Model-based Run-Time Software Adaptation for Distributed Hierarchical Service Coordination

Hassan Gomaa, Koji Hashimoto

Department of Computer Science
George Mason University
Fairfax, VA, USA
hgomaa@gmu.edu, kojihashi@gmail.com

Abstract **- Dynamic software adaptation addresses software systems that need to change their behavior at run-time. A software adaptation pattern models how the components that make up an architecture pattern cooperate to change the software configuration at run-time. This paper describes a model-based run-time adaptation pattern for distributed hierarchical service coordination in service-oriented applications, in which multiple service coordinators are organized in a distributed hierarchical configuration.**

Keywords: service-oriented architecture; dynamic software adaptation; model-based software adaptation pattern; hierarchical service coordination adaptation.

I. INTRODUCTION

Dynamic software adaptation addresses software systems that need to change their behavior at run-time [1]. With model-based dynamic software adaptation, models are used to describe and sequence the adaptation of the software architecture and executable system at run-time [2]. A model-based software adaptation pattern defines how the 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. Because control and sequencing is so important in dynamic run-time adaptation, this research focuses on dynamic models, using in particular state machine models and object communication models.

Previous work has described model-based adaptation patterns for distributed component-based systems [2] and service-oriented architectures (SOA) [3][4]. In typical SOA applications, services are self-contained, loosely coupled, and orchestrated by coordination services [8]. This research addresses dynamic adaptation based on SOA coordination patterns. Previous work addressed independent SOA service coordination [3] and transaction-based distributed software adaptation [4], in which there is one service coordinator orchestrating multiple services. This paper extends this research to SOA applications with hierarchical service coordination by describing and validating a dynamic software adaptation pattern for distributed hierarchical service coordination in which a higher-level coordinator communicates with multiple lower-level coordinators.

This paper describes related work in Section II, provides an overview of software adaptation for SOA in Section III, describes in detail the hierarchical service coordination adaptation pattern in Section IV, describes its validation in Section V, and provides concluding remarks in Section VI.

II. RELATED WORK

Dynamic software architectures and dynamic reconfiguration approaches have been applied to dynamically adapt software systems. Research into self-adaptive, self-managed or self-healing systems includes approaches for monitoring the environment and adapting a system's behavior in order to support run-time adaptation [11]. Kramer and Magee [1] describe how a component must transition to a quiescent state before it can be removed or replaced in a dynamic software configuration. Ramirez and Cheng [5] 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. [9] describe an adaptable service connector model, so that services can be dynamically composed. Irmert et al. [10] provide a framework to adapt services at run-time without affecting application execution and service availability. A related research area is dynamic adaptation of software product lines, in which the different software configurations are organized as a product line, with dynamic adaptation from one member configuration to another managed through a feature model [6].

In comparison with the previous approaches, this paper focuses on dynamic self-adaptation in service-oriented architectures. This paper describes a software adaptation pattern for distributed hierarchical service coordination, in order to adapt not only services but also distributed hierarchical coordinator components.

III. SOFTWARE ADAPTATION FOR SOA

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 (also referred to as coordinators) in situations where access to multiple services needs to be coordinated and/or sequenced [3].

 ISBN: 978-1-61208-341-4

A. *Software Coordination and Adaptation*

In SOA systems, loose coupling is ensured by separating the concerns of individual services from those of the coordinators, which sequence the access to the services. As there are many different types of service coordination, it is helpful to develop service coordination patterns to capture the different kinds of service coordination. For each of these coordination patterns, there is a corresponding dynamic adaptation pattern [3]. The software adaptation patterns described in this paper were developed as part of Self-Architecting Software Systems (SASSY), which is a model-driven framework for run-time self-architecting and re-architecting of distributed service-oriented software systems [8].

B. *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 [2][3]. 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. To enable adaptation patterns, as well as the corresponding code that realizes each pattern, to be more reusable, adaptation state machines are encapsulated in software adaptation connectors as discussed next.

C. *Software Adaptation Connectors*

Software adaptation connectors [3][4] are used to encapsulate adaptation state machines so that adaptation patterns can be more reusable. The adaptation patterns described in this paper include two different types of adaptation connector, *coordinator connector* and *service connector*. The goal of an adaptation connector is to separate the concerns of an individual component (service or coordinator) from its dynamic adaptation. An adaptation connector models the adaptation mechanism for its corresponding service or coordinator. An adaptation connector behaves as a proxy for a component, such that its clients can interact with the connector as if it were the component, as shown in Fig. 1.

IV. HIERARCHICAL SERVICE COORDINATION ADAPTATION PATTERN

In the hierarchical service coordination adaptation pattern for SOA, a higher-level coordinator orchestrates lower-level coordinators, whereas each of the lower-level coordinators is responsible for distributed service coordination. The communication diagram depicted in Fig. 1 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.

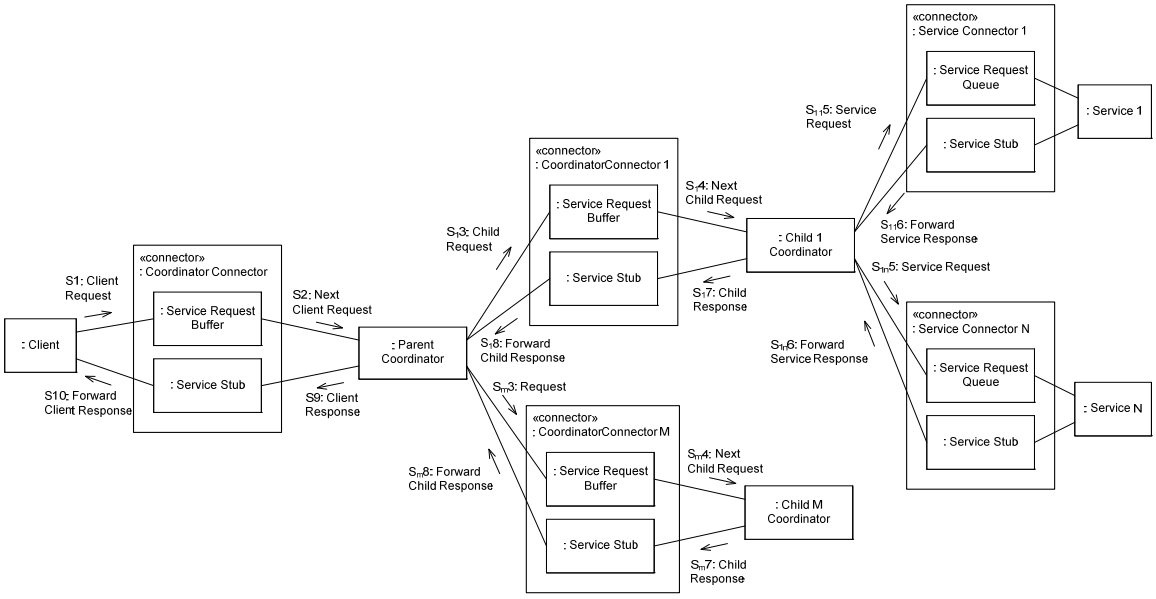

Fig. 1 Hierarchical service coordination communication diagram

An example of hierarchical coordination is a client trip request to the parent coordinator consisting of an airline reservation, a hotel reservation and a car reservation. The parent coordinator decomposes the client request into three smaller requests, which are sent to child coordinators for airline, hotel, and car reservations using a combination of sequential and concurrent coordination (e.g., hotel reservation followed by concurrent hotel and car reservations. Each child coordinator interacts with several individual services (e.g., airline companies) in order to select the most appropriate service. The parent coordinator receives the child coordinator responses and then responds to the client.

The hierarchical service coordination adaptation pattern is organized as follows:

- A parent coordinator is instantiated for each client.
- Two or more child coordinators are instantiated for each parent coordinator.
- A client interacts with a parent coordinator using synchronous message communication; thus, it sends a new request only when it receives a response to its previous request.
- A parent coordinator receives a client request and decomposes it into smaller requests, which are sent to child coordinators. The parent coordinator communicates with the child coordinators either sequentially or concurrently.
- A child coordinator communicates with multiple services sequentially or concurrently. It uses independent service coordination for stateless services [3] and transaction based communication (e.g., two phase commit protocol) for stateful services [4].
- The parent coordinator responds to the client after it has received responses from each of the child coordinators.

To address hierarchical service adaptation it is necessary to consider adaptation of parent coordinators, adaptation of child coordinators, and adaptation of individual services.

A. Dynamic Run-time Adaptation for Hierarchical Coordination

Using the hierarchical service adaptation pattern, the parent coordinator component can be removed or replaced after it has received all the responses from the child coordinators 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 its 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.

The solution involves one coordinator connector for the parent coordinator and one coordinator connector for each child coordinator, as depicted in Fig. 1. Each connector encapsulates the adaptation state machine for its corresponding coordinator. This is possible because a connector tracks the states of its corresponding coordinator, since it receives (and forwards) each upstream message sent to the coordinator and each downstream message sent by the coordinator.

Figures 2 and 3 depict the adaptation state machines executed by the coordinator connectors for the parent coordinator and the child coordinator respectively. Applying separation of concerns, parent and child coordinators deal with coordination decisions while their corresponding connectors address adaptation decisions. Thus, the parent coordinator connector encapsulates the adaptation state machine of the parent coordinator it communicates with, whereas the parent coordinator interacts with multiple child coordinators via their coordinator connectors.

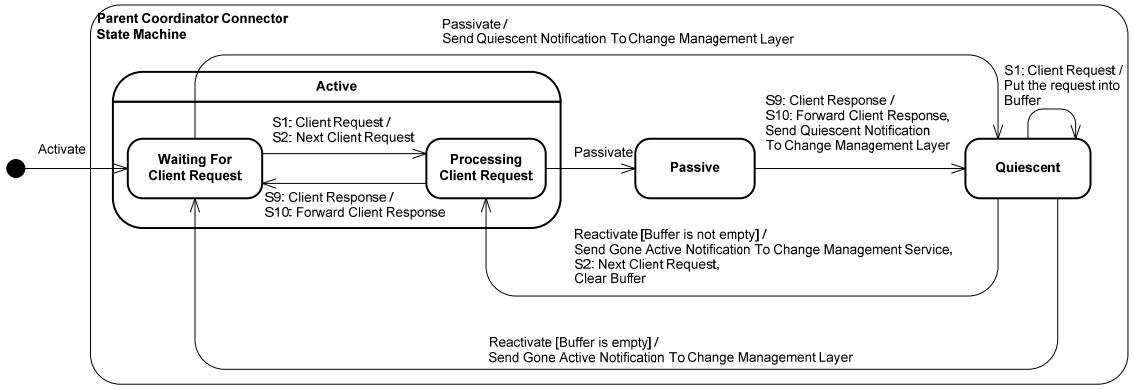

Fig. 2 Parent coordinator adaptation connector state machine

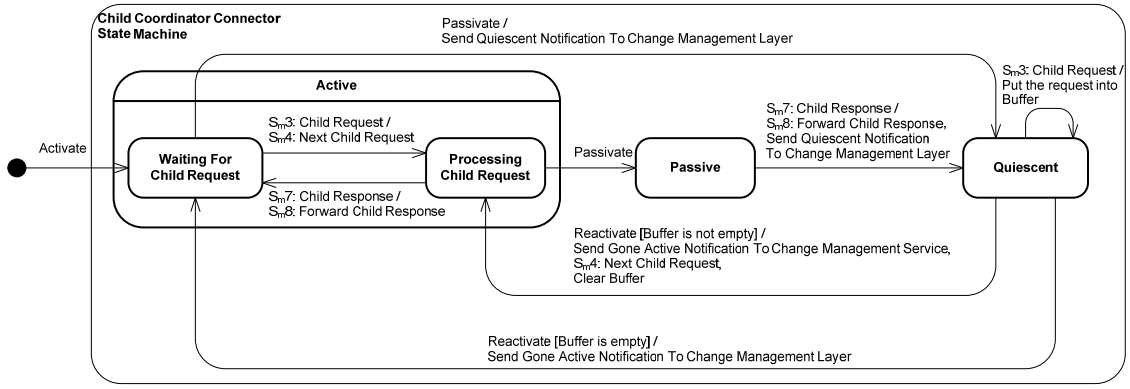

Fig. 3 Child coordinator adaptation connector state machine

B. *Adaptation of Parent Coordinator*

As described in the previous subsection, the parent coordinator connector encapsulates and executes the adaptation state machine for the parent coordinator, shown in Fig 2. (Because of this, the state names reflect the states of the coordinator and not the connector). There are three main states, Active, Passive, and Quiescent. In the Active state, the coordinator is operating normally and its state machine is in one of the two substates of the composite Active state. As shown in Fig. 2, the parent coordinator connector is initially in Waiting for Client Request substate. When it receives a request from the client (message S1 in Fig 1), the connector transitions to Processing Client Request substate (event S1 in Fig 2) and forwards the next client request to the Parent Coordinator (action S2 in Fig 2 and corresponding outgoing message S2 in Fig 1). The parent coordinator then interacts with the child coordinators. When the parent receives the responses from all its children, it sends the client response (message S9 on Fig.1) to the connector. The parent connector transitions back to Waiting for Client Request state (event S9 on Fig. 2) and forwards the response to the client message (action S10 on Fig.2 and corresponding message S10 on Fig. 1).

To initiate dynamic adaptation of the parent coordinator, a Change Manager (CM) [2][3], which is part of the SASSY adaptation framework (see IIIA and [8]), sends the Passivate command to the parent coordinator connector. If the connector is in the Waiting for Client substate (Fig 2), it transitions directly to the Quiescent state; the action is to send a quiescent notification message to CM. Alternatively, if the connector is in the Processing Client Request substate when it receives a Passivate command, it transitions to the Passive state because the parent coordinator is still interacting with the child coordinators to complete the client request. When the connector receives the Client Response (message S9 on Fig.1) from the Parent Coordinator (indicating that the coordinator has completed the client request), it transitions to Quiescent state (event S9 on Fig. 2). The actions are to forward the response to the client (action S10 on Fig. 2 and message S10 on Fig. 1) and to send a quiescent notification to the CM. In Quiescent state, the parent coordinator is idle and ready to be replaced. If a new client request arrives in Quiescent state, the request is stored in a buffer. After the coordinator has been replaced, CM sends a Reactivate command to the coordinator. If the buffer is empty, the connector transitions to Waiting for Client Request. Otherwise, the connector transitions from Quiescent state to Processing Client Request and sends the buffered client request to the reactivated parent coordinator (action S2 on Fig. 2 and corresponding message on Fig. 1).

C. *Adaptation of Child Coordinator*

Each child coordinator connector in Fig. 3 encapsulates the state machine for its corresponding child coordinator. It receives child requests from the parent coordinator and forwards these to the child coordinator. The connector receives child responses from the child coordinator and forwards these to the parent coordinator. When the child connector receives a Passivate command from CM, it transitions to Quiescent state (if it is waiting for a client request) or to Passive state (if it is processing a child request). In the latter case, when the connector receives the child response from the child coordinator, it transitions to Quiescent state and forwards the child response to the parent coordinator.

If the child coordinator coordinates stateless services independently, independent coordination adaptation patterns [3] are applied to the adaptation of a service in the hierarchical coordination pattern, as depicted in Fig. 3 and described above. If a child coordinator orchestrates stateful services using a Two-Phase Commit Protocol, the two-phase commit coordination adaptation pattern described in [4] is applied.

D. *Adaptation of Services*

A concurrent service services multiple client requests concurrently. The adaptation state machine for a concurrent service connector is shown in Fig. 4. The service connector receives service requests from a child coordinator as well as from other clients and forwards them to the service. For a concurrent service, the service can be

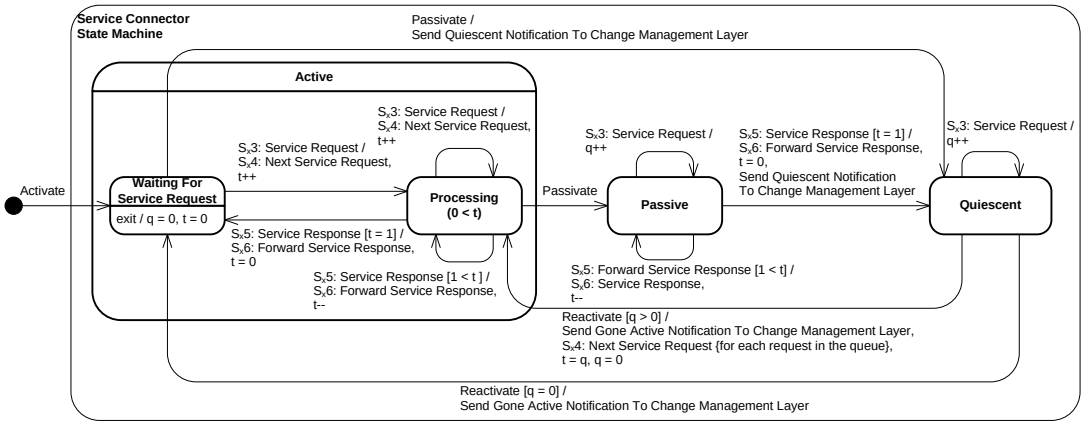

Figure 4 Concurrent service adaptation connector state machine

removed or replaced after it has completed the service requests it has received via the service adaptation connector. The service connector keeps a count t of the requests currently being executed by the service, incrementing the count when a new request is sent to the service and decrementing the count when the response is received and then forwarded to the appropriate client or child coordinator.

If a passivate command is received from CM, the adaptation connector transitions to Passive state if busy, where it waits for the current service requests to complete. New service requests are queued in a service request queue, which is managed by a queue counter q. When the current service requests are completed, the adaptor transitions to Quiescent state. When it receives the reactivate command from CM, the service connector sends the queued service requests to the replacement service and transitions to Processing state.

V. Validation of Hierarchical Service Coordination Adaptation Pattern

The SOA adaptation patterns were validated using the SASSY dynamic run-time software adaptation framework [3][8]. The prototype implementation of the SASSY framework is based on Web services and was developed using open-source SOA frameworks, namely Eclipse Swordfish and Apache CXF. A prototype emergency response system was developed using this framework. Using this framework, validation of a service adaptation pattern consists of executing change management scenarios, performing the run-time adaptation from one configuration to another, and resuming the application after the adaptation.

For the validation of the hierarchical service coordination adaptation pattern, the emergency response system consisted of a region (parent) emergency coordinator that assigned emergency requests to three district (child) emergency coordinators, which each coordinated their local fire engine and ambulance services. Separate adaptation scenarios were executed for the parent and child coordinators and were monitored using execution traces for the parent and child adaptation connectors. The execution trace for the parent coordinator connector is shown in Fig. 5, during which adaptation of the parent coordinator is carried out. The trace depicts the sequence of states the connector transitions through, starting in Idle state. The connector receives a client request, transitions from Idle to Processing state, and sends the new transaction to the parent coordinator. It then receives a Passivate command from CM and transitions to Passive state. When the transaction completed response is received from the parent coordinator, the connector transitions to Quiescent state. In this state, the parent coordinator can be replaced. While in Quiescent state, a new request arrives at the connector from the client and is queued. After adaptation is completed, the connector receives the Reactivate command from CM, transitions to Processing state, and sends the queued request to the new parent coordinator. After the transaction is completed, the connector transitions back to Idle state.

An execution trace for adaptation of a child coordinator is shown in Fig. 6. This scenario shows that child coordinator connector transitions to Processing state after receiving a request from the parent coordinator, which it then sends to the child coordinator. After receiving a Passivate command, the connector transitions to Passive state. When the connector receives the completion message from the child coordinator, it transitions to Quiescent state. In this state, the child coordinator can be adapted. While in Quiescent state, the connector receives a new request, which it queues. After receiving the Reactivate command, the connector then transitions to Processing state and sends the queued request to the child coordinator. When this request is completed, the child coordinator connector transitions to idle state.

In summary, the validation scenarios confirm that the parent and child coordinator adaptation connectors behaved as specified, transitioning from Processing to Passive to Quiescent states and then back to Processing state, while sending and receiving the expected messages.

 ISBN: 978-1-61208-341-4

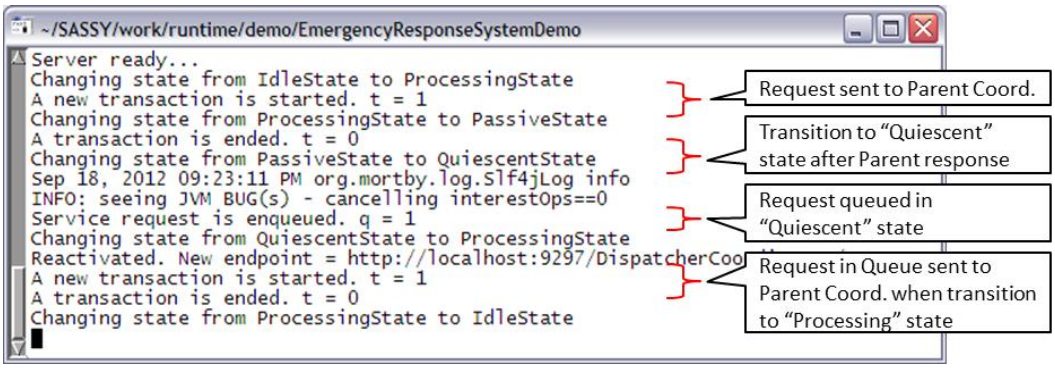

Fig. 5 Execution trace of Parent Coordinator Connector in hierarchical service coordination

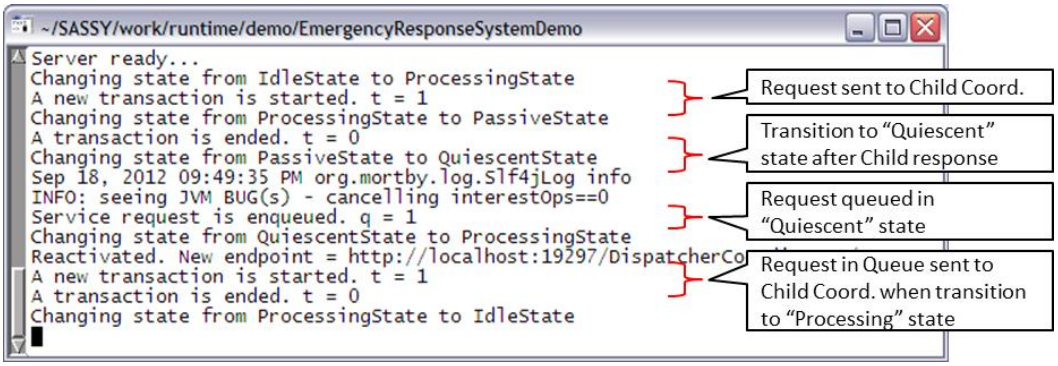

Fig. 6 Execution trace of Child Coordinator Connector in hierarchical service coordination

VI. CONCLUSIONS

This paper has described how software adaptation can be applied to hierarchical coordination in service oriented systems. The main contributions of this paper are:

1. Adaptation pattern for distributed hierarchical service coordination, which can operate with either stateless or stateful services. For hierarchical service coordination with distributed transactions, the pattern corresponds to the compound transaction pattern [6], in which a compound transaction is decomposed into two or more atomic transactions.
2. Design of adaptation connectors for distributed service coordination. Adaptation connectors encapsulate the adaptation state machines for the adaptation pattern to separate the concerns of an individual service or coordinator from software adaptation.

Future work consists of investigating performance issues of dynamic adaptation for service-oriented architectures, developing additional adaptation patterns, and considering recovery from service failure.

ACKNOWLEDGMENTS

This research was partially supported by grant CCF-0820060 from the National Science Foundation. The authors gratefully acknowledge the contributions of D. Menasce, S. Malek, J. Sousa, N. Esfahani, and J. Ewing to the SASSY project.

REFERENCES

[1] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Transactions on Software Eng., Vol. 16, No. 11, 1990, pp. 1293-1306.

[2] H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Springer Verlag LNCS 4199, 2006, pp 1-15.

[3] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. Menasce "Software Adaptation Patterns for Service-Oriented Architectures", Proc. ACM Symp. on Applied Computing, March 2010, pp. 462-469, Sierre, Switzerland.

[4] H. Gomaa and K. Hashimoto, "Dynamic Self-Adaptation for Distributed Service-Oriented Transactions", Proc. SEAMS Symposium, Zurich, Switzerland, June 2012, pp. 12-20.

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

[6] H. Gomaa and K. Hashimoto, "Dynamic Software Adaptation for Service-Oriented Product Lines", in Proc. Intl Wkshp on Dynamic Software Product Lines, Munich, Germany, August 2011.

[7] H. Gomaa, "Software Modeling and Design", Cambridge University Press, 2011.

[8] D. Menasce, H. Gomaa, S. Malek, and J. Sousa, SASSY: A Framework for Self-Architecting Service-Oriented Systems", IEEE Software, Vol. 28, No. 6, 2011, pp. 78-85.

[9] 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.

[10] F. Irmert, T. Fischer, and K. Meyer-Wegener, "Runtime adaptation in a service-oriented component model", Proc. SEAMS Symposium, May 2008, pp. 97-104.

[11] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge", Proc Intl. Conference on Software Engineering, Minneapolis, MN, May 2007, pp. 259-268.

 ISBN: 978-1-61208-341-4