

# SASSY:

## A Framework for Self-Architecting Service-Oriented Systems

Daniel A. Menascé, Hassan Gomaa, Sam Malek, and João P. Sousa,  
George Mason University

// The SASSY self-architecting approach makes systems self-adaptive, self-healing, self-managing, and self-optimizing. //



**DESIGNING LARGE-SCALE** distributed software systems presents the challenge of providing a way for the software to adapt to changes in the computing environment (for example, workload changes and failures) and requirements. Self-adaptive software systems monitor the computing environment and adjust their structure and behavior at runtime in response to changes in the environment. Software architectures provide an appropriate level of granularity for adaptation.<sup>1</sup> However, approaches to self-adaptive software systems<sup>2</sup> assume that a satisfactory architecture is determined a priori.

SASSY (Self-architecting Software Systems) is a model-driven framework targeted at dynamic settings in which a system's requirements might change. SASSY extends the state of the art through self-architecting of distributed software systems. Throughout the system's life cycle, SASSY maintains a near-optimal architecture for satisfying functional and quality-of-service (QoS) requirements. A given architecture's quality is expressed by a user-provided utility function and represents one or more desirable system objectives. SASSY aims for an environment in which many service providers are

available to deliver functionally equivalent services at different QoS levels and different costs.

Unfortunately, the goal of self-architecting directly from requirements is unrealistic. SASSY addresses a more constrained but challenging and increasingly important class of software systems: service-oriented architectures (SOAs).<sup>3</sup> SASSY deals with the system's composition at deployment and provides runtime adaptation in response to changing operating conditions. It lets practitioners construct highly flexible and dynamic service-oriented systems.

### The Challenge of Self-Architecting

Tuning a software system's architecture to deliver optimal QoS is complex. Once software architects define the application logic, they can apply architectural patterns that promote the desired system qualities. Unfortunately, choosing a pattern that addresses certain QoS concerns might negatively affect other concerns.<sup>4</sup> For example, replication to improve a component's reliability creates a system that's more reliable but also more costly to deploy and operate. The architect must make trade-offs reflecting the stakeholders' priorities.

In SASSY, domain experts capture the intended application requirements using a visual activity-modeling language. From this, SASSY automatically derives an architecture that specifies which services are part of the system and how they're coordinated. Consistent with the fundamental premise of SOA, we assume that third parties author service implementations, which SASSY can find using service discovery mechanisms.<sup>5</sup> The key challenges are how to automatically determine

the original architecture and how to continuously and dynamically adapt the architecture and runtime system to maintain QoS goals.

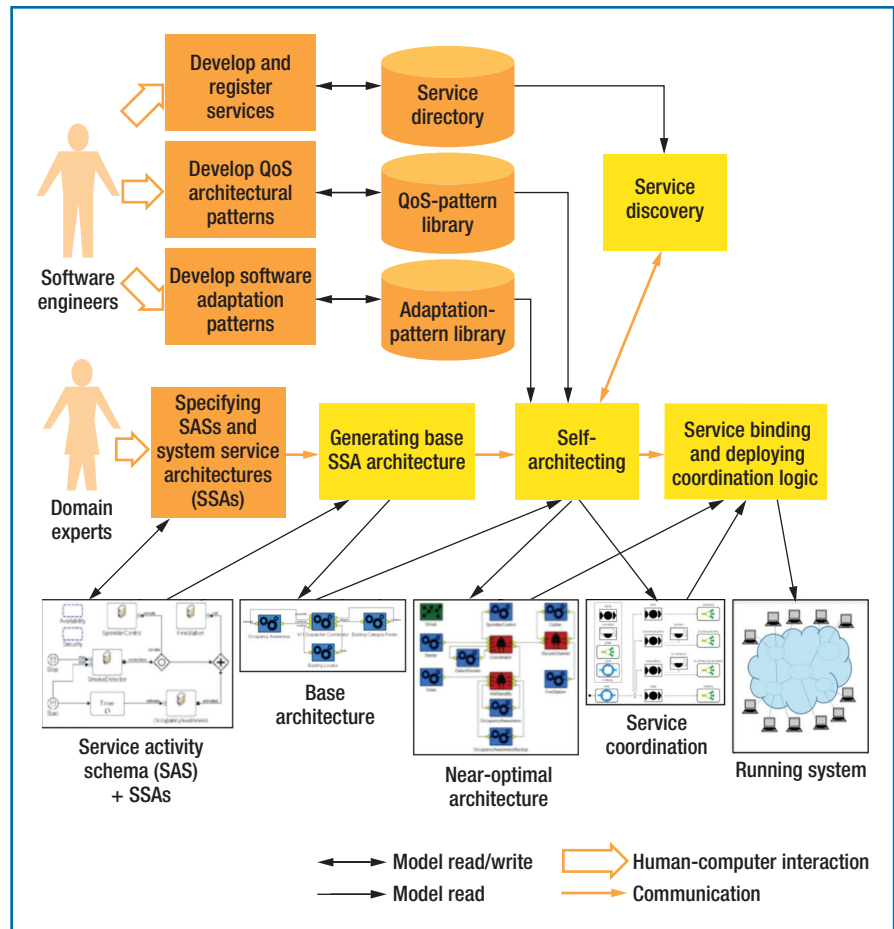
Automatic architecting and rearchitecting is an NP-hard optimization problem. Through an efficient heuristic, SASSY generates near-optimal architectures and service selections to mitigate the problem's computational complexity.

## The SASSY Approach

Figure 1 shows how SASSY initially generates a software architecture. As we just mentioned, third-party software engineers develop services and register them in a service directory so that SASSY can discover them. Software architects develop QoS architectural patterns, which are patterns of service composition (such as replication for fault tolerance, load balancing for increased throughput, and mediation for secure communication). A software performance engineer associates these patterns with parameterized QoS analytic models that determine how a particular pattern influences various QoS metrics of interest. In addition, the architects develop software adaptation patterns to dynamically adapt an executing system from its current architecture to another at runtime.

Domain experts specify *service activity schemas* (SASs), which express system requirements. The modeling constructs are defined in a domain ontology that unambiguously distinguishes different concepts and elements to facilitate service discovery. The domain experts annotate the SAS models with QoS goals for system attributes such as performance, availability, and security.

SASSY uses the SAS requirements to automatically generate a base *system service architecture* (SSA) composed of structural and behavioral views. These views consist of components (associated with service providers) and con-



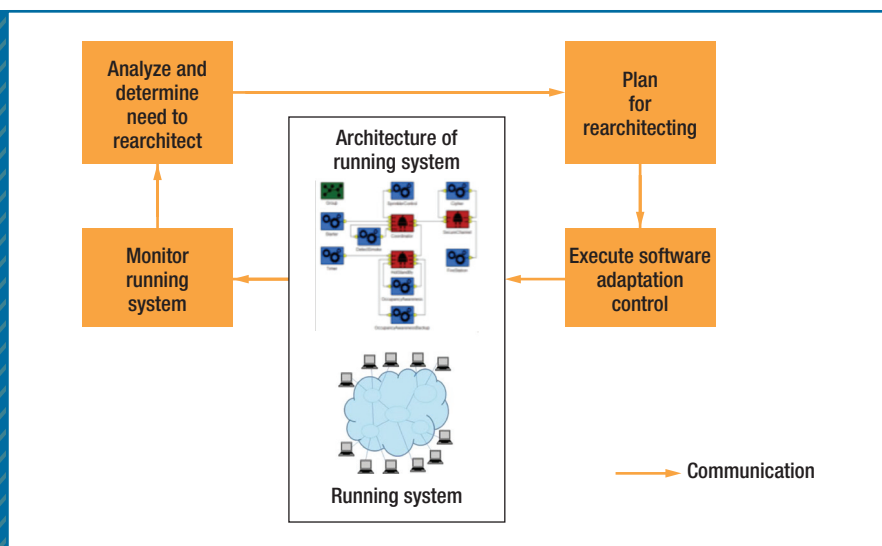
**FIGURE 1.** SASSY architecture generation. SASSY changes software architects' function from working on ad hoc solutions for each application to encoding generic knowledge about architectural patterns and their influence on QoS.

nectors. SASSY derives an optimized architecture from the base architecture by selecting the most suitable service providers and by applying QoS architectural patterns. It determines this optimized architecture with the help of QoS analytic models and optimization techniques aimed at finding near-optimal choices that maximize system utility. SASSY produces a running system by instantiating the optimized architecture through service binding and deploying the coordination logic.

Figure 2 illustrates SASSY's autonomic capabilities, which follow the MAPE-K (monitor, analyze, plan, execute, knowledge) loop.<sup>6</sup> SASSY's moni-

toring component gathers QoS metric values and passes them to the analyzer, which aggregates the data and computes the system's utility. If the utility falls below a stakeholder-specified threshold, the system sends a request to the architecture planner to automatically determine a near-optimal architecture and a corresponding set of service providers. The self-adaptation component executes the changes to the running system through the adaptation patterns.

SASSY handles changing requirements through modifications at the SAS level that trigger the generation of the revised architecture and the runtime adaptation to the new architecture.



**FIGURE 2.** SASSY's runtime self-rearchitecting. SASSY maintains the near optimality of the system at runtime, adapting to changes in operating conditions and to changes in QoS goals.

## Service Activity Schemas

SAS is a visual requirements specification language inspired by the Business Process Modeling Notation (BPMN; [www.bpmn.org](http://www.bpmn.org)). Like BPMN, SAS is intuitive. Unlike BPMN, SAS is formally specified and has the semantics for generating executable architectural models.<sup>7</sup> Figure 3a shows the modeling constructs in SAS.

The first step in constructing an SAS model is to select the required service usages and activities from the domain ontology. The language distinguishes local activities from service usages—that is, activities that external entities perform as services for requesters. Next, the domain expert specifies the sequence of interactions among service usages and activities. The expert does this by using gateways that manage the flow of events, which represent messages exchanged between service usages. Supported gateways include

- inclusive (Conditional-Or),
- exclusive (Fork), and
- parallel (And-Join).

Finally, the domain expert specifies

the QoS requirements through *service sequence scenarios* (SSSs). An SSS is a well-formed subgraph of SAS—that is, it satisfies all the syntactic constraints of a complete SAS. To manage complexity and ensure autonomy, SAS models can be registered and reused as services in other SAS models.

Figure 3b shows an SAS for monitoring fire emergencies in public buildings equipped with smoke detectors and fire sprinklers. This SAS contains three service usages and a composite activity. The reception of the *smokeDet* event starts two parallel threads of control. First, the emergency phone system tries to contact the building occupants. Second, an external building-locator service finds the incident's physical address. If the system makes contact with the occupants, it forwards the phone call to an operator; otherwise, it sends an *investigate* event to the police station.

Following the building-locator service, two other external services, occupancy awareness and building-category finder, determine the number of occupants in the building and the building's type. The system then uses

this information to request appropriate assistance.

Figure 3c shows an SSS associated with end-to-end availability. The SAS model of Figure 3b is reused as the 911 dispatcher service in Figure 3d, which is a higher-level SAS that coordinates capabilities across several organizations in response to a fire emergency. The *smokeDet* event in Figure 3d initiates the execution of the SAS model in Figure 3b.

## Generating the Base Architecture

The SSA offers structural and behavioral models for an SOA system. Unlike traditional software architectural models, which are used mainly during the design phase SASSY uses the SSA at runtime. The SSA is an up-to-date representation of the running software system, and supports (manual or automatic) runtime reasoning with respect to evolution and adaptation.

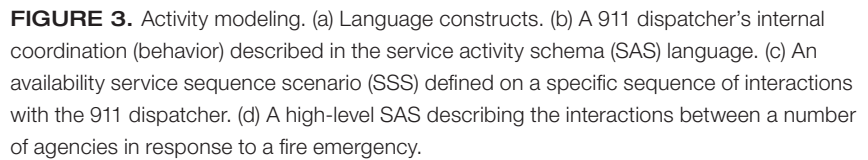
The SSA's structural models are based on xADL (eXtensible Architectural Description Language),<sup>8</sup> which provides the traditional component-and-connector view of software architecture. We extended xADL by introducing service instances modeled as software components. A service instance is the realization of a service type defined in the ontology. The SSA also maps each service instance with the concrete service provider. The middleware facilities enabling integration and communication among the services are modeled as software connectors. Finally, the components and connectors bind to one another using both required and provided interfaces. Figure 4 shows the structural view of two alternative SSA models corresponding to the example in Figure 3b.

The SSA's behavioral models show how service instances collaborate to fulfill the system's requirements. A behavioral model corresponds to the executable logic of service coordination

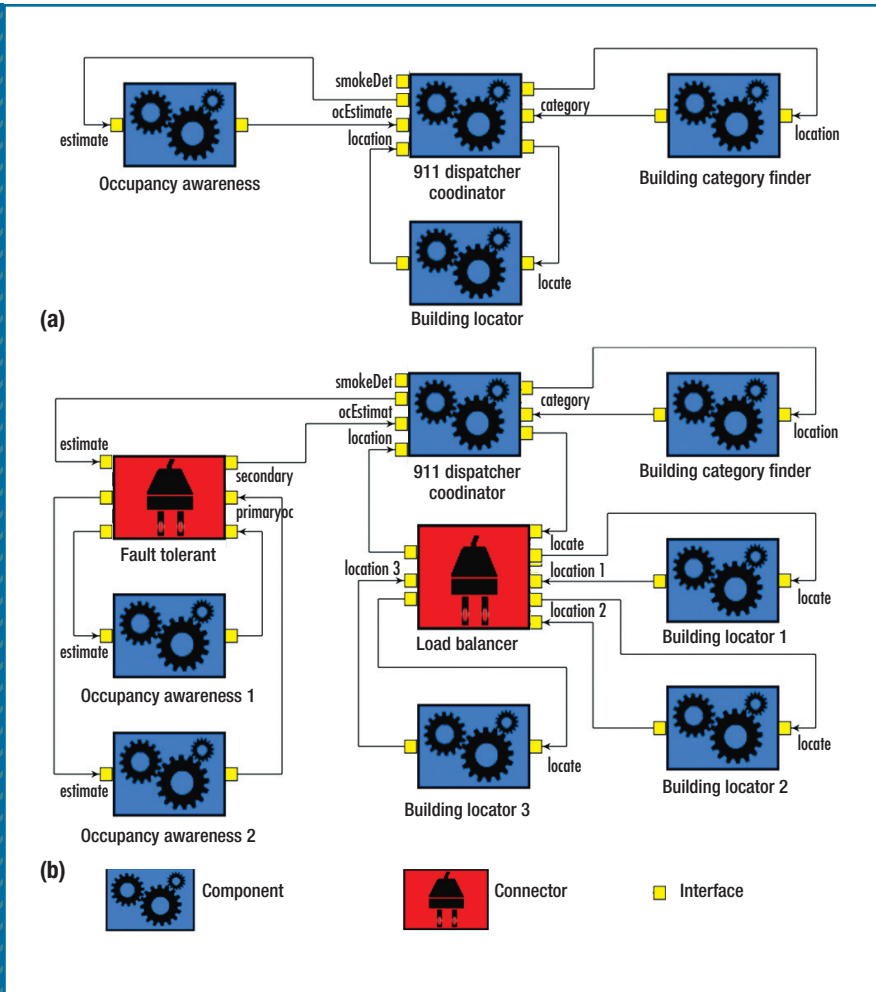
To generate the base architecture from an SAS model, we used a model-driven engineering (MDE) approach, realized on top of the Generic Modeling Environment (GME; [www.isis.vanderbilt.edu/Projects/gme](http://www.isis.vanderbilt.edu/Projects/gme)) and the associated Graph Rewriting and Transformation engine (Great; [www.isis.vanderbilt.edu/tools/GReAT](http://www.isis.vanderbilt.edu/tools/GReAT)). GME's metamodeling language enables a precise specification of a language's semantics, in this case, the semantics of SAS. We developed a model-to-model transformer that executes graph transformation rules to generate SSA models from SAS models. Service usages in SAS are transformed to the equivalent components and connectors to form the SSA's structural model in xADL. The SAS sequence of activities and services is transformed to a coordinator component, forming the states and transitions of the SSA's behavioral model, specified in FSP.

## Self-Architecting

QoS architectural patterns capture strategies known to promote specific



QoS metrics and a corresponding QoS model. For instance, the fault-tolerant architectural pattern in Figure 4b influences two QoS metrics: availability and execution time. This pattern’s behavior is such that its execution is considered complete whenever the first of the two components (occupancy



**FIGURE 4.** Structural views of the architectural model generated for the 911 dispatcher in Figure 3. (a) The base architecture. (b) An adapted architecture. SASSY uses these models at runtime to assess system optimality and adaptation.

awareness 1 and 2) responds. This pattern's availability is a function of the individual components' availabilities, and its execution time is a function of the individual components' execution times.

SASSY's self-architecting method determines an architecture with a set of service providers that maximizes a utility function for the software system. Utility functions, commonly used in autonomic computing, express a system's usefulness as a function of its attributes' values.<sup>2</sup> For example, a software system's utility might progres-

sively decrease to zero as its response time increases.

Utility functions quantify QoS trade-offs. When multiple attributes (such as response time, availability, or security) are of interest, a different utility function for each attribute combines (such as using a weighted sum or a weighted geometric mean) into a single multivariate utility function for the entire system.

SASSY combines a utility function for each SSS into a single global utility function. SASSY monitors the utility's value for the entire system and triggers

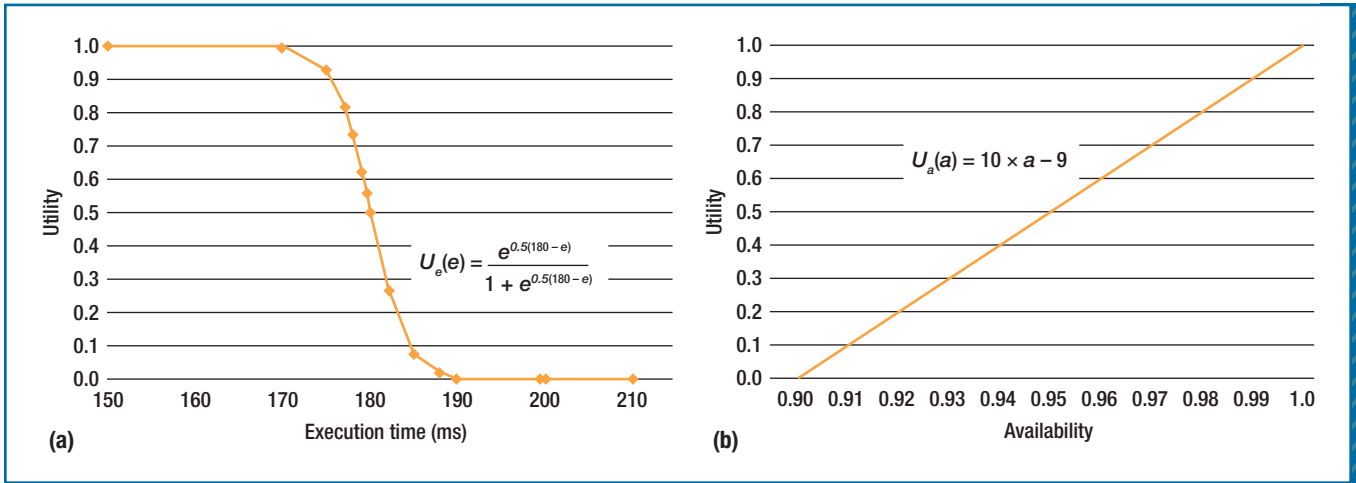
self-adaptation when the utility falls below a predefined threshold.

This self-architecting problem necessitates finding an architecture (including a set of service providers) in a way that maximizes the software system's overall utility function, subject to a set of constraints (such as cost). For  $p$  possible architectural patterns,  $s$  service providers for each service usage, and  $n$  service instances in the SSA,  $(ps)^n$  possible architectures exist.

To solve this problem, we use a hill-climbing-based combinatorial search. Starting from the current architecture, we search the space of possible architectures by generating a set of its neighbors. We generate neighbors by replacing components or composite components with QoS architectural patterns that promote increased utility. Then, we perform a quasi-optimal service allocation for each service provider for each architecture in the neighborhood.<sup>11</sup> The architecture with the largest utility becomes the new visited point in the search space. The search stops if no architecture in the neighborhood increases the utility's value. The hill-climbing search eliminates solutions that violate cost constraints.

Figure 5 represents the results of a numerical example. Figure 5a shows the utility function for availability,  $U_a(a)$ , for the SSS of Figure 3c. Figure 5b shows the utility function for execution time,  $U_e(e)$ , for an SSS with the same structure (invocation of the building-locator service followed by an invocation of the occupancy-awareness service) as in Figure 3c. The global utility in this example is  $0.4 U_a(a) + 0.6 U_e(e)$ .

Table 1 shows the execution time and availability for the service providers in our numerical example. Table 2 shows the global utility values for eight combinations of QoS architectural patterns involving the service types and providers in Table 1. The last row indicates that SASSY instantiated the



**FIGURE 5.** Utility functions defined by domain experts for the system in Figure 3, concerning (a) execution time,  $U_e(e)$ , and (b) availability,  $U_a(a)$ .  $U_e$  is optimal for execution times under 180 ms and then decreases rapidly, becoming useless for execution times over 190 ms.  $U_a$  is only acceptable for an availability over 90 percent, increasing linearly to become optimal for an availability of 100 percent.

building-locator service type using a load balancer (LB) QoS architectural pattern with service providers BL1, BL2, and BL3. To instantiate the occupancy-awareness service, SASSY used a fault-tolerant pattern with service providers OA1 and OA3. The example in this row, which corresponds to Figure 4b, achieved the largest global utility. The next-best utility used the basic component (BC) pattern with service provider BL3, and the first-respond fault-tolerant (FFT) pattern with providers OA1 and OA3.

**TABLE 1**

**The execution time and availability for the service providers in our numerical example.**

Service type	Service provider	Execution time (ms)	Availability	Monthly cost (US\$)
Building locator	BL1	80	0.99	400
	BL2	75	0.98	380
	BL3	70	0.96	350
Occupancy awareness	OA1	110	0.97	400
	OA2	125	0.98	380
	OA3	100	0.96	420

## Self-Adaptation

A software adaptation pattern prescribes the steps needed to dynamically adapt a system at runtime from one configuration to another without jeopardizing its functionality.<sup>12</sup> An adaptation pattern can be modeled as a state machine that defines the sequence of states a service goes through to transition from an active to a quiescent state. A service is active when it engages in its normal operation; it transitions to quiescence when it's no longer operational and its clients no longer communicate with it. Before arriving at quiescence, a service might transition

through several other states (such as a passive state).

For each QoS architectural pattern, a corresponding software adaptation pattern specifies how the system self-adapts to incorporate the pattern into the configuration. Consider the adaptation steps required to apply the fault-tolerant pattern to the architecture of Figure 4a. Software adaptation control (see Figure 2) needs to drive occupancy awareness (see Figure 4a) to quiescence and to request the 911 dispatcher coordinator to suspend communication

with occupancy awareness. The coordinator continues to communicate with the initial configuration's other components.

When occupancy awareness reaches quiescence, software adaptation control removes the connection between occupancy awareness and the 911 dispatcher coordinator. SASSY then adds the second occupancy-awareness service and the fault-tolerant connector to the configuration. Next, it links components to the new configuration to arrive at the architecture of Figure 4b.

TABLE 2

The global utility for eight combinations of QoS architectural patterns, for the service types and providers in Table 1.\*

Combination of QoS architectural patterns		Global utility	Monthly cost (US\$)
Building locator	Occupancy awareness		
BC (BL1)	BC (OA1)	0.245	800
BC (BL1)	BC (OA2)	0.281	780
BC (BL3)	FFT (OA1, OA2)	0.505	1,130
BC (BL3)	FFT (OA1, OA3)	0.831	1,170
LB (BL1, BL2)	BC (OA1)	0.246	1,180
LB (BL1, BL2, BL3)	BC (OA1)	0.286	1,530
LB (BL1, BL2)	FFT (OA1, OA2)	0.357	1,560
LB (BL1, BL2, BL3)	FFT (OA1, OA3)	0.877	1,950

\*If the cost constraint accommodates \$1,950 or more, SASSY chooses the solution in red, which maximizes the global utility. BC stands for the basic component pattern, LB stands for the load balancer pattern, and FFT stands for the first-respond fault-tolerant pattern.

SASSY sends a reactivate command to the 911 dispatcher coordinator. Then, it delivers the estimate message to the fault-tolerant connector, which in turn invokes the two occupancy-awareness services but forwards only the response of the first service responding to the requester.

A similar approach dynamically replaces the building-locator service in Figure 4a with the LB pattern in Figure 4b.

## The SASSY Development Environment


We used GME to build the SASSY modeling environment, which lets domain experts construct SAS models and visualize generated architectural models. SASSY's monitoring, analysis, planning, and adaptation components are services that rely on an enterprise service bus technology (such as Apache ServiceMix) to subscribe to and receive messages of interest in a distributed setting. We developed the model transformation and optimization capabilities as GME plug-ins that can read and manipulate GME models.

SASSY's coordination support is based on the XTEAM environment, a tool that executes FSP models.<sup>13</sup> Once SASSY discovers the appropriate service providers and finds a suitable architecture, it uses XTEAM to execute the architecture model with actual service invocations. XTEAM also allows temporary storage of results returned from services to enable stateful coordination among services and long-living activities.

Our experience with applying SASSY across diverse domains provides us with several avenues for future research, including

- modeling distributed transactions among services,
- extending SASSY for execution in decentralized settings,
- dealing with space and time when discovering cyberphysical services, and
- automatically reconciling conflict-

ing QoS requirements in collaborative multiuser systems.

Currently, the automated architecture generation assumes that SAS models are functionally complete and correct. We're investigating how to incrementally build SOA systems and extend the existing tool support for simulation and testing. 

## Acknowledgments

US National Science Foundation grant CCF-0820060 supported this work. We thank Naeem Esfahani, John Ewing, and Koji Hashimoto for their contributions to SASSY.

## References

1. J. Kramer and J. Magee, "Self-Managed Systems: An Architectural Challenge," *Proc. Future of Software Eng. (FOSE 07)*, IEEE CS Press, 2007, pp. 259–268.
2. M.C. Huebscher and J.A. McCann, "A Survey of Autonomic Computing—Degrees, Models, and Applications," *ACM Computing Surveys*, vol. 40, no. 3, 2008, pp. 1–28.
3. M.P. Papazoglou et al., "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, 2007, pp. 38–45.
4. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.
5. S. Weerawarana et al., *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*, Prentice Hall, 2005.
6. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003, pp. 41–50.
7. N. Esfahani et al., "A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems," *Proc. 12th Int'l Conf. Model Driven Eng. Languages and Systems*, Springer, 2009, pp. 591–605.
8. E.M. Dashofy, A. van der Hoek, and R.N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," *Proc. Working IEEE/IFIP Conf. Software Architectures*, IEEE CS Press, 2001, pp. 103–112.
9. J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd ed., John Wiley & Sons, 2006.
10. D.A. Menascé et al., "QoS Architectural Patterns for Self-Architecting Software Systems," *Proc. 7th Int'l Conf. Autonomic Computing and Communications (ICAC 10)*, ACM Press, 2010, pp. 195–204.
11. D.A. Menascé et al., "A Framework for Utility-Based Service Oriented Design in Sassy," *Proc. 1st Joint WOSP/SIPEW Int'l*

*Conf. Performance Eng. (WOSP/SIPEW 10)*, ACM Press, 2010, pp. 27–36.

12. H. Gomaa et al., “Software Adaptation Patterns for Service-Oriented Architectures,” *Proc. ACM Symp. Applied Computing (SAC 10)*, ACM Press, 2010, pp. 462–469.
13. G. Edwards, S. Malek, and N. Medvidovic, “Scenario-Driven Dynamic Analysis of Distributed Architecture,” *Proc. 10th Int’l Conf. Fundamental Approaches to Software Eng.*, Springer, 2007, pp. 125–139.

## IEEE Software Call for Papers:

Special Issue on

# TECHNICAL DEBT

Submission deadline:

1 April 2012

Publication:

Nov./Dec. 2012

[www.computer.org/  
software/cfp6](http://www.computer.org/software/cfp6)

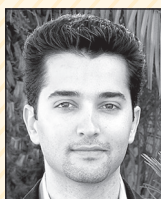
## ABOUT THE AUTHORS



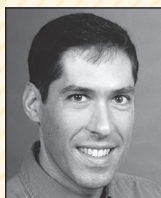
**DANIEL A. MENASCÉ** is a senior associate dean and professor of computer science at George Mason University's Volgenau School of Engineering. His research interests include autonomic computing, service-oriented computing, software performance engineering, and modeling and analysis of computer systems. Menascé has a PhD in computer science from the University of California, Los Angeles. He's a senior member of IEEE and a fellow of the ACM. Contact him at [menasce@gmu.edu](mailto:menasce@gmu.edu).



**HASSAN GOMAA** is a professor in George Mason University's Department of Computer Science and is a former department chair. His research interests include software engineering, software modeling and design, the design of real-time and distributed software, software product-line engineering, software architectures and patterns, dynamic software adaptation, and software performance engineering. Gomaa has a PhD in computer science from Imperial College London. He's a member of IEEE and the ACM. Contact him at [hgomaa@gmu.edu](mailto:hgomaa@gmu.edu).



**SAM MALEK** is an assistant professor of computer science at George Mason University. His research interests include architecture-based software development and deployment, embedded and distributed systems, middleware solutions, and quality-of-service analysis. Malek has a PhD in computer science from the University of Southern California. He's a member of IEEE and the ACM. Contact him at [smalek@gmu.edu](mailto:smalek@gmu.edu).



**JOÃO P. SOUSA** is an assistant professor of computer science at George Mason University. His research interests include software architectures, design languages, and security for ubiquitous computing, with applications to smart spaces, energy grids, and self-configuring and autonomic cyberphysical systems. Sousa has a PhD in computer science from Carnegie Mellon University. He's a member of IEEE. Contact him at [jpsousa@gmu.edu](mailto:jpsousa@gmu.edu).

## Take the CS Library wherever you go!



All 2011 issues of IEEE Computer Society magazines and Transactions are now available to subscribers in the portable ePub format.

Just download the articles from the Computer Society Digital Library, and you can read them on any device that supports ePub. For more information, including a list of compatible devices, visit

[www.computer.org/csdl/epub\\_info.html](http://www.computer.org/csdl/epub_info.html)



IEEE

IEEE  computer society