QoS Architectural Patterns for Self-Architecting Software Systems

Daniel A. Menascé, João P. Sousa, Sam Malek, and Hassan Gomaa
Department of Computer Science
George Mason University
Fairfax, VA 22030, USA
{menasce,jpsousa,smalek,hgomaa}@gmu.edu

ABSTRACT

This paper discusses the automated application of architectural patterns for tuning the quality of service of service-oriented software. The paper first presents an overview of prior work in self-architecting, SASSY, and a motivating example in the emergency response domain. After summarizing the heuristic used for self-architecting, the paper discusses a number of architectural patterns and the corresponding quantitative models for two concrete aspects of quality of service: availability and response time. A case study illustrates the role of patterns in the application of the self-architecting heuristic to the motivating example.

Categories and Subject Descriptors

C.4 [**Modeling Techniques**]: Experimentation; D.2.11 [**Software Architectures**]: Patterns; D.4.8 [**Performance**]: Stochastic analysis; G.1.6 [**Optimization**]: Global optimization

General Terms

Performance, Experimentation

Keywords

Service-Oriented Architecture, software architectures, autonomic computing, quality of service, software optimization, architecture patterns

1. INTRODUCTION

Tuning the architecture of software systems is hard. Once an application's logic is defined, an architect is concerned with choosing architectural styles and patterns that promote desired system qualities. Unfortunately, choosing a pattern that promotes certain aspects of quality normally has a negative effect on some other aspects of quality [4, 14]. The task of an architect is to make tradeoffs that reflect the priorities of stakeholders. This task is especially complex for large systems, where the architectural decisions made to optimize certain features may cause adverse effects in other parts of the system.

Situated systems present an additional challenge since both the specific quality goals and the available means, such as service providers, may only become known on the spot, when the system is needed to address a specific situation. For example, in the emergency response domain, different means may be employed or given different priorities, e.g., to fight a threat vs. to evacuate the affected area, depending on the kind of threat and on the number of potential victims.

Automation is a promising approach to help architects address such challenges. In service-oriented systems, automatic service discovery brings scalability and effectiveness to the process of selecting the set of service providers that will deliver the preferred Quality of Service (QoS) to the system. Furthermore, once service discovery is automated, it can be supported by a run-time infrastructure thus making *self-healing* and *self-adaptation* possible [1, 9, 25]. Further improvements to QoS may be achieved by leveraging the automatic generation of service coordinators based on high-level descriptions of service flow. For example, JOpera automatically tunes coordination parameters, such as thread replication, taking into account system characteristics and load [23].

Although steps in the right direction, the mechanisms above fall short of providing the kinds of QoS improvements that expert application of architectural patterns can deliver. Work on architectural patterns is well-know in the art [8]. However, this paper is about automated techniques for applying QoS architectural patterns.

Recent work in *self-architecting* attempts to bridge that gap by automating the application of architectural patterns [5, 18]. This paper extends prior work in SASSY (Self-Architecting Software SYstems), where the automatic application of patterns is informed by QoS goals defined by system stakeholders [18]. SASSY uses efficient and scalable search heuristics to identify the optimal patterns, making it possible to perform self-architecting both at system deployment and at run time for purposes of self-adaptation [20].

The contributions of this paper are, first, the analysis of a number of patterns suitable for self-architecting, and second, quantitative models to derive the overall QoS of a system after the application of several such patterns.

In the remainder of this paper, Sections 2 and 3 respectively present an overview of SASSY and a motivating example in the emergency response domain. Section 4 summarizes the self-architecting heuristic, and Section 5 presents

ICAC'10, June 7–11, 2010, Washington, DC, USA.

the patterns currently supported in SASSY and their corresponding QoS models. Section 5 also illustrates the application of such patterns with a case study, and Section 6 compares with related work. Finally, Section 7 summarizes the main points of this paper.

2. OVERVIEW OF SASSY

SASSY takes optimization of QoS to a level beyond the optimal selection of service providers by automatically generating a number of candidate architectural patterns to replace each service and searching for the pattern that best serves user-defined QoS goals.

Broadly, SASSY provides a framework and infrastructure for self-adaptation, self-healing, self-optimization and evolution of service-oriented software systems.

Figure 1 shows the relationships between the top-level components and models used in the SASSY framework. This framework is consistent with the classical monitoring and feedback control loop for software self-management (e.g., [17]) and builds on prior work by the authors and others [13, 19]. Specifically, starting at the bottom left, the Monitoring Support component leverages probes on the SOA implementation stack and transforms those observations into measures of QoS at system level via monitoring and gage services. Observations of service failure and QoS are reflected on the System Service Architecture, at the center, thus keeping the latter consistent with the status of the running system. The Adaptation Support component, at the bottom right, manages the operational aspects of effecting changes to the running system based on architectural descriptions of those changes.

What is distinctive about SASSY is the self-architecting and re-architecting component and associated models, at the top of the figure. Service Activity Schemas (SAS) describe the features and logic of the application and are written by domain experts in a graphical notation similar to Business Process Modeling Notation (BPMN) [2] (more in Section 3). BPMN is too loosely defined and does not have the tight semantics and the QoS annotation capabilities as SAS [7].

In contrast to BPMN, SAS can be annotated with QoS goals expressed in the form of *utility functions*. Utility functions originate in economics and have been extensively used in autonomic computing to assign a value to the usefulness of a system based on its attributes (e.g., [1, 25]). For example, a utility function associated with system availability, a might be:

$$U_{\text{Availability}}(a) = \begin{cases} 0 & a < 0.9 \\ 0.5 & 0.9 \leq a < 0.95 \\ 1 & 0.95 \leq a < 1 \end{cases} \quad (1)$$

This expresses that the system is not useful (utility 0) if its availability is less than 0.9, it is only half useful for an availability in the interval [0.9,0.95), and it is entirely adequate (utility 1) when the availability equals to or exceeds 0.95.

SASSY is innovative in the way that it supports the expression of such QoS goals. This is done by highlighting paths of interest in SAS, so called Service Sequence Scenarios (SSS), and specifying *end-to-end* goals along those paths in the form of utility functions as above. The overall utility of a system is defined as a composition of the individual utilities according to their relative importance. Such utility functions and their composition are defined by domain experts in consultation with the system stakeholders.

SASSY uses analytical models to derive the end-to-end QoS attributes, and hence the utility of the system, as functions expressed in terms of the QoS provided by each service along each SSS (more in Section 5.6). These analytical models play a key role in the self-architecting process.

The self-architecting and re-architecting component automatically generates a near-optimal System Service Architecture, and maintains that optimality in the face of changes detected by the Monitoring Support—self-healing and adaptation—and in the face of changes made by users in the SAS and SSS—system evolution. For that, it focuses on a set of SSS with greater room for improving their contribution to the overall utility. Then it generates variations to the system architecture by replacing each service along an SSS with candidate architectural patterns that are functionally equivalent but improve some aspect of QoS. Prior work has evaluated the effectiveness and scalability of several heuristics for generating candidate replacements and searching for the optimal architecture [20].

This paper describes the progress made to further improve the self-architecting process by leveraging knowledge about which architectural patterns promote or detract which QoS attributes.

3. MOTIVATING EXAMPLE

For illustrating our research, we rely on an emergency response system intended for use by government agencies to automatically detect, respond, and manage various crises. The system targets SOA enabled smart spaces, which are comprised of various types of sensors, such as smoke detectors, fire sprinklers, and cameras. Each sensor exposes its functionality via a discoverable web service, which complies to the specification of the service type from the domain ontology. The emergency response crew is also equipped with a variety of SOA enabled devices (e.g., PDAs) and platforms (e.g., fire engines), which allow the crew to communicate their status, coordinate activities, and operate remote sensors. A system such as this is innately dynamic and intended to deal with a variety of emergency scenarios. The SAS language provides a suitable method of specifying the system's requirements under different emergency conditions (e.g., fire, earthquake, hurricane). SASSY in turn selects the appropriate SAS model based on the situation at hand. Below we describe a SAS model constructed for a fire emergency at a smart building, which is further used in this paper to illustrate self-architecting and QoS analysis in SASSY.

Figure 2a shows a subset of the modeling constructs available in the SAS language. *Events* are messages exchanged between two separate entities, and *Gateways* manage the flow of control. Supported gateways include *Conditional-Or*, *Fork*, and *And-Join*, with the same meaning as in BPMN. The language distinguishes local activities from *service usages*, i.e., activities performed by external entities as services provided to the requester. Furthermore, local activities may be *composite*, which enable hierarchical decomposition, or *tasks*, which cannot be decomposed any further. Activities and service usages are represented by rectangles with round corners, where composites show a plus sign, for bringing up the internal composition, and service usages show a server icon. Communication with a service is via *input* and *output*

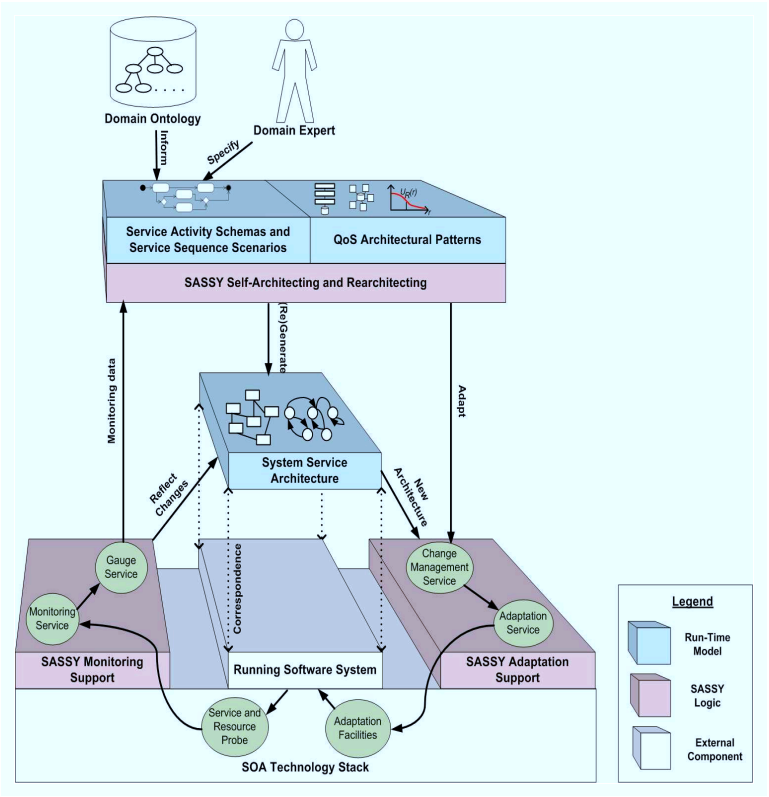

Figure 1: High-level view of the SASSY framework.

events, which are, respectively, indicated as white and black envelopes on the border of the corresponding rectangle.

Figure 2b shows an example SAS made by a city for monitoring fire emergencies in public buildings equipped with smoke detectors and fire sprinklers. If smoke is detected, the building sends a *smokeDet* event to the *911 Dispatcher*, which may issue events targeted at the *Police Station*, *Hospital*, and *Fire Station*. These organizations in turn may take further actions and coordinate their activities with other entities. In this example, the *Fire Station* may decide to turn on the fire sprinklers in the building.

Figure 2c shows the SAS for *911 Dispatcher*, where the reception of a *smokeDet* event starts two parallel threads of control. First, an attempt is made to contact the building occupants using the *Emergency Phone System*, described in a local composite activity. Second, an external *Building Locator* service is used to find the physical address of the incident. Once both have finished, if contact was made with the occupants, the phone call is forwarded to an operator; otherwise, an *investigate* event will be sent to the *Police Station*.

Following the *Building Locator* service, two other external services, *Occupancy Awareness* and *Building Category Finder*, are requested to determine the number of occupants in the building and the type of the building. If there are occupants in the building but no contact was possible, a *reqHelp* event is sent to the *Hospital* for dispatching an ambulance to the scene. Finally, depending on the type of the building, either *reqMultFS* or *reqSingFS* events are sent to the *Fire Station*, indicating a request for multiple or single fire engines, respectively. Organizations such as the *Hospital*, *Police Station*, and *Fire Station*, might develop their SAS models independently, the details of which are not shown in here for brevity.

The purpose of *Service Sequence Scenarios (SSS)* is the specification of QoS objectives, which play a key role in the generation and evaluation of candidate architectures. Formally, SSS are sub-graphs of SAS that are well-formed, in the sense of satisfying all the syntactic constraints of a complete SAS. With the current tool support, SSS are accessible via dashed rectangles at a corner of an SAS (see Figure 2c). Figure 2d shows what happens when the Availability SSS is selected: the corresponding sub-graph is highlighted, while the rest of the SAS is grayed out. Specific end-to-end QoS objectives associated with an SSS are captured in a property sheet. For example, the SSS in Figure 2d has a utility function that depends on that SSS availability. This utility function is composed with the utility function of all the other SSS and is used during the self-architecting process to determine the system's ability in satisfying the user's QoS requirements (see Section 4).

A formal specification of SASSY's activity-oriented requirements modeling language can be found in [7].

4. SASSY'S SELF-ARCHITECTING AND RE-ARCHITECTING

Figure 3 shows the structural view of the base architecture automatically generated by SASSY for the 911 Dispatcher of Fig. 2(c). SASSY automatically generates a base architecture from an SAS specification. In this base architecture, each service type is represented by a single compo-

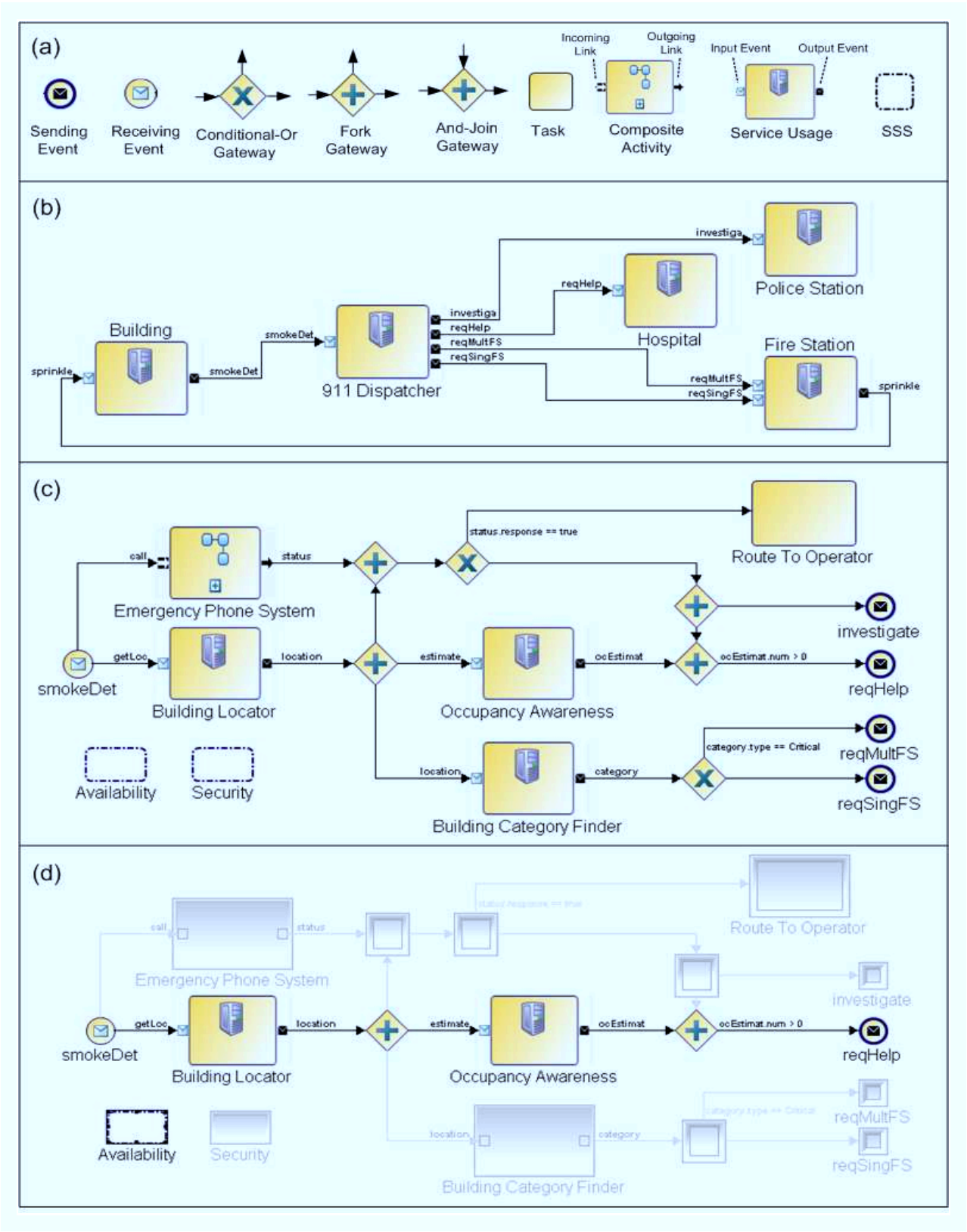

Figure 2: Requirements for a fire emergency response scenario in SAS: (a) SAS language constructs; (b) a high-level SAS describing the interactions between a number of agencies in response to a fire emergency; (c) 911 Dispatcher's internal coordination (behavior) described in the SAS language; and (d) an availability SSS defined on a specific sequence of interactions in 911 Dispatcher.

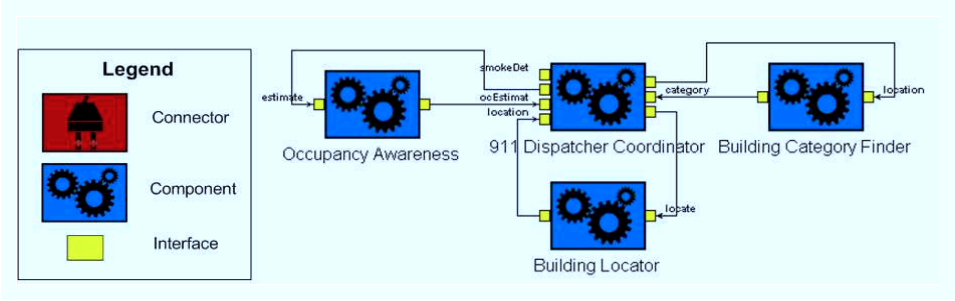

Figure 3: Structural view of the base architectural model generated for the 911 Dispatcher of Fig. 2.

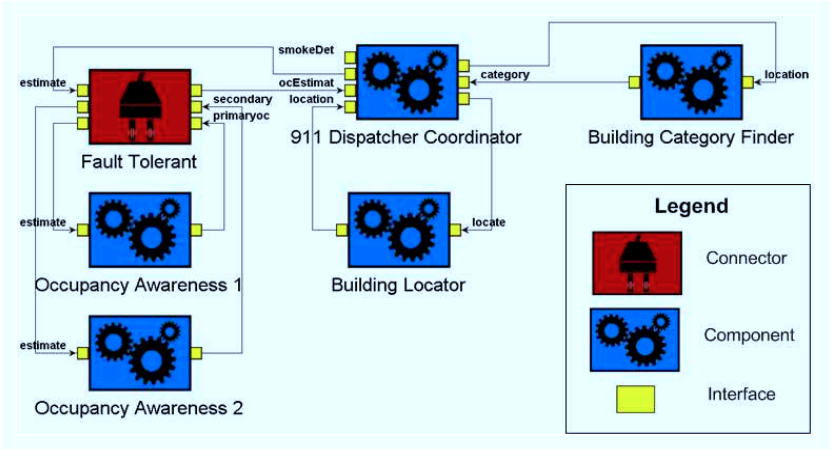

Figure 4: Structural view of an adapted architecture derived from the base architectural model depicted in Fig. 3.

nent. Moreover, a component that carries out the coordinator logic in the SAS is added to the architecture. This component coordinates the communication between the other components. Note that unlike services, activities are not represented in the architectural model that is used for optimizing the software system. This is because activities correspond to locally available libraries and resources that are not necessarily discoverable, and thus are outside the scope of automatic (re)architecting.

The base architecture initially generated by SASSY is not necessarily the best architecture that meets the QoS requirements of the application—expressed as a utility function. Through the process outlined in this section, SASSY searches and finds an architecture, along with a corresponding set of service providers, that maximizes the utility function for the application.

Thus, the optimization problem addressed by SASSY deals with the issue of finding an architecture and a set of service providers (SPs) that implement the service types in the SAS in a way that optimizes the global utility function U_g for a given SAS. This problem is clearly NP-complete. Therefore, SASSY uses a heuristic-based search technique that works as described below.

The search used by SASSY works by replacing **QoS architectural patterns** (see Section 5) for other QoS architectural patterns. Our work assumes that any architectural pattern is totally, and not partially, replaced during the search by another QoS architectural pattern that has the same interface with the rest of the architecture as the pattern being replaced. For example, the *Occupancy Awareness* component in the architecture of Fig. 3 is replaced by a composition of *Occupancy Awareness 1* and *Occupancy Awareness 2* connected by the *Fault Tolerant* connector as illustrated in Fig. 4. It should be noted that the interface with the rest of the architecture was preserved in this replacement. Single components in the architecture are trivial cases of basic architectural patterns and are referred as such in the description below. In what follows, A_0 in Step 1 stands for either the base architecture generated by SASSY or for an instantiated architecture that needs to be re-architected because of the need to adapt.

- Step 1. Start with architecture A_0 and corresponding service selection Z_0 of service providers for the service types of A_0.
- Step 2. Identify the SSSs with the lowest contribution towards overall utility. How many SSSs to consider is a parameter of the heuristic.
- Step 3. Find a neighborhood $\mathcal{N}$ of architectures derived from A_0 by replacing QoS architectural patterns in A_0 by other candidate QoS architectural patterns that improve the utility of the SSSs identified in Step 2 (see Section 5).
- Step 4. Perform a near-optimal service provider allocation for each architecture in $\mathcal{N}$. This is also an

NP-complete problem for which SASSY uses a heuristic described in [21].

- Step 5. Compute the global utility U_g for each architecture in $\mathcal{N}$ and pick the architecture A_{best} with the largest utility in $\mathcal{N}$.
- Step 6. If the utility of A_{best} represents a "good enough" improvement over the previous value of the global utility, stop and return A_{best}. Otherwise, make A_0 equal to A_{best} and go to step 2.

This optimization problem may be modified by adding a cost constraint. In the cost-constrained case, one assumes that there is a cost associated with each SP for providing a certain QoS level. The optimization approach used by SASSY is driven by QoS Architectural Patterns for software self-adaption. These are the main focus of this paper and are discussed below.

Once the architecture is deployed, there may be a need to re-architect due to changes in the environment (e.g., failures of service providers or changes in their QoS characteristics). SASSY uses the same approach described in this section to automatically generate a new architecture. Once that architecture is generated, SASSY's run-time infrastructure effectuates the change through a run-time adaptation process described in details in [12].

A detailed description and evaluation of the heuristic approach used by SASSY in its self-architecting and re-architecting process can be found in [20].

5. QOS ARCHITECTURAL PATTERNS

A QoS architectural pattern $\mathcal{P}$ is defined by the tuple $(S, B, \mathcal{Q})$ where

- S is the structural view of $\mathcal{P}$, which in our case is expressed in a modified xADL [6]. The structural view consists of a set of one or more components C_j ($j = 1, \cdots, C$), connected by zero or more connectors. Each component is associated to a service provider when the architecture is instantiated. Each service provider has associated with it various QoS metrics. Let $v_{m,c}$ be the value of the QoS metric m associated with component c.
- B is the behavioral view of $\mathcal{P}$, which in our case is expressed using Finite State Processes (FSP) [16].
- $\mathcal{Q} = \{(m_i, \mathcal{M}_i) \mid i = 1, \cdots, n\}$ is a set of n QoS metrics m_i, and their corresponding analytical models $\mathcal{M}_i$. A model $\mathcal{M}_i$ can be viewed as a function

$$m_i = \mathcal{M}_i(\vec{V}_1, \cdots, \vec{V}_C) \tag{2}$$

where $\vec{V}_j = (v_{m_1,C_j}, \cdots, v_{m_n,C_j})$, for $j = 1, \cdots, C$. The vector $\vec{V}_j$ represents the set of values of the metrics $m_1, \cdots, m_n$ for component C_j.

The following subsections provide examples of various QoS Architectural Patterns using the notation described above. The structural and behavioral view of each pattern will be described in words and not in xADL or FSP due to space limitations. For illustration purposes, and due to space considerations, the examples below focus on two metrics: a for availability and e for execution time.

5.1 Basic Pattern

This is the simplest possible pattern in terms of structure. It consists of a single component c and no connectors. Its behavior corresponds to asynchronous message-passing. The availability of the basic pattern reflects the probability that it is available to receive the message and its execution time reflects the time it takes to act on the message received.

The set $\mathcal{Q} = \{(a, \mathcal{M}_a), (e, \mathcal{M}_e)\}$ for this pattern is such that

$$\begin{aligned} a &= \mathcal{M}_a(v_{a,c}) = v_{a,c} \\ e &= \mathcal{M}_e(v_{e,c}) = v_{e,c} \end{aligned} \tag{3}$$

Eq. (3) simply says that the value of all QoS metrics for the pattern correspond to that of its component.

5.2 Fault-Tolerant First-to-Respond

This pattern consists of C components $C_1, \cdots, C_C$ and a connector that receives requests and sends them in parallel to all C components. All components process the request and send their replies to the connector, which replies to its requester as soon as the first component replies. It is assumed in what follows that the C components fail independently of one another.

The availability model $\mathcal{M}_a$ for this pattern is

$$a = \mathcal{M}_a(v_{a,C_1}, \cdots, v_{a,C_C}) = 1 - \prod_{j=1}^{C}(1 - v_{a,C_j}) \tag{4}$$

Eq. (4) indicates that the availability of the QoS pattern is 1 minus the unavailability of the set of all C components.

Intuitively, the execution time for this pattern is the minimum execution time among the components that are up. A probabilistic model of $\mathcal{M}_e$ consists of the sum of all the probabilities that a given configuration of components is available multiplied by the minimum execution time of the components that are up. We provide expressions for $C = 2$ and $C = 3$ and then generalize for any value of C. The expression for $C = 2$ is

$$\begin{aligned} e = \frac{1}{a}\,[&v_{a,C_1}(1 - v_{a,C_2})v_{e,C1} + v_{a,C_2}(1 - v_{a,C_1})v_{e,C2} + \\ &v_{a,C_1}v_{a,C_2}\min\{v_{e,C1}, v_{e,C2}\}]. \end{aligned} \tag{5}$$

The expression for $C = 3$ is

$$\begin{aligned} e = \frac{1}{a}\,[&v_{a,C_1}(1 - v_{a,C_2})(1 - v_{a,C_3})v_{e,C_1} + \\ &v_{a,C_2}(1 - v_{a,C_1})(1 - v_{a,C_3})v_{e,C_2} + \\ &v_{a,C_3}(1 - v_{a,C_1})(1 - v_{a,C_2})v_{e,C_3} + \\ &v_{a,C_1}v_{a,C_2}(1 - v_{a,C_3})\min\{v_{e,C_1}, v_{e,C_2}\} + \\ &v_{a,C_1}v_{a,C_3}(1 - v_{a,C_2})\min\{v_{e,C_1}, v_{e,C_3}\} + \\ &v_{a,C_2}v_{a,C_3}(1 - v_{a,C_1})\min\{v_{e,C_2}, v_{e,C_3}\} + \\ &v_{a,C_1}v_{a,C_2}v_{a,C_3}\min\{v_{e,C_1}, v_{e,C_2}, v_{e,C_3}\}]. \end{aligned} \tag{6}$$

The general expression for any value of C is

$$\begin{aligned} e = \frac{1}{a}\sum_{\forall\ \vec{\epsilon}}\prod_{j=1}^{C}&[1 - (\epsilon_j + (-1)^{\epsilon_j}v_{a,C_j})] \times \\ &\min\{\frac{v_{e,C_1}(1+\alpha)}{\epsilon_1 + \alpha}, \cdots, \frac{v_{e,C_C}(1+\alpha)}{\epsilon_C + \alpha}\} \end{aligned} \tag{7}$$

where $\vec{\epsilon} = (\epsilon_1, \cdots, \epsilon_C)$, $\epsilon_j \in \{0, 1\}$, $\prod_{j=1}^{C}\epsilon_j \neq 0$, and $\alpha = 10^{-10}$ (i.e., a very small number). It should be noted that the

terms of the form $\frac{v_{e,C_j}(1+\alpha)}{\epsilon_j+\alpha}$ are equal to v_{e,C_j} when $\epsilon_j = 1$ and equal to a very large number when $\epsilon_j = 0$. In the latter case, a very large number makes the term irrelevant for the min operator. The term ϵ_j is equal to one when component C_j is available and zero otherwise.

As an example, the component *Occupancy Awareness* in Fig. 3 was replaced by a fault-tolerant architectural pattern in Fig. 4. If this pattern is of the first-to-respond type, its availability and execution times would be given by

$$a = 1 - [(1 - v_{a,\text{OccupancyAwareness1}}) \times (1 - v_{a,\text{OccupancyAwareness2}})] \quad (8)$$

and

$$\begin{aligned} e = \frac{1}{a}[&v_{a,\text{OccupancyAwareness1}}(1 - v_{a,\text{OccupancyAwareness2}}) \\ &v_{e,\text{OccupancyAwareness1}} + \\ &v_{a,\text{OccupancyAwareness2}}(1 - v_{a,\text{OccupancyAwareness1}}) \\ &v_{e,\text{OccupancyAwareness2}} + \\ &v_{a,\text{OccupancyAwareness1}} v_{a,\text{OccupancyAwareness2}} \\ &\min\{v_{e,\text{OccupancyAwareness1}}, v_{e,\text{OccupancyAwareness2}}\}]. \end{aligned} \quad (9)$$

This pattern primarily promotes availability, although it also benefits the expected response time in the case of failures: even if the fastest component fails, it guarantees the response time of the next fastest component. However such benefits come at the cost of redundant usage of resources, which negatively impacts scalability.

5.3 Fault-Tolerant Two-Phase Commit

The fault-tolerant two-phase commit pattern has the same structure as the fault-tolerant first-to-respond pattern but has a different behavior. In this case, the connector receives a request, sends it for processing to all C components, waits for all to respond, and then sends a commit request to all of them. Thus, the availability model $\mathcal{M}_a$ for this pattern is

$$a = \mathcal{M}_a(v_{a,C_1}, \cdots, v_{a,C_C}) = \prod_{j=1}^{C} v_{a,C_j} \quad (10)$$

since all components need to be available to complete the operation.

The execution time model $\mathcal{M}_e$ is

$$e = 2 \times \max\{v_{e,C_1}, \cdots, v_{e,C_C}\} \quad (11)$$

This component promotes fault-tolerance when information has to be maintained at more than one location to allow continued operation in the face of failures. However, the increased availability comes at the expense of reduced execution time.

5.4 Load Balancing

This pattern has the same architectural structure as the two previous ones. Its behavior is different though. The connector sends requests to one and only one of the C components at a time. When it receives a reply, the connector issues a reply. The load balancing pattern may follow many different disciplines. We assume a simple round-robin load balancing discipline. If a component is not available it is not included in the round-robin cycle. Thus, the availability model $\mathcal{M}_a$ for this pattern is

$$a = \mathcal{M}_a(v_{a,C_1}, \cdots, v_{a,C_C}) = \frac{1}{C}\sum_{j=1}^{C} v_{a,C_j} \quad (12)$$

because $1/C$ is the probability that a component is selected and v_{a,C_j} is the probability that it is available when selected. If the load balancer pattern uses a mechanism to track when components are available, such as heart-beat or time-out on requests, then its availability is the same as in Eq. (4).

The execution time model $\mathcal{M}_e$ is

$$e = \frac{1}{C}\sum_{j=1}^{C} v_{a,C_j} \times v_{e,C_j} \quad (13)$$

because the execution time is v_{e,C_j} if component C_j is selected and available. This occurs with probability $v_{a,C_j}/C$.

This pattern primarily promotes scalability. Similarly to the first-to-respond pattern, it also promotes availability but the benefits in the execution time are less pronounced: the expected time is a weighted average of the response times, in contrast to a guaranteed fastest available response time.

5.5 Parallel Invocation

This pattern consists of a connector that receives a request and breaks it down into sub-requests that are sent in parallel to all C components. The connector merges all replies from the C components and replies to the original request. The connector and all components have to be available for the pattern to be available. Thus, the availability model $\mathcal{M}_a$ for this pattern is

$$a = v_{a,\text{connector}} \prod_{j=1}^{C} v_{a,C_j} \quad (14)$$

The execution time model $\mathcal{M}_e$ is

$$e = v_{e,\text{connector}-\text{prior}} + \max\{v_{e,C_1}, \cdots, v_{e,C_C}\} + v_{e,\text{connector}-\text{post}} \quad (15)$$

where $v_{e,\text{connector}-\text{prior}}$ and $v_{e,\text{connector}-\text{post}}$ are the execution times at the connector prior to farming all subrequests to the C components and after receiving their replies, respectively.

This pattern primarily promotes the reduction of execution time given that the overall work can be broken down in smaller pieces to be executed in parallel. However, this may come at the cost of reduced availability.

5.6 Composition of QoS Architectural Patterns

This section describes how the various QoS metrics are affected by the composition of patterns along an SSS. In SASSY, a utility function, associated to a QoS metric, is assigned to each SSS. Then, all these utility functions are combined into a global utility function.

The end-to-end QoS metric along an SSS depends on the values of that metric for each component or pattern in the SSS.

We illustrate here how these end-to-end metrics are computed for execution time and availability. The expression for the end-to-end execution time is obtained as the sum of the execution times of all components along the SSS. Consider for example the SSS in Fig. 2d to which the stakeholders associated goals for availability and end-to-end execution time. Suppose that the architecture in Fig. 4 is being evaluated,

where the first-to-respond pattern has been applied for *Occupancy Awareness*. Then, the end-to-end execution time for that SSS is

$$e_{SSS} = v_{e,\text{BuildingLocator}} + e_{\text{Fault-Tolerant First-to-Respond}} \quad (16)$$

where $e_{\text{Fault-Tolerant First-to-Respond}}$ is given by Eq. (9).

The availability of the SSS in Fig. 2d is given by

$$a_{SSS} = v_{a,\text{BuildingLocator}} \times a_{\text{Fault-Tolerant First-to-Respond}} \quad (17)$$

where $a_{\text{Fault-Tolerant First-to-Respond}}$ is given by Eq. (9).

In general, the end-to-end execution time is the sum of the execution times of basic components or composite components (i.e., patterns) along an SSS. The end-to-end availability is the product of the availabilities of basic components or composite components (i.e., patterns) along an SSS.

5.7 Case Study

This section provides an example of the use of QoS Architectural patterns for two SSSs. Consider the availability SSS shown in Fig. 2d and consider a similar SSS (i.e., the same structure) but with execution time associated to it. These SSSs use two service types: *Building Locator* and *Occupancy Awareness*. Consider that there are three possible service providers (BL1, BL2, and BL3) that can be used to support the *Building Locator* service type and three service providers (OA1, OA2, and OA3) that can support the *Occupancy Awareness* service type. Table 1 shows the execution times (in msec) and availabilities guaranteed by these service providers.

Table 2 illustrates the effect of using eight different combinations of QoS architectural patterns for these SSSs. We use three types of patterns in this example: Basic Component (BC), Load Balancing (LB), and Fault Tolerant First-to-Respond (FFT). In parentheses, next to each pattern in Table 2 is the list of actual service providers used in the pattern. For example, LB (BL1, BL2) indicates a load balancing pattern that uses service providers BL1 and BL2.

The availability a and execution time e for *Building Locator* and *Occupancy Awareness* are shown in the table and were computed using the expressions given in Section 5. Then, the table shows a_{SSS} and e_{SSS}, the availability and execution time for the SSSs, respectively. As it can be seen, the combinations of patterns and service provider selections that provides the best availability are 4 and 8, both with an SSS availability of 0.9792. The lowest execution time for the SSS is given by combination number 4, which has an execution time equal to 150.38 msec.

The last two columns of Table 2 show the utilities for these values of availability and execution time. The utility for availability was computed using Eq. (1). The utility for execution time was computed using the following sigmoid:

$$U_e(e) = \frac{e^{0.5\ (160-e)}}{1 + e^{0.5\ (160-e)}}. \quad (18)$$

The above expression is commonly used in autonomic computing [1]. The value 160 represents the desirable QoS goal for execution time.

The highest value for both U_a and U_e is obtained for combination number 4, which uses a basic component with service provider BL1 for the *Building Locator* service type and a fault tolerant first-to-respond pattern using service providers OA1 and OA3 for the *Occupancy Awareness* service type. The architecture that uses these patterns is illustrated in Fig. 4. In general, the search procedure tries to optimize a global utility that is a function of all the individual utilities.

This example illustrates just a few possible combinations of patterns and service selections. The reader will certainly note that the number of possible combinations of QoS architectural patterns and service providers grows in a combinatorial way. SASSY automates the search process (see Section 4) using heuristic procedures that are driven by QoS architectural patterns. The interested reader may refer to [20] for a detailed description of these heuristics.

6. RELATED WORK

Over the past few years, researchers and practitioners have developed a variety of frameworks and techniques intended to support the construction of self-adaptive systems [3, 17]. Below we provide an overview of the most notable approaches in this area and examine them in light of our work.

IBM's Autonomic Computing initiative advocates a reference model, known as MAPE-K [15], that is structured as a hierarchical set of feedback-control loops, each of which consists of the following activities: Monitor, Analyze, Plan, and Execute. In their seminal work [22], Oreizy et al. pioneered the architecture-based approach to run-time adaptation and evolution management. In [10], Garlan et al. present Rainbow framework, a style-based approach for developing reusable self-adaptive systems. Rainbow monitors a running system for violation of the invariant imposed by the architectural model, and applies the appropriate adaptation strategy to resolve such violations. Georgiadis et al. [11] propose a decentralized adaptation approach, where each self-organizing component manages its own adaptation with respect to the overall system goal.

All of the above approaches, including numerous others (e.g., see [3, 17]), share three key traits: (1) use analytical models for making adaptation decisions at runtime, and (2) rely on architectural models for managing the complexity of analysis, and (3) effect a new solution through architecture-based adaptation. These works have clearly formed the foundation of our work. However, in our research we are extending these approaches by employing QoS architectural patterns, which allow SASSY to satisfy the users' QoS requirements through automatic (re)generation of the architecture.

Our previous work [13, 12] investigated the concept of *adaptation pattern*, which provides a reusable mechanism for ensuring the consistency of the software during and after the adaptation. It corresponds to a state-based model that describes how to transition a software component from *active* state to the *quiescence* state prior to its replacement. The work presented in this paper is different, as the QoS architectural patterns are used to determine the best architecture, and not the low-level adaptation activities. We believe QoS architectural patterns and adaptation patterns to be complementary to one another.

In [26], Zhang and Cheng present an approach to formally model the behavior of adaptive programs, automatically analyze them, and generate an implementation of the system. A more recent work [24] shows how this model-driven approach could be employed for developing patterns, which through reuse could reduce the effort required for developing self-adaptive software systems. The proposed patterns are of three types: monitoring, decision-making, and dynamic

Service Type	Service Provider	Execution Time (msec)	Availability
Building Locator	BL1	50	98%
	BL2	70	97%
	BL3	60	99%
Occupancy Awareness	OA1	120	99%
	OA2	150	95%
	OA3	100	98%

Table 1: Service types, service providers (SPs) and their characteristics.

No	Building Locator	a	e	Occupancy Awareness	a	e	a_{SSS}	e_{SSS}	$U_a(SSS)$	$U_e(SSS)$
1	BC (BL1)	0.980	50.000	BC (OA1)	0.960	120.00	0.9408	170.00	0.5	0.99331
2	BC (BL1)	0.980	50.000	BC (OA2)	0.950	150.00	0.9310	200.00	0.5	0.00005
3	BC (BL1)	0.980	50.000	FFT (OA1,OA2)	0.998	121.14	0.9780	171.14	1.0	0.98821
4	BC (BL1)	0.980	50.000	FFT (OA1, OA3)	0.999	100.38	0.9792	150.38	1.0	1.00000
5	LB (BL1, BL2)	0.975	58.450	BC (OA1)	0.960	120.00	0.9360	178.45	0.5	0.68460
6	LB (BL1, BL2, BL3)	0.980	58.767	BC (OA1)	0.960	120.00	0.9408	178.77	0.5	0.64946
7	LB (BL1, BL2)	0.975	58.450	FFT (OA1,OA2)	0.998	121.14	0.9731	179.59	1.0	0.55079
8	LB (BL1, BL2, BL3)	0.980	58.767	FFT (OA1, OA3)	0.999	100.38	0.9792	159.15	1.0	0.99997

Table 2: Eight combinations of QoS Architectural Patterns for the SSS of Fig. 2d and various selections of SP. BC = Basic Component; LB = Load Balancer; FFT = Fault Tolerant First-to-Respond

reconfiguration. The patterns investigated by the authors are general, and not specifically targeted at improving QoS properties in SOA systems. Moreover, the focus of our work has been on an automated quantitative method of selecting and composing patterns to achieve the QoS objectives.

Finally, related to our work are the approaches for adaptive QoS management in SOA settings. The two most related approaches to SASSY are JOpera [23] and MOSES [5] frameworks. JOpera [23] provides an engine for managing the execution of service providers to satisfy multiple QoS properties, such as efficiency and latency. Thus, JOpera's objective is on enabling the service providers to satisfy their SLA requirements, while in SASSY given a set of service providers, it aims to find an architecture that satisfies the users' QoS requirements. Using a BPEL specification of a software system, MOSES [5] dynamically determines the best composition of service providers. Unlike MOSES, SASSY uses a higher-level and more intuitively understood language than BPEL. Moreover, SASSY maintains an explicit representation of the system's architecture, which is continuously revised at runtime through the application of QoS architectural patterns. MOSES uses only two patterns: fault-tolerant one-at a-time, referred to as sequential alternate execution of services, and fault-tolerant first-to-respond; this is in contrast with the richer set of patterns in Section 5. Furthermore, MOSES uses linear programing to search for an optimal solution, which severely limits its scalability since the problem is NP-complete.

7. CONCLUDING REMARKS

This paper has described the autonomic application of architectural patterns for tuning the quality of service of service-oriented software systems. In particular, the paper has described a number of QoS patterns and the corresponding quantitative models for two concrete aspects of quality of service: availability and execution time. The emphasis here is on automatic application of QoS patterns to obtain an architecture that maximizes a utility function for the software system. This paper has also discussed how this research extends prior work in SASSY, where the service activity schemas are annotated with QoS goals as defined by domain experts.

This paper extends prior work in SASSY by providing autonomic system adaptation based on QoS patterns, where the system continuously monitors QoS goals and takes autonomic action if QoS goals are not met. SASSY uses efficient and scalable search heuristics to identify the optimal patterns, making it possible to perform self-architecting both at system deployment and at run time for purposes of self-adaptation. A case study has illustrated the role of QoS patterns in a self-architected service-oriented application.

The contributions of this paper are: 1) the analysis of a number of patterns suitable for self-architecting and self-adaptation and 2) quantitative models to derive the overall QoS of a system after the application of several such patterns.

Future work involves integrating the QoS patterns described in this paper with our research into self-adaptive systems [12]. Our research is consistent with the 3-layer model for self-management [17], in which the QoS autonomic adaptation corresponds to the highest layer of Goal Management. The next layer, Change Management, determines the configuration changes needed to autonomically adapt the affected part of the system, while the rest of the system continues to be operational. For example, in the case study (Section 5.7) in order to replace the basic Occupancy Awareness component with the FFT QoS pattern, it would be necessary to (a) drive the Occupancy Awareness (OA1) service to a quiescent state by queueing up new OA service requests and letting OA1 complete old requests, (b) unlink the connection from

OA1 to the Dispatcher Coordinator (DC), (c) instantiate the Fault Tolerant connector (FTC), link FTC to the OA1 and OA2 services, (d) link DC to FTC, and (e) release queued OA service requests to FTC to restart the operation of the OA service, operating with the FFT QoS pattern. The bottom layer of the 3-layer model, Component Control, actually coordinates the execution of the adaptation scenario.

Acknowledgments

This work is partially supported by award no. CCF-0820060 from the National Science Foundation.

8. REFERENCES

[1] M.N. Bennani and D.A. Menascé. Resource allocation for autonomic data centers using analytic performance models. In *Proc. 2nd IEEE Intl. Conf. Autonomic Computing (ICAC'05)*, Seattle, WA, June 2005.

[2] Object Management Group (OMG). BPMN Spec. ver 1.1. http://www.omg.org/spec/BPMN/1.1/

[3] B. Cheng et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems, LNCS Hot Topics*, 2009.

[4] P. Clements, R. Kazman, and M. Klein. Evaluating software architectures: methods and case studies. Addison Wesley, 2001.

[5] V. Cardellini, E. Casalicchio, V. Grassi, F.L. Presti, R. Mirandola. QoS-Driven Runtime Adaptation of Service-Oriented Architectures. In *7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, Amsterdam, Netherlands, Aug 2009.

[6] E. Dashofy, A. van der Hoek, and R.N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proc. 24th International Conference on Software Engineering*, pages 266–276, Orlando, FL, May 2002.

[7] 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. In *Proc. 12th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MODELS'09*, Denver, CO, Oct. 2009.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, ISBN 0-201-63361-2, 1995.

[9] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. IBM Systems Journal, vol. 42, 2003, pp. 5-18.

[10] D. Garlan, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In *IEEE Computer*, Vol. 37(10), Oct. 2004.

[11] I. Georgiadis, J. Magee, and J. Kramer. Self-Organizing Software Architectures for Distributed Systems. In *Workshop on Self-Healing Systems*, Newport Beach, CA, Oct 2004.

[12] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D.A. Menascé, Software Adaptation Patterns for Service Oriented Architectures, In *Proc. 25th ACM Symposium on Applied Computing, Dependable and Adaptive Distributed Systems*, Sierre, Switzerland, March 22 - 26, 2010.

[13] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Proc. 4th Working IEEE/IFIP Working Conf. Software Architecture*, pages 79–88, Oslo, Norway, June 2004.

[14] N. Harrison and P. Avgeriou. Leveraging Architecture Patterns to Satisfy Quality Attributes. Software Architecture, Springer LNCS, 2007.

[15] J.O. Kephart, and D.M. Chess. The Vision of Autonomic Computing. In *IEEE Computer*, vol. 36(4), Jan 2003.

[16] J. Kramer and J. Magee. Analyzing dynamic change in software architectures: A case study. In *Proceedings of the 4th IEEE Intl. Conf. Configurable Distributed Systems*, pages 91–100, Annapolis, MD, May 2007.

[17] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE'07)*, pages 259–268, Minneapolis, MN, May 2007.

[18] S. Malek, N. Esfahani, D.A. Menascé, J.P. Sousa, and H. Gomaa. Self-architecting software systems (SASSY) from QoS-annotated models. In *Principles of Engineering Service Oriented Systems (PESOS'09)*, pages 62–69, Vancouver, Canada, May 2009.

[19] S. Malek, M. Mikic-Raki, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Tr. Software Engineering*, 31(3):256–272, Mar. 2005.

[20] D.A. Menascé, J. Ewing, H. Gomaa, S. Malek, and J.P. Sousa. A Framework for Utility-Based Service Oriented Design in SASSY. Joint WOSP/SIPEW Intl. Conf. Performance Engineering, San José, California, Jan 28-30, 2010.

[21] D.A. Menascé, E. Casalicchio, and V. Dubey. On Optimal Service Selection in Service Oriented Architectures, Performance Evaluation Journal, Elsevier, www.elsevier.com/locate/peva, doi:10.1016/j.peva.07.001

[22] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. In *Intl. Conf. Software Engineering*, Kyoto, Japan, April 1998.

[23] C. Pautasso, T. Heinis, and G. Alonso. JOpera: Autonomic Service Orchestration. IEEE Data Engineering Bulletin, vol. 29, Sep. 2006, pp. 32-39.

[24] A.J. Ramirez and B.H. C. Cheng. Applying Adaptation Design Patterns. In *Intl. Conf. Autonomic Computing and Communications*, Barcelona, Spain, June 2009.

[25] J.P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw. Task-based Adaptation for Ubiquitous Computing. IEEE Tr. Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems, vol. 36, pp. 328-340, 2006.

[26] J. Zhang and B.H. C. Cheng, Model-based development of dynamically adaptive software. In *Intl. Conf. Software Engineering*, Shanghai, China, May 2006.