A Framework for Utility-Based Service Oriented Design in SASSY

Daniel A. Menascé, John M. Ewing, Hassan Gomaa, Sam Malek, and João P. Sousa
Department of Computer Science
George Mason University
Fairfax, VA, USA
{menasce,jewing2,hgomaa,smalek,jpsousa}@gmu.edu

ABSTRACT

The architecture of a software system has a significant impact on its quality of service (QoS) as measured by several performance metrics such as execution time, availability, throughput, and security. This paper presents a framework that is part of a large project called SASSY (Self-Architecting Software Systems), whose goal is to allow domain experts to specify the system requirements using a visual activity-based language. The SASSY framework automatically generates a base architecture that corresponds to the requirements. Then SASSY generates a new architecture, derived from the base architecture, that optimizes a utility function for the entire system. The utility function is a multivariate function of several QoS metrics. The paper shows a complete example and illustrates how SASSY automatically adapts to changes in the environment's QoS features.

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, utility functions, QoS, optimization, heuristic

1. INTRODUCTION

The architecture of a software system has a significant impact on its quality of service (QoS) as measured by several performance metrics such as execution time, availability, throughput, and security. Large and complex software systems are deployed on environments that may exhibit significant variations at run-time due to changes in the workload intensity and failures. A service-oriented architecture (SOA) is a software architecture that consists of multiple distributed autonomous services. The same service type, e.g., airline reservation, is capable of being offered by different service providers (SP), where services and their service providers can be discovered at run time [4, 16, 17]. New SPs may be brought into existence at any time and existing SPs may fail or stop operating entirely. Different functionally equivalent SPs may exhibit different QoS levels at different costs. In such environments, it is important for software systems to self-architect so they can adapt to changes in the environment in an autonomous way. Software adaptation refers to software systems that change their behavior during execution. Self-adaptive software systems monitor the environment and adapt their behavior in response to changes in the environment [10].

The framework presented in this paper is part of a large project called SASSY (Self-Architecting Software Systems), whose goal is to allow domain experts to specify the system requirements using a visual activity-based language. The SASSY framework automatically generates a base architecture that corresponds to the requirements. Then, as described here, SASSY generates a new architecture, derived from the base architecture, that optimizes a utility function for the entire system. The utility function is a multivariate function of several QoS metrics.

The contributions of this paper are the following. First, it presents SASSY and its main concepts. Second, it formalizes the optimal self-architecting problem for software systems and presents a heuristic to find a near optimal solution. Third, the paper discusses a complete example and illustrates how SASSY adjusts the architecture automatically in response to changes in the QoS characteristics of the underlying environment.

The paper is organized as follows: Section 2 presents an overview of the SASSY system. Section 3 describes how SASSY's activity-based language is mapped to a system service architecture. The next section describes the optimization problem, including the heuristic search technique used to find a near optimal architecture. Section 5 presents a complete numerical example. Concluding remarks are presented in Section 6.

2. OVERVIEW OF SASSY

SASSY (Self-Architecting Software Systems) is a model-driven framework for run-time self-architecting and re-ar-

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.

chitecting of distributed software systems. SASSY provides a uniform approach to automated composition, adaptation, and evolution of software systems based on service-oriented architectures (SOAs). This framework provides mechanisms for self-architecting and re-architecting that determine the near-optimal architecture for satisfying functional and QoS requirements. The quality of a given architecture is expressed by a utility function provided by end-users and represents one or more desirable system objectives.

Figure 1 illustrates, at a very high level, how SASSY uses run-time models for self adaptation of SOA-based software. SASSY uses four types of run-time models: 1) service activity schemas with their corresponding service sequence scenarios, 2) system service architecture models, 3) QoS architectural pattern models, and 4) QoS analytical models.

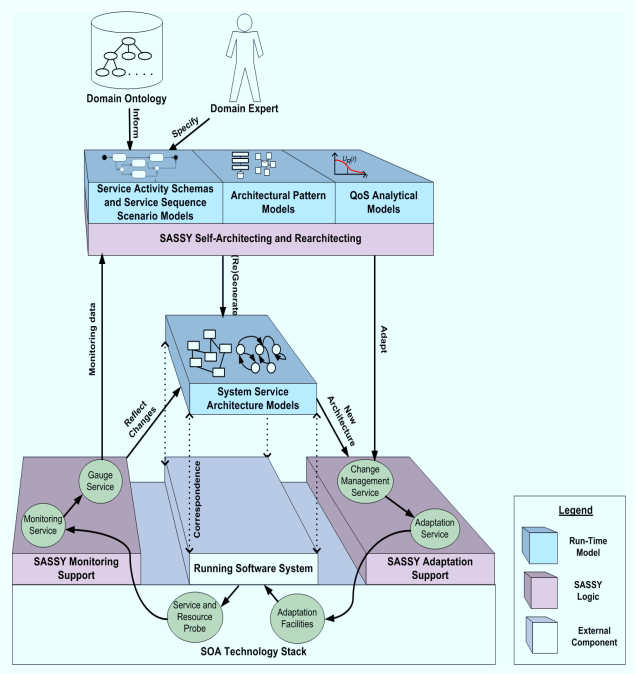

Figure 1: High-level view of the SASSY framework.

A domain expert, as opposed to a software engineer, expresses the system requirements in the form of a service activity schema (SAS). Activities represent tasks that need to be performed. The modeling constructs (e.g., activities, service type, domain entities) are defined in a domain ontology that unambiguously distinguishes different concepts and elements in order to facilitate service discovery and resources in support of activities.

Figure 2 provides an example of an SAS for an emergency response application and was obtained as a screenshot from the tool we developed for the project. This tool supports domain experts performing SASSY-related development. The dashed boxes on the left-hand side of Fig. 2 correspond to service sequence scenarios (SSS) to be introduced later in this section. The SAS shows five service types (see boxes with rounded corners with a server icon inside): `Road Map`, `Weather`, `ID Possible Threats`, `Make Evac Plan`, and `Eval Evac Plan`. The application can be described as follows. When there is an emergency, three service types are invoked simultaneously (see fork gateway in the diagram specified by a rhombus with a plus sign inside): `Road Map`, `Weather`, and `ID Possible Threats`. The `Road Map` service type is used to obtain machine readable maps of the roads in the region. The `Weather` service type provides current and predicted weather information about the region and the `ID Possible Threats` service type identifies possible threats for the region and assigns probabilities to each threat. After all three activities complete, i.e., join, the `Make Evac Plan` service type is invoked to generate an evacuation plan, which is passed to the `Eval Evac Plan` service type for an analysis and evaluation of the evacuation plan. The result of the evaluation is tested at the conditional gateway (see rhombus with an "×" in the middle at the output of `Eval Evac Plan`). If the plan is not approved, a new one will have to be produced.

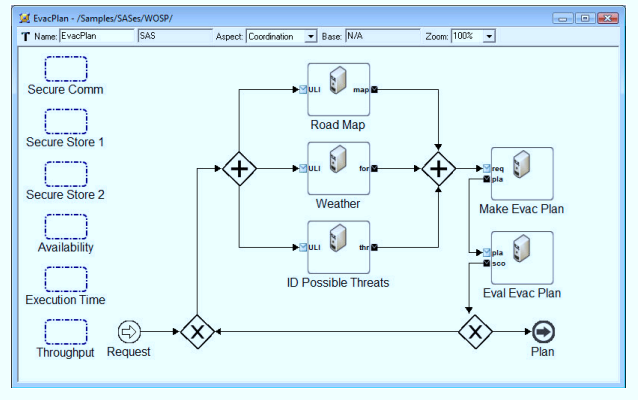

Figure 2: Example of a service activity schema (SAS) for an emergency response application.

Using a list of QoS metrics (e.g., performance, availability, and security) defined in the domain ontology, the domain expert can specify desired QoS properties in an SAS through one or more service sequence scenarios. Each SSS is associated to one of the QoS metrics. An SSS is a set of one or more service types in an SAS connected by links in an SAS.

Figure 3 displays three SSSs: the `Execution Time` SSS, the `Availability` SSS, and the `Secure Comm` SSS. The `Execution Time` SSS includes the fork-and-join of `Road Map`, `Weather`, and `ID Possible Threats` followed by the execution of `Make Eval Plan` and `Eval Evac Plan`. The QoS metric associated with this SSS is execution time. The `Availability` SSS has the same structure as the `Execution Time` SSS except that its metric is availability. Thus, two or more SSSs may have the same structure as long as they have different metrics associated with them.

The `SecureComm` SSS includes the connection between `ID Possible Threats`, `Make Eval Plan`, and `Eval Evac Plan`. The metric associated to this SSS is the security level of its communication.

A utility function is associated with each SSS. Utility functions are used in economics and autonomic computing to assign a value to the usefulness of a system based on its attributes [3]. The attribute associated with the utility function of each SSS is its QoS metric. Consider for example the `Availability` SSS (top part of Fig. 3) and let a be the value of its availability. Then, an example of the utility function,

$U_{\text{Availability}}(a)$, of this SSS is:

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

In this example, the utility of this SSS is zero if its availability is less than 0.9. The utility is 0.5 for an availability in the interval [0.9,0.95) and it is 1 when the availability is equal to 0.95 or more.

Utility functions are established by the domain experts in consultation with all stakeholders. Note that the actual value of the availability a, or more generally, the value of the metric associated with any SSS can only be computed when actual service providers (SPs) are selected for each service type in the SAS. However, one can derive an analytic model for the value of the metric associated with an SSS. This model can be expressed as a function of the metrics obtained from the SPs of the service types appearing in that SSS. For example, the expression for the availability a is simply

$$a = a_{\texttt{Road Map}} \times a_{\texttt{Weather}} \times a_{\texttt{ID Possible Threats}} \times a_{\texttt{Make EvacPlan}} \times a_{\texttt{Eval Evac Plan}} \quad (2)$$

because all services involved need to be available in order for the SSS to be available.

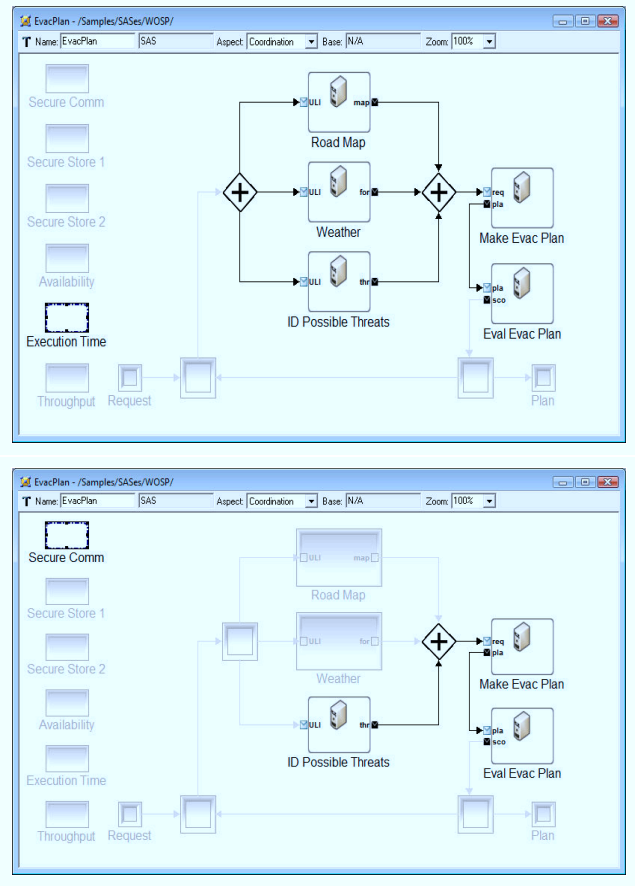

Figure 3: Top: `Execution Time` or `Availability` SSS (they have the same structure); Bottom: `Secure Comm` SSS

Consider now the `Execution Time` SSS. The expression for the execution time e for this SSS is:

$$e = \max\{e_{\texttt{Road Map}}, e_{\texttt{Weather}}, e_{\texttt{ID Possible Threats}}\} + e_{\texttt{Make EvacPlan}} + e_{\texttt{Eval Evac Plan}}. \quad (3)$$

The utility, U_g, for the entire SAS is defined by the domain expert as a function of the utilities of all SSSs. This global utility function encompasses domain information by reflecting SSS priorities and any interactions between SSSs.

A system service architecture (SSA) consisting of components, associated to SPs, and connectors is automatically generated from the requirements above. The architecture is optimized with respect to QoS requirements through the selection of the most suitable SPs and application of QoS architectural patterns. This architecture is determined with the help of QoS analytic models and optimization techniques aimed at finding near optimal choices that maximize system utility as described here.

SASSY's monitoring support services generate triggers that cause self-adaptation. SPs may be automatically replaced and the software architecture may be automatically regenerated by SASSY when the system utility degrades beyond a certain threshold. The search space for self-adaptation includes a set of alternative candidate software architectures, derived from an analysis of critical SSSs, and the SPs that support the services in each architecture. For example, a candidate architecture that uses replication to address fault tolerance may be considered if the observed availability degrades considerably. Similarly, a load balancing architectural pattern can be used to improve response time and availability at the same time.

The SASSY approach is self-healing, self-managing, and self-optimizing since, when SPs fail or are unable to meet their QoS goals, SASSY's monitoring services trigger SP replacement through a new round of service discovery, optimal SP selection, and possible determination of alternative architectures. Our approach for dynamic monitoring and adaptation builds on previous research on dynamic software architectures [8, 13]. This approach is consistent with the widely accepted three layer architecture model of self-management [10], which consists of: 1) Goal Management Layer—planning for change—often human assisted, 2) Change Management Layer—execute the change 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—actually implements the run-time adaptation. More details about SASSY's activity modeling language can be found in [7, 12].

3. FROM SAS TO SSA

The System Service Architecture (SSA) offers structural and behavioral models for an SOA system. Unlike traditional software architectural models that are mainly used during the design phase, an SSA is intended for use at run-time by the SASSY framework. The SSA is an accurate representation of the running software system, and is utilized to support run-time reasoning with respect to evolution and adaptation. The SSA's structural model is based on the eXtensible Architectural Description Language (xADL) [6], which provides the traditional component-and-connector view of the software architecture.

We have extended the core xADL language by introducing the concept of service instances, which are modeled as software components or composition of software components. A *service instance* is the realization of a service type defined in the ontology. The SSA also provides a mapping between each service instance and the concrete SP that is used at a given point in time. The middleware enabling integration and communication among the services is modeled as software connectors. Finally, the components and connectors bind to one another using required and provided interfaces.

The top part of Fig. 4 shows the structural view of a base SSA that corresponds to the SSA of Fig. 2. The base SSA is automatically generated by SASSY using GReAT [1, 2], which is a graph transformation tool. The base architecture associates one component to each service type and creates one component to represent the logic of a coordinator that orchestrates the communication between all service types. The bottom part of Fig. 4 shows a modified version of that base SSA in which the component `Eval Evac Plan` was replaced by a fault-tolerant component.

The SSA's behavioral models provide a state machine view for representing how the service instances collaborate with one another to fulfill the system requirements. In other words, a behavioral model corresponds to the executable logic of service coordination in SOA. SSA's behavioral models are based on Finite State Processes (FSP) [9], which unlike many other state machine languages provides a rich set of abstraction constructs, thereby making it scalable for modeling the behavior of large-scale software systems. SASSY automatically generates the SSA's behavioral models based on the requirements specified by the domain expert in the SAS. Since the SSA contains both structural and behavioral models, the SSA needs only a set of selected service instances and bindings with those services to become executable.

The relationship between SSSs, components, service types, QoS metrics, and utility functions can be better described with the help of the class diagram of Fig. 5. Each SSS has a single utility function associated with it and that utility function is a function of a single QoS metric. An analytic QoS model is used to compute the value of that QoS metric. See for example Eq. (3), which is a QoS model for the case of execution time for the `Execution Time` SSS. The SASSY framework uses a library of *architectural patterns* to assist in the self-architecting process. Each pattern consists of one or more components which may be linked by connectors. Each of these components may be associated with one or more service types, which are instantiated by one or more SPs. The structural and behavioral aspect of a pattern are described in xADL [6] and FSP [9], respectively. A pattern also includes one or more QoS metrics and the corresponding QoS model for the metrics they influence. Some architectural patterns such as load balancing and fault tolerant have more than one component associated to them.

Consider for example the fault tolerant architectural pattern used in the bottom of Fig. 4. This pattern influences two QoS metrics: availability and execution time. If the behavior of this pattern is such that its execution is considered to be complete whenever the first of the two components, (`Eval Evac Plan 1` and `Eval Evac Plan 2`), complete, then the availability is a function of the availabilities a_1 and a_2 of the individual components and the execution time is a function of the execution times e_1 and e_2 of the

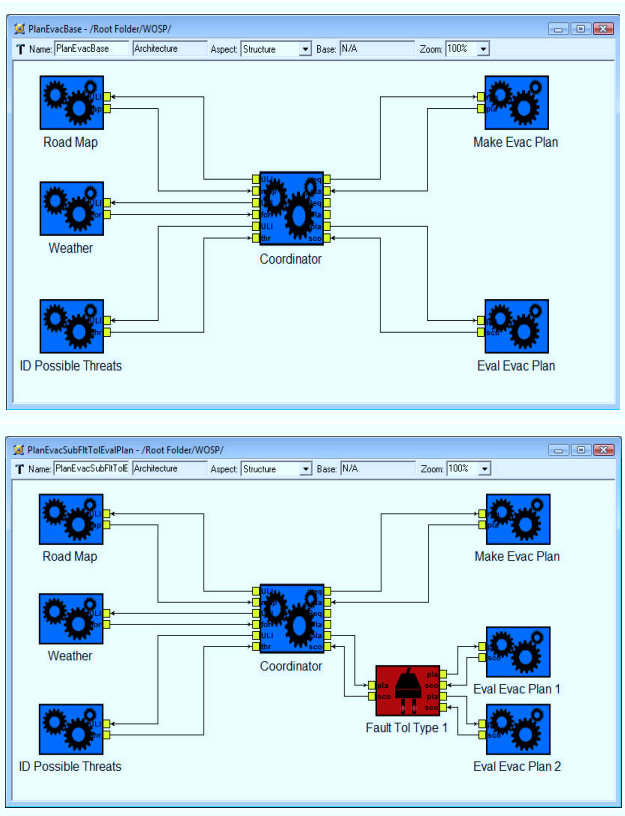

Figure 4: Top: Base SSA corresponding to the SAS of Fig. 2; Bottom: SSA showing the replacement of component `Eval Evac Plan` with a fault tolerant component.

individual components. Then, the QoS models for the availability A and the execution time E for the fault tolerant pattern are:

$$A = 1 - (1 - a_1)(1 - a_2) \tag{4}$$

assuming failure independence, and

$$E = \frac{a_1(1-a_2)}{A} e_1 + \frac{a_2(1-a_1)}{A} e_2 + \frac{a_1 a_2}{A} \min\{e_1, e_2\}. \tag{5}$$

4. THE OPTIMIZATION PROBLEM

The optimization problem addressed in this paper 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 SAS utility function U_g.

More formally, the optimization problem can be expressed as:

Find an architecture A^* and a corresponding SP allocation Z^* such that

$$(A^*, Z^*) = \text{argmax}_{(A,Z)}\ U_g(A, Z) \tag{6}$$

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 example we describe in the experimental section uses a cost constraint.

The number of different architectures is $O(p^n)$ where p is the average number of architectural patterns that can be used to replace any component and n is the number of components in the architecture. The number of possible SP selections for an architecture with n components is $O(s^n)$ where s is the average number of SPs that can be used to implement each component. Thus, the size of the solution space for this optimization problem is $O((p \times s)^n)$. It is very clear that the solution space is huge even for small values of p, s and n. For example, for $p = 5$, $s = 2$, and $n = 10$, the size of the solution space is on the order of 10^{10}, i.e., 10 billion possible solutions! In fact, the problem is NP-hard. We describe in what follows a near optimal solution technique to avoid an exhaustive and computationally unfeasible search.

4.1 Solving the Optimization Problem

The general approach for obtaining a near optimal architecture and service selection consists of the following steps. First, a *base architecture* is built. As explained in Section 3, the SASSY framework automatically generates a base architecture from the SAS. In this architecture, each service type is mapped to a single component in the architecture. Then, an optimal selection of SPs for that architecture is carried out which maximizes the global utility function U_g.

The optimal selection of SPs for a given architecture is similar to the problem of optimal service allocation for business processes in SOAs as described in [15]. The difference is that in [15] the goal was to minimize end-to-end execution time and here the goal is to maximize global utility. In this paper, we do not dwell on the optimal service allocation problem. We focus on near optimal architecture selection.

4.2 General Approach

The general approach is described in very simple terms by Algorithm 1. We start by assigning the base architecture A_{base} to the currently visited architecture A_{visited}. Perform an optimal selection of SPs for A_{visited} (line 3) and compute the value of its utility function (line 4). Then, iteratively, using local search techniques such as hill climbing, we select a new architecture to visit within a neighborhood $\mathcal{N}$ of A_{visited} using the function GenerateNeighborhood (line 8). For each architecture in $\mathcal{N}$, perform an optimal SP selection (line 10). Select as the next architecture to visit, the neighbor that provides the largest improvement in the value of the utility U_g.

One may stop the search if there are no such neighbors, in which case the near optimal architecture is the last visited architecture. Other stopping criteria such as a limited search budget (i.e., maximum number of iterations) can be used. One can also re-start the search from a random architecture if a neighborhood does not provide an improvement in the value of the utility function. This is done to reduce the chances of the search being stuck in a local optimum. Our implementation uses re-start by generating a random architecture from the base architecture as follows. We randomly select an architectural pattern to replace a component in the base architecture and randomly select the number of instances of multi-service provider patterns. We also randomly select the security level of each security option.

Algorithm 1 General Search Approach (k)

```
1: function GeneralSearch ()
2: A_visited ← A_base; /* initialization */
3: OptimalServiceProviderSelection (A_visited);
4: U_g ← Utility (A_visited) /* utility computation */
5: Searching ← TRUE
6: while Searching do
7:    A_opt ← A_visited
8:    N ← GenerateNeighborhood (A_visited, k)
9:    for all A_i ∈ N do
10:      OptimalServiceProviderSelection (A_i)
11:   end for
12:   A ← argmax_{A_i ∈ N} {Utility(A_i)}
13:   if Utility (A) > U_g then
14:      A_visited ← A
15:      OptimalServiceProviderSelection (A_visited);
16:      U_g ← Utility(A_visited) /* utility computation */
17:   else
18:      Searching ← FALSE
19:   end if
20: end while
21: end function
```

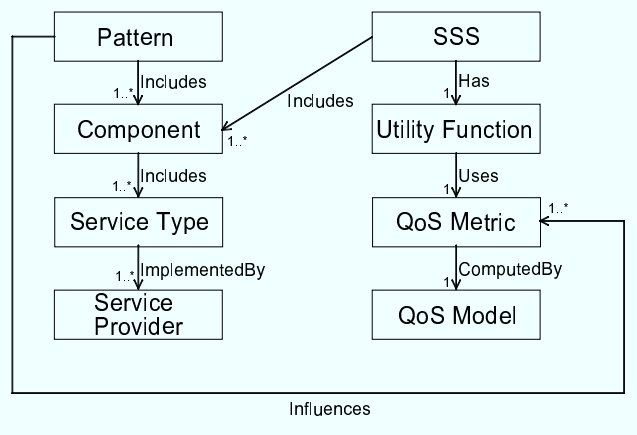

Figure 5: Class diagram for SSSs.

There are many ways to define the neighborhood of a given architecture. One of them, which we call *unfiltered*, replaces every component of every SSS with an architectural pattern that improves the metric associated with the SSS. This approach tends to generate a rather large neighborhood. The other approach, called *filtered*, reduces the size of the neighborhood by concentrating on the SSSs that can provide better gains to the utility function. The smaller the neighborhood, the higher the likelihood that the search will be trapped in a local optimum. On the other hand, large neighborhoods imply larger computational costs.

Our filtering approach is described at a very high level in Algorithm 2. We start by determining the set $\mathcal{S}_k$ of problem SSSs, i.e., the set of k (a tunable parameter) SSSs that exhibit the lowest numerical contribution to the global utility function (line 3). These are the SSSs that, if improved, could contribute the most towards improving the global utility. The contribution of an SSS towards the global utility function depends on the value of its own utility function and

on the weight of the SSS's utility function with respect to the global utility function.

For each SSS s in $\mathcal{S}_k$, determine the set $\mathcal{P}$ of architectural patterns in the library that improve the metric associated with that SSS (line 7). Then, for each component c in the set $\mathcal{C}$ of components of SSS s (line 8) and for each pattern p in $\mathcal{P}$ (line 9) add a new architecture to the neighborhood $\mathcal{N}$ of A_{visited} by replacing c by p in A_{visited}.

There are other details of the process of finding the neighborhood that are not described in detail in Algorithm 2 due to lack of space. They are however implemented in SASSY. The first has to do with multiple instances of SPs for patterns such as load balancing or fault tolerant. A new architecture can be generated from A_{visited} by increasing or decreasing by one the number of service instances of the architectural pattern. The second aspect not described in the algorithm has to do with the security option. Neighbors may be generated by increasing or decreasing by one notch the security level of a security option.

Algorithm 2 Generate Neighborhood

```
1: function GenerateNeighborhood (A_visited, k)
2: N ← φ; /* initialize with empty neighborhood */
3: S_k ← Set of SSSs with k lowest contribution to U_g
4: for all s ∈ S_k do
5:    m ← s.QoSmetric
6:    C ← Set of components of s
7:    P ← Set of patterns that improve m
8:    for all c ∈ C do
9:       for all p ∈ P do
10:         N ← N ⋃ Replace(A_visited, c, p)
11:      end for
12:   end for
13: end for
14: return N
15: end function
```

5. EXPERIMENTAL RESULTS

This section describes a detailed example of the use of the framework presented here for the emergency response example presented before. The section presents the utility functions used, the list of SPs used to support the service types of the application, the patterns available in the library and their QoS models, the QoS models of the SSSs, and the results of applying the approach to the base architecture.

5.1 Utility Functions Used

In this emergency response example, the domain expert has defined utility functions for execution time, availability, and throughput SSSs that follow a sigmoid curve. Sigmoid functions are used to connote sensitivity to a particular threshold value of a QoS metric [11]. However, the domain expert could choose other utility functions for these QoS metrics. For example, a log function could be applied to throughput if the application is not sensitive to a particular rate threshold. The specific form of the sigmoid function used here is as follows:

$$U_i(v) = K_i \frac{e^{\alpha_i\ (\beta_i - v)}}{1 + e^{\alpha_i\ (\beta_i - v)}} \tag{7}$$

where K_i is a normalizing factor equal to

$$K_i = \begin{cases} (1 + e^{\alpha_i \beta_i})/e^{\alpha_i \beta_i} & \text{for execution time} \\ 1 & \text{for throughput} \\ (1 + e^{\alpha_i (\beta_i - 1)})/e^{\alpha_i (\beta_i - 1)} & \text{for availability,} \end{cases} \tag{8}$$

β_i is the QoS goal for the metric associated with SSS_i, and α_i is a sensitivity parameter that defines the sharpness of the curve. The sign of α_i determines whether the sigmoid decreases ($\alpha_i > 0$) with v or increases ($\alpha_i < 0$) with v. The maximum value of the utility function defined above is 1. These maximum values occur at $v = 0$ for execution time, $v = 1$ for availability, and for $v \to \infty$ for throughput. A decreasing utility function is used for execution time and an increasing one is used for throughput and availability. The values of α_i and β_i for the `Execution Time`, `Availability`, and `Throughput` SSSs are shown in Table 2.

For SSSs that have a security metric associated to them, the utility function is discretized. The `Secure Comm` utility function is:

$$U_{\texttt{Secure Comm}} = \begin{cases} 0.2 & \text{low security} \\ 0.8 & \text{medium security} \\ 1.0 & \text{high security.} \end{cases} \tag{9}$$

The different levels of security are determined by the type of encryption algorithm used and the key length.

The utility function of the `Secure Store1` and `Secure Store2` SSSs are the same

$$U_{\texttt{Secure Store1}} = U_{\texttt{Secure Store2}} = \begin{cases} 0.4 & \text{low security} \\ 0.7 & \text{medium security} \\ 1.0 & \text{high security.} \end{cases} \tag{10}$$

The use of security mechanisms has an impact on other metrics [14] and can increase execution times and reduce capacity. Table 1 quantifies these impacts, which changes the values of QoS metrics, thereby affecting the value of the utility. The execution time for each service instance within a component is modified by adding the delay values in Table 1 for the component's selected security levels. Similarly, the capacity is modified by the capacity reduction values shown in Table 1. These delay and capacity impacts are fixed and not scaled.

Level	Delay	Capacity
Secure Communication		
Low	0.00	0.0
Medium	0.05	-1.5
High	0.12	-3.0
Secure Storage		
Low	0.00	0.0
Medium	0.04	-1.4
High	0.10	-2.8

Table 1: Performance impacts of security levels.

The global utility function U_g used in the experiments is the weighted geometric mean of the utility functions of all SSSs (the weights used are shown in Table 2):

$$U_g = \left(\prod_{i=1}^{N} U_i^{w_i} \right)^{1/\sum_j^N w_j}. \tag{11}$$

SSS No.	SSS Name	Weight in U_g	α_i	β_i
1	`Secure Comm`	0.2	-	-
2	`Secure Store 1`	0.1	-	-
3	`Secure Store 2`	0.1	-	-
4	`Execution Time`	0.2	0.5	65.0
5	`Availability`	0.2	-100.0	0.95
6	`Throughput`	0.2	-0.5	14.0

Table 2: Weights of the SSSs in the global utility function and values of α_i and β_i.

5.2 List of Service Providers Used

Table 3 shows a list of service providers (SPs) used in our numerical example. There are four SPs for `Road Map`, four for `Weather`, three for `ID Threat`, three for `Make Plan`, and three for `Eval Plan`. These SPs have different performance and cost characteristics. The characteristics include the capacity C_i in tps, the execution time E_i in sec, the availability A_i, and the cost per monthly subscription, in US$. The examples described in this section assume a cost constraint of US$ 5,500.00. We assume that the service description language used for registering the services is QoS-aware similar to [5].

5.3 Patterns Available

For the experiments reported here we consider three types of architectural components: basic (B), load balancing (LB), and fast fault tolerant (fFT). The *basic* architectural pattern consists of a single component that can be instantiated by a single SP. The *load balancing* architectural component consists of a front-end load balancer connector that dispatches requests to one of n components using a weighted round-robin rule. The weight applied to component i is proportional to its capacity, i.e., $C_i/\sum_{j=1}^{n} C_j$. The *fast fault tolerant* architectural pattern consists of a connector that sends a request to n functionally equivalent components and waits for the first to reply.

We present now the analytic models used to derive the availability A, the throughput X, and the execution time E of the three architectural patterns described above. These analytic models consider individual service instances to be black boxes. That is to say the models assume that a service instance when offered a load less than its advertised capacity will provide its advertised levels of QoS. In the expressions below, A_i, X_i, and E_i denote, respectively, the availability, throughput, and execution time of the i-th component of a pattern once allocated to an SP. Note that $X_i = A_i\, C_i$.

The analytic models for the basic architectural pattern are quite trivial: $A = A_i$, $X = X_i = A_i\, C_i$, and $E = E_i$. The analytic models for the LB pattern are as follows. The probability that a request is sent to component i is $C_i/\sum_{j=1}^{n} C_j$ and the availability of that component is A_i. Thus, the average availability is given by

$$A = \sum_{i=1}^{n} \frac{C_i}{\sum_{j=1}^{n} C_j} A_i. \tag{12}$$

The throughput of component i is equal to its capacity C_i multiplied by the probability, A_i, it is available. Thus, the average throughput of the LB pattern is given by

$$X = \sum_{i=1}^{n} \frac{C_i}{\sum_{j=1}^{n} C_j} (C_i \times A_i). \tag{13}$$

A request is completed by component i when the component is available, which happens with probability A_i and when the component i is selected. Thus, the average execution time is

$$E = \sum_{i=1}^{n} \frac{C_i\, A_i}{\sum_{j=1}^{n} C_j\, A_j} E_i. \tag{14}$$

The ratio $(C_i\, A_i)/(\sum_{j=1}^{n} C_j\, A_j)$ represents the fraction of completed requests that are submitted to component i.

The analytic models for the fFT pattern are given below. The fFT pattern is available when at least one of the n components is available. Thus,

$$A = 1 - \prod_{i=1}^{n} (1 - A_i). \tag{15}$$

The throughput for fFT is given by

$$X = X_{\text{argmin}_{i=1,\ldots,n}\{X_i\}}. \tag{16}$$

The execution time E depends on the combination of components that are available. In order to rule out the case in which none of of the components are available, one needs to divide the probability of each combination occurring by A, So, for $n = 2$ we would get

$$\begin{aligned} E \;=\; & \frac{A_1 \times A_2}{A} \min\{E_1, E_2\} + \\ & \frac{A_1(1 - A_2)}{A} E_1 + \frac{A_2(1 - A_1)}{A} E_2. \end{aligned} \tag{17}$$

5.4 QoS Models for the Various SSSs

As indicated in Section 2, an analytic model can be derived for the computation of the various QoS metrics for an SSS as a function of the metrics of the components that compose the SSS. The following general rules, when applied recursively, can be used to obtain the SSS QoS models for availability, execution time, and throughput.

- Availability: the availability of a sequence is the product of the availabilities of the components in the sequence. The availability of a fork-and-join is the product of the availabilities of all of its branches.
- Execution time: the execution time of a sequence is the sum of the execution times of the components in the sequence. The execution time of a fork-and-join is the maximum execution time of each of its branches.
- Throughput: the throughput of a sequence is the minimum throughput of the components in the sequence. The throughput of a fork-and-join is the minimum throughput of its branches.

5.5 Numerical Results

Figure 6 illustrates the variation of the global utility as a function of the number of evaluated architectures during the search. We ran four experiments for different types of neighborhood-finding approaches: unfiltered, $k = 1$, $k = 2$, and $k = 3$, where k is the number of SSS's considered when filtering the neighborhood. We can observe that:

SP i	Capacity C_i (in tps)	E_i (in sec)	Availability (A_i)	Cost (in US$)
Road Map Acme	15.0	0.2	0.900	50
Road Map Pinnacle	12.5	3.0	0.990	100
Road Map ServiceTron	7.5	0.3	0.985	150
Road Map Apex	17.5	1.0	0.975	250
Weather Acme	16.5	0.1	0.980	100
Weather Pinnacle	13.5	5.0	0.999	200
Weather ServiceTron	10.0	0.8	0.995	300
Weather Apex	18.0	0.6	0.990	250
ID Threat Intellifort	13.0	1.5	0.990	500
ID Threat InfoSafe	15.5	2.9	0.985	400
ID Threat CryptIT	17.0	1.8	0.995	550
Make Plan DataCrunch	15.0	48.5	0.940	1500
Make Plan OR Gurus	19.0	92.0	0.990	2000
Make Plan Master Plan	7.0	83.2	0.965	1600
Eval Plan DataCrunch	17.0	5.2	0.985	150
Eval Plan OR Gurus	14.5	9.8	0.995	200
Eval Plan Master Plan	7.5	3.9	0.990	250

Table 3: Service providers (SPs) and their characteristics.

- After 780 evaluations, all approaches converge to the same architecture, described in what follows, that has a global utility equal to 0.656.
- The $k = 2$ approach converges more rapidly than the other approaches (after 54 evaluations). This is followed by the $k = 3$ approach, which converges after 66 evaluations. The $k = 1$ approach converges after 530 evaluations. The unfiltered approach is the last to converge and that happens after 780 evaluations. A single-threaded C++ implementation of the $k = 2$ approach was able to evaluate 3,000 architectures in 2.25 seconds on a Linux system with two dualcore 2.6 GHz Opteron CPUs. The other approaches executed in a similar amount of time. This demonstrates that these architecture search techniques can be used in near real-time.

The final architecture obtained through this process is shown in Fig. 7. Specific details about the SPs used by each service type, architectural patterns, number of instances of each, and security levels are given in Table 4. The performance characteristics of this architecture are as follows. The numbers in parentheses indicate the corresponding QoS values for the base (starting architecture): execution time = 59.4 (56.7) sec, availability = 0.965 (0.889), and throughput = 12.1 (12.4) tps. The cost of the near optimal architecture is US$ 5,500, which is equal to the cost constraint. The cost of the base architecture is US$2,350. It should be noted from Table 2 that security accounts for 40% of the weight of the global utility and availability for 20%. Table 4 indicates that the near-optimal architecture uses secure communication and secure storage while the base architecture did not. Also, the availability of the near optimal architecture is higher than that of the base architecture. A different utility function would yield different results.

One of the advantages of the approach presented here is that it can be used for autonomous adaptation of an archi-

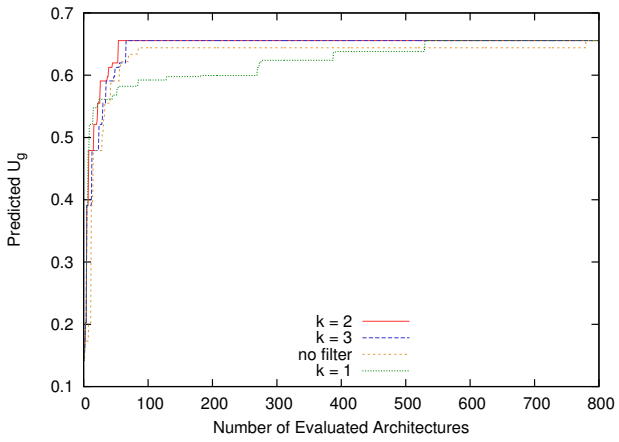

Figure 6: Variation of the global utility during the search.

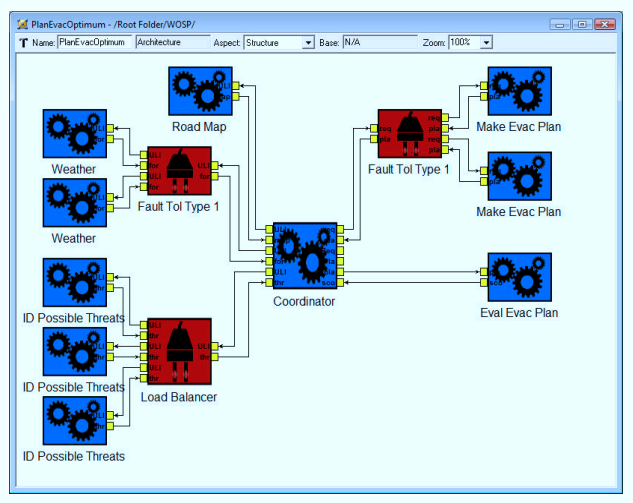

Figure 7: Optimal architecture.

Service Type	Service Provider(s)	Pattern	No. Instances	Sec. Comm. Level	Sec. Sto. Level
`Road Map`	Pinnacle	Basic	1	Low	Low
`Weather`	Acme and Pinnacle	fFT	2	Low	Low
`ID Possible Threats`	Intellifort, InfoSafe, and CryptIT	LB	3	Medium	Medium
`Make Eval Plan`	Data Crunch and OR Gurus	fFT	2	Medium	Medium
`Eval Evac Plan`	Data Crunch	Basic	1	Medium	Low

Table 4: Details of the near optimal architecture.

Service Type	Service Provider(s)	Pattern	No. Instances	Sec. Comm. Level	Sec. Sto. Level
`Road Map`	Acme and Apex	fFT	2	Low	Low
`Weather`	Acme and Pinnacle	fFT	2	Low	Low
`ID Possible Threats`	Intellifort and InfoSafe	LB	2	Medium	Low
`Make Eval Plan`	Data Crunch and OR Gurus	fFT	2	Medium	Medium
`Eval Evac Plan`	Data Crunch and OR Gurus	fFT	2	Medium	Low

Table 5: Details of the near optimal architecture after adaptation.

tecture. Consider that after we settled for the near optimal architecture described above, the Pinnacle Road Map SP degrades its performance. In particular, its capacity goes down to 6.5 tps from the original 12.5 tps, its execution time goes up from 3 sec to 12 sec, and its availability decreases from 0.99 to 0.75. With the performance problem of the Pinnacle Road Map SP, U_g plummets to 0.003. The approach presented here first replaces the Pinnacle Road Map service with the Acme Road Map service, and U_g improves to 0.160 (it was previously 0.656). It then searches for a new architecture and finds one with U_g equal to 0.643. Figure 8 shows the variation of U_g during the adaptation process. The details of the adapted architecture can be seen in Fig. 9 and Table 5. The performance characteristics of the new adapted archictecture are (values in parentheses are pre-adaptation): execution time = 58.8 sec (68.4 sec), availability = 0.985 (0.731), and throughput = 12.1 tps (4.9 tps). In the adapted architecture, the secure storage level is reduced from medium to low.

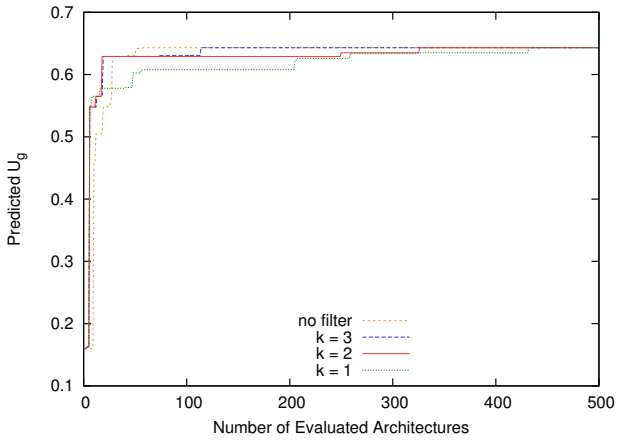

Figure 8: Variation of the global utility during the search due to adaptation.

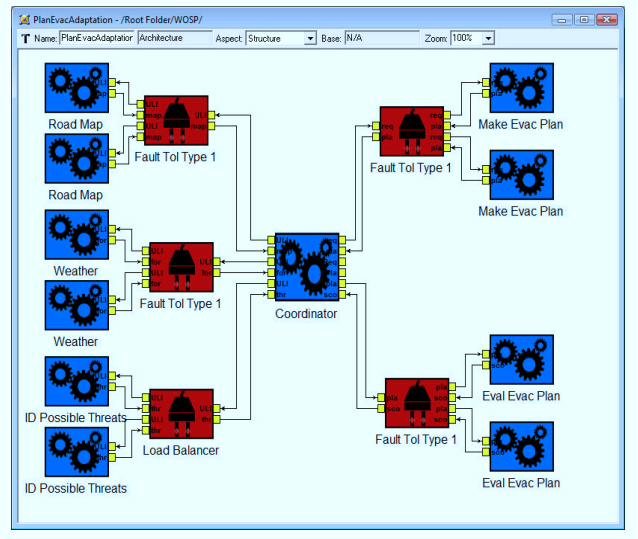

Figure 9: Optimal architecture after adaptation.

6. CONCLUDING REMARKS

SASSY brings several new contributions with respect to the existing autonomic service composition approaches (e.g., JOpera [18]): (1) models the system's requirements in a high-level visual language that is intuitively understood by the domain experts, and semantically well-defined through a domain ontology; (2) maintains four types of run-time models, which are synchronized with one another, and collectively support a rich set of alternatives for satisfying the system's requirements; and (3) provides a uniform approach to automated composition, adaptation, and evolution of SOA software systems.

The experimental results demonstrate that SASSY's approach to architectural search can find near-optimal architectures in near real-time on small and medium sized problems. In fact, our implementation was able to perform 3,000 architecture evaluations in just over 2 seconds. The discovered near-optimal architecture provided a substantial im-

provement over the initial SASSY base architecture. Our results also show that the search procedures used for initial optimization were able to successfully adapt to the degradation of a key service instance. The adaptation example presented here indicates that a small change in the environment can lead to substantial changes in the structure and features of near-optimal architectures. This point illustrates the value of autonomic management in SOAs: autonomic management can elevate the end user experience by reacting to emergent problems on the scale of seconds, while human administrators would need hours, possibly days, to devise and implement a new architecture that would properly restore the application's performance.

In our future work, we will devise an algorithm for the autonomatic generation of SSS performance models to make SASSY's architecture optimization more generally applicable. We already have some of the rules for this model generation process (see Section 5.4), but more work is required to devise a general algorithm.

While this paper introduces a relatively simple neighborhood filter for hill-climbing, more investigation is required to assess its value. The filter could be expanded to focus on other aspects of the problem including the top n components contributing to a poor metric value in an SSS. An effective neighborhood filter could also allow us to expand the neighborhood by constructing neighbors through multiple modifications of the currently visited architecture. This could help the hill-climbing heuristic to avoid local optima and converge more rapidly.

In addition to hill-climbing, we plan to consider other local search techniques including beam search, simulated annealing, and tabu search. Evolutionary approaches such as genetic algorithms may also prove effective and could be readily applied to the SASSY architecture search process. We would like to test the SASSY architecture optimization process against a large set of test problems ranging in size and composition. This will help us to quantify the strengths and weaknesses of various architecture optimization search procedures in SASSY.

Finally, it is worth noting that SASSY's approach can also be used for evolution. When requirements change, they need to be reflected at the SAS level. From that point, SASSY regenerates a near optimal architecture that satisfies the requirements of the evolved system.

7. REFERENCES

[1] A. Agrawal, G. Karsai, and F. Shi. Generative programming via graph transformations in the model-driven architecture. In *OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, pages 229–240, Seattle, WA, Nov. 2002.

[2] A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. Technical report, Institute for Software Integrated Systems, Nov. 2003.

[3] M.N. Bennani and D.A. Menascé. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC'05)*, pages 229–240, Seattle, WA, June 2005.

[4] M.B. Blake. Decomposing composition: Service-oriented software engineers. *IEEE Software*, 24:68–77, Nov. 2007.

[5] A. D'Ambrogio. Model-driven WSDL extension for describing the qos of web services. In *IEEE International Conference on Web Services (ICWS'06)*, pages 789–796, Chicago, IL, Sept. 2006.

[6] E. Dashofy, A. van der Hoek, and R.N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 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 *Proceedings of the 12th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MODELS'09*, Denver, CO, Oct. 2009.

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

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

[10] 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.

[11] J.W. Lee, R. Mazumdar, and N. B. Shroff. Non-convex optimization and rate control for multi-class services in the internet. *IEEE/ACM Transactions on Networking*, 13(4):827–840, Aug. 2005.

[12] 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.

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

[14] D.A. Menascé. Security performance. *IEEE Internet Computing*, 7(3):84–87, May 2003.

[15] D.A. Menascé, E. Casalicchio, and V. Dubey. A heuristic approach to optimal service selection in service oriented architectures. In *Proceedings of the 7th International Workshop on Software and Performance (WOSP 2008)*, pages 13–24, Princeton, NJ, June 2008.

[16] O. Nano and A. Zisman. Realizing service-centric software systems. *IEEE Software*, 24(6):28–30, Nov. 2007.

[17] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, Nov. 2007.

[18] C. Pautasso, T. Heinis, and G. Alonso. JOpera: Autonomic service orchestration. *IEEE Data Engineering Bulletin*, 29(3):32–39, Sept. 2006.