# Self-Architecting Software SYstems (SASSY) from QoS-Annotated Activity Models

Sam Malek, Naeem Esfahani, Daniel A. Menascé, João P. Sousa, Hassan Gomaa
*Department of Computer Science*
*George Mason University*
*{smalek, nesfaha2, menasce, jpsousa, hgomaa}@gmu.edu*

## Abstract

*As the complexity associated with software development has increased, software engineers have sought novel ways to represent, reason about, and compose large-scale software systems. However, the majority of these approaches are geared to technically well versed engineers, making them unwieldy for use in a growing class of real-world pervasive computing systems. In this paper, we propose a new approach intended to address the current shortcomings in service-oriented software systems. Given the functional and QoS requirements specified by a domain expert in an activity oriented modeling language, an architecture satisfying the requirements is generated. We describe our approach in the context of a framework, entitled Self-Architecting Software SYstems (SASSY), which shapes our ongoing research and aims to automate the composition, analysis, adaptation, and evolution of service-oriented software systems.*

## 1. Introduction

Service-Oriented software systems are associated with a number of advantages, including the ability to abstract away the heterogeneity of the communication and computational substrate, the decoupling of service providers from consumers, and the flexibility of dynamically discovering and binding to services. As a result, service-oriented systems are fast becoming pervasive in a variety of computing domains. Service-Oriented Architecture (SOA) [13][14] is seen as the solution to many of the problems facing modern day enterprise and e-commerce software systems, which continue to grow in size and complexity. Similarly, the proliferation of portable and embedded computing devices and the recent advances in wireless network connectivity have made the SOA paradigm a viable option for an emerging class of pervasive systems.

On top of the traditional complexity associated with the construction of any large scale software system, software developers are forced to cope with additional sources of complexity introduced by the growing class of mobile and pervasive systems, which are innately dynamic and unpredictable. Moreover, these systems are often long-lived, and are expected to adapt not only to the fluctuating execution conditions (e.g., network throughput, available battery power), but also to changes in their operational requirements (e.g., functional features).

As the complexity associated with software development has increased, software engineers have sought novel ways to represent, reason about, and synthesize large-scale software systems. However, many of these approaches have relied heavily on human reasoning and manual intervention, making them unwieldy for use in highly dynamic and unpredictable pervasive settings. Moreover, the majority of existing approaches have primarily targeted technically savvy software engineers, as opposed to domain users, thereby making them difficult to use in many real-world industrial settings.

In this paper, we present a novel approach targeted at the challenges of automatically composing an SOA software system in dynamic, unpredictable, and pervasive SOA software systems. We rely on the domain expert's functional and Quality of Service (QoS) requirements expressed in an activity-oriented language to automatically generate an "optimal" architecture. Architecture generation consists of (1) selection of the appropriate service providers that satisfy the user's activities, and (2) application of suitable architectural patterns to compose the services into a cohesive software system. The approach ensures that the generated architecture satisfies the key quality attributes (e.g., latency, security) specified by the user. The work presented in this paper is part of a framework, entitled *Self-Architecting Software SYstems (SASSY)*, which shapes our ongoing research effort and aims to automate the composition, analysis, adaptation, and evolution of SOA software systems.

The remainder of the paper is organized as follows. Section 2 introduces a motivating example used throughout this paper for exposition purposes. Section 3 provides an overview of the SASSY framework. The rest of the paper focuses on a particular aspect of the SASSY framework dealing with the process of generating a viable architecture. To that end, Section 4 describes the architecture generation methodology, Section 5 describes the activity-oriented language used to model the system's

requirements, Section 6 discusses the architectural modeling language, and finally Section 7 presents the architecture generation process. The paper concludes with an outline of our future work.

## 2. Motivating Example

As a motivating example, consider the following application scenario taken from the emergency response domain. Smart buildings equipped with various sensors, such as smoke detectors, seismic probes, and cameras, provide monitoring data for emergency phenomena, such as fire, structural damage, and burglary. Other sensors, such as fire sprinklers, electronic locks, and exit lights, provide means of responding to an emergency situation. The services provided by these sensors is made available publicly (to authorized consumers) via service discovery directories (e.g., UDDI). Information fusion services provisioned by various emergency response agencies (hospitals, police headquarters, firefighting stations) aggregate and process the data received from the sensors, determine the occurrence of emergency situations, and publish the results onto a wide-area event notification system (e.g., Siena). A management service listens to emergency events and deals with classification and determination of the appropriate course of action, as well as with supporting the field work of emergency response teams.

The above scenario is an example of pervasive software systems that are increasingly deployed in a variety of domains, including emergency response. Such systems consist of a heterogeneous set of smart spaces (e.g., buildings equipped with sensors, autonomous vehicles) that through the commonly adopted SOA standards (i.e., Web Services enabling standards [14][18], such as SOAP, WSDL, and UDDI) can find each other's provided services and integrate to form a cohesive software system. Unlike traditional systems, they are expected to be utilized in a variety of scenarios (e.g., different emergency situations), many of which may not be known a priori. In other words, the system's actual functional and QoS requirements become known at run-time. These systems require significantly more flexible and dynamic software composition techniques than those that are currently at our disposal. In the next section, we describe a framework that aims to alleviate these challenges.

## 3. Overview of SASSY

Figure 1 illustrates the SASSY framework at a high level. A domain expert, as opposed to a software engineer, expresses the system's requirements in the form of an activity diagram, called *Service Activity Schemas (SAS)*. A SAS is a graph whose nodes correspond to activities that a user needs to accomplish. An *activity* describes a task that needs to be performed. Therefore, it is a technology independent concept that represents a transformation of inputs, potentially artifacts from another activity, into outputs. An activity may be *long-lived*, where the activity generates an indefinite number of outputs before termination, or *short-lived*, where the activity terminates after generating an output. For example, in an emergency response system that supports victims of a hurricane, activities may correspond to "contact the state national guard", "declare state of emergency", "contact the fire department", "contact the police department", "order food supplies", "order medication", "order tents", etc. The terms used to name activities come from a *domain ontology*. The domain ontology provides the means for unambiguously distinguishing different concepts and elements, which as outlined further below facilitate the discovery of services and resources in support of activities. The domain ontology is created and maintained by a consortium of domain experts, who specify the various domain activities and concepts, including the properties of respective services that realize them.

Given the requirements defined in the SAS, a base *System Service Architecture (SSA)* is automatically generated. A SSA is essentially a system's software architecture, with the exception that the component types are *service types*. A *service type* is the specification (e.g., name of operations, type of parameters) of a service. A *service type* is defined in the ontology. The service interface is defined in terms of the operations it provides, where each operation corresponds to one or more activities in the SAS. A service type is realized by a *service instance*, which is made publicly available for use by a *service provider*. Given the SAS, the corresponding service types are

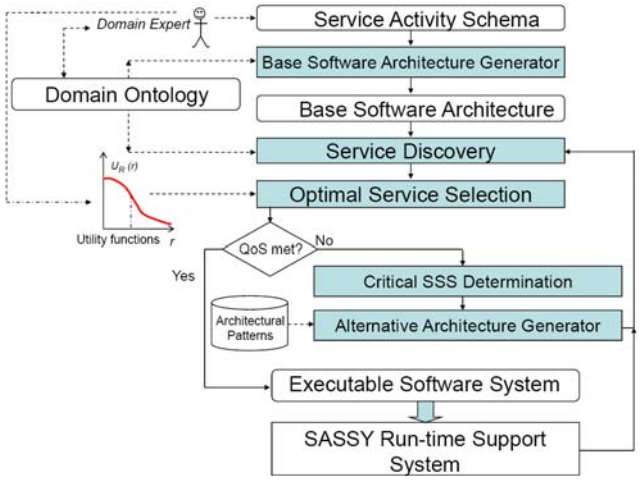


**Figure 1. High-level view of SASSY framework.**

found in the ontology, and a base SSA is composed. QoS goals are specified in an SAS through *Service Sequence Scenarios* (SSS), as detailed in Section 5.

The next step entails discovering a set of candidate service providers that can support the execution of the service types described in the SAS. There may be more than one service provider for each service type, each exhibiting a different QoS level at potentially different cost. The framework performs a service selection that maximizes a *utility function* provided by the domain-expert. A utility function reflects the usefulness of the software system based on the value of its quality attributes (e.g., response time, throughput, availability, security). If all QoS goals specified in the SSSs are met, the base architecture is effected by binding the associated services, generating the coordinator's logic, and executing the system on top of the SASSY run-time support system, which includes the following common services: monitoring of QoS metrics, consumer report, service rating, change management, goal management, repository services, service discovery, QoS brokering services [20], and others.

If however, the QoS goals are not met, SASSY determines the critical SSSs that violate the QoS goals. Then, it uses a library of *architectural patterns* [6] to generate alternative architectures that may ameliorate the QoS issues (e.g., replication for availability, encrypted channels for security). See Section 7 for specific examples. New service discovery has to be performed for each of the candidate architectures followed by optimal service allocation and QoS verification in an iterative process.

The SASSY approach is used for evolution and runtime adaptation of the software systems as well. We are going to extend our previous work [19] to support runtime adaptation. In the case of software evolution, the new system requirements (i.e., new SAS) are compared with the current requirements, and an architecture that satisfies the new requirements is generated (see Section 7). In the case of runtime adaptation, when the services fail to meet their QoS contracts, the QoS monitoring services trigger exploration of alternative architectures, and subsequently the revised architecture is reconfigured via the SASSY infrastructure support. As the backward arrow in Figure 1 indicates, monitors within the SASSY run-time support system may trigger adaptation, which may result in the generation of a new architecture and new service discovery.

In the remainder of this paper, we focus on two aspects of the SASSY framework: (1) modeling the system's requirements via a user-friendly, relatively informal, activity-oriented language, and then (2) automatically transforming the requirements to concrete architectural artifacts.

## 4. Architecture Generation Methodology

Our approach is grounded in the *Model Driven Architecture (MDA)* methodology [7]. MDA distinguishes between business and software models. To that end, MDA advocates the construction of a *Computation Independent Model (CIM)*, which represents an organization's business requirements. The CIM is then leveraged to arrive at a *Platform Independent Model (PIM)* of a software system that satisfies the business requirements. The PIM provides an appropriate-level of granularity (i.e., architectural-level) for assessing a software system's ability to satisfy its key functional and QoS requirements. The PIM is then translated into a *Platform-Specific Model (PSM)* that is more detailed and closely aligned with the implementation technology.

Researchers have shown that, in the general case, automatic derivation of software architectural models (PIM) from business models (CIM) to be infeasible [7]. To make the problem manageable, we have (1) constrained our research to SOA software systems, which allow us to make several simplifying assumptions, and (2) limited the expressiveness scope of the requirements language (SAS) to a predetermined set of concepts available in the ontology.

Figure 2 depicts our methodology. An MDA environment, such as the *Generic Modeling Environment* [5], is utilized to develop visual *Domain Specific Modeling Languages*. This process is depicted in Figure 2: a meta-model codifying the semantics of the SAS language is developed and passed to the *Meta-Model Interpreter*, which in turn customizes the MDA environment accordingly and provides visual support for constructing models in the SAS language. Support for the SSA language is provided similarly.

The domain user specifies the requirements of the system using the SAS language. The *Model Generator* leverages the constructed SAS model, the domain

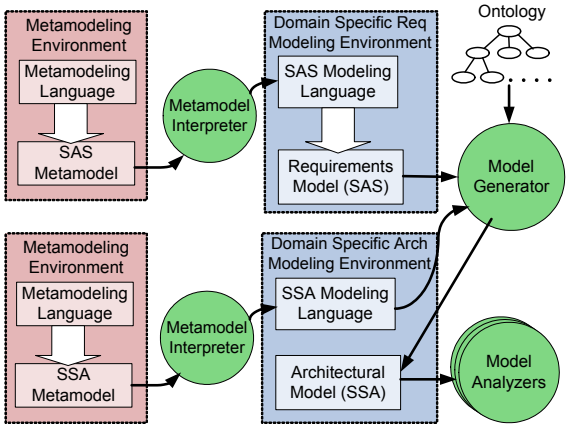


**Figure 2. Architecture generation and analysis methodology.**

ontology relating activities to service types, and the specification of the SSA modeling language to produce an architectural model of the system that satisfies the requirements. The architectural models are then utilized by the *Model Analyzer* engines, which may perform either further analysis (e.g., service provider selection) or software composition (e.g., applying architectural patterns). In the remainder of this paper, we take a closer look at the application of this methodology in the context of our research.

# 5. Service Activity Modeling

This section describes the rationale and elements of the activity-oriented modeling language we propose for defining SAS and SSS. The two most important requirements of this language are that: (a) it should be usable by domain experts with little training in software technologies, and (b) it should fit easily to the class of real-world problems illustrated in Section 2.

Specifically, it should easily support distribution and concurrency (joins, forks, etc.), specification of QoS objectives (see SSS below), awareness of geographic location, activities with different durations (ongoing versus short duration), and different styles of interaction (synchronous, asynchronous, streaming, etc.). Furthermore, it should manage the scalability of specifications via hierarchical decomposition of activities (i.e., sub-activities).

A comprehensive survey of existing languages led us to the Business Process Modeling Notation (BPMN) [1], which has recently been designated as an OMG standard. Unlike many others, e.g., UML activity diagrams [9], BPMN is intended to be used by domain experts, as opposed to software engineers. Additionally, previous research has shown that appropriately constrained BPMN models can be mapped to the executable Business Process Execution Language (BPEL) [15][17], which increases our confidence that a language based on BPMN can be automatically translated to executable software.

Nevertheless, the current specification of BPMN has three significant shortcomings with respect to the goals of our research. First, in its general form, it is overly flexible and open-ended to support fully automatic generation of software architecture. Second, it does not support modeling of QoS objectives. Finally, it does not support awareness of geographic location for scalable and context-aware discovery of services. While the first limitation prompted us to define a subset of BPMN with precise semantics, the other two prompted us to extend the language in those directions.

## 5.1. Service Activity Schemas

The SAS modeling language is specified by the meta-model depicted in Figure 3. As mentioned in Section 4, the meta-model is used to provide support for the language in the *Generic Modeling Environment* [5]. For brevity, we do not describe the details of the meta-model; instead, we provide a description of the language in this section. However, we would like to point out the importance of the meta-models in our approach. Meta-models provide precise semantic underpinnings for the languages used in our research, addressing a shortcoming of many existing languages (e.g., BPMN). This in turn enables systematic transformation of our models.

From BPMN, SAS retains the following elements: (1) *Events*, depicted as circles, for sending and receiving messages between possibly distributed logical or physical entities; (2) *Gateways*, depicted as diamonds, for controlling the flow of execution within an SAS; and (3) a generic notion of activity, or task, represented as a round-corner rectangle. Like BPMN, activities may be annotated as looping, based on a condition, and multiple, contingent on an expression to determine the number of instances.

Unlike the BPMN, SAS explicitly distinguishes local activities from those that are performed externally. An external entity, also referred to as *service usage*, is represented as a round-corner rectangle annotated with a server icon. An entity could be either logical (i.e., a conceptual element in the domain) or physical (e.g., a sensor). Each activity carried out by an external entity is represented as a port (a small envelop in the round-corner rectangle).

Also unlike BPMN, SAS does not use the notation of swim lanes to represent the concerns of separate activities (see Figure 4a). Each SAS is depicted from the local point of view of the component responsible for the activity (i.e., coordinator), and the interactions with external entities are shown as exchanges of

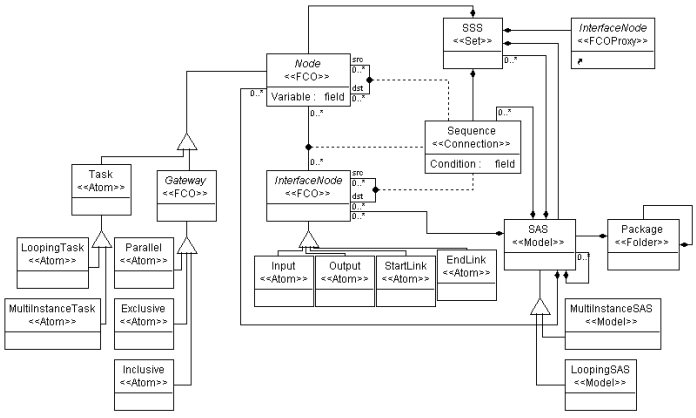


**Figure 3. Subset of the SAS meta-model in GME.**

events. In other words, one could map the SAS notation to BPMN's swim lanes by dragging each service usage into a separate swim lane, decomposing it into its underlying activities, and keeping local activities and gateways in a central lane, which contains the logic for coordinating the invocation of external activities. We have pushed the intermediate message events to the boundary of the external entities residing in their pools; in this way we have removed the need for extra communication type (i.e. *Message Flow*) and made the language even simpler.

The reason we chose to drop the swim lane notation is the key role played by service discovery in the class of systems targeted by our research. Swim lanes suggest a stable, predefined mapping of functions to components. In contrast, in SAS, which component actually provides a service is contingent on service discovery, and immaterial to the specification of the service usage within the activity.

To better illustrate these ideas, Figure 4a shows a simple example of a scenario that monitors fire emergencies in a smart building. The scenario uses the services of external smoke detectors, sprinklers, building occupancy estimation, and a fire station.

Similarly to BPMN, the arrival and emission of such messages is represented as events: a dark envelope (not shown in the figure) denotes the event of sending a message, and a clear envelope, the event of receiving one. In the example, the monitoring activity is started upon receipt of a *start* message, after which the relevant *SmokeDetector* and *Timer* are started. *Timer* is a local sub-activity which generates periodic ticks requesting *OccupancyAwareness* to provide an estimation on the number of the occupants in the monitored building.

If a *SmokeDetector* ever senses smoke, it sends a message describing the level of smoke and Carbon Monoxide (CO), as well as the specific location. The gateway marked with a circle, an *inclusive* (Conditional-Or) gateway in BPMN, proceeds to activate *SprinklerControl*, if the concentration of CO is above 600ppm. In any case, it further proceeds to activate the gateway marked with a cross, a *Parallel* (And-Join) gateway in BPMN, which makes sure an occupancy estimate has been received before sending a *fire call* message to the *FireStation*.

## 5.2. Service Sequence Scenarios

The second shortcoming of BPMN, dealing with the specification of QoS objectives, is addressed by *Service Sequence Scenarios* (SSS). These are well-formed sub-graphs of SAS, and correspond to a sequence of interest to the user, to which QoS objectives are associated. An SSS is well-formed if it satisfies the syntactic constraints of an SAS: a condition easily checked by the modeling tool.

Currently, SSS are accessible via dashed rectangles at the upper left corner of an SAS (see Figure 4a). Figure 4b 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 QoS objectives associated with this scenario, e.g., an availability of 99%, are captured in a property sheet associated with the SSS (not shown for the sake of space). Furthermore, to facilitate the job of domain experts, the tool relies on ontology to support specifying domain-specific objectives either at a high-level of abstraction, e.g. *high-resolution* images, or at a technical level, e.g., 1024x1024 pixel.

The QoS objectives associated with an SSS are

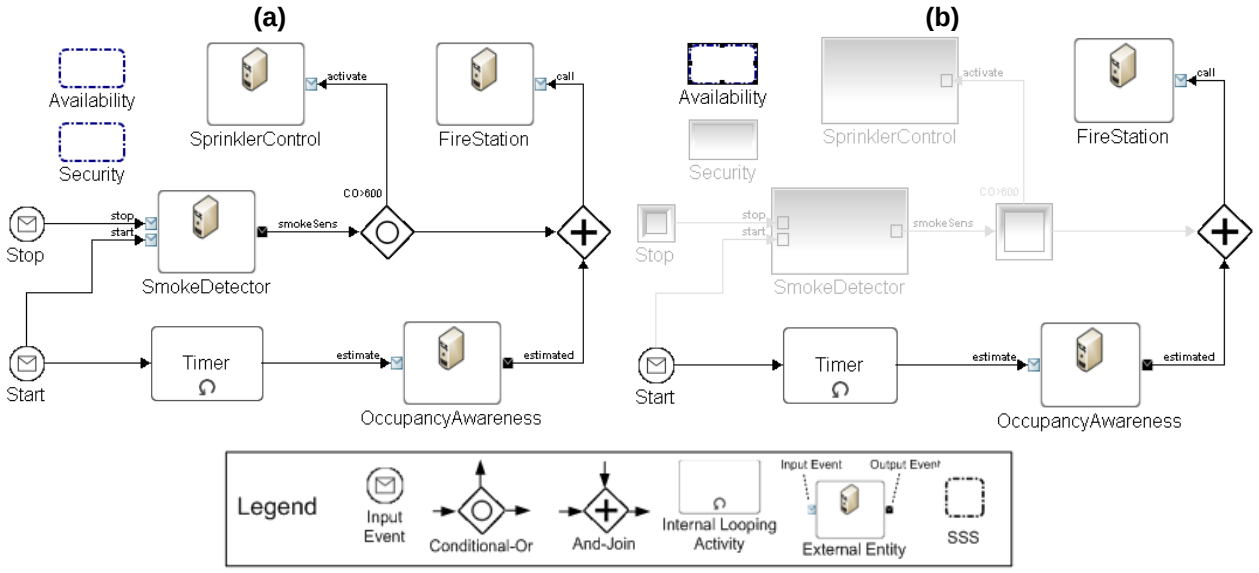


**Figure 4. (a) A simple SAS for a Fire Emergency Response Scenario, (b) A selected availability SSS.**

used in the generation of the architecture and analysis, as discussed in the next section. These annotations are expressed in a language that is inspired by WSLA [16].

### 5.3. Scaling Discovery

The third limitation of BPMN, dealing with location-aware service discovery, is addressed by associating location constraints to service usages. Referring to the example in Figure 4, after a *smoke sensed* message is received, the monitoring activity should discover the fire sprinklers that are close to the affected area in the building, using the location information received from the smoke detector, and send the *activate* message only to those. Geographic constraints on service discovery can be used to pin-point services across wide geographic distances. For example, given a specific emergency, a federal agency located in Washington DC might need to discover and inquire the state of all the emergency response vehicles currently at a particular district in New Orleans.

We are currently experimenting with a hierarchical representation of location, with support for aliasing, similar to the notation used for URLs in the internet. For example, *//fairfax.va.us/22030/4400-university-drv* represents the same location as *//gmu.edu*.

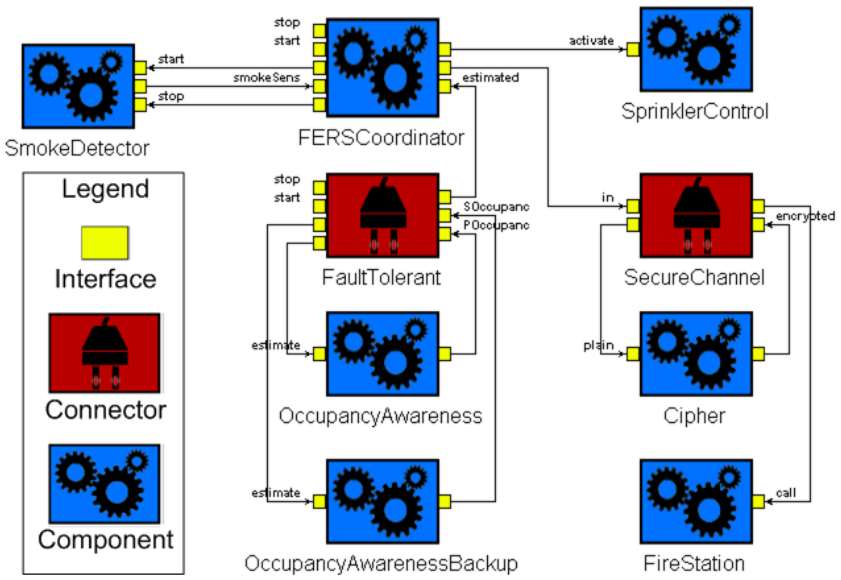


**Figure 5. Structural view of SSA for the example of Figure 4.**

## 6. System Service Architecture

*Architectural Description Languages (ADLs)* [10] have been shown to be effective in modeling the crucial characteristics of a software system that determine its ultimate capabilities, properties, and qualities. However, a traditional shortcoming of ADLs is that each is specialized to a particular type of concern (e.g., structural vs. behavioral, dynamic vs. static, computation vs. interaction). In SASSY, we needed a comprehensive architectural modeling approach to model the different aspects of the *System Service Architecture (SSA)*. To this end, we have extended our previous work [4] that allows for the construction of a software system's architectural concerns from different points of view using multiple ADLs. In this approach, each ADL is supported via a meta-model (see Section 4). More importantly, the ADLs are linked together at the meta-model level, allowing for the development of composite ADLs. The approach provides innate consistency among the multiple views of the same system, even if they are developed using different ADLs.

Our SSA models are based on two well-known ADLs that have been adapted for the purposes of modeling SOA systems. We have utilized a widely used and extensible ADL, *eXtensible Architectural Description Language (xADL)* [3] to represent the structural properties of a system's software architecture. xADL provides the traditional component-and-connector view of a software systems architecture [3]. In SASSY, we model services as components. Figure 5 shows an example of a xADL model that is further discussed in the next section.

To represent the behavioral aspects of an SOA system (i.e., the coordinator's logic) we have used *Finite State Processes (FSP)* [8]. FSP is a type of state machine language intended to capture a software system's high-level behavioral and interaction properties. Figure 6 shows an example of a FSP model in SSA, which is also discussed in more detail in the next section. A more detailed description of support for xADL and FSP, including the corresponding meta-models, which form the basis of SSA, is provided in [4]. Next, we describe our approach in the generation of SSA models from SAS models.

## 7. Architecture Generation

As described in Section 3 and depicted in Figure 1, the process of generating the architecture consists of (1) generating a base architecture, (2) solving the QoS objectives via service provider selection, and finally (3) if QoS objectives are not satisfied, find an alternative architectural solution. The first step is easily achieved using an appropriate domain ontology referring activities to service types. The second step is

a complex problem which is the subject of our ongoing research, and early results are provided in [11]. In this section, we are going to focus on the third step.

When existing service providers capable of satisfying the QoS objectives of the base architecture cannot be found, SASSY applies a variety of *software architectural patterns* [6] to find an architectural solution to the problem. To demonstrate this capability, we revisit the example of Figure 4. In this example the user has specified an availability requirement of 99% for the two services involved in the availability SSS. Moreover, the user has specified a security requirement of "High" for any communication with the *FireStation* service (i.e., the user has requested encrypted communication with the *FireStation*). Note that the security SSS is not selected in Figure 4b; therefore its details are not highlighted.

Let us assume that an *OccupancyAwareness* provider that is 99% available and a *FireStation* provider that support encrypted communication are not found. However, the SASSY infrastructure has found two potential *OccupancyAwareness* providers that are 90% available. Similarly, a generic *Cipher* component with the ability to encrypt and decrypt messages is found. We describe the process of generating an alternative architectural model that satisfies the QoS requirements of this example next.

## 7.1. Generating the Structural View

Figure 5 shows the structural view of the SSA that is generated. The *FERSCoordinator* component organizes the interaction among the services. Its logic is generated from the SAS example of Figure 4 and detailed in the next section. *SmokeDetector* and *SprinklerControl* components correspond to the two service providers that satisfy the QoS requirements.

Since an *OccupancyAwareness* provider capable of satisfying the availability requirement is not found, SASSY applies the *replication* pattern through the use of a specialized *FaultTolerant* connector [12]. A *FaultTolerant* connector broadcasts the service requests to several service providers, but only the response from the *primary* provider is sent back to the client. If the primary provider becomes unavailable, the *FaultTolerant* connector detects it, and promotes one of the *secondary* providers to the role of the primary. As shown in Figure 5, by utilizing the replication pattern, SASSY satisfies the user's stringent QoS requirement of 99% availability, via the two *OccupancyAwareness* services, each of which is available 90% of time.

Similarly, SASSY satisfies the security requirement associated with the *FireStation* by applying the *mediator* pattern. The *FERSCoordinator* component relies on a *SecureChannel* connector that encrypts messages using the *Cipher* service. The resulting architecture is depicted in Figure 5. The above patterns can be more generally utilized for other purposes. For example, the mediator pattern can also be used in conjunction with a compression service to satisfy the throughput and latency requirement.

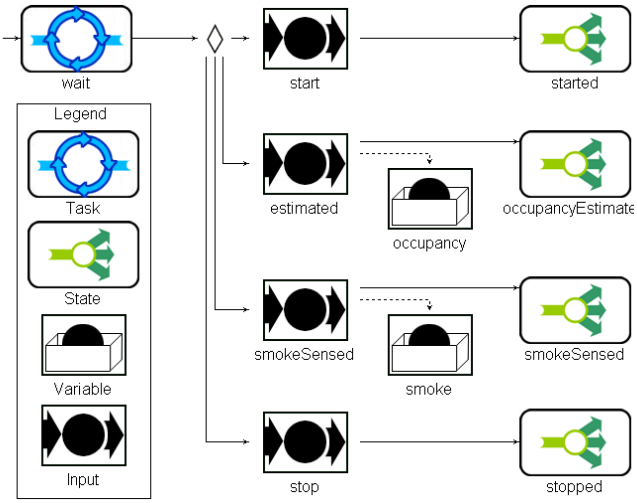


**Figure 6. Coordinator's behavior in handling the inputs for the example of Figure 4.**

The SSA in Figure 5 can be extended to support a highly distributed software architecture consisting of multiple buildings, each of which containing several instances of *SmokeDetector* and *SprinklerControl,* one instance of the *FERSCoordinator* and *FaultTolerant* connector, and two instances of *OccupancyAwareness*. In addition, the architecture may consist of several instances of *FireStation*, where each *FERSCoordinator* would discover the *FireStation* instance geographically closest to it using the approach described in Section 3. Finally, the coordinator itself could be both logically and physically distributed. For the sake of simplicity, our example does not show such a scenario. A more complex system would involve multiple SAS models, each of which may result in one or more, potentially distributed, set of coordinator components.

## 7.2. Generating the Behavioral View

In the previous section, we described the structural view of the generated SSA. Figure 6 shows a subset of the coordinator's logic in FSP that is generated from the SAS example of Figure 4. As detailed in [4] we have slightly modified the traditional FSP for the purposes of simulation (not discussed in this paper).

FSP allows for the hierarchical composition of state machines. The portion of the coordinator behavior depicted shows the highest level, dealing with the receipt of messages. The initial entry point to the coordinator is a *wait task,* which represents the initial

storage of the messages before they are handled by a processing thread.

Afterwards, one of the potential paths is selected based on the type of the input received. The inputs in the FSP model are generated based on the events in the SAS model of Figure 4. A coordinator needs to be able to handle the *Event* messages marked with a white envelope, which denote messages received by the coordinator in SAS. Some of the inputs (*estimated* and *smoke sensed*) carry data that are stored in local variables (*occupancy* and *smoke*). This data corresponds to the *Event* attributes (e.g., the *Carbon Monoxide* value of the *smoke sensed* event) specified in the SAS model and used by the SAS *Gateway*s for decision making. Based on the selected path, the coordinator is put into the corresponding state to handle the received input. The details of the other states not shown in this figure are generated similarly.

Finally, the generated architectures may have to evolve as a result of changes in the requirements. For instance, an additional entity, such as an ambulance, may need to be added to the example of Figure 4 . The user specifies the new requirements by revising the existing SAS models. Afterwards, the SASSY framework adapts the existing software architecture by modifying the structural view and generating the behavioral view for the new coordinator. This could result in changes to the set of service providers and to previous architectural decisions.

## 8. Conclusion

The SOA paradigm is increasingly employed in the construction of a growing class of pervasive real-world software systems. This paper presented a novel approach intended to streamline the composition of SOA software system in such settings. Our approach deviates from related approaches (e.g. Jopera [21]) through its explicit reliance on the system's software architecture. Given the functional and Quality of Service (QoS) requirements expressed in an activity-oriented language, an "optimal" architecture is generated. The architecture indicates specifically the service providers as well as the interaction patterns that should be utilized to satisfy the requirements.

This work is part of an ongoing research effort on Self-Architecting Software Systems (SASSY), a framework intended to automate the composition, analysis, adaptation, and evolution of real-world SOA software systems. Avenues of future work include, fully automatic analysis and generation of architecture, building a QoS related pattern repository, empirical evaluation of the approach, development of runtime architecture adaptation capability, and the generation of BPEL code for distributed coordination of activities.

## 9. Acknowledgements

This work is partially supported by grant CCF-0820060 from the National Science Foundation.

## 10. References

[1] BPMN Spec. ver 1.1. *Object Management Group*, 2008.
[2] P. Clements, et al. Documenting Software Architectures: Views and Beyond. *Addison Wesley*, 2003.
[3] E. Dashofy, et al. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *Int'l Conf. on Software Engineering*, Orlando, FL, May 2002.
[4] G. Edwards, S. Malek, and N. Medvidovic. Scenario-Driven Dynamic Analysis of Distributed Architecture. *Int'l Conf. on Fundamental Approaches to Software Engineering*, Braga, Portugal, March 2007.
[5] GME. http://www.isis.vanderbilt.edu/Projects/gme/
[6] H. Gomaa, et al. Composition of Software Architectures from Reusable Architecture Patterns. *Int'l Workshop on Software Architecture*, Orlando, Florida, 1998.
[7] A. G. Klepp, et al. MDA Explained: Practice and Promise, *Addison-Wesley*, Boston, MA, 2003.
[8] J. Magee, et al. Behaviour Analysis of Software Architectures. Int'l *Working Conf. on Software Architecture*, Deventer, Netherlands, Feb, 1999.
[9] S. Martin, et al. UML Distilled: Applying the Standard Object Modeling Language. *Addison-Wesley*, 1997.
[10] N. Medvidovic, et al. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Soft. Eng..*, vol. 26, 2000.
[11] D. A. Menascé, et al. A Heuristic Approach to Optimal Service Selection in Service Oriented Architectures, *Workshop on Software and Performance*, Princeton, NJ, June 2008.
[12] M. Rakic, et al. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *Symp. on Software Reusability*, Toronto, Canada, May 2001.
[13] M. P. Papazoglou, et al. Service-oriented Design and Development Methodology. *Int'l Journal on Web Engineering and Technology*, vol. 2, pp. 412-442, 2006.
[14] M. P. Papazoglou. Web Services: Principles and Technology. *Prentice Hall*, 2008.
[15] W.M.P. van der Aalst and K.B. Lassen. Translating unstructured workflow processes to readable BPEL: Theory and implementation. *Information and Software Technology*, vol. 50, pp. 131-159, 2008.
[16] WSLA. www.research.ibm.com/wsla/
[17] WS-BPEL ver 2.0, *OASIS*, 2006.
[18] S. Weerawarana et al. Web Services Platform Architecture. *Prentice Hall*, 2005.
[19] H. Gomaa, et al. Model-Based Software Design and Adaptation. Workshop on *Soft. Eng. for Adaptive and Self-Managing Systems*, Minneapolis, MN, May 2007.
[20] D. A. Menascé, et al. QoS Management in Service-Oriented Architectures, *Journal of Performance Evaluation*, vol. 64, Issues 7+8, Pp. 646-663, Aug 2007.
[21] C. Pautasso, T. Heinis, and G. Alonso. JOpera: Autonomic Service Orchestration. *IEEE Data Engineering Bulletin*, vol 29, no 3, Sep. 2006.