Software Adaptation Patterns for Service-Oriented Architectures

Hassan Gomaa, Koji Hashimoto, Minseong Kim, Sam Malek, Daniel A. Menascé
Department of Computer Science
George Mason University
Fairfax, VA 22030
{hgomaa, khashimo, mkim12, smalek, menasce}@gmu.edu

ABSTRACT
This paper 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. In our approach, adaptation connectors are introduced to encapsulate adaptation state machine models so that the adaptation patterns can be more reusable. A change management model for dynamically evolving service-oriented applications is also described with a case study.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *State diagrams,* D.2.11 [**Software Engineering**]: Software Architectures – *Domain-specific Architectures, Patterns.*

Keywords
component, service, service-oriented architecture, dynamic adaptation, software adaptation patterns.

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 paper describes the concept of software adaptation patterns and how they can be used in service oriented architectures. Previous papers have described how software architectural patterns can be used to help in building software systems and product lines [1][7][8]. This paper describes how software adaptation patterns can be used to help with the adaptation of service-oriented software systems after original deployment. The adaptation patterns are part of the Self-Architecting Software Systems (SASSY) framework [20].

SAC'10, March 22-26, 2010, Sierre, Switzerland.

In this paper, we base our research 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 paper builds on software reconfiguration patterns in our previous work [6] and advances these concepts by describing patterns to support dynamic adaptation in service-oriented applications.

This paper first describes different kinds of software adaptation. It then describes software adaptation patterns for service oriented architectures, with a detailed example of a sequential coordination adaptation pattern. Adaptive change management is also described.

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 paper focuses on service coordination in service-oriented architectures. We develop software adaptation patterns for different kinds of service coordination, in order to adapt not only services but also coordinator components.

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.

3.1 Three-layer Architecture Model for Software Adaptation

Our approach 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. OVERVIEW OF SASSY FRAMEWORK

The software adaptation patterns described in this paper are developed as part of the Self-Architecting Software Systems (SASSY), which is a model-driven framework for run-time self-architecting and rearchitecting 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. Specific details are provided in the sections that follow.

A system service architecture, consisting of components, (associated with services) and connectors (associated with middleware facilities), is optimized 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].

SASSY's monitoring support services (Monitoring Service and Gauge Service in Figure 1) generate triggers that cause self-adaptation. Services are automatically replaced and the software architecture is automatically regenerated by the Goal Management layer when the system utility degrades beyond a certain threshold. SASSY's adaptation support services (Change Management Service and Adaptation Service) are used to transition from one version of the architecture to a new one. When services fail or are unable to meet their QoS goals, the SASSY's monitoring services trigger service replacement through a new round of service discovery, optimal service selection, and possible determination of alternative architectures.

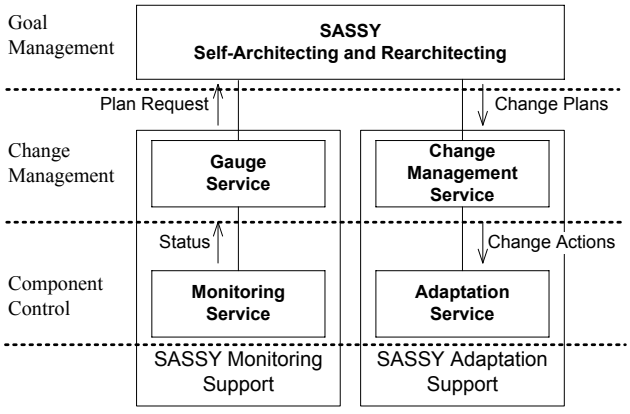


Figure 1. High-level view of the SASSY framework

More information about SASSY is given in [20][21][22]. In the remainder of this paper, we describe the role of adaptation patterns in enabling adaptation of SOA software systems in SASSY.

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 SASSY, the software architecture ensures this loose coupling 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 two properties: 1) type of coordination and 2) degree of concurrency. 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. In this paper, we describe independent coordination patterns with sequential and/or concurrent coordination.

With 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.

With 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.

With 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.

With 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.

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

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 (Figure 1) 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, our focus in this paper 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]. We have previously developed several adaptation patterns, including 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.

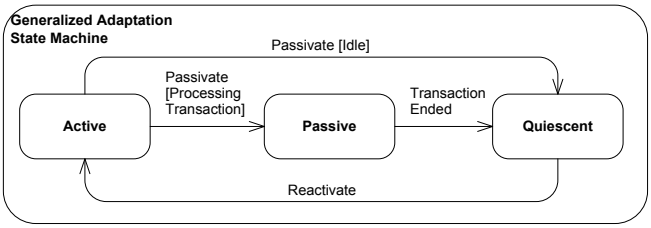


Figure 2. Basic adaptation state machine

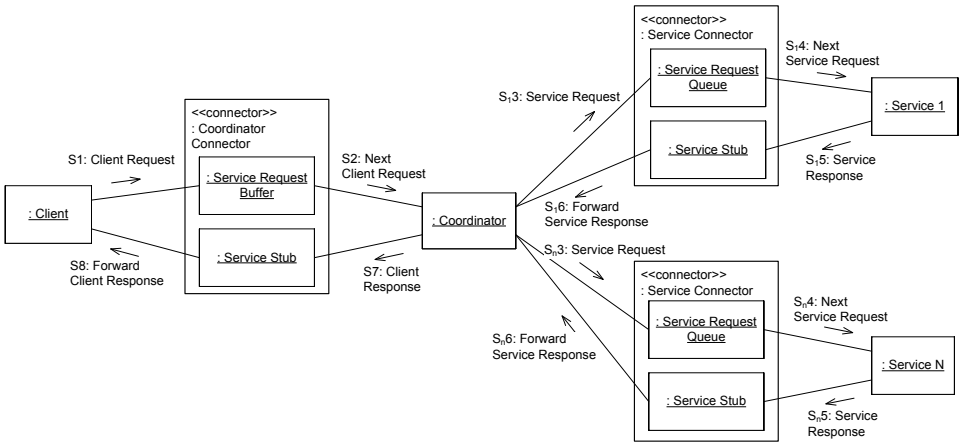


Figure 3. Independent coordination communication diagram

However, for more complicated patterns, there is often some interaction between the two state machines. In this paper, we investigate 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.

7. SOA ADAPTATION PATTERNS

As described in Section 5, a coordination pattern can be characterized by the type of coordination and the degree of concurrency. This paper 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. We make the following assumptions:

- A coordinator component is instantiated for each client.
- A client interacts with a service 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.

In the following sections, we first describe SASSY's Adaptation connectors, followed by the adaptation patterns for three coordination patterns: 1) sequential coordination, 2) concurrent coordination, and 3) combined concurrent/sequential coordination patterns.

7.1 SASSY Adaptation Connectors

The SASSY framework provides 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 section) 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 three software adaptation patterns are described.

7.2 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.

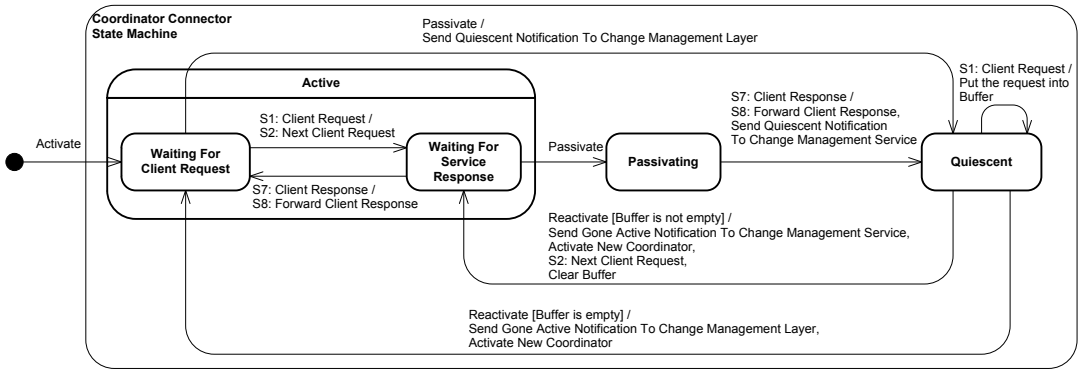


Figure 4. Coordinator connector state machine

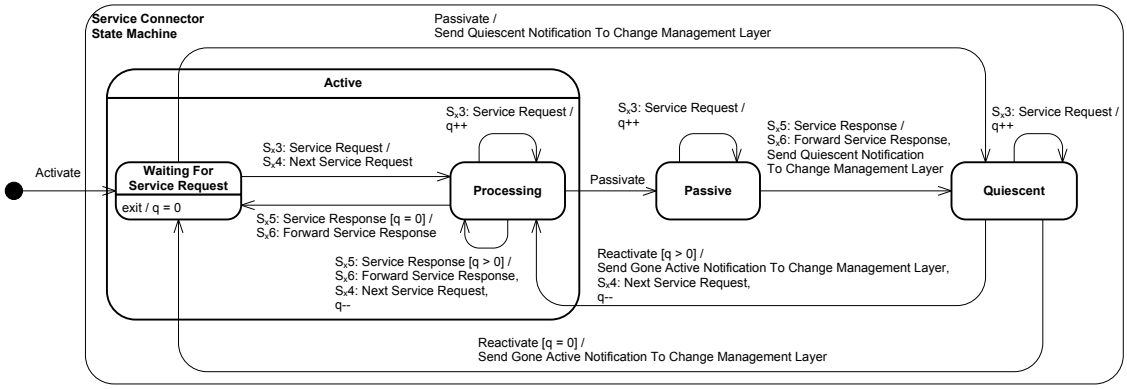


Figure 5. Service connector state machine for a sequential service

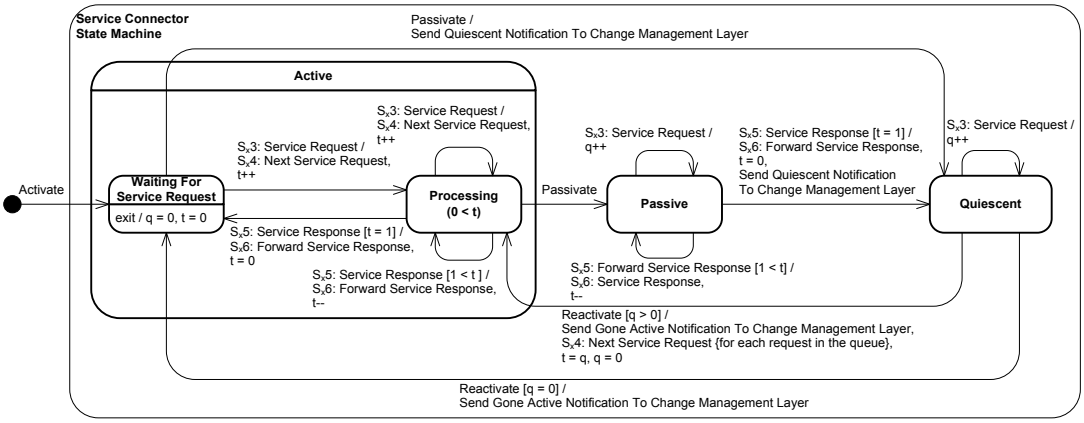


Figure 6. Service connector state machine for a concurrent service

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 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.

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.

7.3 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.

7.4 Combined Coordination Adaptation Pattern

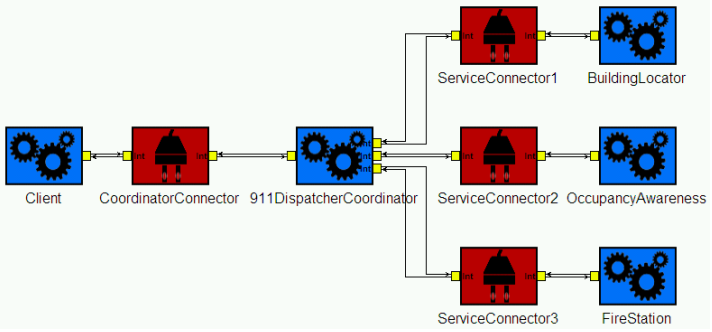
For the combined sequential and concurrent coordination pattern described in Section 5, the operating state machines for the coordinator and service connecters are the same as those in the sequential and concurrent coordination patterns, for the same reasons described in Section 7.3. Therefore, the adaptation connectors can be reused in the independent coordination pattern, regardless of the coordinator's degree of concurrency (sequential, concurrent, or combined sequential/concurrent).

8. ADAPTIVE CHANGE MANAGEMENT

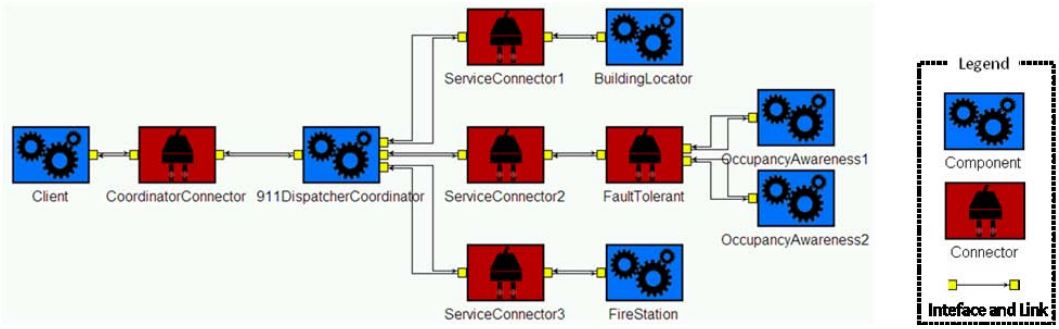
Adaptive change management is provided by a Change Management Model, which is used to establish a region of quiescence [9] so that dynamic adaptation can take place. For each adaptation pattern, the change management model describes a process for controlling and sequencing the steps in which the configuration 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. In the SASSY framework (Section 4), services are discovered by the Goal Management layer.

8.1 Example of Dynamic Software Adaptation

As an example of dynamic software adaptation, consider an emergency response system shown in Figure 7.



a) Initial software configuration



b) Revised software configuration

Figure 7. Dynamic software adaptation in emergency response system

The initial software configuration is shown in Figure 7a before dynamic software adaptation, while the revised configuration after dynamic software adaptation is shown in Figure 7b.The emergency system uses the 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 7b.

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 (Figure 7b). 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 7b). 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.

9. VALIDATION OF SOFTWARE ADAPTATION PATTERNS

The emergency response system example described in Section 8 was modeled using XTEAM [23], which is an architectural modeling and analysis environment. XTEAM provides a structural Architectural Description Language (ADL), xADL [24], with a behavioral ADL, Finite State Processes (FSP) [25], to generate executable system simulations.

In the emergency response system example, we used XTEAM to model the system's structure shown in Figure 7 in xADL, and the behavior of Coordinator and Service Connectors by translating their state machines described in Figures 4-6 into equivalent FSP models. In the simulation, we modeled CM as a component in xADL, which sends the adaptation commands to the adaptation connectors by executing the change management scenario described in Section 8. Since XTEAM does not currently support run-time dynamic adaptation of the software architecture and executable simulation, we simulated the example using behavioral adaptation (Section 3). We therefore developed, in advance, the Fault Tolerant connector and the two Occupancy Awareness services connected with Service Connector 2, 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 0 and the dynamic software adaptation scenario described in Section 8. 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.

The validation consisted 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. For both the above two scenarios, the XTEAM simulation recorded the trace of FSP state transitions for each xADL component in execution logs. Analysis of these logs showed that the above three validation steps 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.

10. CONCLUSIONS

This paper has described how software adaptation patterns can be used in service oriented architectures to dynamically adapt coordinator components and services at run-time. We have developed software adaptation patterns for independent coordinators, which can be state dependent, and either sequential or concurrent services. An independent coordinator, which is instantiated for each client, can be state dependent if the client has multiple interactions with one or more services. Future research will investigate software adaptation patterns with distributed and hierarchical coordination.

In this paper, we assumed services that are stateless. Future research will investigate services that are stateful and participate in a transaction, such as in the two phase commit protocol. The current approach supports long-living transactions, such as check flight availability before booking, in which the long-living transaction is actually executed as two separate independent stateless transactions.

The adaptation patterns are part of the self-architecting SASSY framework. The patterns are described in terms of a three-layer reference architecture for self-management. The adaptation 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.

Future work will consist of investigating additional adaptation patterns for service-oriented architectures and incorporating them into the SASSY framework.

11. ACKNOWLEDGMENTS

This work is partially supported by grant CCF-0820060 from the National Science Foundation. The authors thank João P. Sousa for his comments on an earlier draft.

12. REFERENCES

[1] H. Gomaa, "Building Software Systems and Product Lines from Software Architectural Patterns", ECOOP Wkshp. 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] 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.

[24] 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.

[25] 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.