities are defined precisely within the architecture, but not attributes such as identity and name. The architecture document provides definitions that are applicable regardless of the meta-model selected for a specific instantiation of Picasso. As we gain more experience with what can be generalized across domains, we may provide additional definitions. These things also depend heavily on the meta-model selected, and the architecture currently expects these to be defined by the domain architect as part of the domain definition.

3. *Is it the case that models only represent services? This seems to mean that the only thing you can represent in the system is services of one sort or another. This may be ok, but it implies there can never be a "free" entity that's not part of a service of one sort or another. This may not hold up at the end.*

   Yes, that is the intent. The model is the external representation of the service, which includes its behavior, state, and operations (see definition of model in section 2). It will be possible to represent things other than services (for example, managed elements), but only as long as those things are part of the exposed state from some service. If no triggers are generated when you create a model at the SAP (which will be the case if no service cares about that model), then the current semantics in the model exchange pattern allows that creation to succeed. Again, assuming nothing cares about changes, any client can make any changes in that model, because the SAP will also allow that to succeed. Thus, in the worst case, any "free" entities have to become a part of the exposed state from the SAP (which represents a service).

4. *While foundation services are mentioned in the document, you do not provide any definitions or implementation guidelines for them.*

That is correct. We expect to provide more detailed models and guidelines for foundation services as we create a reference implementation. Those details are expected to be part of a separate document.

5. *I do not see a service mentioned here that I believe is critical to SOAs.*

   The list of foundation services only includes some services that are frequently mentioned. Omission from the list does not mean that the service is not necessary. We expect the domain architect to define the necessary foundation services based on the needs of the domain under consideration.

6. *You do not provide any specific mechanisms for cross-domain communication.*

   Cross domain communication is expected to happen through entities that are visible in both domains (see definition for domain). Since it is possible that the SOA protocols will differ in the two domains, the implementation of the shared entities is expected to provide any translation capabilities necessary.

7. *Is it just the service instances that have identity, or also the model instances?*

   Only the service instances have identity. Model instances are distinguished by using *model references*. Note that models will contain things like state, so service identity or name is not the same as a model reference. Clearly a model reference will be used to access the service identity, but model references provide separation between model elements, while identity is a service level notion.

8. *You have not provided use cases for service update and versioning.*

   Many other use cases are possible, and the list is evolving. We may add other use cases if they represent common patterns that are useful across architecture instances. Feel free to send suggestions (or better yet, actual use case definitions) for inclusion in the document.

9. *The model operations defined do not provide operations to create triggers.*

   Note that triggers do not have an implementation-independent representation. Triggers are generated by the SAP based on the presence of Filter instances (which are model elements) at the SAP. Filters can be defined using the type and instance operations defined in Section 5. The exact form for filters depends on the domain meta-model (and SAP models) selected by the domain and service architect.

10. *This document is too abstract. It would be helpful to have example mappings of the architecture (for example using WS-Management standards).*

    We considered that. However, given the number of such "standards," it is difficult to include them all. We will provide some of these mappings in separate documents as we create a reference implementation, and will probably include references to those documents in the near future.

11. *You label this as an SOA for model-based automation. I don't see anything in the document about automation.*

    That is true ☺. Automation in IT environments was the driver for the architecture, and we anticipate that we will use it for that purpose. Many of the definitions (e.g., model events) and communication constructs (e.g., model exchange pattern) within are present precisely because we have observed automation environments fail because of lack of such constructs in other SOAs. As we gain more experience with implementation, we expect to document (separately) automation-specific guidelines when using this architecture.

12. *There is a concept of a domain, but no distinction between managed and management domains. I assume I can define a domain as a management domain and only have management services as part of it. Then I could define a managed domain with all the managed services in it. Are cross-domain service interactions allowed?*

    Cross domain interactions are allowed by using entities (most likely services) that participate in multiple domains (see definition of domain in Section 2 and paragraph right under it). The reason for doing it this way (as opposed to explicitly modeling cross domain interactions) is that the SOA protocols, the modeling language and the meta-models may well be different in the two domains. While the initial thinking within the document deals with the management domain, it is possible to do the same thing within the managed domain. The domain architect is free to define domains and service scope within domains based on need (it is part of domain definition in use case DO01).

13. *I see that the document primarily describes the management domain. The management services are entities in the management domain. Entities from the managed domain need to be modeled as well and eventually be passed on as arguments to management services (e.g., ServerBootService.start(server234); ServerBootService is from the management domain, server234 an entity from the managed domain (assuming that the managed domain contains models about managed elements, and not the elements themselves); Alternatively, one could say: server234.start().*

    Managed entities are modeled as part of the exposed state from services and operations can be defined on those elements (see Footnote 14 and Steps 1a and 1b in use case 4.1.2: SD01 – Service Development). However, if a service does not expose a managed element directly to other services, then there is no obligation to represent it in the model. Whether the first or the second form of method invocation is used depends on the meta-model defined for the domain.

14. *Does the architecture encompass services that may be provided by humans, or is it restricted to automated services?*

    Humans are explicitly part of the ecosystem. However, since human actions are not easy to model, they are part of the external environment that can cause spontaneous changes in state of services (see definition of service in section 2, and paragraph right under it). The same view is represented in Figure 3 where humans interact directly with services. How that interaction takes place is left service dependent and the underlying SOA makes no assumptions about it. The important part is not who performs a service, how it is performed or how it communicates with humans, but the corresponding model that is exposed to the SOA. Thus you can represent

a person/role as a proxy object (a service) which exposes the corresponding state. It is left up to the proxy object implementation to decide how it would interact with the humans it represents.

15. *You are allowing state to be encapsulated as part of the services. This causes problems in maintaining state coherence. Why do you not restrict services to be stateless?*

    The architecture does not prohibit stateless services—a service can be stateless. However, in any large environment, state has to be maintained somewhere, and providing access to that state still requires a service within a SOA context. For example, an inventory service that can be queried for number of inventory items available is exposing state. Thus at least some of the services will need to encapsulate state. The domain and service architects decide based on the domain requirements whether a few services carry the majority of the state, or whether state can be distributed across a large number of services.

    If by state, you mean session state (i.e., a particular message sent to a service does not assume that the service retains information contained in previous messages), that decision is an implementation choice made by the service architect, and the architecture neither requires nor precludes that choice.

16. *You state that multiple versions of the same model may be present in the domain. How are the different versions differentiated?*

    The versioning conventions are determined by the underlying meta-model chosen for representing domains. The architecture recognizes that multiple versions may need to co-exist for evolution of services with the domain; however it leaves open the exact form that are used within the domain as a choice to be made by the domain architect.

17. *The model exchange pattern does not provide any "atomicity" guarantees. I can see how a single operation can be rolled back if it does not succeed—what happens if my operation requires changes to multiple operations or if it cannot be rolled back?*

    Without more understanding of the exact domain models, it is difficult to make atomicity claims because a change in one model may lead to changes in other models elsewhere. In practical scenarios, it may not be possible to roll back operations. The model exchange pattern attempts to insure that operations are rolled back when possible, but more importantly, that the clients become aware of state corruption as soon as possible to avoid propagating errors because of state corruption. Within a given instance of this architecture, it may be possible to make stronger

Glossary

Term	Page	Definition
Approval Request	13	An approval request is a trigger that requires approval from some listener (*the change approver*) before the pending change can be committed.
Change Notification	13	A change notification is a trigger that that simply informs the listener of a model change that has been committed at the SAP.
Domain	8	A *Domain* is the set of related entities that interact with one another to accomplish some purpose. Domains define the scope of discourse between related entities, and hence the types of entities necessary, the vocabulary (ontology) used to describe the entities, and the syntactic constructs that are understood by all entities within the domain. A given entity may participate in multiple domains.
Entity	7	An entity is a thing or a concept that is relevant for the operation of the system and thus needs representation inside the system.
Indication	13	An *Indication* is a model element that is passed to a service (or service client) as result of a model event trigger. Indications provide sufficient information to the recipient to allow it to obtain the model state as proposed by the change
Indication Listener	13	A service that is capable of receiving indications is an *indication listener*.
Model	9	See "Service Model"
Model Event	12	A *Model Event* is a pending change of some service model element caused by a model operation performed by some service at a service access point. The model operation may target either the model type or the model instance at the SAP. Model operations or changes in external state that cannot cause in changes in models hosted by an SAP do not represent model events.
Model Event Filter	14	A *Model Event Filter* is a model element that defines a query on service models held at a service access point. The query specifies model operations (create/ update/ delete/ invoke) on specified model elements, as well as references to one or more indication listeners that need to be notified if the corresponding query succeeds
Model Event Trigger	13	A *Model Event Trigger* is a model event that requires the service

		access point to notify some service (or service client) of a pending change in some model element
Role	14	A *Role* represents the expectation that a certain service (or human) entity within a domain is assigned or required to perform a given task or activity
Service	9	A *Service* is an entity that represents an encapsulation of functionality and state useful to other entities within a domain
Service Access Point (SAP)	11	A *Service Access Point* is a service end-point that provides a standard interface for interaction between services using a model exchange pattern. A service access point represents a distributed proxy service that provides the ability to utilize other services through a restricted set of model operations.
Service Model	9	A *Service Model* is the representation of a service within the SOA. It defines the externally visible description, behavior, state, and operations available from a service to other services. Within a domain, each service defines its own service model and is responsible for exposing it to other services within that domain.