SOFTWARE ADAPTATION PATTERNS FOR SERVICE-ORIENTED ARCHITECTURES

by

Koji Hashimoto
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Software Engineering

Committee:

_____	Dr. Hassan Gomaa, Thesis Director
_____	Dr. João P. Sousa, Committee Member
_____	Dr. Sam Malek, Committee Member
_____	Dr. Hassan Gomaa, Department Chair
_____	Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering
Date:_____	Spring Semester 2010 George Mason University Fairfax, VA

Software Adaptation Patterns for Service-Oriented Architectures

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Koji Hashimoto
Ph.D.
Osaka University, 2000

Director: Hassan Gomaa, Professor
Department of Computer Science

Spring Semester 2010
George Mason University
Fairfax, VA

DEDICATION

To my beloved parents, Junko and Takao
To my wonderful wife Kaori
To my lovely daughters, Riko and Fumi

ACKNOWLEDGEMENTS

I would like to thank my dissertation director Dr. Hassan Gomaa for his guidance, support, and encouragement during the course of this research. I would also like to thank my committee members Dr. João P. Sousa and Dr. Sam Malek for serving on my committee and their guidance throughout my education at George Mason University. I would like to extend my thanks to Dr. Daniel. A. Menascé for his invaluable advice and support during the course of this research.

I would like to thank my research colleagues for their cooperation and support. I am especially grateful to Mohammad A. Abu-Matar, John Ewing, Minseong Kim, Naeem Esfahani, Zeynep Zengin and Ahmed Elkhodary.

I am thankful to my parents, parents-in-law, siblings and extended family & friends for their unconditional love and continuous encouragement throughout my life.

Lastly, I am thankful to my beloved wife, Kaori, for her continuous effort, understanding, and patience during my research, and to my wonderful and lovely daughters, Riko and Fumi, who gave me a reason to survive and complete my masters' degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

SOFTWARE ADAPTATION PATTERNS FOR SERVICE-ORIENTED ARCHITECTURES

Koji Hashimoto, MS

George Mason University, 2010

Thesis Director: Dr. Hassan Gomaa

This thesis describes the concept of software adaptation patterns and how they can be used in software adaptation of service-oriented architectures. The patterns are described in terms of a three-layer architecture for self-management. A software adaptation pattern defines how a set of components that make up an architecture pattern dynamically cooperate to change the software configuration to a new configuration.

The software architecture in SOA is characterized by service coordination where services are orchestrated and/or sequenced by coordinators. As there are many different types of service coordination, this thesis focuses on SOA coordination patterns to capture the different kinds of coordination. Thus, a software adaptation pattern is developed for each coordination pattern.

This thesis introduces adaptation connectors to encapsulate adaptation state machine models so that the adaptation patterns can be more reusable. A change management model for dynamically adapting service-oriented applications is also described with a case study.

CHAPTER 1 INTRODUCTION

Service-Oriented Architectures (SOA) are becoming increasingly widespread in a variety of computing domains such as enterprise and e-commerce systems, which continue to grow in size and complexity. These systems are expected to adapt not only to the fluctuating execution environments but also to changes in their operational requirements.

This thesis describes the concept of software adaptation patterns and how they can be used in service oriented architectures. Previous research has investigated how software architectural patterns can be used to help in building software systems and product lines [1][7][8]. This thesis describes how software adaptation patterns can be used to help with the adaptation of service-oriented software systems after original deployment.

This thesis focuses on three areas as follows: a) research into software architectural and design patterns [2][3][7] applied in particular to service-oriented architectures [16][17], b) research into dynamic reconfiguration and change management [6][9], and c) research into self-adaptive, self-managed or self-healing systems [4][10]. The research described in this thesis builds on software reconfiguration patterns in the previous work [6] and advances these concepts by describing patterns to support dynamic adaptation in service-oriented applications.

In this thesis, adaptation connectors are introduced to encapsulate adaptation state machine models so that adaptation patterns can be more reusable. The goal of an adaptation connector is to separate the concerns of an individual service from dynamic adaptation, i.e., the adaptation connector implements the adaptation mechanism for its corresponding service.

In typical SOA applications, services are self-contained and loosely coupled, and orchestrated by coordination services. As there are many different types of service coordination, this thesis develops SOA coordination patterns to capture the different kinds of coordination.

This thesis first describes different kinds of software adaptation. It then describes software adaptation patterns according to SOA coordination patterns mentioned above. Adaptive change management is also described with a case study. The SOA adaptation patterns described in this thesis are validated by simulation and Web Service based prototype implementation.

CHAPTER 2 RELATED WORK

Dynamic software architectures and dynamic reconfiguration approaches have been applied to dynamically adapt software systems [4][10]. These approaches address incorporating reconfigurability into the architecture, design and implementation of software systems for the purpose of run-time change and evolution. In [6][11][12], dynamic reconfiguration is applied to changing the configuration of a system from one configuration to another in a software product line while the system is operational. Research into self-adaptive, self-managed or self-healing systems [4][10] includes various approaches for monitoring the environment and adapting a system's behavior in order to support run-time adaptation.

Kramer and Magee [9][10] describe how a component must transition to a quiescent state before it can be removed or replaced in a dynamic software configuration. Ramirez et al. [13] describe applying adaptation design patterns to the design of an adaptive web server. The patterns include structural design patterns and reconfiguration patterns for removing and replacing components.

For service-oriented computing and service-oriented architectures, Li et al. [14] suggest the adaptable service connector model, so that services can be dynamically composed.

Irmert et al. [15] provide a framework to adapt services at run-time without affecting application execution and service availability.

In comparison with the previous approaches, this thesis focuses on service coordination in service-oriented architectures. This thesis develops software adaptation patterns for different kinds of service coordination, in order to adapt not only services but also coordinator components.

CHAPTER 3 SOFTWARE ADAPTATION

Software adaptation addresses software systems that need to change their behavior during execution. In self-managed and self-healing systems, systems need to monitor the environment and adapt their behavior in response to changes in the environment [10]. Garlan and Schmerl [4] have proposed an adaptation framework for self-healing systems, which consists of monitoring, analysis/resolution, and adaptation. Kramer and Magee [9] have described how in an adaptive system, a component needs to transition from an active (operational state) to a quiescent (idle) state before it can be removed from a configuration.

Adaptation can take many forms. It is possible to have a self-managed system that adapts the algorithm it executes based on changes it detects in the external environment. If these algorithms are pre-defined, then the system is adaptive but the software structure and architecture is fixed. The situation is more complex if the adaptation necessitates changes to the software structure or architecture. In order to differentiate between these different types of adaptation, adaptations can be classified as follows within the context of distributed component-based software architectures:

a) Behavioral adaptation. The system dynamically changes its behavior within its existing structure. There is no change to the system structure or architecture.

b) Component adaptation. Dynamic adaptation involves replacing one component with another that has the same interface. The dynamic replacement of the old component(s) with a new component(s) has to be performed while the system is executing.

c) Architectural adaptation. The software architecture has to be modified as a result of the dynamic adaptation. Old component(s), which may not provide the same interface, must be dynamically replaced by new component(s) while the system is executing. Hence, the changes may impact the architectural configuration (e.g., architectural style) of the system.

Model based adaptation can be used in each of the above forms of dynamic adaptation, although the adaptation challenge is likely to grow progressively from behavioral adaptation through architectural adaptation.

CHAPTER 4 THREE-LAYER ARCHITECTURE MODEL FOR SOFTWARE ADAPTATION

The approach in this thesis for software adaptation is compatible with the widely accepted three-layer reference architecture model for self-management [10]. The architecture model consists of 1) Goal Management layer—planning for change—often human assisted, 2) Change Management layer—execute the adaptation in response to changes in state (environment) reported from lower layer or in response to goal changes from above, and 3) Component Control layer—executing architecture that actually implements the run-time adaptation.

This reference architecture for self-management originally comes from research [18][19] on control architectures for robotic systems. The robot control architectures are composed of three layers, namely deliberate, sequencing, and reactive layers. The deliberate layer is to interface with a user and to execute a planning process. The sequencing layer is to execute the plan by managing the components in the layer below. The reactive layer is responsible for reactive control of robot behavior. As a result, the three layers of the architecture model for self-management, i.e., Goal Management, Change Management, and Component Control layers, are consistent with those of robot control architectures, i.e., deliberate, sequencing, and reactive layers, respectively. This three-layer model

provides a good conceptual architecture that helps identify and organize the necessary features for dynamic software adaptation.

4.1 Three-layer Architecture Example: SASSY Framework

Self-Architecting Software Systems (SASSY) is a model-driven framework for run-time self-architecting and re-architecting of distributed service-oriented software systems [20][21]. SASSY provides a uniform approach to automated composition, adaptation, and evolution of software systems. SASSY provides mechanisms for self-architecting and rearchitecting that determine the best architecture for satisfying functional and Quality of Service (QoS) requirements. The quality of a given architecture is expressed by a utility function, which is provided by end-users and represents one or more desirable system objectives.

Figure 1 illustrates, at a high level, how SASSY uses the three layer architecture model for self adaptation of SOA-based software. The self-architecting and re-architecting component at the Goal Management layer generates a near-optimal system service architecture, consisting of components, (associated with services) and connectors (associated with middleware facilities), with respect to QoS requirements through selection of the most suitable services. This architecture is determined with the help of QoS analytical models and optimization techniques aimed at finding near-optimal choices that maximize system utility [22].

The architecture is executed on top of SASSY run-time infrastructure support which is composed of SASSY Monitoring Support and Adaptation Support in Figure 1. SASSY Monitoring Support services (Monitoring Service and Gauge Service) generate triggers that cause two kinds of self-adaptation, *reactive adaptation* and *proactive adaptation*. In reactive adaptation, the Component Control layer quickly replaces services when they fail. In proactive adaptation, the software architecture is automatically regenerated by the Goal Management layer when the system utility degrades beyond a certain threshold. SASSY Adaptation Support services (Change Management Service and Adaptation Service) are used to transition from one version of the architecture to a new one. Thus, when services are unable to meet their QoS goals, the SASSY's monitoring services trigger proactive adaptation through a new round of service discovery, optimal service selection, and possible determination of alternative architectures.

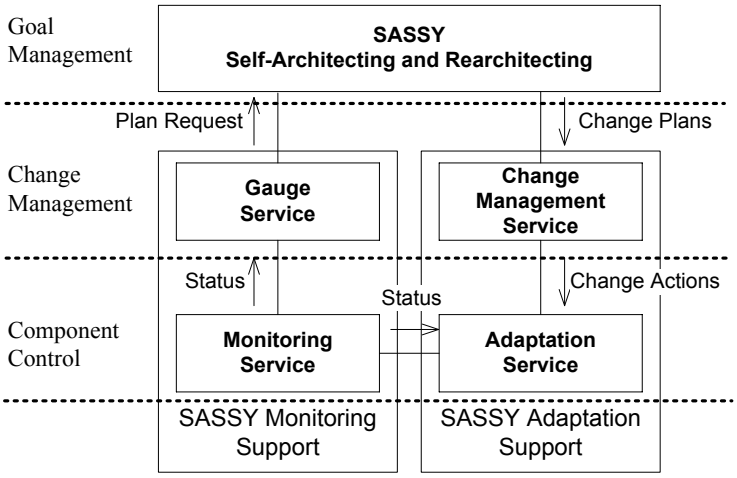


Figure 1 High-level view of SASSY framework

CHAPTER 5 SOFTWARE COORDINATION

In SOA applications, services are intended to be self-contained and loosely coupled, so that dependencies between services are kept to a minimum. Instead of one service depending on another, it is desirable to provide coordination services (coordinators) in situations where multiple services need to be accessed and access to them needs to be coordinated and/or sequenced. In SOA systems, this loose coupling is ensured by separating the concerns of individual services from those of the coordinators, which sequence the access to the individual services.

As there are many different types of service coordination, it is helpful to develop SOA coordination patterns to capture the different kinds of coordination. In particular, coordination can be categorized by the following three properties: 1) type of coordination, 2) degree of concurrency, and 3) type of service. By characterizing the behavioral and structural properties of these coordination patterns, it is possible to analyze and estimate the possible adaptation paths of these patterns.

The type of coordination can be:

- Independent coordination - each coordinator instance operates independently of other coordinators in its interactions with services. E.g., an airline coordination

service that contacts multiple airline services to offer travel alternatives to a customer. There are many instances of this coordinator, one for each customer. This is probably the most common form of coordinator in SOA.

- Distributed coordination – overall coordination of SOA application consists of multiple coordinators that are distributed and need to cooperate with each other. An example is an Emergency Response SOA where a city emergency coordinator seeks help from a neighboring city emergency coordinator.
- Hierarchical coordination – overall coordination of SOA application is hierarchical with a high-level coordinator making major decisions and assigning more detailed coordination tasks to lower-level coordinators. During a hurricane, regional emergency coordinators might sends requests for fire engine and ambulance assistance to local city emergency coordinators.

Degree of concurrency can be:

- Sequential coordination – a coordinator interacts with multiple services sequentially in order to achieve the overall service objective. For example, a travel coordinator needs to first make an airline reservation and determine the airline travel dates before making a hotel reservation.
- Concurrent coordination – a coordinator interacts with multiple services concurrently in order to achieve the overall service objective, e.g., airline coordination service that contacts multiple airline services to offer travel alternative to a customer.

- Combined sequential and concurrent coordination – a coordinator does some sequential coordination and some concurrent coordination. For example, a travel coordinator needs to first make an airline reservation and determine the airline travel dates before making hotel and car rental reservations, which can be made in parallel.

Type of service can be:

- Stateless service – a service bases its response entirely on the request received.
- Stateful service – a service bases its response not only on the request received but also on the current state of the service.

A given coordinator can be characterized by these three properties, for example an independent coordinator that does concurrent coordination of stateful services. It is possible for a coordinator to be state dependent if the coordination needs to follow a specified sequence.

CHAPTER 6 SOFTWARE ADAPTATION PATTERNS

A software architecture is composed of distributed software architectural patterns, such as client/server, master/slave, and distributed control patterns, which describe the software components that constitute the pattern and their interconnections. For each of these architectural patterns, there is a corresponding software adaptation pattern, which models how the software components and interconnections can be changed under predefined circumstances, such as replacing one client with another in a client/server pattern, inserting a control component between two other control components in a distributed control pattern, etc.

A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of adaptation commands. In terms of the three-layer reference architecture described earlier, adaptation patterns correspond to the bottom layer of a self-managed system, i.e., the component control layer, and they realize dynamic adaptation by actually adding or deleting components (and appropriate connectors if necessary) according to adaptation commands sent from the middle layer, i.e., the change management layer. On the other hand, the goal management layer is in charge of producing change management plans (to satisfy QoS goals) that are executed at

the change management layer. Thus, the focus in this thesis is mainly on the component control layer for dynamic software adaptation.

A software adaptation pattern requires state- and scenario-based reconfiguration behavior models to provide for a systematic design approach. The adaptation patterns are described by adaptation interaction models (using communication or sequence diagrams) and adaptation state machine models [6][8]. Previously developed adaptation patterns include the Master-Slave Adaptation Pattern, Centralized Control Adaptation Pattern, and Decentralized Control Adaptation Pattern [6].

6.1 Software Adaptation State Machines

An adaptation state machine defines the sequence of states a component goes through from a normal operational state to a quiescent state. A component is in the Active state when it is engaged in its normal application computations. A component is in the Passive state when it is not currently engaged in a transaction it initiated, and will not initiate new transactions. A component transitions to the Quiescent state when it is no longer operational and its neighboring components no longer communicate with it. Once quiescent, the component is idle and can be removed from the configuration, so that it can be replaced with a different version of the component. Figure 2 shows the basic adaptation state machine model for a component as it transitions from Active state to Quiescent state. The adaptation framework sends a passive command to the component.

If the component is idle, it transitions directly to the quiescent state. However, if the component is busy participating in a transaction, it transitions to the Passive state. When the transaction is completed, it then transitions to the Quiescent state.

In previous research [6], the state machine for each component was modeled using two orthogonal state machines, an operational state machine (modeling normal component operation) and an adaptation (also referred to as reconfiguration) state machine.

However, for more complicated patterns, there is often some interaction between the two state machines. This thesis investigates separating the operational state machine from the adaptation state machine in service-oriented systems. The objective is to encapsulate the adaptation state machine in the service connector (as discussed in the next section), such that the adaptation patterns, as well as the corresponding code that realizes each pattern, can be more reusable.

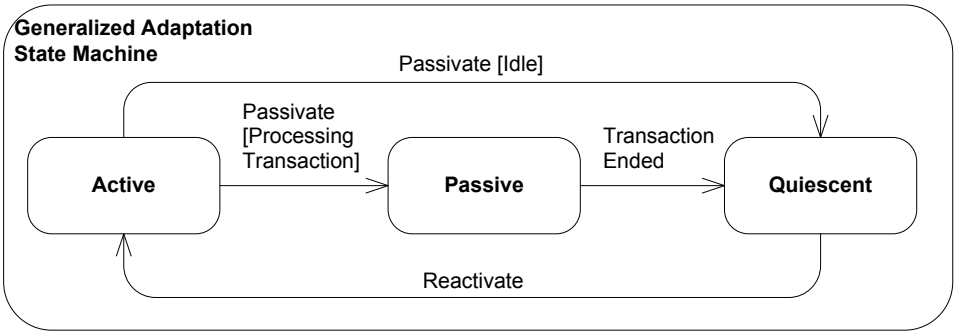


Figure 2 Basic adaptation state machine

6.2 Adaptation Connectors

In this thesis, adaptation connectors are introduced to encapsulate adaptation state machine models so that adaptation patterns can be more reusable. The adaptation patterns described in this thesis include two different types of adaptation connectors, *coordinator connector* and *service connector*, as shown in Figure 3. A service adaptation connector behaves as a proxy for a service, such that its clients can interact with the connector as if it was the service. The goal of an adaptation connector is to separate the concerns of an individual service from dynamic adaptation, i.e., the adaptation connector implements the adaptation mechanism for its corresponding service, including the interaction with the change management service (see next chapter) and the management of the operational states of the service. The adaptation state machine for a given adaptation pattern is encapsulated in the corresponding adaptation connector. In the following sections, the adaptation state machines for SOA software adaptation patterns are described.

CHAPTER 7 SOA ADAPTATION PATTERNS

As described in Chapter 5, a coordination pattern can be characterized by the type of coordination, the degree of concurrency, and the type of service. The following chapters describe the adaptation patterns for the following coordination patterns:

1. Independent coordination of stateless services
    a. Sequential coordination (Section 7.1.1)
    b. Concurrent coordination (Section 7.1.2)
    c. Combined sequential / concurrent coordination (Section 7.1.3)
2. Two-phase commit coordination (coordination of stateful services) (Section 7.2)
3. Hierarchical coordination (Section 7.3)
4. Distributed coordination (Section 7.4)

7.1 Independent Coordination Adaptation Patterns

This section considers independent coordination, which is a common form of coordination in SOA. In the independent coordination pattern, a coordinator orchestrates multiple services independently of other coordinators as shown in Figure 3. The assumptions are as follows:

- A coordinator component is instantiated for each client.

- A client interacts with a coordinator using synchronous communication; thus, it sends a new request only when it receives a response to its previous request.
- Services are stateless and independent of each other.

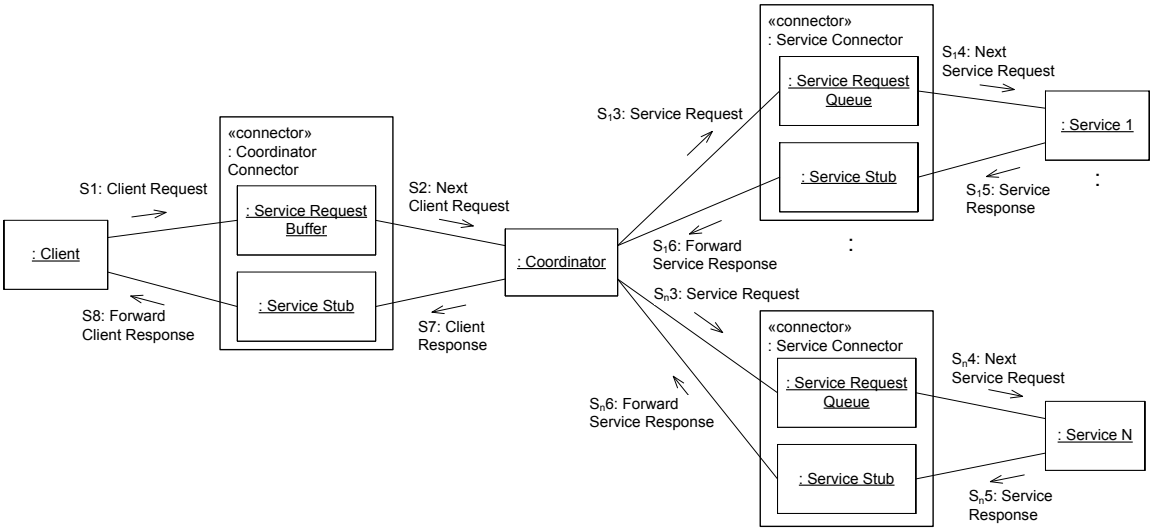


Figure 3 Independent coordination communication diagram

7.1.1 Sequential Coordination Adaptation Pattern

In a sequential coordination adaptation pattern for service-oriented architectures, multiple services are sequentially invoked by a coordinator, e.g., airline reservation followed by hotel reservation. Once the coordinator receives a client request for an application service, it sends a service request to the first service to be invoked. The coordinator sends another service request to the second service after receiving a response from the first service. The coordinator sends a response to the client after a response from the last

service has been received. The communication diagram depicted in Figure 3 shows a general case where a coordinator interacts with N services.

Based on the assumptions described earlier, the coordinator component can only be removed or replaced after it has received all the responses from the services sequentially invoked and sent its response to the client. On the other hand, a service can be removed or replaced after it completes the current service execution in the case of a sequential service, or after completing the current set of service executions in the case of a concurrent service.

The coordinator connector executes its own operating state machine shown in Figure 4. As mentioned in Section 7.1, the adaptation state machine for the coordinator is encapsulated in the coordinator's connector. The connector implements the dynamic adaptation of the coordinator when it receives the Passivate adaptation command from the change management service. In Figure 4, when the connector is Active (idle and not executing a client command) and receives a Passivate command, it transitions to the "Quiescent" state and sends a quiescent notification to the change management service. When the connector is Active and receives a request from the client, it forwards the request to the coordinator and transitions to the "Waiting For Service Response" state, which means that the coordinator is processing a request. If the connector receives a Passivate command, it transitions to the "Passivating" state in which the coordinator is still interacting with the service to accomplish its transaction.

When the coordinator receives a response from the last service, it sends the client response to the connector. The connector then transitions to the "Quiescent" state; the actions are to forward the response to the client, and to send a quiescent notification to the change management service.

While in the quiescent state, the connector could receive a new request from the client, which it puts into a request buffer. When the connector receives a Reactivate command, it transitions to either the "Waiting For Client Request" or the "Waiting For Service Response" state according whether or not there is a client request in the request buffer.

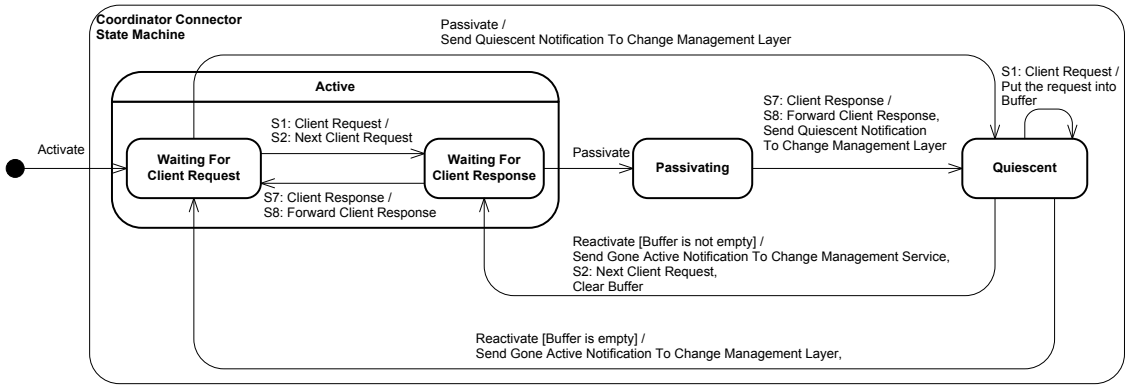


Figure 4 Coordinator connector state machine

Figure 5 depicts the operating state machine executed by the service connector in the case of a sequential service that has a single thread processing requests. As a service may have multiple clients besides the coordinator, the service request queue has an important role

to buffer requests in a sequence. If a service request arrives in the "Waiting For Service Request" state, it is immediately forwarded to the service. In the "Processing" and "Passive" states, the connector queues service requests on the request queue. In other words, clients can send requests to the service regardless of its state, because the connector will queue service requests if necessary. If the change management service sends a Passivate command to the connector, the connector will transition to Quiescent state and continue to queue up any client requests that arrive. In Quiescent state, the service can be replaced. When the newly replaced service becomes active, the connector resumes sending client requests to the service, as depicted on the state machine shown in Figure 5.

As a future work, the service connector could forward client requests to the newly replaced service immediately after it would receive a Passivate command, without waiting for the state machine to transition to Quiescent state.

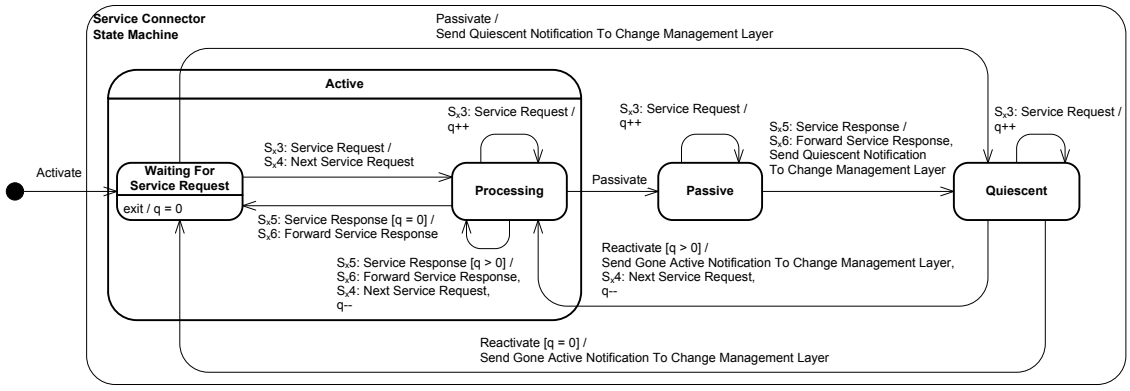


Figure 5 Service connector state machine for a sequential service

In the case of a concurrent service, the service connector executes the operating state machine shown in Figure 6. Since services can be removed or replaced after completing the current set of service executions, the state machine manages the number of requests currently executed by the service by the variable t. Although a request is immediately forwarded to the service in Active or Processing state, the connector still necessitates the request queue because it cannot forward a new request to the service which is passive or quiescent.

In the following adaptation patterns, this thesis considers only a concurrent service because most of common service provider server implementations [27][28][33] support a thread pool mechanism to provide concurrent services, and have a request queue to handle the case where the number of incoming requests exceed the thread pool capacity.

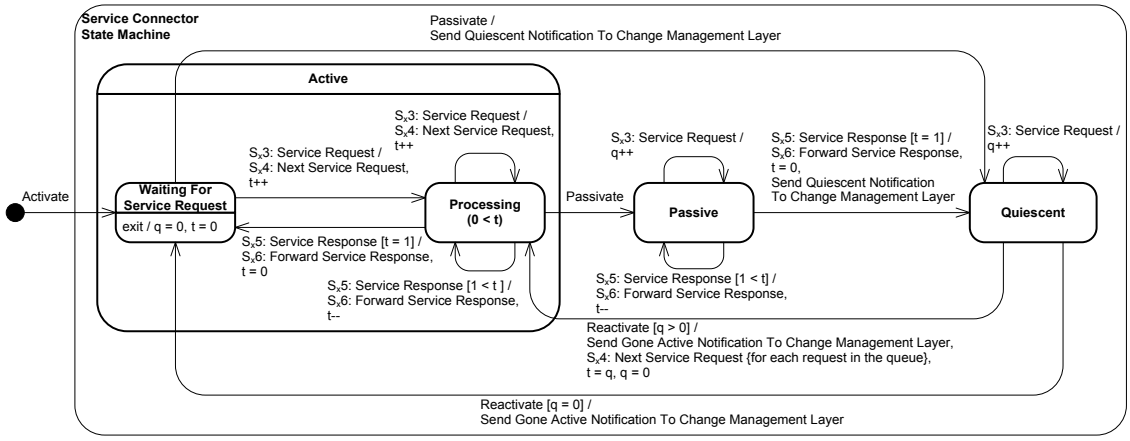


Figure 6 Service connector state machine for a concurrent service

7.1.2 Concurrent Coordination Adaptation Pattern

In the concurrent coordination adaptation pattern, multiple services are invoked by a coordinator concurrently. Once the coordinator receives a client request, it sends service requests concurrently to all the services to be invoked. The coordinator sends a response to the client after the responses from all the services have been received. Since this pattern also involves independent coordination, the structural view of this pattern is the same as Figure 3.

In this adaptation pattern, the operating state machine for the coordinator connector is exactly the same as shown in Figure 4, because the coordinator connector determines when the coordinator is in Quiescent state by monitoring only the message "S7: Client Response" from the coordinator.

The operating state machine for the service connector in the case of a sequential service and a concurrent service are also the same as shown in Figure 5 and Figure 6, respectively. As in the case of the coordinator connector, the service connector determines when its corresponding service is in Quiescent state by monitoring only the message "S_x5: Service Response" sent from the service.

7.1.3 Combined Coordination Adaptation Pattern

For the combined sequential and concurrent coordination pattern described in Chapter 5, the operating state machines for the coordinator and service connecters are the same as those in the sequential and concurrent coordination patterns, for the same reasons described in Section 7.1.2. Therefore, the adaptation connectors can be reused in the independent coordination pattern, regardless of the coordinator's degree of concurrency (sequential, concurrent, or combined sequential/concurrent).

7.2 Two-Phase Commit Coordination Adaptation Pattern

This section describes the adaptation pattern for Two-Phase Commit Coordination, which is a typical coordination of stateful services. Consider an example of a transfer transaction between two bank accounts – e.g., from a savings account to a checking account – in which the accounts are maintained at two separate banks (servers). In this case, it is necessary to debit the savings account and credit the checking account. Therefore, the transfer transaction consists of two operations that must be atomic (i.e., indivisible) – a debit operation and a credit operation – and the transfer transaction must be either committed or aborted:

- Committed – both credit and debit operations occur.
- Aborted – neither the credit nor the debit operation occurs.

One way to achieve this result is to use the Two-Phase Commit Protocol, which synchronizes updates on different nodes in a distributed application. The result of the Two-Phase Commit Protocol is that either the transaction is committed (in which case all updates succeed) or the transaction is aborted (in which case all updates fail).

The communication diagram in Figure 7 (or Figure 8) shows a general case where a coordinator executes the Two-Phase Commit Protocol with N protocol participants (services). The message exchange sequences in the cases of committed and aborted are depicted in Figure 7 and Figure 8, respectively.

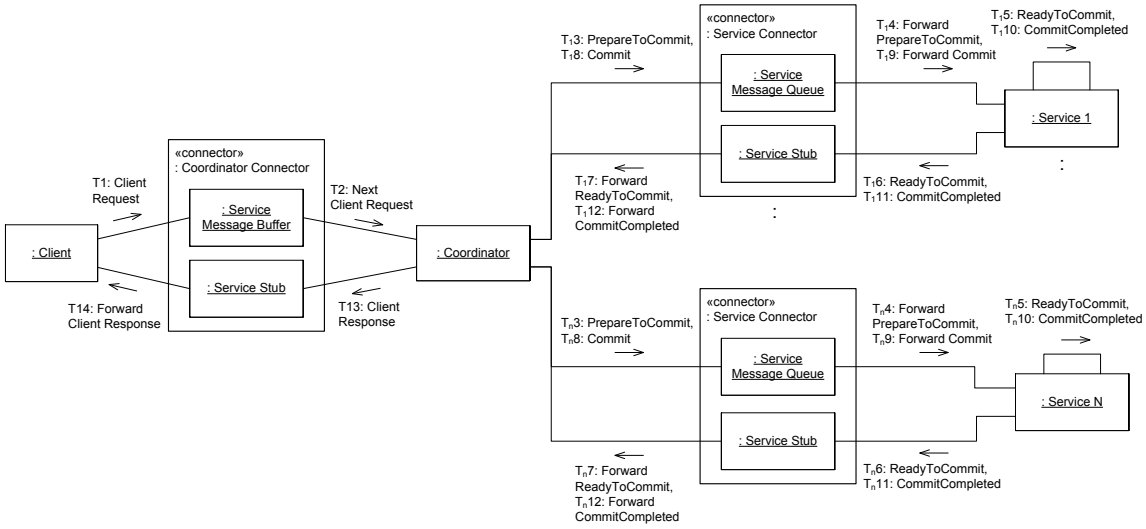


Figure 7 Two-Phase Commit coordination communication diagram: committed case

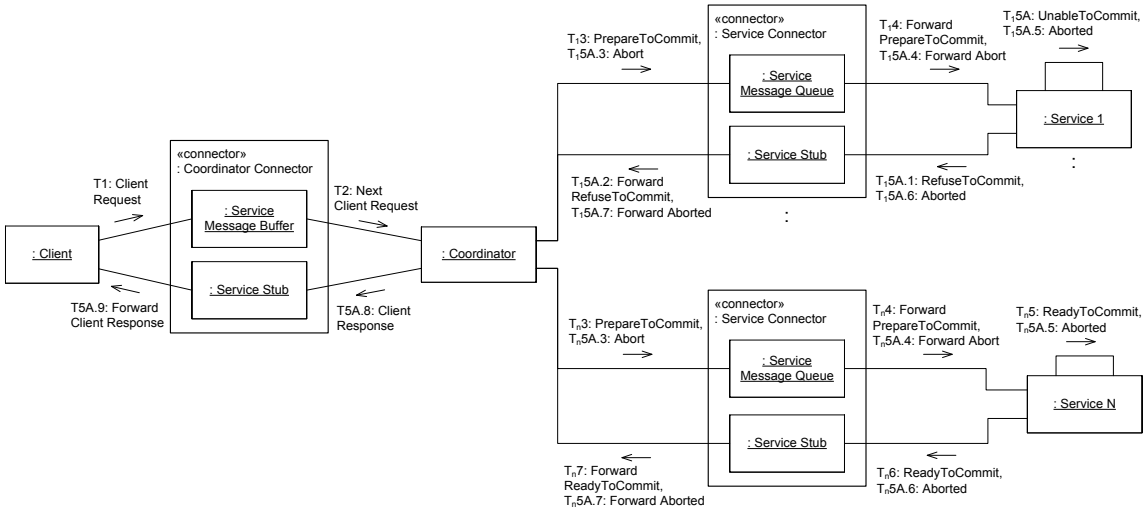


Figure 8 Two-Phase Commit coordination communication diagram: aborted case

Once the coordinator receives a client request, it starts the Two-Phase Commit Protocol. In the first phase, the coordinator sends a "T_x3: PrepareToCommit" message to each participant service. Each participant service locks the record, performs the transaction, and then sends a "T_x6: ReadyToCommit" message to the coordinator (Figure 7). If a participant service is unable to perform the transaction, it sends a "T_x5A.1: RefuseToCommit" message (Figure 8). The coordinator waits to receive responses from all participants.

When all participant services have responded, the coordinator proceeds to the second phase of the Two-Phase Commit Protocol. If all participants have sent "T_x6: ReadyToCommit" messages (Figure 7), the coordinator sends the "Commit" message to each participant service. Each participant service makes the transaction permanent,

unlocks the record, and sends a "T_x11: CommitCompleted" message to the coordinator. The coordinator waits for all "CommitCompleted" messages. The coordinator sends a response to its client after all the messages have been received.

If a participant service responds to the "PrepareToCommit" message with a "ReadyToCommit" message, it is committed to completing the transaction. The participant service must then complete the transaction even if a delay occurs (e.g., even if it goes down after it has sent the "ReadyToCommit" message). If, on the other hand, any participant service responds to the "PrepareToCommit" message with a "RefuseToCommit" message, the coordinator sends an "T_x5A.3: Abort" message to all participants (Figure 8). The participants then roll back the transaction.

Based on the Two-Phase Commit Protocol described above, the coordinator component can only be removed or replaced after it has received "CommitCompleted" or "Aborted" message from all the participant services and sent its response to the client. On the other hand, a participant service can be removed or replaced after it completes the current transaction of the protocol in the case of a sequential service, or after completing the current set of transactions in the case of a concurrent service.

In the adaptation pattern for the Two-Phase Commit coordination, as shown in Figure 9, the operating state machine for the coordinator connector is exactly the same as Figure 4 (only the labels for events and actions are changed according to the communication

diagrams in Figure 7 and Figure 8), because only the message "T13: Client Response" or "T5A.8: Client Response" determines when the coordinator goes to Quiescent state.

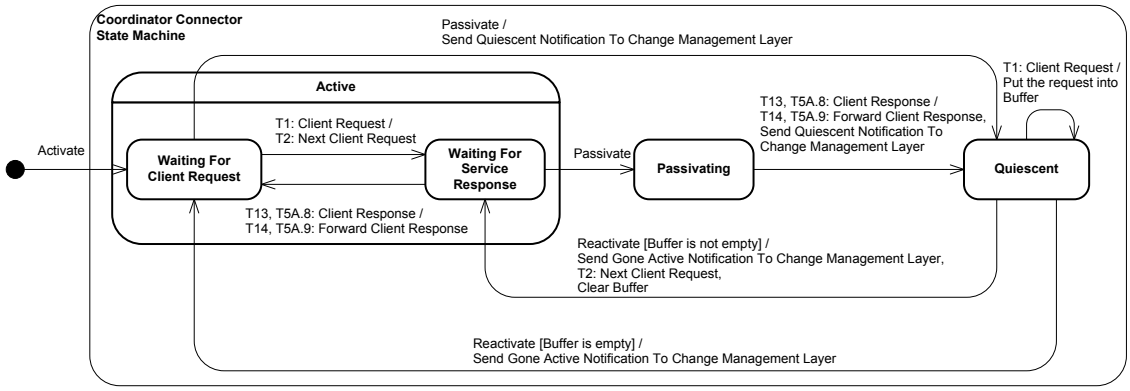


Figure 9 Coordinator connector state machine

Figure 10 depicts the operating state machine executed by the service connector in the case of a concurrent service that has multiple threads processing requests. The key idea behind this state machine is that the connector monitors if its corresponding service is in transactions of the Two-Phase Commit Protocol.

If a "PrepareToCommit" message arrives in the "Waiting For Service Request" state, it is immediately forwarded to the service. The connector stays in the "Processing" state, and forwards intermediate messages such as "ReadyToCommit", "Commit", "RefuseToCommit", and "Abort", because the service is still in the transaction of the Two-Phase Commit Protocol. The state machine manages the number of transactions currently executed by the service by the variable t, because services can be removed or

replaced after completing the current set of transactions. Therefore, even if it receives "CommitCompleted" or "Aborted" message, it remains in the "Processing" state if $t > 1$, i.e., the service is still in other transactions.

If the connector receives a Passivate command from the change management service, the connector transitions to the "Passive" state. In the "Passive" state, the connector queues "PrepareToCommit" messages on the request queue. As in the "Processing" state, the connector forwards intermediate messages, and stays in the "Passive" state. Furthermore, the connector remains in the "Passive" state if $t > 1$ even when it receives "CommitCompleted" or "Aborted" message. On the other hand, if $t = 1$ when the connector receives "CommitCompleted" or "Aborted" message, it transitions to the "Quiescent" state in which the service can be replaced, because the service is executing no transactions of the Two-Phase Commit Protocol.

Note that the two-phase commit coordination adaptation pattern described in this section can be extended to multiple-phase commit protocols such as a three-phase commit protocol. Furthermore, this pattern can be extended to the case of session management between a client and a service. In this case, the adaptation connector for such a stateful service must monitor the message to initiate a new session and the final response to end the session. To develop the adaptation state machine for the session management is a future work.

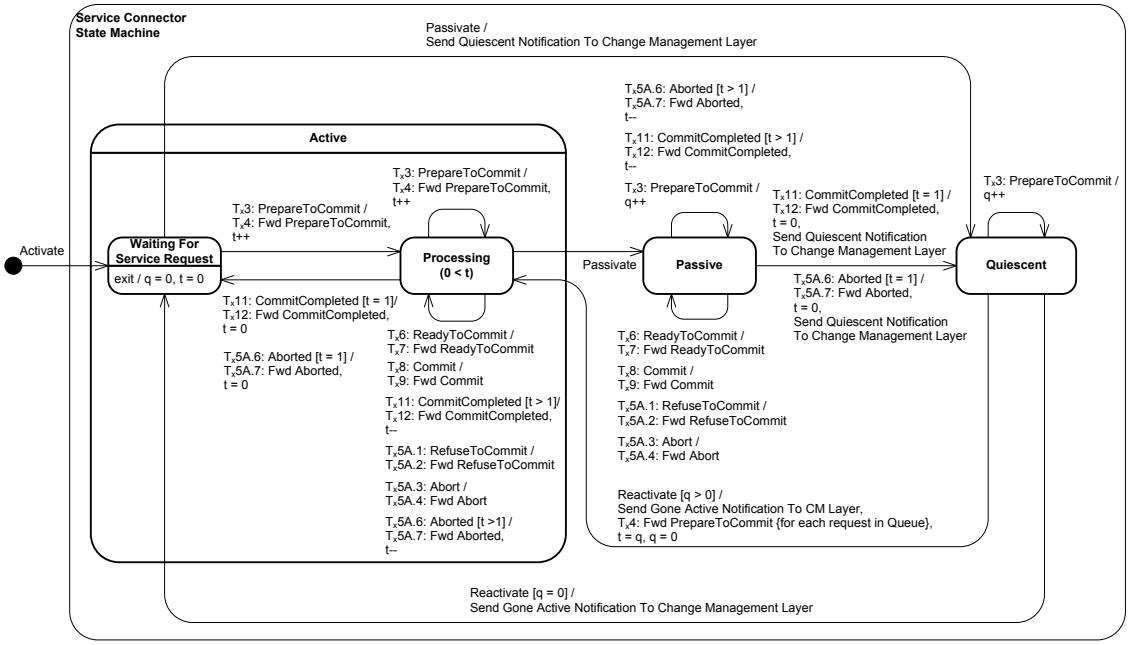


Figure 10 Service connector state machine for a concurrent service

7.3 Hierarchical Coordination Adaptation Pattern

This section describes the hierarchical coordination adaptation pattern for service-oriented architectures. In the hierarchical coordination, a high-level coordinator orchestrates lower-level coordinators, whereas each of the lower-level coordinators is responsible for more detailed SOA coordination. The communication diagram depicted in Figure 11 shows a general hierarchical coordination pattern where a higher-level parent coordinator coordinates M lower-level child coordinators, each of which interacts with multiple services. The assumptions are as follows:

- A parent coordinator is instantiated for each client.
- One or more child coordinators are instantiated for each parent coordinator.

- A client interacts with a coordinator using synchronous communication; thus, it sends a new request only when it receives a response to its previous request.

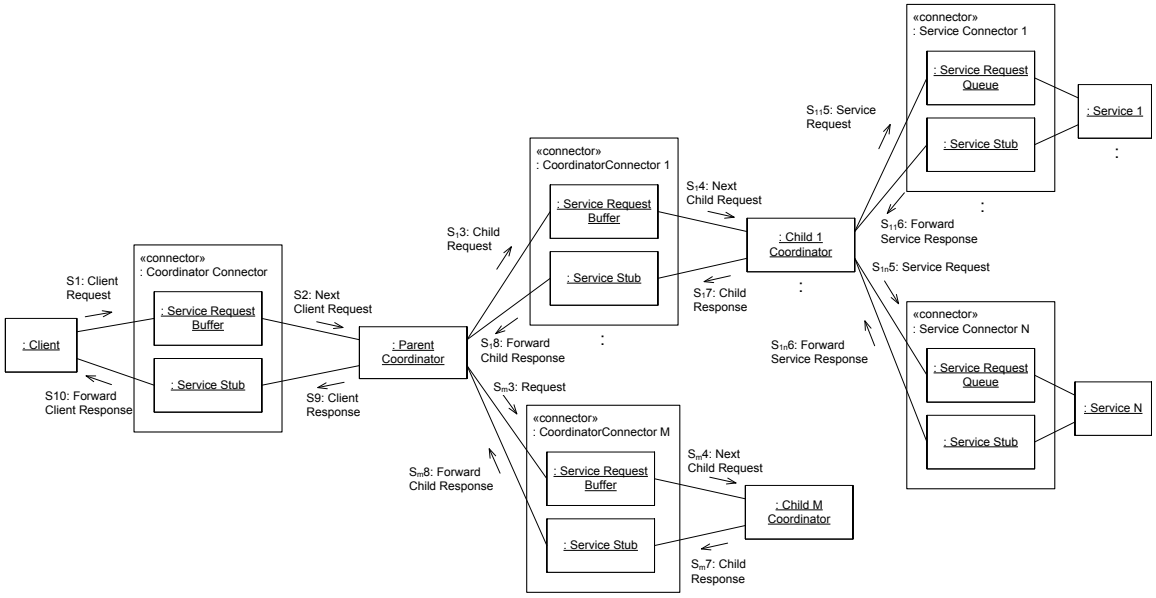


Figure 11 Hierarchical coordination communication diagram

Based on the assumptions mentioned above, the parent coordinator component can be removed or replaced after it has received all the responses from the child coordinators sequentially or concurrently invoked and sent its response to the client. A child coordinator can be removed or replaced after it has received responses from all the services invoked and sent it response to the parent coordinator. On the other hand, a service can be removed or replaced after it completes the current service execution in the case of a sequential service, or after completing the current set of service executions in the case of a concurrent service.

Figure 12 and Figure 13 depict the operating state machines executed by the coordinator connectors for the parent coordinator and the child coordinator, respectively. Except for the labels for events and actions, these state machines are exactly the same as Figure 4, because a coordinator connector determines when a coordinator is in Quiescent state by monitoring only the final response from the coordinator.

For the same reasons as described in Section 7.1.3, the coordinator connectors can be reused in the hierarchical coordination pattern, regardless of the coordinator's degree of concurrency (sequential, concurrent, or combined sequential/concurrent). Moreover, for the same reasons, the coordinator connectors are reusable for multi-level hierarchical coordination that has intermediate coordinators between the parent and child coordinators.

Since services are coordinated by a child coordinator in the form of the independent coordination pattern, the independent coordination adaptation patterns described in Section 7.2 can also be applied to the adaptation for a service in the hierarchical coordination pattern. If a child coordinator orchestrates stateful services by following the Two-Phase Commit Protocol, the two-phase commit coordination adaptation pattern described in Section 7.3 can be applied.

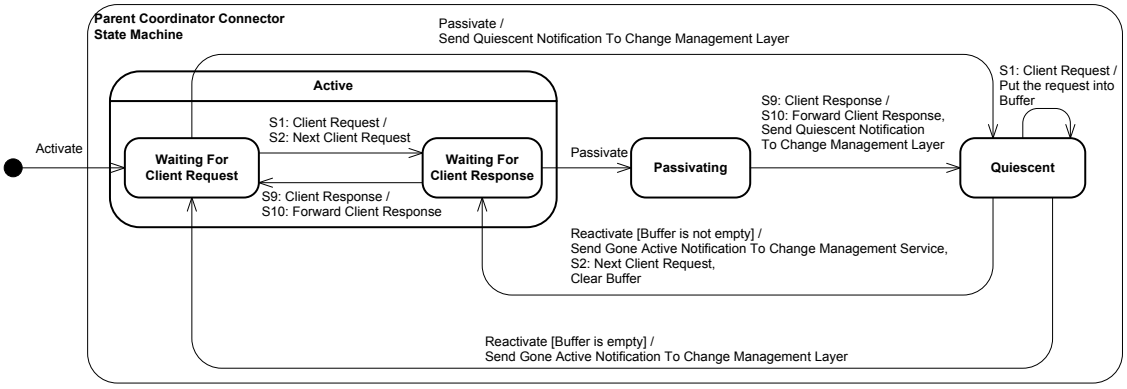


Figure 12 Parent Coordinator connector state machine

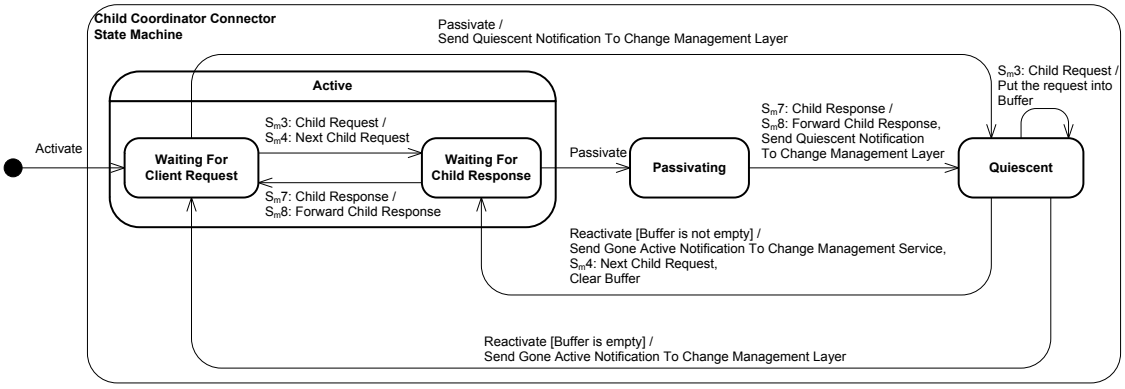


Figure 13 Child Coordinator connector state machine

7.4 Distributed Coordination Adaptation Pattern

In the distributed coordination pattern, an SOA application consists of multiple coordinators that are distributed and cooperate with each other. Figure 14 depicts a communication diagram that shows a general case where Coordinator M asynchronously sends a service request message "A1: Request" to other coordinator L whereas Coordinator M asynchronously receives another service request message "B2: Request"

from another coordinator N (Coordinators L and N could be identical). This section makes the following assumptions:

- A coordinator can send a service request asynchronously to any other coordinator, i.e., the coordinator can send, receive, and process other requests while it is waiting for the response for an outstanding request it has sent to the other coordinator.
- Service requests a certain coordinator receives are independent of each other.

Based on the assumptions described above, a coordinator component can only be removed or replaced after

1. the coordinator has received all the responses for the requests it has sent to other coordinators, and
2. the coordinator has processed all the requests, and has sent out the corresponding responses to the requesters.

In other words, a coordinator can be removed or replaced only after completing the transactions it participates in. A transaction that a coordinator participates in is defined as either 1) the interval between a service request it sends and the corresponding response it receives, or 2) the interval between a service request it receives and the corresponding response it sends out. For example, Coordinator M in Figure 14 initiates a transaction by sending a service request message "A1: Request" to another Coordinator L. The transaction completes when Coordinator M receives a response from Coordinator L. Coordinator M participates in another transaction by receiving a service request message

from another Coordinator N. The transaction completes when Coordinator M sends a response to Coordinator N. Note that, based on these assumptions, a coordinator can initiate multiple transactions asynchronously and a coordinator can receive multiple requests asynchronously.

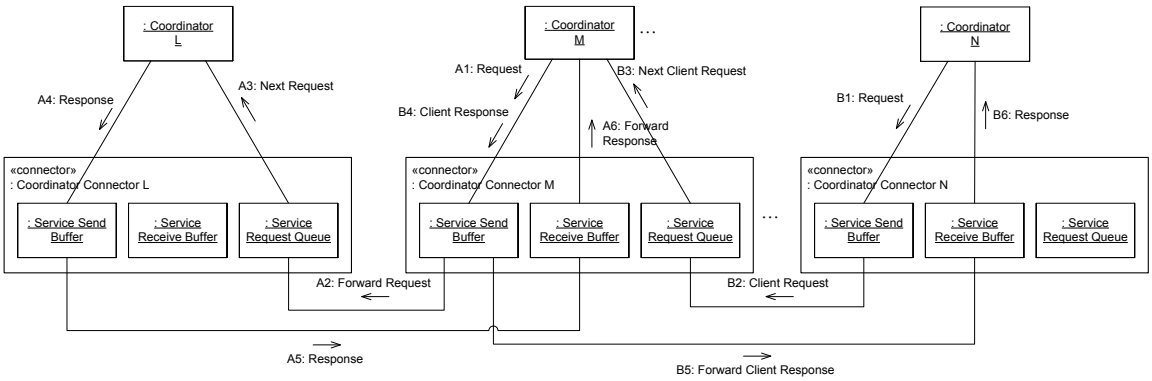


Figure 14 Distributed coordination communication diagram

As shown in Figure 14, a coordinator connector is provided for each coordinator. Unlike the adaptation patterns described in previous chapters, a coordinator connector in the distributed coordination adaptation pattern captures and forwards all incoming and outgoing messages for its corresponding coordinator so that the coordinator connector can manage both 1) transactions the corresponding coordinator initiates and 2) transactions the coordinator participates in by receiving service requests from other coordinators.

The coordinator connector executes the operating state machine depicted in Figure 15. The state machine manages the number of transactions the coordinator currently executes by the variable t. When the connector is in the "Idle" state and receives a "A1: Request" message from its corresponding coordinator, it forwards the request to its destination coordinator and transitions to the "Processing" state, which means that the coordinator has initiated a new transaction and is waiting for a response. On the other hand, when the connector is in the "Idle" state and receives a "B1: Request" message from another coordinator, it forwards the request to its corresponding coordinator and transitions to the "Processing" state, which means, in this case, that the coordinator has participated in a new transaction and is processing the request. While in the "Processing" state, if the coordinator receives more "A1: Request" or "B1: Request" message, it forwards the request accordingly and increments the variable t.

When the connector is in the "Processing" state and receives a Passivate command from the change management service, it transitions to the "Passivating" state in which the corresponding coordinator is still waiting for responses and/or processing requests to accomplish transactions. In the "Passivating" state, the connector rejects any service request message "A1: Request" sent from its corresponding coordinator because the coordinator will be removed or replaced and should not initiate any new transaction. On the other hand, the connector could receive a new "B1: Request" message from other coordinator, which it queues the request on the service request queue.

Once the coordinator accomplishes all the transactions, the connector will transition from the "Passivating" to the "Quiescent" state. The connector captures "A5: Response" and "B4: Response" messages and manages the number of transactions with the variable t to decide when the coordinator accomplishes all the transactions, as shown in Figure 15.

While in the quiescent state, the connector rejects a new request from the corresponding coordinator, whereas the connector queues a new request from other coordinator on the service request queue.

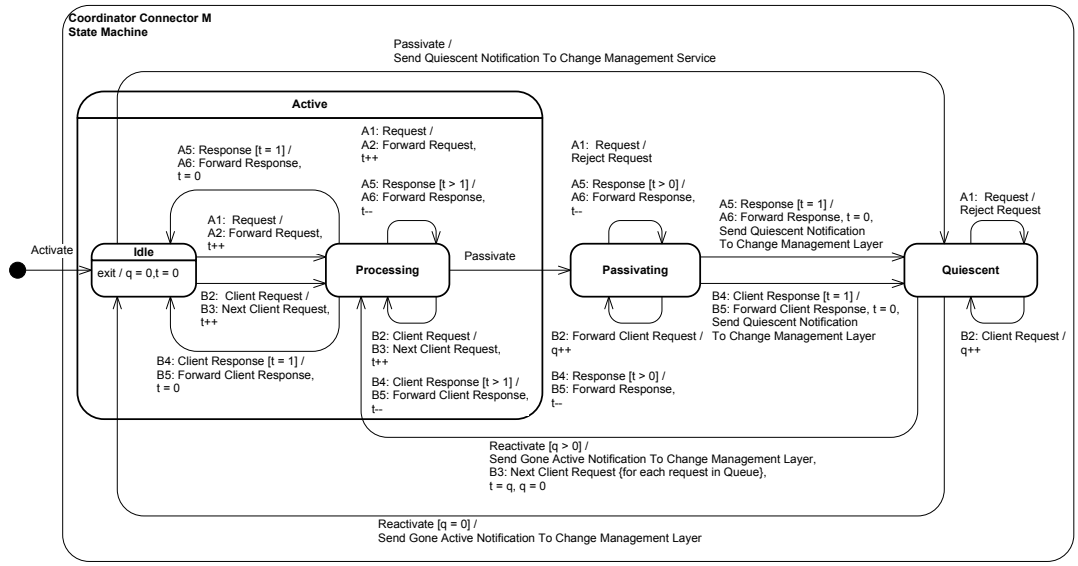


Figure 15 Coordinator connector state machine

CHAPTER 8 SERVICE FAILURE ADAPTATION PATTERN

Software adaptation is triggered not only by planned software configuration change but also by unexpected hardware/software failure in an SOA application. As a preliminary research effort, this thesis considers the failure of a stateless service.

In a typical SOA computing environment, services are often provided by third-party service providers whereas coordinators and connectors are developed in house. Therefore, this chapter makes the following assumptions in addition to those mentioned in Section 7.1:

- Services are stateless.
- A service can go down due to the failure of its provider server.
- The failure of a service is detected and notified by Monitoring service.
- Coordinators and adaptation connectors (coordinator and service connectors) are reliable.

Since this thesis assumes only a service can fail, this chapter describes only the operating state machine executed by a service connector in Figure 16, which is developed based on the one depicted in Figure 6, which is for a concurrent service. In this case, the service request queue has an important role to buffer requests in a sequence, not only because a

service may have multiple clients but also because the service may go down during its processing of the requests. As shown in Figure 16, the connector not only forwards requests it receives but also queues the requests on the request queue, even in the "Waiting For Service Request" state or the "Processing" state. A request in the queue is removed when the connector receives its response from the service. Consequently, the connector manages the number of requests being processed by the service by monitoring the number of requests in the request queue.

When the failure of the corresponding service is detected, the connector transitions to the "Failed" state, regardless of its current state ("Waiting For Service Request", "Processing", or "Passive"). At this moment, the requests kept in the queue have not been completed by the failed service. Therefore, once an alternative service is discovered, the connector forwards the requests in the queue to the new service, and transitions to the "Processing" state.

If a new service request arrives in the "Failed" state, the connector queues the request on the request queue. In other words, clients need not consider either the service failure or the dynamic adaptation because the connector will queue service requests even in the case of service failure.

Note that the service connector can be reused in the independent coordination pattern regardless of the coordinator's degree of concurrency (sequential, concurrent, or combined sequential/concurrent) for the same reasons as described in Section 7.1.3.

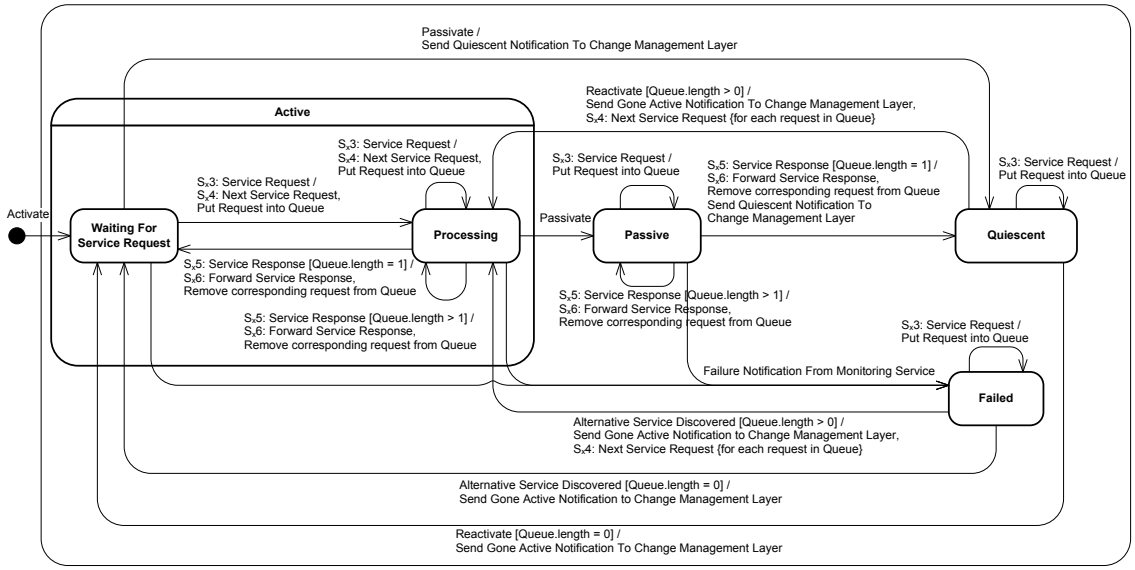


Figure 16 Service connector state machine for a concurrent service

CHAPTER 9 IMPLEMENTATION OF ADAPTATION CONNECTOR

In Chapters 7 and 8, several adaptation state machines executed by adaptation connectors, i.e. coordinator connectors and service connectors, are described according to the coordination patterns and the consideration of service failure. If each of those state machines would be implemented as an individual adaptation connector, it would need to be carefully chosen, instantiated and deployed on an SOA runtime system according to the coordination pattern and the type of service.

Furthermore, as mentioned in Section 6.2, an adaptation connector behaves as a proxy for a service/coordinator. In other words, an adaptation connector must provide the same interface as its corresponding service or coordinator (a list of operations with the signature for each operation). In the case of Web Service, the interface of a service is defined and advertised in the form of WSDL [29]. Therefore, it is possible to develop a generator which automatically generates an adaptation connector from WSDL, although users still need to carefully instantiate and deploy connectors generated according to services.

In order for an adaptation connector to be practically usable, therefore, it must fulfill two key requirements:

1. Be usable for any service, regardless of the interface that the service provides.
2. Be usable for any adaptation pattern, regardless of the type of coordination, the degree of concurrency, and the type of service described in Chapter 5.

The following chapters discuss solutions to satisfy the two requirements.

9.1 General Interface of Adaptation Connectors

As mentioned in the previous section, an adaptation connector must provide the same interface as its corresponding service or coordinator because it behaves as a proxy for the service or coordinator. However, note that an adaptation connector just forwards incoming and outgoing messages while managing the adaptation states by monitoring them for their timing and direction. The signature of operations, therefore, does not matter for the behavior of the adaptation state machines described in this thesis.

In this thesis, as described in Chapter 11, adaptation connectors are implemented using the SOAP with Attachments API for Java (SAAJ) API [30][31], which allows a service to accept any SOAP envelope, i.e., any request message. When a request message arrives, the adaptation connector can handle the corresponding SOAP envelope as a Java object, execute the adaptation state machine, and forward it to the designated service.

By implementing an adaptation connector as shown above, the first adaptation connector implementation requirement can be satisfied, i.e., the adaptation connector can be usable for any service regardless of its interface.

9.2 Integration of Adaptation State Machines

Different adaptation state machines are developed according to the coordination patterns and consideration of service failure as described in Chapters 7 and 8. However, it is noteworthy that all the state machines are similar to each other. As a matter of fact, those state machines can be integrated as depicted in Figure 17, which satisfies the second adaptation connector implementation requirement, i.e., the integrated adaptation connector is usable for both coordinators and services, regardless of the type of coordination, the degree of concurrency, and the type of service. Table 1 to 7 show the message mapping between coordinator/service connector state machines in the adaptation patterns and the integrated adaptation state machine in Figure 17.

First, a service connector can also be used for a coordinator. Even if a coordinator is instantiated for each client which interacts with the coordinator using synchronous communication, the Service Request Queue in the service connector can keep a new request from the client in the Quiescent state. For example, Table 1 shows the message mapping between the coordinator connector state machine (Figure 4) in the independent coordination adaptation patterns and the integrated adaptation state machine in Figure 17.

By mapping messages according to this table, the integrated state machine can execute exactly the same state transitions as the coordinator connector state machine in Figure 4. That is, the integrated connector can also be used as the coordinator connector. The same argument applies to parent and child coordinators in the Hierarchical Coordination Adaptation pattern described in Section 7.4. Table 5 and 6 show the message mapping for parent and child coordinator state machines in Figure 12 and 13, respectively.

Second, a stateless service can be considered as a special case of a stateful service. In the case of a stateless service, there are two kinds of messages from the perspective of adaptation connectors, 1) a request message from a client and 2) a response from the corresponding service.

The adaptation connector transitions from Active to Passive to Quiescent states based on the fact that 1) a request message initiates a new transaction the service participates in, and 2) its response finishes the transaction. For example, in the independent coordination adaptation patterns, messages received by a service connector in Figure 3 are categorized as follows:

- "S_x3: Service Request": 1) a request message from a client which initiates a new transaction
- "S_x5: Service Response": 2) a response from the service which finishes the transaction

On the other hand, in the case of a stateful service, there are three kinds of messages, 1) a request message from a client which initiates a new transaction, 2) intermediate messages

and responses, and 3) a final response from the service which finishes the transaction. For example, in the two-phase commit coordination adaptation pattern, messages received by a service connector in Figure 7 are categorized as follows:

- "T_x3: PrepareToCommit": 1) a request message from a client which initiates a new transaction
- "T_x6: ReadyToCommit": 2) an intermediate response
- "T_x8: Commit": 2) an intermediate message
- "T_x11: CommitCompleted": 3) a final response from the service which finishes the transaction

Thus, a stateless service is a special case of a stateful service where there is no exchange of intermediate messages and responses. As a result, the adaptation state machines for stateless services can be integrated into the state machine depicted in Figure 17. Table 4 show the message mapping for the stateless and stateful (two-phase commit) service connector state machines in Figure 6 and 10, respectively. The implementation issue here is that the integrated adaptation connector must distinguish the different kinds of messages mentioned above. However, the adaptation connector should not depend on the signature of operations provided by services due to the second adaptation connector implementation requirement. The implementation for validation described in Chapter 11, therefore, utilizes a SOAP header attached to a message indicating which kind of message it is, as follows:

1) SOAP header element "beginTransaction" in a request message indicates that the message initiates a new transaction. Its response is to be intermediate.

2) SOAP header element “intermediateTransaction” in a request message indicates that the message is intermediate. Its response is also to be intermediate.
3) SOAP header element “endTransaction” in a request message indicates that its response finishes the transaction.
4) If no SOAP header elements are attached to a request message, the message is considered to be for a stateless service.

For example, in the independent coordination adaptation patterns, no SOAP headers are attached to messages because services are stateless. On the other hand, in the two-phase commit coordination adaptation pattern, SOAP headers are attached to messages received by a service connector in Figure 3 as follows:

- “T_x3: PrepareToCommit”: 1) SOAP header element “beginTransaction” is attached.
- “T_x8: Commit”: 3) SOAP header element “endTransaction” is attached.

(Note that the two-phase commit coordination adaptation pattern has no messages that SOAP header element “intermediateTransaction” is attached to. A protocol with more than two phases could have such messages.)

Third, by providing one more interface only for request messages outgoing from the corresponding coordinator, the integrated adaptation connector can also support the state machine depicted in Figure 15 for the Distributed Coordination Adaptation pattern. As discussed in Section 7.4, the coordinator connector captures request messages not only incoming from other coordinator but also outgoing from the corresponding coordinator.

By adding another interface to the adaptation connector for such outgoing messages, it can distinguish between incoming messages and outgoing messages. Other adaptation patterns simply do not use the interface. Table 7 shows the message mapping between the coordinator connector state machine (Figure 15) in the distributed coordination adaptation pattern and the integrated adaptation state machine in Figure 17.

Finally, this thesis assumes coordinators are reliable, i.e., only services can fail. Therefore, the integrated adaptation connector executing the state machine in Figure 17 can be used for coordinators. (The adaptation connector assigned for a coordinator never transitions to the "Failed" state.) Table 8 shows the message mapping between the coordinator connector state machine (Figure 16) in the service failure adaptation pattern and the integrated adaptation state machine in Figure 17. Note that support for the failure of stateful services is a future work.

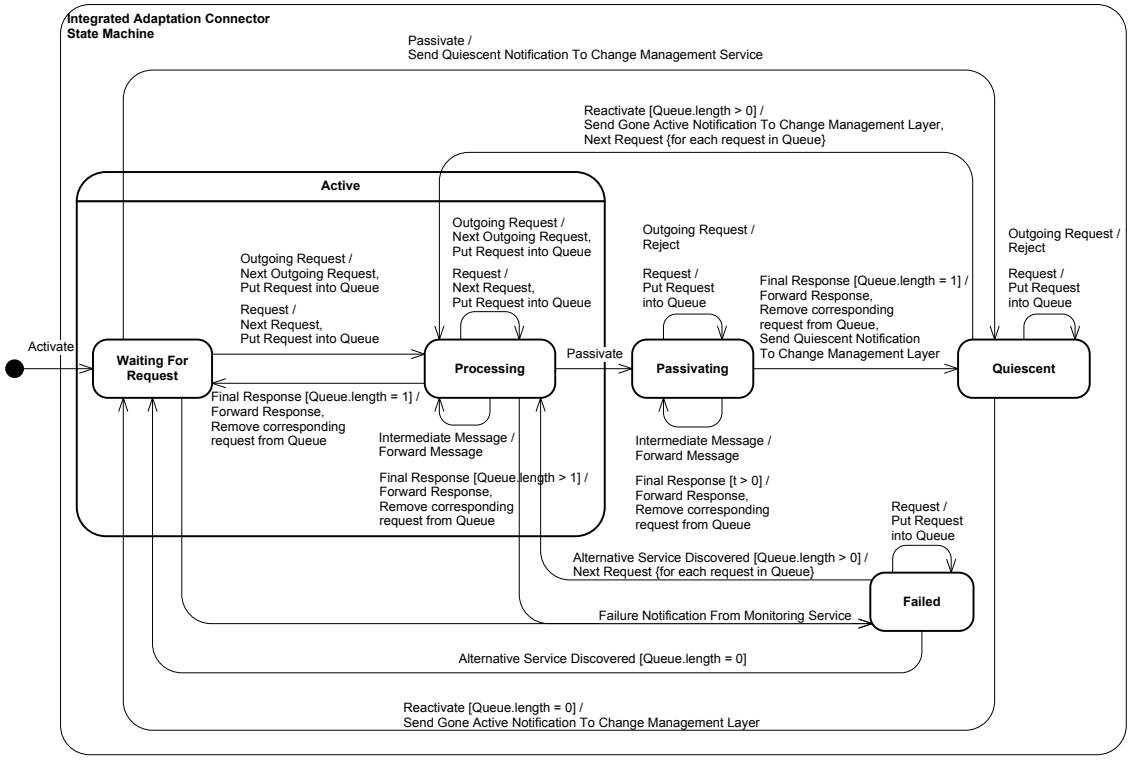


Figure 17 Integrated adaptation state machine

Table 1 Message correspondence for Coordinator connector state machine in Figure 4

Coordinator Connector	Integrated Adaptation Connector
S1: Client Request	Request
S2: Next Client Request	Next Request
S7: Client Response	Final Response
S8: Forward Client Response	Forward Response

Table 2 Message correspondence for Service connector state machine in Figure 6

Service Connector	Integrated Adaptation Connector
S_x3: Service Request	Request
S_x4: Next Service Request	Next Request
S_x5: Service Response	Final Response
S_x6: Forward Service Response	Forward Response

Table 3 Message correspondence for Coordinator connector state machine in Figure 9

Coordinator Connector	Integrated Adaptation Connector
T1: Client Request	Request
T2: Next Service Request	Next Request
T13, T5A.8: Client Response	Final Response
T14, T5A.9: Forward Client Response	Forward Response

Table 4 Message correspondence for Service connector state machine in Figure 10

Service Connector	Integrated Adaptation Connector
T_x3: PrepareToCommit	Request
T_x4: Fwd PrepareToCommit	Next Request
T_x6: ReadyToCommit	Intermediate Message
T_x7: Fwd ReadyToCommit	Forward Message
T_x8: Commit	Intermediate Message
T_x9: Fwd Commit	Forward Message
T_x11: CommitCompleted	Final Response
T_x12: Fwd CommitCompleted	Forward Response
T_x5A.1: RefuseToCommit	Intermediate Message
T_x5A.2: Fwd RefuseToCommit	Forward Message
T_x5A.3: Abort	Intermediate Message
T_x5A.4: Fwd Abort	Forward Message
T_x5A.6: Aborted	Final Response
T_x5A.7: Fwd Aborted	Forward Response

Table 5 Message correspondence for Parent coordinator connector state machine in Figure 12

Coordinator Connector	Integrated Adaptation Connector
S1: Client Request	Request
S2: Next Client Request	Next Request
S9: Client Response	Final Response
S10: Forward Client Response	Forward Response

Table 6 Message correspondence for Child coordinator connector state machine in Figure 13

Coordinator Connector	Integrated Adaptation Connector
S_m3: Child Request	Request
S_m4: Next Child Request	Next Request
S_m7: Child Response	Final Response
S_m8: Forward Child Response	Forward Response

Table 7 Message correspondence for Coordinator connector state machine in Figure 15

Coordinator Connector	Integrated Adaptation Connector
A1: Request	Outgoing Request
A2: Forward Request	Next Outgoing Request
A5: Response	Final Response
A6: Forward Response	Forward Response
B2: Client Request	Request
B3: Next Client Request	Next Request
B4: Client Response	Final Response
B5: Forward Client Response	Forward Response

Table 8 Message correspondence for Service connector state machine in Figure 16

Service Connector	Integrated Adaptation Connector
S_x3: Service Request	Request
S_x4: Next Service Request	Next Request
S_x5: Service Response	Final Response
S_x6: Forward Service Response	Forward Response

CHAPTER 10 ADAPTIVE CHANGE MANAGEMENT

Adaptive change management is provided by a Change Management Model, which is used to establish a region of quiescence [9] so that dynamic adaptation can take place. For each adaptation pattern, the change management model describes a process for controlling and sequencing the steps in which the configuration of components in the pattern is changed from the old configuration to the new configuration. Thus, as stated before, the middle layer of the three-layer model, i.e., change management layer, is responsible in our approach for implementing the Change Management Model, and controlling the adaptation process through adaptation commands. The adaptation commands describe reconfiguration actions associated with user-required changes, which are predefined as reconfiguration scenarios. The adaptation commands for SOA applications are *passivate*, *unlink*, *remove*, *create*, *link*, *activate*, and *reactivate*, as described in more detail below with the aid of an example. Note that remove and create commands are not required for the adaptation of services provided by third parties.

10.1 Example of Dynamic Software Adaptation

As an example of dynamic software adaptation, consider an emergency response system shown in Figure 18. The initial software configuration is shown in Figure 18a before

dynamic software adaptation, while the revised configuration after dynamic software adaptation is shown in Figure 18b.The emergency system uses the independent, sequential coordination pattern for coordination of the three services, Building Locator, Occupancy Awareness, and Fire Station. In the example, Occupancy Awareness is to be replaced by a more reliable service composition as depicted in Figure 18b.

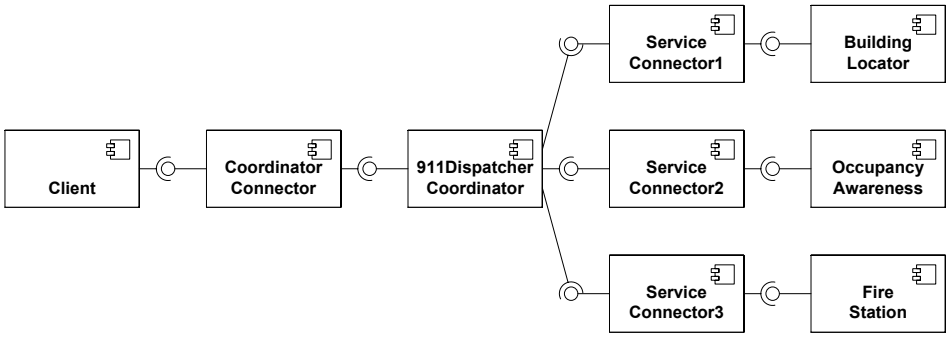
The adaptation is triggered by the availability of Occupancy Awareness operating below 99.999%, which is specified as a QoS requirement. The Goal Management layer determines a possible adaptation, which involves two potential Occupancy Awareness services that are 99.0% available and could be mediated by a Fault Tolerant connector [23] (Figure 18b). The Change Management layer then decides that the change involves adding a second Occupancy Awareness service and the Fault Tolerant connector, which invokes the two Occupancy Awareness services but forwards only the response of the primary service back to the requester.

The Change Management (CM) layer controls and coordinates the dynamic adaptation, and communicates this to the service connectors in the software configuration by sending adaptation commands as follows:

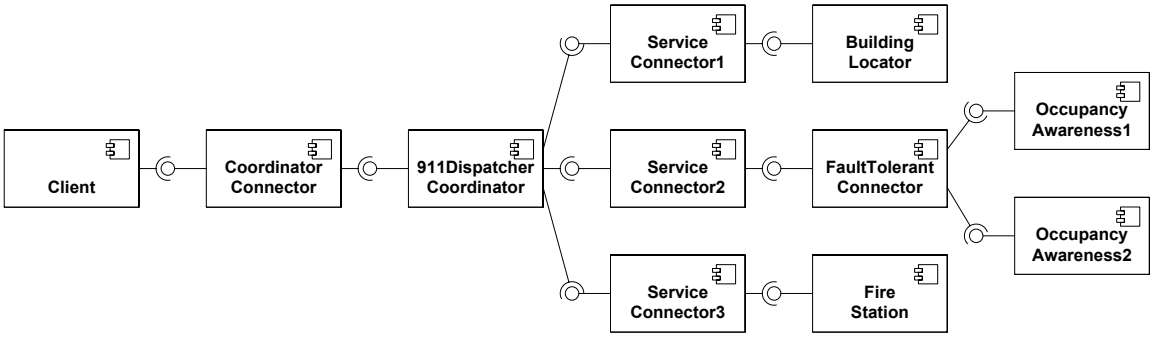
1. CM sends a passivate command to Service Connector 2 for Occupancy Awareness, so that the connector transitions to the quiescent state.
2. Upon transitioning to quiescent state, Service Connector 2 sends the quiescent notification to CM. CM then sends an unlink command to Service Connector 2.

As a result, the interconnection between Service Connector 2 and Occupancy Awareness is unlinked.

3. CM sends Link commands to connect Service Connector 2 with a new service composition, which consists of the Fault Tolerant connector, Occupancy Awareness 1, and Occupancy Awareness 2 (as shown in Figure 18b). In this case, the Fault Tolerant connector has the responsibility to connect the two service instances. Service Connector 2 and the Fault Tolerant connector are linked as the Fault Tolerant connector provides a proxy interface for the Occupancy Awareness service.
4. CM sends a reactivate command to Service Connector 2, which responds with a Gone Active Notification and resumes forwarding service requests.



a) Initial software configuration



a) Revised software configuration

Figure 18 Dynamic software adaptation in emergency response system

CHAPTER 11 VALIDATION OF SOA ADAPTATION PATTERNS

The SOA adaptation patterns described in Chapters 7 and 8 were validated through simulation and prototype implementation using the emergency response system example described in Chapter 10 and other examples. Note that in the validation by prototype implementation, the integrated adaptation state machine in Figure 17 was implemented and used as an adaptation connector for both coordinators and services for all the adaptation patterns.

The validation consists of

1) executing the change management scenario,
2) performing the software adaptation from one configuration to another, and
3) resuming the application after the adaptation.

11.1 Validation by Simulation

The emergency response system example described in Chapter 10 was modeled using XTEAM [24], which is an architectural modeling and analysis environment. XTEAM provides a structural Architectural Description Language (ADL), xADL [25], with a

behavioral ADL, Finite State Processes (FSP) [26], to generate executable system simulations.

In the emergency response system example, the validation used XTEAM to model the system's structure in xADL as shown in Figure 19, and the behavior of Coordinator and Service Connectors by translating their state machines described in Figure 4-6 into equivalent FSP models. In the simulation, as shown in Figure 19, the Change Management layer was modeled as a component in xADL, which sends the adaptation commands to the adaptation connectors by executing the change management scenario described in Chapter 10. Since XTEAM does not currently support run-time dynamic adaptation of the software architecture and executable simulation, the example was simulated using behavioral adaptation (Chapter 3). Therefore, the Fault Tolerant connector connecting the two Occupancy Awareness services was developed and connected in advance with Service Connector 2 as shown in Figure 19, so that XTEAM simulates execution of the unlink and link commands for the case of the Occupancy Awareness service replacement.

The system's simulation in XTEAM, generated from the xADL and FSP models, was executed with the adaptation state machines described in Section 7.1 and the dynamic software adaptation scenario described in Chapter 10. This scenario involves the service connector state machine transitioning from Active to Passive to Quiescent states, replacing one service with another, and then reactivating the service connector. A second

scenario was executed in which the coordinator was replaced. This scenario involves the coordinator connector state machine transitioning from Active to Passive to Quiescent states, replacing the coordinator, and then reactivating the coordinator connector.

For both the above two scenarios, the XTEAM simulation recorded the trace of FSP state transitions for each xADL component in execution logs (see Appendices A and B). Analysis of these logs showed that the three validation steps described in Chapter 11 were carried out as planned. Thus the validation demonstrates that the software adaptation patterns and state machines described in this paper perform the desired software adaptation while ensuring that the service-oriented application does not enter an inconsistent state.

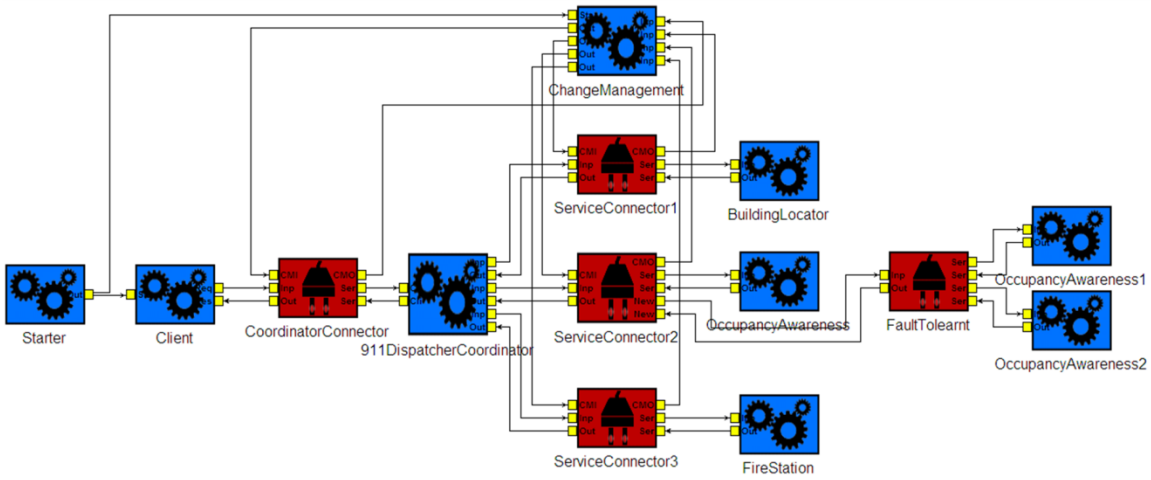


Figure 19 xADL model of emergency response system for simulation

11.2 Validation by Implementation

Since XTEAM does not support run-time dynamic change of software architecture and executable simulation, the initial and revised software configurations in xADL and FSP models are required to be implemented and deployed in advance, as explained in the previous chapter. In addition to the XTEAM simulation, therefore, all the SOA adaptation patterns described in Chapters 7 and 8 were implemented as a part of the prototype of SASSY framework described in Section 4.1, using open-source SOA frameworks, Eclipse Swordfish and Apache CXF.

Eclipse Swordfish [32] is an open-source, extensible ESB (Enterprise Service Bus), built upon Apache ServiceMix [34]. The prototype of SASSY framework described in Section 4.1 is being developed on top of Swordfish which is based on OSGi [35] so that a component newly generated by SASSY framework can be integrated into Swordfish at runtime. In the SASSY framework prototype, the goal management and change management layers are implemented and integrated into Swordfish framework as service components, i.e., Self-Architecting and Reachitecting component, Gauge Service, and Change Management Service. In addition, a coordinator generated as a part of a system service architecture is deployed on Swordfish at runtime as a component. In the emergency response system example, the 911 Dispatcher Coordinator is deployed on Swordfish.

Apache CXF [33] is an open-source web-services framework which supports standard APIs such as JAX-WS and JAX-RS as well as WS standards including SOAP, WSDL, WS-Addressing, WS-Policy, etc. In the emergency response system example, the Building Locator, Occupancy Awareness, and Fire Station services are implemented on top of Apache CXF as if they are external services.

Unlike other SASSY services, the adaptation connector was implemented as an independent web service on top of Apache CXF instead of Swordfish, because Swordfish, as an ESB, doesn't allow a component to dynamically change the service (provider) it invokes. In other words, although Swordfish may change the service for the component based on its service discovery, the component cannot change the service to invoke by itself at runtime. The adaptation connector was implemented using SAAJ API, a low-level Web-Service API so that it is reusable for any service as discussed in Chapter 9.

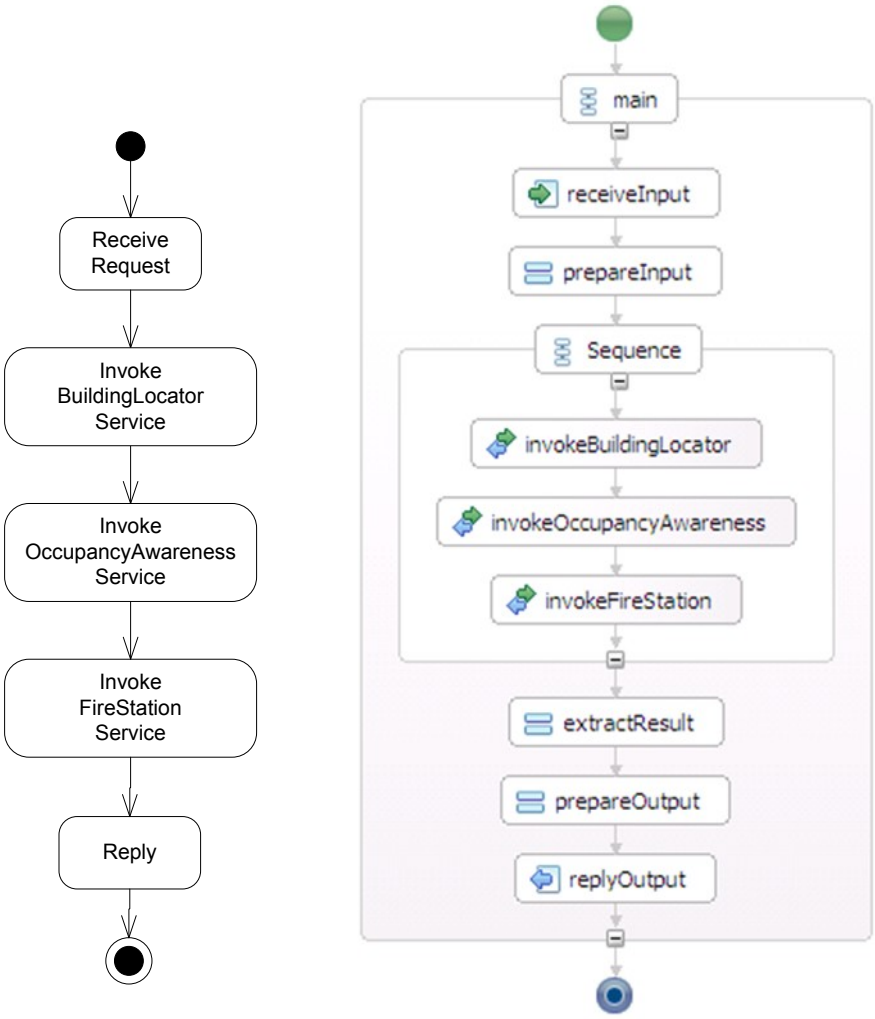
11.2.1 Validation of Independent Coordination Adaptation Patterns

In this chapter, the following three independent coordination adaptation patterns are validated using the emergency response system example:

1. Sequential coordination adaptation pattern (Section 7.1.1)
2. Concurrent coordination adaptation pattern (Section 7.1.2)
3. Combined sequential / concurrent coordination adaptation pattern (Section 7.1.3)

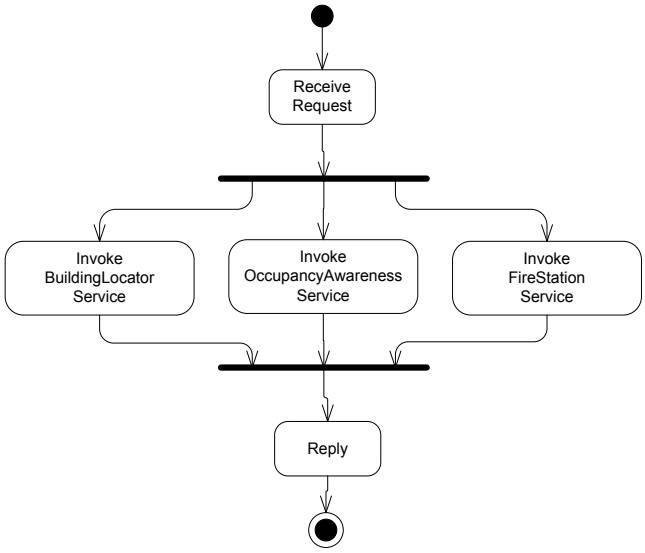
Since the purpose of the implementation is to validate these adaptation patterns, the 911 Dispatcher Coordinator was manually implemented as a BPEL processe and deployed on Apache ODE, an open source BPEL engine. Figure 20, 22, and 23 depict the coordination logics for the 911 Dispatcher Coordinators which perform the above three coordination patterns, respectively. In these figures, (a) shows the activity model of the coordination logic whereas (b) shows the corresponding BPEL process as a screenshot of Eclipse BPEL Designer [36].

Figure 20 shows sequential coordination in the BPEL process which invokes Building Locator, Occupancy Awareness, and Fire Station services sequentially. On the other hand, Figure 21 depicts a BPEL process for current coordination in which the elements of invocation for the three services are put in a BPEL flow element. Figure 22 shows an example of combined sequential and concurrent coordination, in which Occupancy Awareness service and Fire Station service are invoked concurrently after Building Locator is invoked and responded.

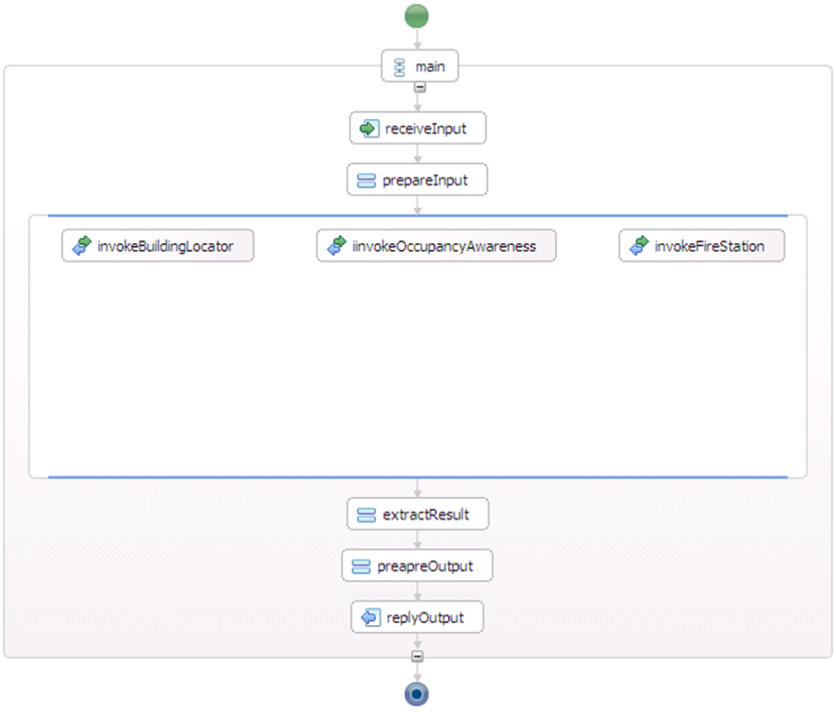


(a) Activity model (b) BPEL

Figure 20 Sequential coordination BPEL process

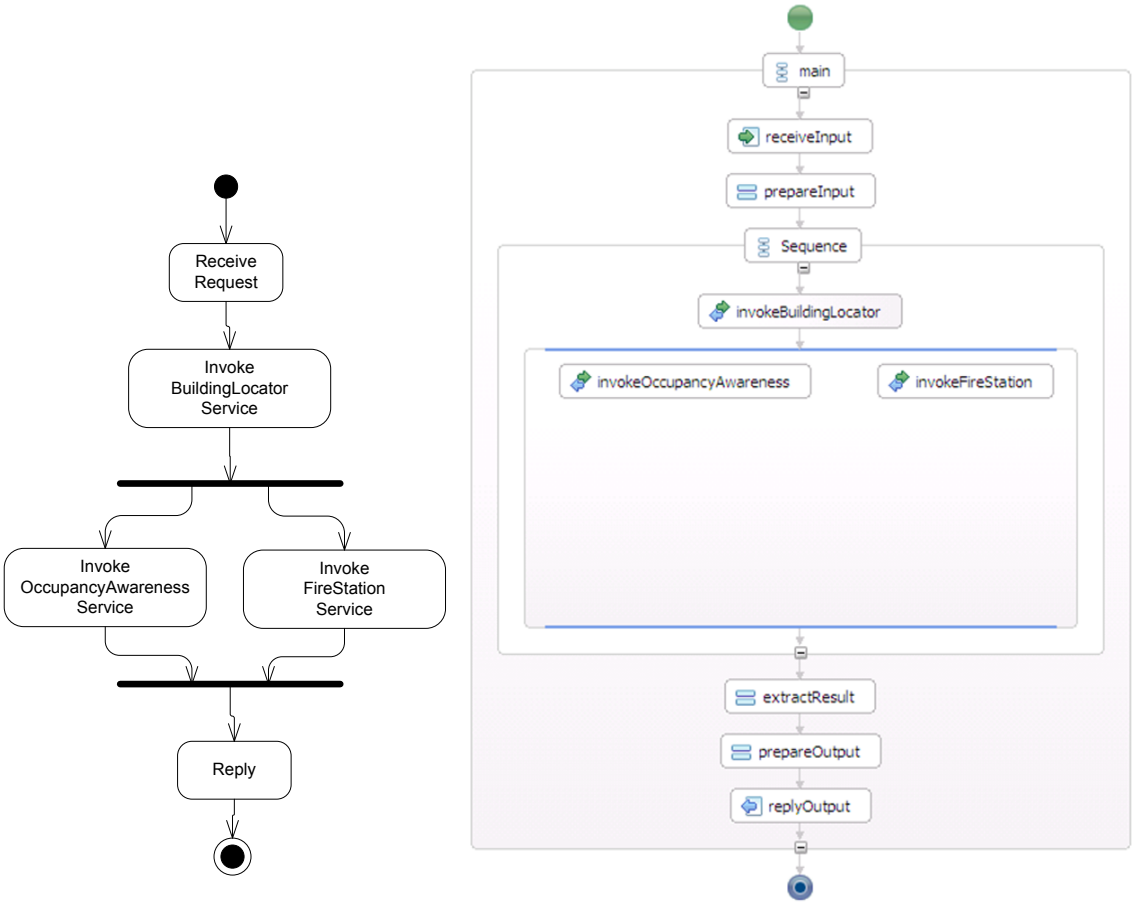


(a) Activity Model



(b) BPEL

Figure 21 Concurrent coordination BPEL process



(a) Activity Model (b) BPEL

Figure 22 Combined sequential / concurrent coordination BPEL process

Since Apache CXF runtime runs a web service as a concurrent service by default (a thread is assigned to a request to be processed), this validation considers only the case of the service connector state machine for a concurrent service depicted in Figure 6.

This implementation executed the adaptation state machines described in Figure 6 and the dynamic software adaptation scenario described in Chapter 10. As the simulation described in the previous chapter, this scenario involves the service connector state machine transitioning from Active to Passive to Quiescent states, replacing one service with another, and then reactivating the service connector. A second scenario was executed in which the coordinator was replaced.

For each of the three independent coordination implementations, the result of the execution showed that the three validation steps described in Chapter 11 were carried out as planned. For example, Figure 23 shows the execution log of Service Connector 2 in the emergency response system, in the first scenario where one service was replace with another. Figure 24 shows the execution log of Coordinator Connector in the second scenario where the coordinator was replaced. Thus the validation demonstrates that the independent coordination adaptation patterns and state machines described in Section 7.1 perform the desired software adaptation while ensuring that the service-oriented application does not enter an inconsistent state.

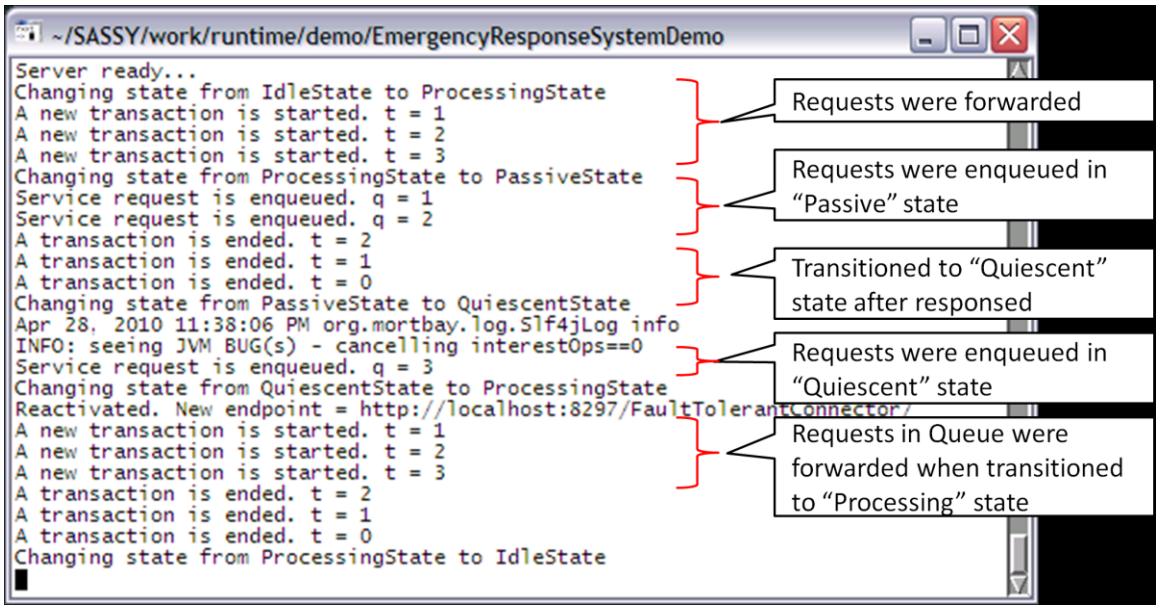


Figure 23 Execution log of Service Connector 2 in emergency response system

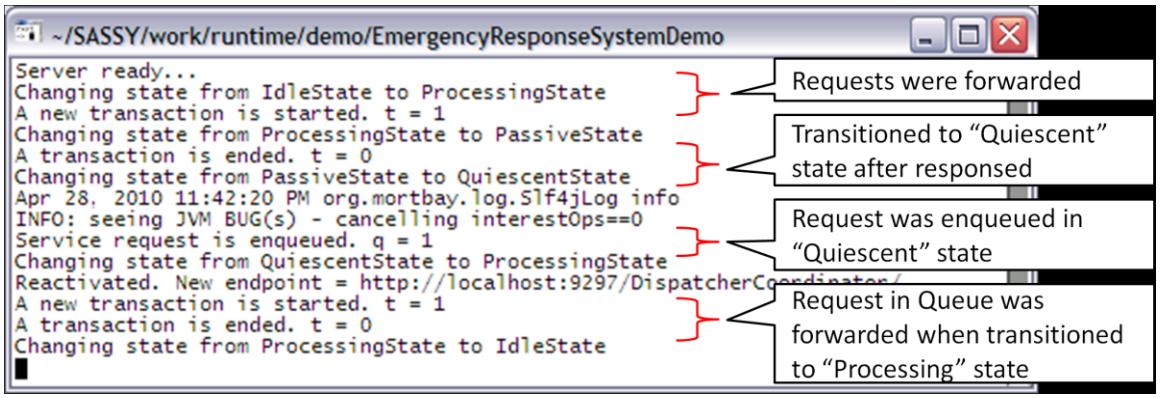


Figure 24 Execution log of Coordinator Connector in emergency response system

11.2.2 Validation of Two-phase Commit Coordination Adaptation Pattern

This section validates the two-phase commit coordination adaptation pattern described in Section 7.2. A general service following the Two-phase Commit Protocol was

implemented in such a way that the service randomly responses either "ReadyToCommit" or "RefuseToCommit" for a "PrepareToCommit" request message. Figure 25 depicts the activity model of an independent coordinator implemented as a BPEL process shown in Figure 26, which coordinates three of the above stateful service instantiated. As shown in the figures, the three services are concurrently invoked in each phase.

This implementation executed the adaptation state machines in Figure 9 and Figure 10. The executions traces of the service connectors are almost the same as Figure 23 except for the fact that they did not transition to Passive or Quiescent state even when they received intermediate messages. On the other hand, the execution log of the coordinator connector is the same as Figure 24. Thus, the result of execution showed that the three validation steps described in Chapter 11 were performed as planned. Particularly for an adaptation of a service replacement, when a passivate command was sent, the adaptation connector didn't transition to the Quiescent state until it received "CommitCompleted" or "Aborted" message.

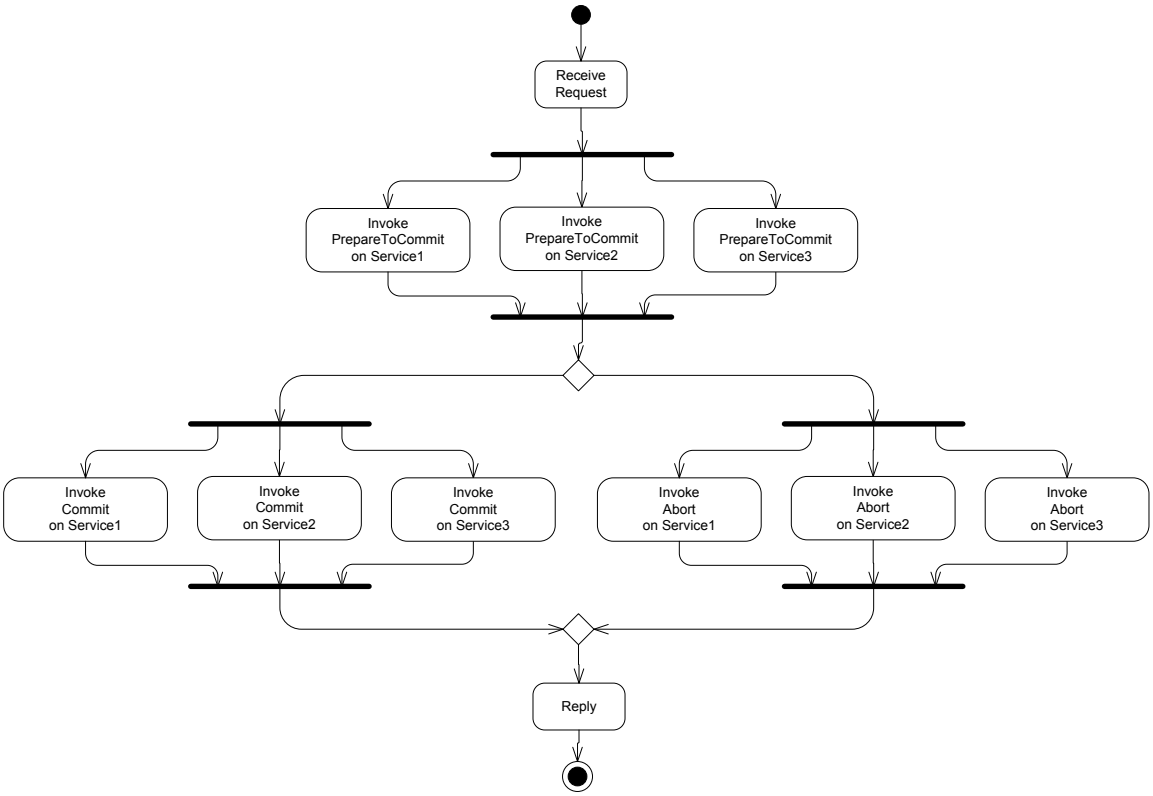


Figure 25 Activity model of the Two-phase commit coordination BPEL process

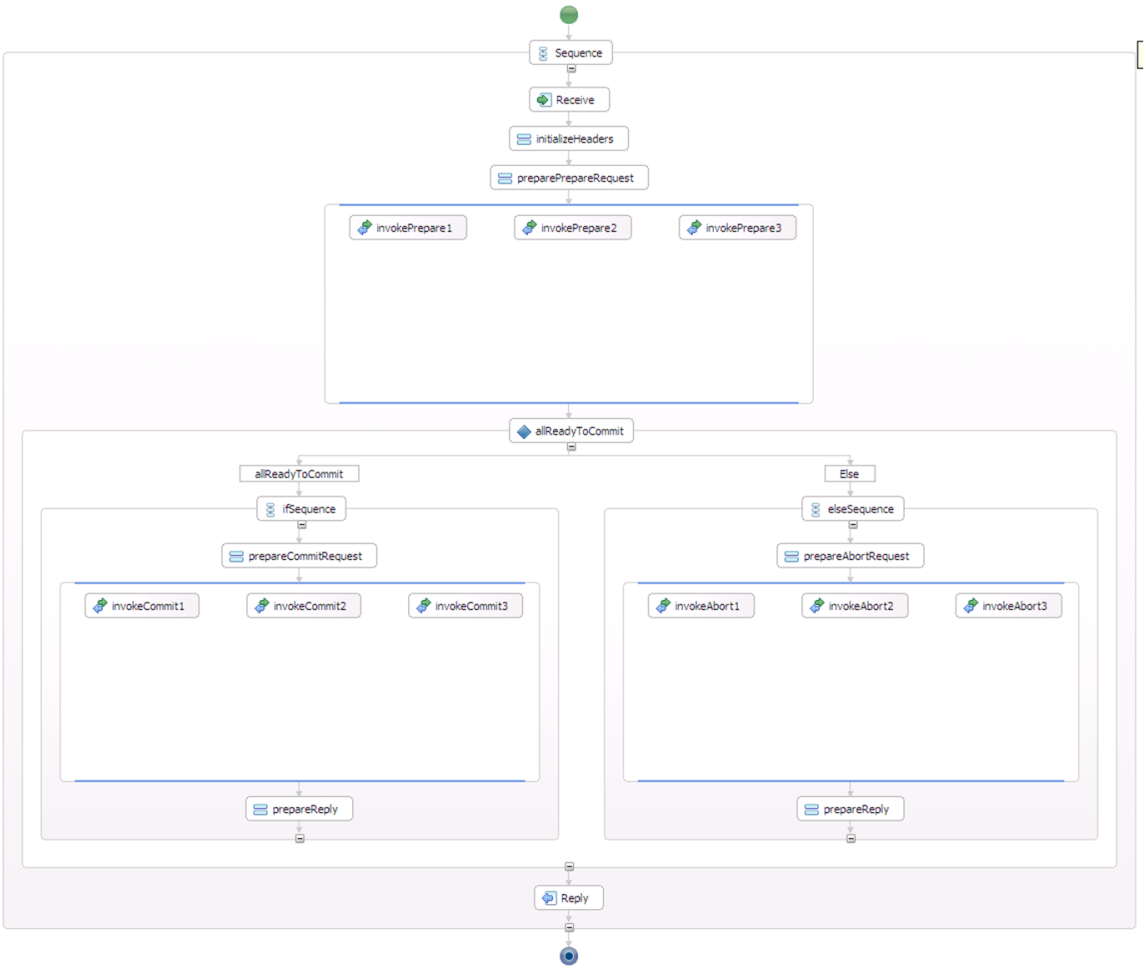


Figure 26 Two-phase commit coordination BPEL process

11.2.3 Validation of Hierarchical Coordination Adaptation Pattern

This section validates the hierarchical coordination adaptation pattern described in Section 7.3. In this validation, three of the implementation of emergency response system example described in Section 11.2.1 were instantiated as shown in Figure 27. The 911 Dispatcher Coordinator for each instantiation corresponds to one of the child coordinators

in Figure 11. On the other hand, a parent coordinator was implemented in the same way as shown in Figure 20, 22, and 23. As a result, in the execution of this implementation, the three validation steps described in Chapter 11 were carried out as planned. The execution traces of the parent and child coordinators are the same as Figure 24. The adaptation connector for a child coordinator to be replaced transitioned to the Quiescent state only after it received responses from all the services, whereas the adaptation connector for the parent coordinator was replaced after all the child coordinators responded and the parent coordinator sent a response to the client. The whole SOA application was resumed after the adaptation.

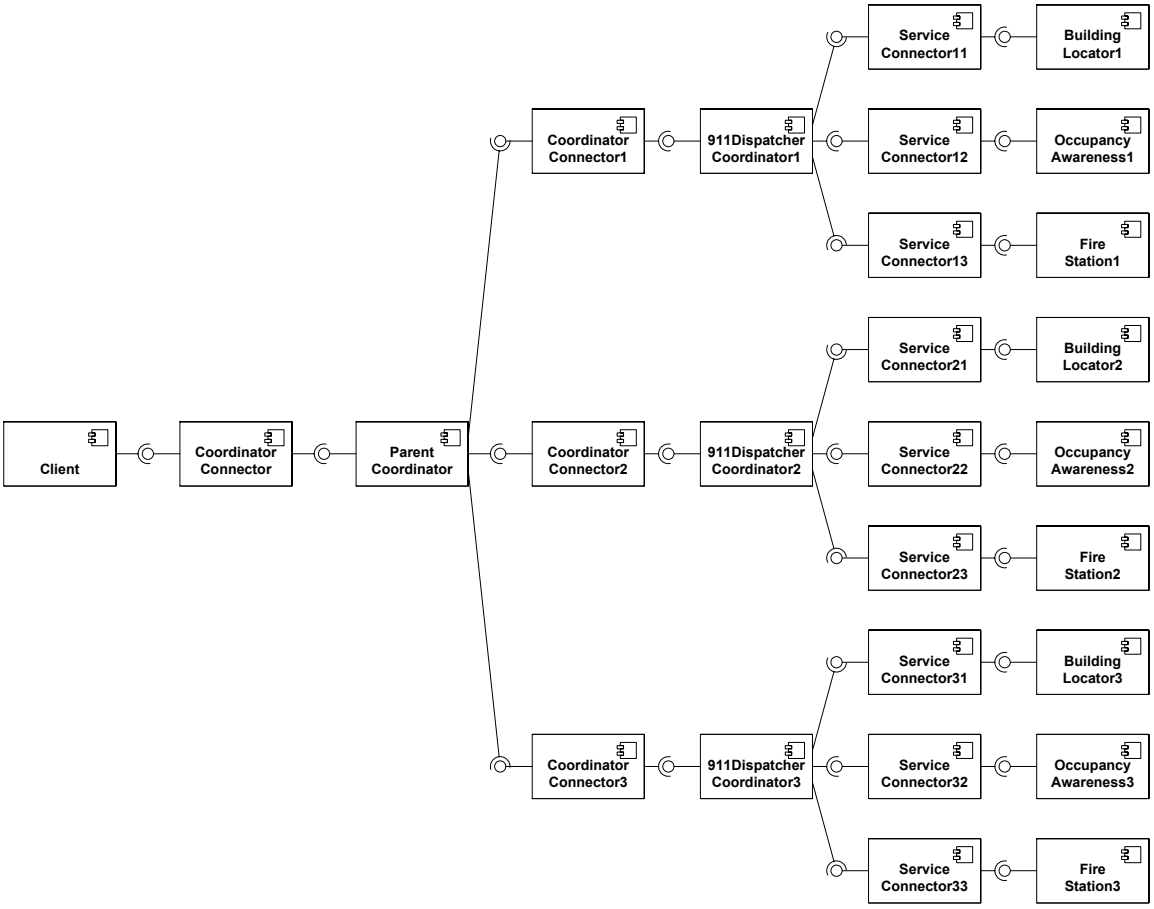


Figure 27 Hierarchical coordination in emergency response system

11.2.4 Validation of Distributed Coordination Adaptation Pattern

In this chapter, the distributed coordination adaptation pattern described in Section 7.4 is validated. As in the validation for the hierarchical coordination adaptation pattern, three of the implementation of emergency response system example described in Section 11.2.1 were instantiated as shown in Figure 28 in this validation. However, each 911

Dispatcher Coordinator was changed to be a distributed coordinator as a multi-threaded concurrent service as follows:

- Threads are assigned to incoming requests for the 911 Dispatcher coordination logic as long as they are available in the Swordfish thread pool.
- Two more threads are created, each of which sends a request to other distributed 911 Dispatcher Coordinator repeatedly in a random period.
- If a request is rejected by the corresponding adaptation connector, the coordinator shuts down its execution because it is removed or replaced for adaptation.

As a result, three instances of the distributed version of 911 Dispatcher Coordinator send requests to each other asynchronously while they coordinate the three services in their domains upon the requests.

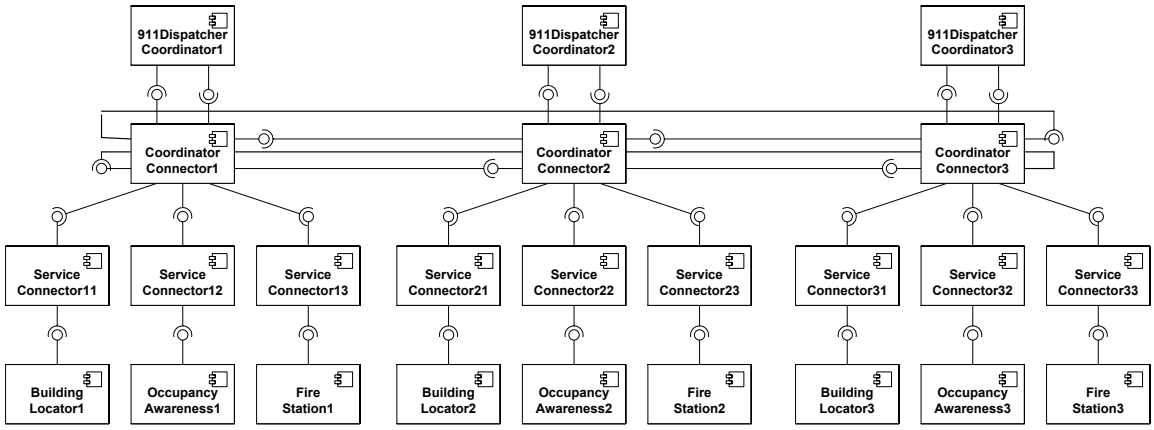


Figure 28 Distributed coordination in emergency response system

The result of execution of this implementation showed that the three validation steps described in Chapter 11 were carried out as planned. Figure 29 shows the execution log

of one of the coordinator connectors in the distributed coordination. As shown in the figure, the adaptation connector for a distributed coordinator to be replaced transitioned to the Quiescent state only after it finished all transactions it participated in. Either in the Passivating state or the Quiescent state, the adaptation connector rejected a request message sent by the coordinator, which shut down its execution.

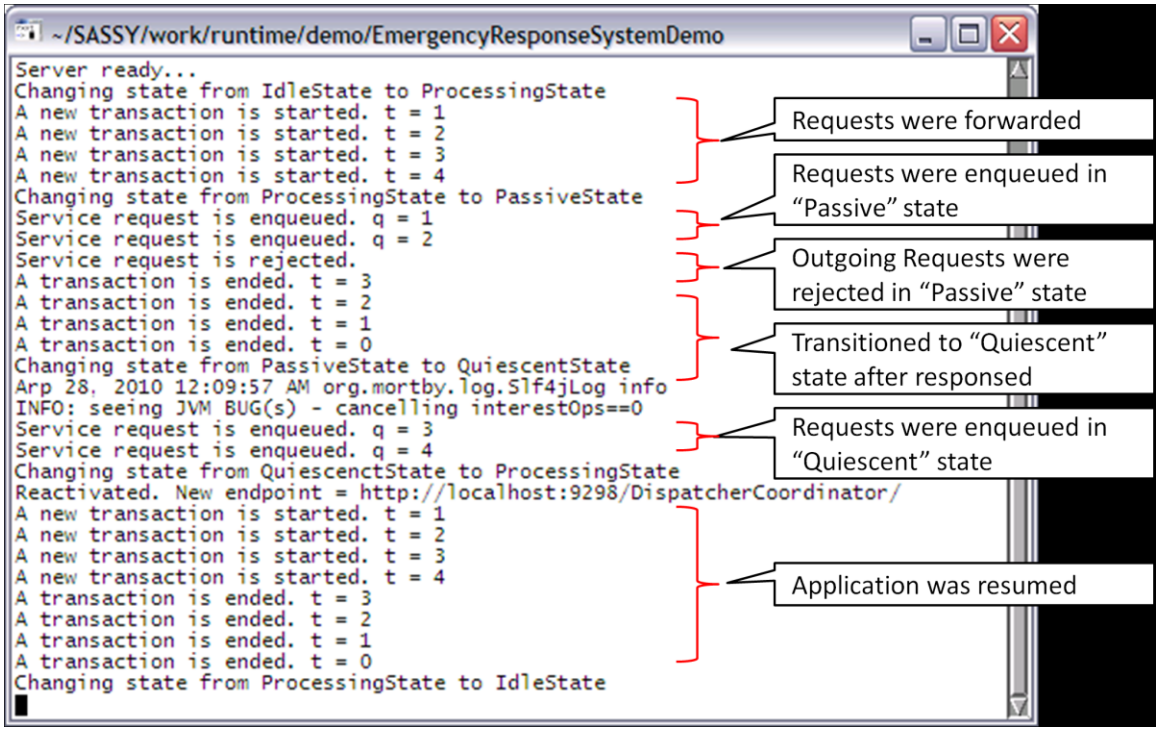


Figure 29 Execution log of Coordinator Connector in distributed coordination

11.2.5 Validation of Service Failure Adaptation Pattern

In this chapter, the service failure adaptation pattern described in Chapter 8 is validated as a part of the validation of SASSY framework. As described in Section 4.1, one of the

goals of SASSY framework is to support reactive adaptation, i.e., quick service failure recovery. In the SASSY framework prototype described in Section 11.2, therefore, the service failure detection and an alternative service discovery features were added to the adaptation connector, which monitors the connection established with a corresponding service/coordinator. If the connection is broken, the adaptation connector decides that the corresponding service/coordinator fails. It then attempts to find an alternative service via Swordfish Service Registry while it executes the adaptation state machine depicted in Figure 16.

This validation is based on the following scenario in the emergency response system example in Figure 18. First, Service Connector 2 performs the reactive adaptation by detecting the service failure of Occupancy Awareness. The adaptation connector quickly performs service replacement by discovering an alternative Occupancy Awareness service. Note that the software configuration after the reactive adaptation is the same as shown in Figure 18b. Second, the proactive adaptation scenario described in Section 10.1 is triggered based on the availability of the newly discovered Occupancy Awareness operating below 99.999%, which is specified as a QoS requirement.

The SASSY prototype with the emergency response system implementation executed the quick service failure recovery and dynamic software adaptation scenario described in Section 10.1. Service Connector 2 detected a service failure of Occupancy Awareness which was manually caused by network disconnection. While in the “Failed” state in

Figure 16, the service connector then replaced the failed service with an alternative Occupancy Awareness service discovered via Swordfish Service Registry. Upon notification, the SASSY self-architecting and re-architecting component generated a new software architecture involving the Fault Tolerant connector described in Section 10.1, which was implemented as an independent Apache CXF service in this example. When Change Management Service receives the newly generated architecture, it 1) extracts the difference between the new architecture and the original one, 2) instantiates and deploys the Fault Tolerant Connector, 3) generates a sequence of adaptation commands described in Section 10.1, and 4) sends the commands to Service Connector 2. As a result, the adaptation connector transitions from Active to Passive to Quiescent as designed in Figure 16. Note that Link command was skipped since there is no notion of "link" in this Web-service based implementation.

The validation consisted of 1) performing the quick service failure recovery, 2) executing the change management scenario, 3) performing the software adaptation from one configuration to another, and 4) resuming the application after the adaptation. The result of the execution of the implementation showed that the above four steps were carried out as planned. Figure 30 shows the execution trace of Service Connector 2 in the emergency response system in the above scenario including the service failure recovery of Occupancy Awareness service. Thus the validation demonstrates that the service failure adaptation pattern and the state machine described in Chapter 8 is integrated in the SASSY three-layer architecture, and perform the desired reactive and proactive

adaptation while ensuring that the service-oriented application does not enter an inconsistent state.

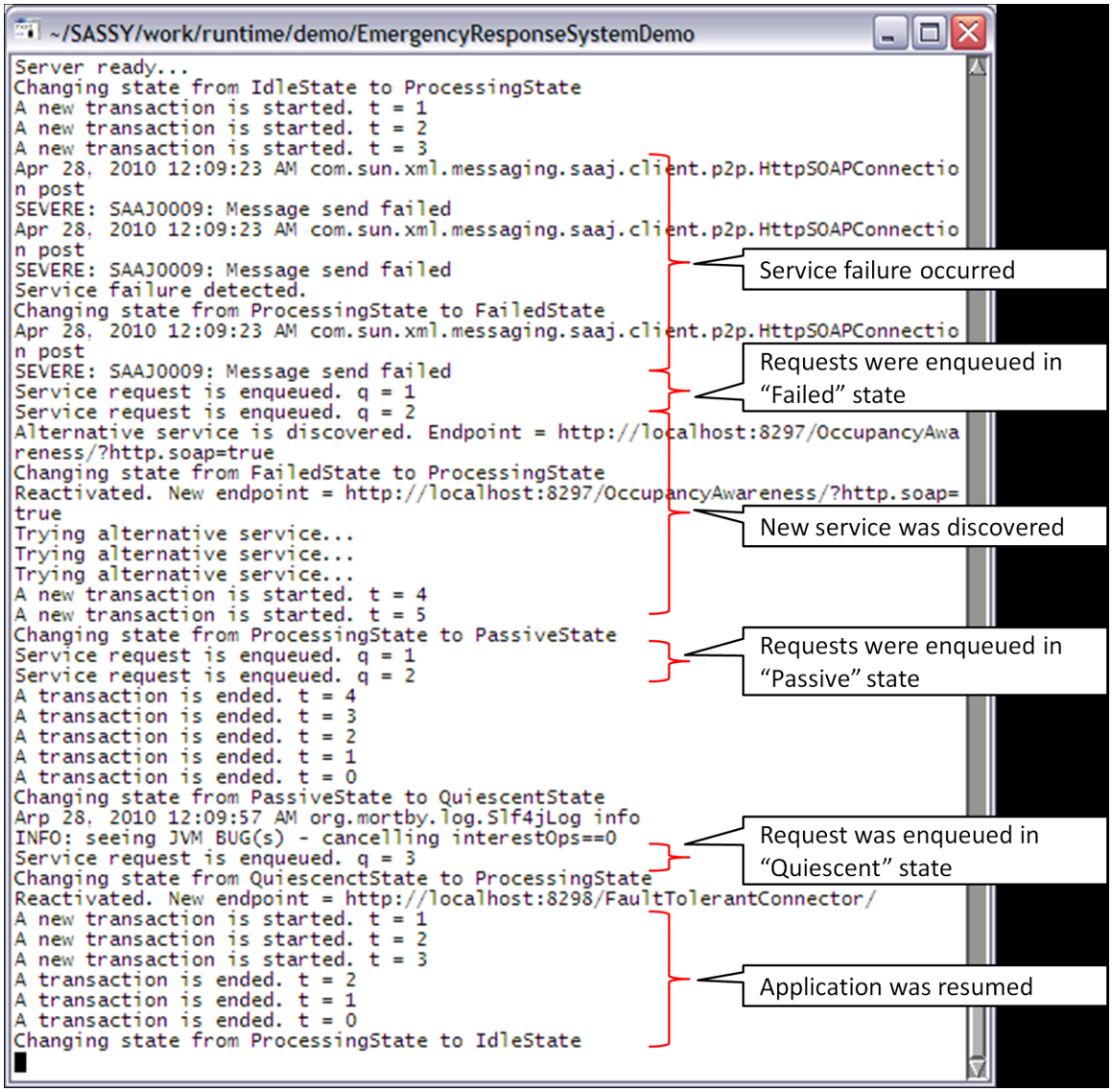


Figure 30 Execution log of Service Connector 2 in emergency response system

CHAPTER 12 CONCLUSIONS

This thesis has described how software adaptation patterns can be used in service oriented architectures to dynamically adapt coordinators and services at run-time. SOA adaptation patterns have been developed according to SOA coordination patterns in which coordinators can be independent, distributed, or hierarchical, and services can be stateless or stateful following the Two-phase Commit Protocol. The adaptation patterns are described by adaptation interaction models and adaptation state machine models encapsulated in adaptation connectors (coordinator connectors and service connectors).

The SOA adaptation patterns are described in terms of a three-layer reference architecture for self-management. The patterns execute at the lowest level, the component control layer. The Change Management Service executes at the second layer, sending change management commands to initiate the coordinator and/or service adaptation.

The contributions of this thesis include

1. SOA adaptation patterns: the adaptation patterns have been developed for SOA coordination patterns, i.e., independent coordination, two-phase commit coordination, hierarchical coordination, and distributed coordination patterns.

2. Introduction of adaptation connectors: adaptation connectors encapsulate the adaptation state machines for a given adaptation pattern to separate the concerns of an individual service from software adaptation. As a result, the SOA adaptation patterns described in this thesis are reusable regardless of the implementation of coordinators and services.
3. Integration of adaptation state machines: this thesis has shown the integrated adaptation connector which is usable for any SOA coordination pattern described in this thesis, by the thorough study of software adaptation for all the coordination patterns.

Future work will consist of investigating performance issues of dynamic adaptation for service-oriented architectures, developing additional adaptation patterns, and considering the failure of stateful services. Future research also includes the change management layer which automatically generates a sequence of adaptation commands by extracting the difference between new software architecture and original one.

APPENDIX A

The extract of the XTEAM simulation log for the first adaptation scenario, i.e., Occupancy Awareness service replacement:

```
                                    :
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 508
ChangeManagement_0000010b sent CMCommand at time 622
ChangeManagement_0000010b : 0 transitioning to process
HANDLEINPUTS at time 622
ServiceConnector1_000000ac received CMCommand at time 622
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSCOMMAND at time 622
ChangeManagement_0000010b : 0 transitioning to process IDLE at
time 623
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSPASSIVATE at time 623
ServiceConnector1_000000ac : 0 transitioning to process PASSIVE
at time 624
ServiceConnector1_000000ac : 0 transitioning to process
HANDLEINPUTS at time 625
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 626
OccupancyAwarenessService_00000008 sent Message at time 706
OccupancyAwarenessService_00000008 : 0 transitioning to process
WAIT at time 706
ServiceConnector1_000000ac received Message at time 706
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSSERVICERESPONSE at time 706
OccupancyAwarenessService_00000008 : 0 transitioning to process
IDLE at time 707
ServiceConnector1_000000ac : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PASSIVE at time 707
ServiceConnector1_000000ac sent Message at time 708
ServiceConnector1_000000ac : 0 transitioning to process SENDING
QUIESCENT NOTIFICATION TO ADAPTATION SERVICE IN PASSIVE at time
708
Coordinator_00000003 received Message at time 708
Coordinator_00000003 : 0 transitioning to process INVOKESERVICE2
at time 708
Coordinator_00000003 sent Message at time 708
Coordinator_00000003 : 0 transitioning to process HANDLEINPUTS at
time 708
ServiceConnector2_0000013a received Message at time 708
```

```
ServiceConnector2_0000013a : 0 transitioning to process
PROCESSSERVICEREQUEST at time 708
ServiceConnector1_000000ac sent CMNotification at time 709
ServiceConnector1_000000ac : 0 transitioning to process QUIESCENT
at time 709
Coordinator_00000003 : 0 transitioning to process IDLE at time
709
ServiceConnector2_0000013a : 0 transitioning to process SENDING
NEXT SERVICE REQUEST at time 709
ChangeManagement_0000010b received CMNotification at time 709
ChangeManagement_0000010b : 0 transitioning to process
PROCESSNOTIFICATION at time 709
ServiceConnector1_000000ac : 0 transitioning to process
HANDLEINPUTS at time 710
ServiceConnector2_0000013a sent Message at time 710
ServiceConnector2_0000013a : 0 transitioning to process
PROCESSING at time 710
ChangeManagement_0000010b : 0 transitioning to process SENDING
UNLINK at time 710
BusinessService2_00000126 received Message at time 710
BusinessService2_00000126 : 0 transitioning to process INPUT at
time 710
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 711
ServiceConnector2_0000013a : 0 transitioning to process
HANDLEINPUTS at time 711
ServiceConnector2_0000013a : 0 transitioning to process IDLE at
time 712
BusinessService2_00000126 sent Message at time 715
BusinessService2_00000126 : 0 transitioning to process WAIT at
time 715
ServiceConnector2_0000013a received Message at time 715
ServiceConnector2_0000013a : 0 transitioning to process
PROCESSSERVICERESPONSE at time 715
BusinessService2_00000126 : 0 transitioning to process IDLE at
time 716
ServiceConnector2_0000013a : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PROCESSING at time 716
ServiceConnector2_0000013a sent Message at time 717
ServiceConnector2_0000013a : 0 transitioning to process WAITING
FOR SERVICE REQUEST at time 717
Coordinator_00000003 received Message at time 717
Coordinator_00000003 : 0 transitioning to process INVOKESERVICE3
at time 717
Coordinator_00000003 sent Message at time 717
Coordinator_00000003 : 0 transitioning to process HANDLEINPUTS at
time 717
ServiceConnector3_0000017d received Message at time 717
```

```
ServiceConnector3_0000017d : 0 transitioning to process
PROCESSSERVICEREQUEST at time 717
ServiceConnector2_0000013a : 0 transitioning to process
HANDLEINPUTS at time 718
Coordinator_00000003 : 0 transitioning to process IDLE at time
718
ServiceConnector3_0000017d : 0 transitioning to process SENDING
NEXT SERVICE REQUEST at time 718
ServiceConnector2_0000013a : 0 transitioning to process IDLE at
time 719
ServiceConnector3_0000017d sent Message at time 719
ServiceConnector3_0000017d : 0 transitioning to process
PROCESSING at time 719
BusinessService3_00000130 received Message at time 719
BusinessService3_00000130 : 0 transitioning to process INPUT at
time 719
ServiceConnector3_0000017d : 0 transitioning to process
HANDLEINPUTS at time 720
ServiceConnector3_0000017d : 0 transitioning to process IDLE at
time 721
BusinessService3_00000130 sent Message at time 724
BusinessService3_00000130 : 0 transitioning to process WAIT at
time 724
ServiceConnector3_0000017d received Message at time 724
ServiceConnector3_0000017d : 0 transitioning to process
PROCESSSERVICERESPONSE at time 724
BusinessService3_00000130 : 0 transitioning to process IDLE at
time 725
ServiceConnector3_0000017d : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PROCESSING at time 725
ServiceConnector3_0000017d sent Message at time 726
ServiceConnector3_0000017d : 0 transitioning to process WAITING
FOR SERVICE REQUEST at time 726
Coordinator_00000003 received Message at time 726
Coordinator_00000003 : 0 transitioning to process RESPONDCLIENT
at time 726
ServiceConnector3_0000017d : 0 transitioning to process
HANDLEINPUTS at time 727
ServiceConnector3_0000017d : 0 transitioning to process IDLE at
time 728
Coordinator_00000003 sent Message at time 736
Coordinator_00000003 : 0 transitioning to process IDLE at time
736
CoordinatorConnector_000001f7 received Message at time 736
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSSERVICERESPONSE at time 736
CoordinatorConnector_000001f7 : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PROCESSING at time 737
```

```
CoordinatorConnector_000001f7 sent Message at time 738
CoordinatorConnector_000001f7 : 0 transitioning to process
WAITING FOR SERVICE REQUEST at time 738
Client_00000039 received Message at time 738
Client_00000039 : 0 transitioning to process FINAL at time 738
CoordinatorConnector_000001f7 : 0 transitioning to process
HANDLEINPUTS at time 739
Client_00000039 : 0 transitioning to process SENDREQUEST at time
739
CoordinatorConnector_000001f7 : 0 transitioning to process IDLE
at time 740
Client_00000039 sent Message at time 740
Client_00000039 : 0 transitioning to process HANDLEINPUTS at time
740
CoordinatorConnector_000001f7 received Message at time 740
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSSERVICEREQUEST at time 740
Client_00000039 : 0 transitioning to process IDLE at time 741
CoordinatorConnector_000001f7 : 0 transitioning to process
SENDING NEXT SERVICE REQUEST at time 741
CoordinatorConnector_000001f7 sent Message at time 742
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSING at time 742
Coordinator_00000003 received Message at time 742
Coordinator_00000003 : 0 transitioning to process INVOKESERVICE1
at time 742
Coordinator_00000003 sent Message at time 742
Coordinator_00000003 : 0 transitioning to process HANDLEINPUTS at
time 742
ServiceConnector1_000000ac received Message at time 742
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSSERVICEREQUEST at time 742
CoordinatorConnector_000001f7 : 0 transitioning to process
HANDLEINPUTS at time 743
Coordinator_00000003 : 0 transitioning to process IDLE at time
743
ServiceConnector1_000000ac : 0 transitioning to process
INCREMENTINGQ IN QUIESCENT at time 743
CoordinatorConnector_000001f7 : 0 transitioning to process IDLE
at time 744
ServiceConnector1_000000ac : 0 transitioning to process QUIESCENT
at time 744
ServiceConnector1_000000ac : 0 transitioning to process
HANDLEINPUTS at time 745
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 746
ChangeManagement_0000010b sent CMCommand at time 1210
```

```
ChangeManagement_0000010b : 0 transitioning to process SENDING
LINK at time 1210
ServiceConnector1_000000ac received CMCommand at time 1210
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSCOMMAND at time 1210
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSUNLINK at time 1211
ServiceConnector1_000000ac : 0 transitioning to process
HANDLEINPUTS at time 1212
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 1213
ChangeManagement_0000010b sent CMCommand at time 1710
ChangeManagement_0000010b : 0 transitioning to process SENDING
REACTIVATE at time 1710
ServiceConnector1_000000ac received CMCommand at time 1710
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSCOMMAND at time 1710
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSLINK at time 1711
ServiceConnector1_000000ac : 0 transitioning to process
HANDLEINPUTS at time 1712
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 1713
ChangeManagement_0000010b sent CMCommand at time 2210
ChangeManagement_0000010b : 0 transitioning to process
HANDLEINPUTS at time 2210
ServiceConnector1_000000ac received CMCommand at time 2210
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSCOMMAND at time 2210
ChangeManagement_0000010b : 0 transitioning to process IDLE at
time 2211
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSREACTIVATE at time 2211
ServiceConnector1_000000ac : 0 transitioning to process SENDING
GONE ACTIVE NOTIFICATION TO ADAPTATION SERVICE at time 2212
ServiceConnector1_000000ac sent CMNotification at time 2213
ServiceConnector1_000000ac : 0 transitioning to process SENDING
NEXT SERVICE REQUEST IN QUIESCENT at time 2213
ChangeManagement_0000010b received CMNotification at time 2213
ChangeManagement_0000010b : 0 transitioning to process
PROCESSNOTIFICATION at time 2213
ServiceConnector1_000000ac : 0 transitioning to process SEND NEXT
NEWSERVICE REQUEST IN QUIESCENT at time 2214
ChangeManagement_0000010b : 0 transitioning to process
HANDLEINPUTS at time 2214
ServiceConnector1_000000ac sent Message at time 2215
ServiceConnector1_000000ac : 0 transitioning to process
DECREMENTINGQ IN QUIESCENT at time 2215
```

```
ChangeManagement_0000010b : 0 transitioning to process IDLE at
time 2215
FaultTolerantService_000001d4 received Message at time 2215
FaultTolerantService_000001d4 : 0 transitioning to process INPUT
at time 2215
ServiceConnector1_000000ac : 0 transitioning to process
PROCESSING at time 2216
ServiceConnector1_000000ac : 0 transitioning to process
HANDLEINPUTS at time 2217
ServiceConnector1_000000ac : 0 transitioning to process IDLE at
time 2218
FaultTolerantService_000001d4 sent Message at time 2220
                                    :
```

APPENDIX B

The extract of the XTEAM simulation log for the second adaptation scenario, i.e., the coordinator replacement case:

```
                                    :
ChangeManagement_0000010b sent CMCommand at time 621
ChangeManagement_0000010b : 0 transitioning to process
HANDLEINPUTS at time 621
ServiceConnector1_0000024f : 0 transitioning to process SENDING
NEXT SERVICE REQUEST at time 621
Coordinator_000002bc : 0 transitioning to process IDLE at time
621
CoordinatorConnector_000001f7 : 0 transitioning to process
HANDLEINPUTS at time 621
CoordinatorConnector_000001f7 received CMCommand at time 621
CoordinatorConnector_000001f7 : 1 transitioning to process
PROCESSCOMMAND at time 621
ChangeManagement_0000010b : 0 transitioning to process IDLE at
time 622
CoordinatorConnector_000001f7 : 0 transitioning to process IDLE
at time 622
ServiceConnector1_0000024f sent Message at time 622
ServiceConnector1_0000024f : 0 transitioning to process
PROCESSING at time 622
BusinessService1_00000008 received Message at time 622
BusinessService1_00000008 : 0 transitioning to process INPUT at
time 622
CoordinatorConnector_000001f7 : 1 transitioning to process
PROCESSPASSIVATE at time 623
ServiceConnector1_0000024f : 0 transitioning to process
HANDLEINPUTS at time 623
CoordinatorConnector_000001f7 : 1 transitioning to process
PASSIVE at time 624
ServiceConnector1_0000024f : 0 transitioning to process IDLE at
time 624
CoordinatorConnector_000001f7 : 1 transitioning to process
HANDLEINPUTS at time 625
CoordinatorConnector_000001f7 : 1 transitioning to process IDLE
at time 626
BusinessService1_00000008 sent Message at time 627
BusinessService1_00000008 : 0 transitioning to process WAIT at
time 627
ServiceConnector1_0000024f received Message at time 627
```

```
ServiceConnector1_0000024f : 0 transitioning to process
PROCESSSERVICERESPONSE at time 627
BusinessService1_00000008 : 0 transitioning to process IDLE at
time 628
ServiceConnector1_0000024f : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PROCESSING at time 628
ServiceConnector1_0000024f sent Message at time 629
ServiceConnector1_0000024f : 0 transitioning to process WAITING
FOR SERVICE REQUEST at time 629
Coordinator_000002bc received Message at time 629
Coordinator_000002bc : 0 transitioning to process
PROCESSRESPONSEFROMSERVICE1 at time 629
NewCoordinator_00000292 received Message at time 629
NewCoordinator_00000292 : 0 transitioning to process
PROCESSRESPONSEFROMSERVICE1 at time 629
Coordinator_000002bc : 0 transitioning to process INVOKESERVICE2
at time 629
NewCoordinator_00000292 : 0 transitioning to process HANDLEINPUTS
at time 629
Coordinator_000002bc sent Message at time 629
Coordinator_000002bc : 0 transitioning to process HANDLEINPUTS at
time 629
ServiceConnector2_0000013a received Message at time 629
ServiceConnector2_0000013a : 0 transitioning to process
PROCESSSERVICEREQUEST at time 629
ServiceConnector1_0000024f : 0 transitioning to process
HANDLEINPUTS at time 630
ServiceConnector2_0000013a : 0 transitioning to process SENDING
NEXT SERVICE REQUEST at time 630
Coordinator_000002bc : 0 transitioning to process IDLE at time
630
NewCoordinator_00000292 : 0 transitioning to process IDLE at time
630
ServiceConnector2_0000013a sent Message at time 631
ServiceConnector2_0000013a : 0 transitioning to process
PROCESSING at time 631
ServiceConnector1_0000024f : 0 transitioning to process IDLE at
time 631
BusinessService2_00000126 received Message at time 631
BusinessService2_00000126 : 0 transitioning to process INPUT at
time 631
ServiceConnector2_0000013a : 0 transitioning to process
HANDLEINPUTS at time 632
ServiceConnector2_0000013a : 0 transitioning to process IDLE at
time 633
BusinessService2_00000126 sent Message at time 636
BusinessService2_00000126 : 0 transitioning to process WAIT at
time 636
```

```
ServiceConnector2_0000013a received Message at time 636
ServiceConnector2_0000013a : 0 transitioning to process
PROCESSSERVICERESPONSE at time 636
BusinessService2_00000126 : 0 transitioning to process IDLE at
time 637
ServiceConnector2_0000013a : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PROCESSING at time 637
ServiceConnector2_0000013a sent Message at time 638
ServiceConnector2_0000013a : 0 transitioning to process WAITING
FOR SERVICE REQUEST at time 638
Coordinator_000002bc received Message at time 638
Coordinator_000002bc : 0 transitioning to process
PROCESSRESPONSEFROMSERVICE2 at time 638
NewCoordinator_00000292 received Message at time 638
NewCoordinator_00000292 : 0 transitioning to process
PROCESSRESPONSEFROMSERVICE2 at time 638
Coordinator_000002bc : 0 transitioning to process INVOKESERVICE3
at time 638
NewCoordinator_00000292 : 0 transitioning to process HANDLEINPUTS
at time 638
Coordinator_000002bc sent Message at time 638
Coordinator_000002bc : 0 transitioning to process HANDLEINPUTS at
time 638
ServiceConnector3_0000017d received Message at time 638
ServiceConnector3_0000017d : 0 transitioning to process
PROCESSSERVICEREQUEST at time 638
ServiceConnector2_0000013a : 0 transitioning to process
HANDLEINPUTS at time 639
ServiceConnector3_0000017d : 0 transitioning to process SENDING
NEXT SERVICE REQUEST at time 639
Coordinator_000002bc : 0 transitioning to process IDLE at time
639
NewCoordinator_00000292 : 0 transitioning to process IDLE at time
639
ServiceConnector3_0000017d sent Message at time 640
ServiceConnector3_0000017d : 0 transitioning to process
PROCESSING at time 640
ServiceConnector2_0000013a : 0 transitioning to process IDLE at
time 640
BusinessService3_00000130 received Message at time 640
BusinessService3_00000130 : 0 transitioning to process INPUT at
time 640
ServiceConnector3_0000017d : 0 transitioning to process
HANDLEINPUTS at time 641
ServiceConnector3_0000017d : 0 transitioning to process IDLE at
time 642
BusinessService3_00000130 sent Message at time 645
```

```
BusinessService3_00000130 : 0 transitioning to process WAIT at
time 645
ServiceConnector3_0000017d received Message at time 645
ServiceConnector3_0000017d : 0 transitioning to process
PROCESSSERVICERESPONSE at time 645
BusinessService3_00000130 : 0 transitioning to process IDLE at
time 646
ServiceConnector3_0000017d : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PROCESSING at time 646
ServiceConnector3_0000017d sent Message at time 647
ServiceConnector3_0000017d : 0 transitioning to process WAITING
FOR SERVICE REQUEST at time 647
Coordinator_000002bc received Message at time 647
Coordinator_000002bc : 0 transitioning to process
PROCESSRESPONSEFROMSERVICE3 at time 647
NewCoordinator_00000292 received Message at time 647
NewCoordinator_00000292 : 0 transitioning to process
PROCESSRESPONSEFROMSERVICE3 at time 647
ServiceConnector3_0000017d : 0 transitioning to process
HANDLEINPUTS at time 648
ServiceConnector3_0000017d : 0 transitioning to process IDLE at
time 649
Coordinator_000002bc : 0 transitioning to process RESPONDCLIENT
at time 657
NewCoordinator_00000292 : 0 transitioning to process HANDLEINPUTS
at time 657
NewCoordinator_00000292 : 0 transitioning to process IDLE at time
658
Coordinator_000002bc sent Message at time 667
Coordinator_000002bc : 0 transitioning to process IDLE at time
667
CoordinatorConnector_000001f7 received Message at time 667
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSSERVICERESPONSE at time 667
CoordinatorConnector_000001f7 : 0 transitioning to process
FORWARDING SERVICE RESPONSE IN PASSIVE at time 668
CoordinatorConnector_000001f7 sent Message at time 669
CoordinatorConnector_000001f7 : 0 transitioning to process
SENDING QUIESCENT NOTIFICATION TO ADAPTATION SERVICE IN PASSIVE
at time 669
Client_00000039 received Message at time 669
Client_00000039 : 1 transitioning to process FINAL at time 669
CoordinatorConnector_000001f7 sent CMNotification at time 670
CoordinatorConnector_000001f7 : 0 transitioning to process
QUIESCENT at time 670
Client_00000039 : 1 transitioning to process SENDREQUEST at time
670
ChangeManagement_0000010b received CMNotification at time 670
```

```
ChangeManagement_0000010b : 0 transitioning to process
PROCESSNOTIFICATION at time 670
CoordinatorConnector_000001f7 : 0 transitioning to process
HANDLEINPUTS at time 671
Client_00000039 sent Message at time 671
Client_00000039 : 1 transitioning to process HANDLEINPUTS at time
671
ChangeManagement_0000010b : 0 transitioning to process SENDING
UNLINK at time 671
CoordinatorConnector_000001f7 received Message at time 671
CoordinatorConnector_000001f7 : 1 transitioning to process
PROCESSSERVICEREQUEST at time 671
CoordinatorConnector_000001f7 : 0 transitioning to process IDLE
at time 672
Client_00000039 : 1 transitioning to process IDLE at time 672
CoordinatorConnector_000001f7 : 1 transitioning to process
INCREMENTINGQ IN QUIESCENT at time 673
CoordinatorConnector_000001f7 : 1 transitioning to process
QUIESCENT at time 674
CoordinatorConnector_000001f7 : 1 transitioning to process
HANDLEINPUTS at time 675
CoordinatorConnector_000001f7 : 1 transitioning to process IDLE
at time 676
ChangeManagement_0000010b sent CMCommand at time 1171
ChangeManagement_0000010b : 0 transitioning to process SENDING
LINK at time 1171
CoordinatorConnector_000001f7 received CMCommand at time 1171
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSCOMMAND at time 1171
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSUNLINK at time 1172
CoordinatorConnector_000001f7 : 0 transitioning to process
HANDLEINPUTS at time 1173
CoordinatorConnector_000001f7 : 0 transitioning to process IDLE
at time 1174
ChangeManagement_0000010b sent CMCommand at time 1671
ChangeManagement_0000010b : 0 transitioning to process SENDING
REACTIVATE at time 1671
CoordinatorConnector_000001f7 received CMCommand at time 1671
CoordinatorConnector_000001f7 : 1 transitioning to process
PROCESSCOMMAND at time 1671
CoordinatorConnector_000001f7 : 1 transitioning to process
PROCESSLINK at time 1672
CoordinatorConnector_000001f7 : 1 transitioning to process
HANDLEINPUTS at time 1673
CoordinatorConnector_000001f7 : 1 transitioning to process IDLE
at time 1674
ChangeManagement_0000010b sent CMCommand at time 2171
```

```
ChangeManagement_0000010b : 0 transitioning to process
HANDLEINPUTS at time 2171
CoordinatorConnector_000001f7 received CMCommand at time 2171
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSCOMMAND at time 2171
ChangeManagement_0000010b : 0 transitioning to process IDLE at
time 2172
CoordinatorConnector_000001f7 : 0 transitioning to process
PROCESSREACTIVATE at time 2172
CoordinatorConnector_000001f7 : 0 transitioning to process
SENDING GONE ACTIVE NOTIFICATION TO ADAPTATION SERVICE at time
2173
CoordinatorConnector_000001f7 sent CMNotification at time 2174
CoordinatorConnector_000001f7 : 0 transitioning to process
SENDING NEXT SERVICE REQUEST IN QUIESCENT at time 2174
ChangeManagement_0000010b received CMNotification at time 2174
ChangeManagement_0000010b : 0 transitioning to process
PROCESSNOTIFICATION at time 2174
CoordinatorConnector_000001f7 : 0 transitioning to process SEND
NEXT NEWSERVICE REQUEST IN QUIESCENT at time 2175
ChangeManagement_0000010b : 0 transitioning to process
HANDLEINPUTS at time 2175
CoordinatorConnector_000001f7 sent Message at time 2176
CoordinatorConnector_000001f7 : 0 transitioning to process
DECREMENTINGQ IN QUIESCENT at time 2176
ChangeManagement_0000010b : 0 transitioning to process IDLE at
time 2176
NewCoordinator_00000292 received Message at time 2176
NewCoordinator_00000292 : 0 transitioning to process
INVOKESERVICE1 at time 2176
NewCoordinator_00000292 sent Message at time 2176
NewCoordinator_00000292 : 0 transitioning to process HANDLEINPUTS
at time 2176
                                    :
```

REFERENCES

REFERENCES

[1] H. Gomaa, "Building Software Systems and Product Lines from Software Architectural Patterns", ECOOP Workshop. on Building Systems from Patterns, Glasgow, UK, July 2005.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, "Pattern Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[4] D. Garlan and B. Schmerl, "Model-based Adaptation for Self-Healing Systems", Proc. Workshop on Self-Healing Systems, ACM Press, Charleston, SC, 2002.

[5] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley, Reading MA, 2000.

[6] H. Gomaa and M. Hussein, "Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures", Proc. Fourth Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, June, 2004.

[7] H. Gomaa, "Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures", Addison-Wesley, 2005.

[8] H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Keynote paper, Proc. 9th Intl. Conf. on Model-Driven Engineering, Languages, and Systems (MoDELS), Genova, Italy, Oct. 2006.

[9] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Transactions on Software Eng., Vol. 16, No. 11, 1990.

[10] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge", Proc Intl. Conference on Software Engineering, Minneapolis, MN, May 2007.

[11] M. Kim, J. Jeong, and S. Park, "From Product Lines to Self-Managed Systems: An Architecture-Based Runtime Reconfiguration Framework," Proc. Design and Evolution of Autonomic Application Software (DEAS2005), ICSE05, St. Louis, MO, May 2005, pp. 66-72.

[12] J. Lee and K. Kang, "A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering," Proc. 10th Int. Soft. Product Line Conf. (SPLC 2006), Baltimore, Maryland, 2006.

[13] A. J. Ramirez and B. H. Cheng, "Applying Adaptation Design Patterns," Prof. 6th Intl. Conf. on Autonomic Computing (ICAC), pp. 69-70, Jun. 2009.

[14] G. Li, et al., "Facilitating Dynamic Service Compositions by Adaptable Service Connectors", International Journal of Web Services Research, Vol. 3, No. 1, 2006, pp. 67-83.

[15] F. Irmert, T. Fischer, K. Meyer-Wegener, "Runtime adaptation in a service-oriented component model", Proc. Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems, May 2008, pp. 97-104.

[16] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges", Computer, vol. 40, pp. 39-45, 2007.

[17] E. Thomas. Service-Oriented Architecture. Prentice Hall PTR, Upper Saddle River, 2005.

[18] E. Gat, Three-layer Architectures, "Artificial Intelligence and Mobile Robots", MIT/AAAI Press, 1997.

[19] M. Kim et al., "Service Robot Software Development with the COMET/UML Method", IEEE Robotics and Automation, Vol. 16, No. 1, March 2009, pp. 34-45.

[20] S. Malek, N. Esfahani, D. Menascé, J. Sousa, and H. Gomaa, "Self-Architecting Software Systems (SASSY) from QoS-Annotated Activity Models", in Proc ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), Vancouver, Canada, May 2009.

[21] N. Esfahani, S. Malek, J. P. Sousa, H. Gomaa, and D. A. Menascé, "A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems", Proc. ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 09), Denver, Colorado, Oct. 2009.

[22] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa, "A Framework for Utility-Based Service Oriented Design in SASSY", Proc. First Joint WOSP/SIPEW International Conf. on Performance Engineering, Jan. 2010.

[23] D. A. Menascé, J. P. Sousa, S. Malek, and H. Gomaa, "QoS Architectural Patterns for Self-Architecting Software Systems", 7th IEEE Intl. Conf. on Autonomic Computing and Communication, Washington, DC, June, 2010.

[24] G. Edwards, S. Malek, and N. Medvidovic, "Scenario-Driven Dynamic Analysis of Distributed Architecture", Proc. Intl. Conf. on Fundamental Approaches to Software Engineering, Braga, Portugal, March 2007.

[25] E. Dashofy, A. van der Hoek, and R.N. Taylor, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages", Proc. 24th Intl. Conference on Software Engineering, pp. 266 - 276, 2002.

[26] J. Magee, et al., "Behaviour Analysis of Software Architectures", Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), pp. 35 - 50, 1999.

[27] Apache Tomcat, http://tomcat.apache.org/

[28] GlassFish, https://glassfish.dev.java.net/

[29] Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl

[30] SOAP Specifications, http://www.w3.org/TR/soap/

[31] SOAP with Attachments API for Java (SAAJ), http://java.sun.com/xml/saaj/index.html

[32] Eclipse Swordfish Project, http://www.eclipse.org/swordfish/.

[33] Apache CXF, http://cxf.apache.org/.

[34] Apache ServiceMix, http://servicemix.apache.org/.

[35] OSGi Alliance, http://www.osgi.org/.

[36] Eclipse BPEL Project, http://www.eclipse.org/bpel/

CURRICULUM VITAE

Koji Hashimoto graduated with a Ph.D. in Computer Science from Osaka University, Osaka, Japan in 2000. He worked as a researcher at Hitachi Research Laboratory, Hitachi Ltd., Japan from 2000 to 2008. He will work as a Software Development Engineer at Amazon.com, Seattle, WA, USA.