Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. Hill, L. Moench, and O. Rose, eds.

A SIMULATION FRAMEWORK FOR SERVICE-ORIENTED COMPUTING SYSTEMS

Hessam Sarjoughian
Sungung Kim
Muthukumar Ramaswamy
Stephen Yau

Arizona Center for Integrative Modeling and Simulation
School of Computing and Informatics
Arizona State University, Tempe, Arizona, 85271-8809, USA

ABSTRACT

An SOA-compliant DEVS (SOAD) simulation framework is proposed for modeling service-oriented computing systems. A set of novel abstract component models that conform to the SOA principles and are grounded in the DEVS formalism is developed. The approach supports construction of hierarchical composition of service models with feedback relationships. A SOAD Simulator (SOADS) is designed and implemented. An exemplar model of a basic service-oriented computing system is described. A representative experiment capturing throughput and timeliness QoS attributes for the exemplar model is devised, simulated, and described. The paper concludes with the concept of community-based development of the SOAD framework and tools.

1 INTRODUCTION

Many of today's computer-based systems are challenging to build since they are distributed and operating in changing environments. A central requirement for a system is to be flexible in that its parts are loosely coupled while the system as a whole can satisfy quality attributes known as run-time (e.g., performance and availability) and non-run-time (e.g., reusability and integrability) observable. To build such systems, service-oriented computing paradigm based on Service Oriented Architecture (SOA) framework has been proposed (Erl 2006; Chen and Tsai 2008).

To achieve the goals set forth for service-oriented computing, a growing number of researchers are formulating detailed concepts, methods, and techniques that can be used to build service-based systems. The most common approach in defining a system's structure and behavior is to develop models. The choice of a model is driven by the role it can play in the system development and operation lifecycle. For example, a model can be at the architectural level or be complete and sufficiently detailed to be automatically implemented. Models can be developed to define technical requirements and architectural design of a service-based system. Such models may, for example, represent dynamics of the services and their interactions in such a way to study the system's capability to support the quality of service attributes such as performance, timeliness, accuracy, and security.

To design service-based software systems capable of satisfying multiple Quality of Service (QoS) attributes, simulation-based modeling is desirable. For instance, in the context of our research, simulation plays a central role in enabling tradeoff study among time-based quality of service attributes. The basic need is to have an Adaptive SBS (ASBS) where its QoS can be observed by a Monitoring system and controlled by an Adaptation system. The users can select services and list their expected QoS under the presence of some uncontrollable, but predictable environmental fluctuations.

To develop the ASBS framework – design, implement, and test the Monitoring and Adaptation systems – we can develop a set of real composite and simulated services (Yau et al. 2008). Together, real and simulated services enable analysis and design capabilities that are impractical to support by either real or simulated services alone. There are important differences between real and simulated services. First, the dynamics of real services are considered partially controllable as compared to their simulated counterparts. Second, the details of simulated services can be at a high-level of abstraction relative to real services (e.g., details of how a publisher generates its services can hide some details). Abstraction and flexibility afforded by simulation allows early architecture and design of service-based systems traits such as scalability, adaptability, and performance. Third, the control of the simulation execution is at the discretion of the user, something that is generally impractical to achieve in real systems. Forth, simulated services may be used with real services, a capability that has proven very useful in development of component-based complex, distributed systems. With simulated services, the Monitoring and Adaptation systems can themselves be real services and thus support carrying out design and experimentation under normal and unusual settings. Here the flexibility of simulated services allows devising experiments for studying alternative designs for the Monitoring, Adaptation, and the ASBS as a whole. The result would be a mixed simulated/real testbed for conceptualization, design, evaluation, and validation of adaptive service-based systems.

In order to develop and use simulated services, it is important to have a modeling and simulation framework that is theoretically sound, has one or more robust implementations, and simple to use. One such framework is the Discrete Event System Specification (DEVS) formalism (Zeigler et al. 2000) with implementation such as DEVSJAVA (ACIMS 2001). This modeling formalism is positioned to specify and simulate *SOA-compliant* simulation models.

In the rest of the paper, we will discuss the need for developing a simulator for service-based software systems (Section 2). Then, we review the SOA framework and the DEVS formalism (Section 3). Next we detail the basic concepts of SOA and DEVS and formulate the basic concept of SOAD framework (Section 4). The basic elements of the proposed simulation framework are then presented (Section 5). Next a realization of the SOADS in DEVSAJAVA with an example is described (Section 6). Finally, related works are reviewed (Section 7) and the paper's summary and future work is presented (Section 8).

2 MOTIVATION

There exist basic differences between SOA principles and the underlying concepts of all existing simulation approaches. Therefore, we should expect difficulty in using general-purpose simulation tools for service-oriented software systems. These difficulties are due to theoretical limitations of modeling and simulation frameworks – e.g., lack of support for basis SOA concepts such as services autonomy and loose coupling. The differences between the fundamental concepts supported in a modeling and simulation framework and the basic characteristics of a class of systems to be simulated are not uncommon. For example, a general-purpose simulation framework such as DEVS can be used to model computer networks. This requires extending the SOA software concept with that of the hardware concept. The resulting model abstractions can represent both software components and network nodes of service-oriented software systems. A domain-specific simulation framework such as ns-2 simulator (ns-2 2002) recognizes the importance of modeling computer network systems and their communication protocols. The simulation models represent the basic traits of computer network systems in a set of core and reusable model abstractions (i.e., components and relationships among them). Therefore, modelers can start using a set of network model abstractions that can be synthesized to create and simulate computer network systems having varying degrees of complexity.

Based on the above observation and the anticipated growth in simulating service-oriented software systems, we propose developing an SOA-compliant simulation framework. A suitable modeling framework is Discrete Event System Specification (DEVS). A set of generic model abstractions for services and their relationships are needed. The simulation of the models should capture the inherent properties of SOA-compliant software systems. Specifically, the simulation framework must have a set of SOA elements (i.e., publisher, subscriber, broker services, and messages) and relationships (e.g., subscriber can find out published services only via a service broker) that comply with the SOA principles. The resulting SOA-compliant simulation framework can support creating different user-specific simulation models that are built on the top of verifiably correct SOA model components.

Before we proceed further, we note that simulation of SOA-based software system should account for both software and hardware aspects. The software aspect refers to the core SOA principles. The concept of SOA-compliant DEVS Simulator is based on partitioning the core SOA principles into two parts called simple and complex. The term simple is used to refer to service-based systems that cannot have services added/removed at run-time; there is limited support for loose coupling. The term complex is used to refer to service-based systems that can have their structures changed at run-time. Therefore, separation of the SOA principles into simple and complex parts is to help in building the proposed SOAD framework in two stages. The simple part focuses on the autonomy, abstraction, service contract, reusability, composability and statelessness principles. The complex part focuses on the loose coupling and discoverability principles. The hardware aspect refers to physical and non-service components that are responsible for the execution of services and their interactions. Modeling of hardware – i.e., a collection of computing nodes (processors and routers/switches) and network links – is essential for capturing the dynamics of the services. This is because hardware components responsible for the execution of the services and communication of messages directly impacts QoS attributes. These considerations lead us to the development of the SOAD framework and realization:

1. SOAD models represent the static software aspect of the SOA capabilities.
2. A simple model of the hardware which delays messages and limit data transmission is used with the SOAD models.
3. The SOAD models defined in Item 1 is extended to represent the dynamic aspect of the SOA capabilities.
4. A detailed model of the hardware representing, computing nodes responsible for executing the services with the links and routing devices for message transmission is used.

In this paper, a basic SOAD Simulator is developed to support Items 1 and 2. The advanced concepts and capabilities contained in Items 3 and 4 can be introduced into SOADS to simulate service-based software systems having dynamic structures and complex behaviors arising from mixed software and hardware interactions.

3 BACKGROUND

The DEVS and SOA share important concepts even though their uses are intrinsically different – one is intended to build service-oriented software systems and the other to simulate component-based systems. Their commonality lies in their view of (i) software or simulated systems to be either flat or hierarchical, (ii) there can be feed forward and feedback interactions, and (iii) sequential and parallel execution. However, these system-level concepts have different abstractions. The SOA framework's abstractions are relatively at a higher level compared with those of the DEVS framework. The basic concepts, principles, and artifacts of these frameworks are described next. Some of their main similarities and differences are also exposed.

3.1 SOA Framework

The desire for enterprise systems that have flexible architectures, detailed designs, implementation agnostic, and operate efficiently continues to grow. A major effort toward satisfying this need is to use Service Oriented Architecture. Moreover, there is new research and development in order to achieve more demanding capabilities (e.g., workflow service composition with run-time adaptation to changing QoS attributes) that have been proposed for service-based systems, especially in the context of system of systems.

A variety of concepts and definitions have been proposed to satisfy different needs. A basic concept is for SOA to enable specifying the creation of services that can be automatically composed to deliver desired system dynamics while satisfying multiple QoS attributes. The principal artifacts of SOA are publisher, subscriber, and broker services (Erl 2006). The communication protocols for these general-purpose services are supported with WSDL, UDDI, and SOAP (Møller and Schwartzbach 2006). The publisher and subscriber services are also sometimes referred to as provider and requester, respectively. A publisher registers its service descriptions (WSDL) with the broker service and a subscriber can find services it is searching for if they are registered with a broker. The broker uses its service registry using UDDI to identify matched service descriptions. Then, a subscriber can invoke a publisher and obtain the requested service. The message interactions among the services are supported by the SOAP mechanism.

A fundamental SOA concept is to enable flexible composition of independent services in a simple way. The simple concept is crucial since it separates details of how a service is created and how it may be used. This kind of modularity is defined based on the concept of broker and its realization as the broker service. The SOA conceptual framework lends itself to the separation of concerns ranging from application domains (e.g., business logic) IT infrastructure to the choices of programming languages and operating systems. The interoperability at the level of services means loose coupling of reusable services.

Service can have different levels of resolution – it may be a software component, a business process, a deployed application, or a whole system. The software components form the basic services from which composite services may be built. The ability to model composition of services as software components, therefore, is a basic necessity for developing service-based systems.

The high-level description of the SOA principals does not account for the operational dynamics of SOA, especially with respect to time-based operations. Therefore, understanding the dynamics of service-based system using simulation is important. Simulation can also support specific kinds of service-based software systems that are targeted for business processes with specialized domain knowledge. For example, given the steps in creating a service (e.g., defining service capabilities, selecting services, specifying service flows, and deploying services) they may be supported with component-based, scalable, and efficient simulation. A simulation framework capable of modeling SOA-compliant software systems offers a basis that can be extended to conceptualize and evaluate interesting aspects of higher-levels of services (e.g., automated service composition) for different application domains.

3.2 DEVS Framework

Simulation is considered useful and increasingly indispensible across all phases of system development lifecycle (i.e., conceptualization, design, implementation, deployment, and operation). This observation applies to service-based systems since component-based simulation and service-based systems are based on fundamental concept of components and their interactions. A component-based modeling framework such as Discrete Event Systems Specification (DEVS) (Zeigler et al. 2000) is well positioned to create model abstractions for service-based systems. The SOA principles including autonomy, composability, and reusability of services with message-based interactions fit well the DEVS modeling formalism. This is because the dynamics of a typical SBS system can be characterized in terms of time-based modular and hierarchical reactive simulation model components. These simulation model components can process input events (messages) and generate output events (messages). The DEVS formalism provides abstract formulation for describing concurrent processing and the event-driven nature of arbitrary system configurations and executions. Parallel atomic/coupled DEVS models can be executed in distributed settings (including grid services), and therefore is a suitable modeling framework to characterize complex, large-scale service-based systems.

An atomic model (formalized as ⟨X, S, Y, δ_{ext}, δ_{int}, δ_{conf}, λ, ta⟩) characterizes the structure and behavior of individual components in terms of inputs (X), outputs (Y), states (S), and functions. The external (δ_{ext}), internal (δ_{int}), confluent (δ_{conf}), output (λ), and time advance functions (ta) define a component's behavior over time. For example, a subscriber can be conceptualized, formalized, and implemented as an atomic model. Internal and external transition functions describe autonomous behavior and response to external stimuli, respectively. The time advance function represents the passage of time. Time is specified as real number. The output function is used to generate outputs. Atomic models are basic components and thus can be combined to represent aggregate behavior of coupled models which have tree structures hierarchy). Parallel DEVS, which extends the classical DEVS, is capable of processing multiple input events and concurrent occurrences of internal and external transition functions. Its confluent transition function provides local control by handling simultaneous internal and external transition functions. .

A coupled model (formalized as ⟨X, Y, D, {M_d}, EIC, IC, EOC⟩) is defined in terms of its constituent atomic and/or coupled models. A coupled model can be constructed by composing models into hierarchical tree structures, and is defined in terms of its constituent (atomic and/or coupled) models. The input and output sets X and Y have the same specification as those of the atomic model. D is a set of component names and M_d is a set of atomic and/or coupled components, and EIC, EOC, and IC are external input, external output, and internal couplings, respectively. The closure under coupling property allows a coupled model to be treated as a basic or atomic model. When a component sends messages, the (external input, external output, and internal) couplings between input and output ports immediately relay the messages from the sender to receiver components. Upon receipt of messages by atomic models, the messages are processed, which may result in new states and generation of new outputs for other models.

The behavioral semantics of the DEVS models are defined in atomic and coupled abstract simulation protocols. The execution ordering of the atomic model functions is determined by the atomic simulator. Similarly, the transmission of the messages among the atomic and coupled models is determined by the coupled simulator. One of the object-oriented realizations of the DEVS formalism and its associated simulation protocol is DEVSJAVA (Zeigler and Sarjoughian 1997). It is a simulation tool that is in use in academic and industrial settings for nearly a decade. The formal foundation of DEVS, its object-orientation extension and design, efficient execution, and the availability of sequential, parallel, or distributed simulation engines using alternative computational environments such as HLA and Web-services are useful for large-scale simulations.

3.2.1 Dynamic Structure and SW/HW Models

The basic atomic and coupled models are not sufficient for modeling the kinds of SOA complexities that need to be simulated. For example, to model addition or removal of services at run-time, it is important for a DEVS simulation model to change its structure dynamically which can be by adding or removing atomic and coupled models. This concept is known as variable or dynamic structure DEVS (Zeigler et al. 2000). One realization of this concept is called Dynamic Structure DEVS, which at its core has an Executive model component with rules for adding and deleting model components during simulation (Barros 1997). Another important contributor to the complexity of SOA is the dependency on hardware. While atomic and coupled models can represent software aspect of a service, it is also important to model hardware aspect of the resources (e.g., processors, switches, and network links) on which the services execute and interact. To model both software and hardware aspects and the mapping of the former to the latter, the DEVS/DOC, a software/hardware co-design approach has been developed (Hild et al. 2002). In this environment, disparate software components executing on distributed hardware components can be modeled and simulated. This approach supports quantum level abstraction of software and hardware components.

4 SOAD FRAMEWORK CONCEPT

Earlier, we described the details of the SOA and DEVS frameworks. An important consideration in choosing a modeling and simulation framework is its direct support for message-based communication among independent model components. This is important since the concept of SOA is grounded in autonomous services that can only influence each other via messages. The combination of publisher and subscriber interaction via messages matches well the strict modularity of the DEVS framework. Furthermore, as noted above, the capability for parallel simulation of services with arbitrary combined feed forward and feedback message flows has been a key consideration in the selection of the DEVS framework in this work.

The SOAD framework concept is shown in Figure 1. The basic idea for this simulator is to enable modeling and simulating primitive and composite services as if they were real service. (The concept of simulated services is distinct from that of simulated objects – the DEVS and object-orientation concepts, compared to DEVS and SOA, are closely related.) The concept of simulated services is distinct from that of simulated objects. The DEVS and object-orientation concepts, compared to DEVS and SOA, are closely related. The SOA is defined in terms of principles that are intended to guide architecture, design, implementation, testing, and operation of service based systems. These principles may be used to develop details of SOA which can result in different realizations both for the SOA itself as well as user applications. The DEVS formalism, on the other hand, is a mathematical specification intended for developing time-based models that can be simulated. We also note that while atomic and coupled models require abstract atomic and coupled simulators in order to be executed, the services (publisher, subscriber, and broker) contain their own execution logics (see Figure 1 where abstract simulators can be executed in either centralized or distributed computing settings).

Given the disparities between DEVS and SOA frameworks, our aim is to develop a framework for SOAD. Two basic approaches can be taken. One is to infuse the concept and capabilities of DEVS concept and capabilities into SOA framework. The other is to extend the DEVS framework such that it can account for the SOA concept and capabilities. In this work, we choose the latter approach.

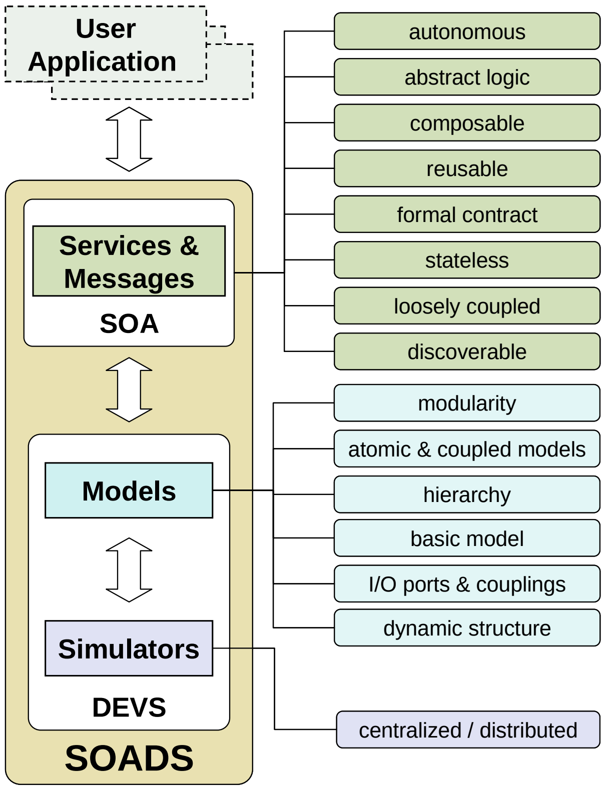

Figure 1: SOAD simulation framework concept

Before we consider the SOA and DEVS frameworks together, it is important to recall that one is intended to build real services and the other to build simulated services. In this section, DEVS framework refers to DEVS with Dynamic Structure capability. Also, it is useful to appreciate that while a primitive (subscriber or publisher) service and atomic model can be considered as components (or objects), their underlying concepts are inherently distinct. Furthermore, the concept of a composite service (either as publisher or subscriber) differs from that of a coupled model. The following examination of the SOA and DEVS reveals similarities and differences between the SOA and DEVS frameworks (see Table 1).

- The concept of autonomous services corresponds to the concept of modularity of atomic and coupled models. DEVS models are defined in terms of generic transition (δ_{ext}, δ_{int}, δ_{conf}), output (λ) and time advance (ta) functions.
- The formal contract corresponds to the input/output ports and messages (X and Y), and their couplings (EIC, EOC, IC) subject to the strict coupled model specification. The couplings in DEVS are fixed, although the use of coupling in a simulation can be decided during simulation. The concept of coupling components via ports is absent in SOA.
- The concept of service composability is similar to coupled model hierarchy. SOA composability is not constrained to have strict hierarchy. This is because DEVS hierarchy requires strict tree structure relationships among (atomic and coupled) model components. In SOA, composability is based on the broker service which is not defined in DEVS. In DEVS, input and output messages are sent and received via direct couplings – i.e., the coupled model contains the coupling relations between model components.
- The concept of abstract logic in DEVS has a theoretical basis (abstract structural and behavior syntax with operational semantics) whereas SOA does not. For example, δ_{ext} has template syntax that has to be completed given a component's specific functions. In contrast, a service has an interface template, but without functionality.
- The basic concept of reusability in SOA is more powerful than that of DEVS. This is because the broker concept with support for publishing services and identifying services are not defined in DEVS.
- The concept of stateless service promotes loose coupling of composite services. Atomic and coupled model components require state information which includes time t ($t \in S$) in order to allow input and output event synchronization.

The concepts of loosely coupled and discoverable services are similar to dynamic structure DEVS where the structure of a model can change during simulation execution – i.e., capability is provided for adding and removing atomic and coupled models. The concept of executive in dynamic structure resembles that of broker service, but it is not the same as described above.

As noted earlier, the fundamental difference between DEVS and SOA is the 'broker' concept. SOA is grounded in the separation of publisher and subscriber services which can send and receive messages. The message-based interactions between the publisher and subscriber services can only be established by the broker service. The concept of broker is not defined in the DEVS formalism. This is because services (i.e., publisher, subscriber, and broker) have special roles and functionality based on the broker concept. That is, due to the lack of broker concept, the DEVS atomic or coupled components are not "service-enabled" – i.e., the basic syntax and semantics of the atomic and coupled components are insufficient for describing service-based software systems. Furthermore, the SOA is not the same as dynamic structure DEVS even though the structure of a coupled model can be modified during simulation. This is because DS-DEVS does not account for the SOA concepts of publishing and subscribing. An atomic model does not publish its services to the executive model nor can a subscriber inquire about services of a publisher.

Table 1: Association between DEVS and SOA frameworks.

SOA	DEVS
autonomous	atomic and coupled models modularity
association composable	hierarchy and closure under coupling
formal contract	inputs/output ports, variables, and couplings
abstract logic	$\langle X, S, Y, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta\rangle$ $\langle X, Y, D, \{M_d\}, EIC, IC, EOC\rangle$
reusable	basic models
stateless	state-based
loosely coupled	dynamic structure
discoverable	dynamic structure

It is possible to model service-based system using the elementary atomic and coupled model components by embellishing them with SOA properties (see Figures 1 and 2). One way to model the SOA publish and subscribe functionality for a pair of publisher and subscriber services is to introduce to every atomic model a pair of input and output ports. These ports with their couplings are dedicated to finding published services and notifying the subscriber. That is, Steps 1, 2, and 3 shown in Figure 2 can be modeled as the ((subscriber, identify-publisher), (publisher, identify-publisher)) and ((publisher, found-request), (subscriber, found-request)) couplings as shown in Figure 3. Here, DEVS messages represent service descriptions (WSDL) and data (SOAP) messages. The DEVS simulation protocol that manages the sending/receiving of messages corresponds to the communication SOA messaging framework (SOAP).

The DEVS "Publisher/Subscriber" model can be an abstraction for the Notification and Eventing model – a basic model of a subscriber (Travel Agent) requesting for a service from a publisher (Ski Resort). The publisher is assumed to be able to provide the requested service based on the response provided to the subscriber service by the broker service. As shown in Figure 3, the subscriber and publisher relationship is modeled via two uni-directional couplings (see the dashed lines). The messages sent and received messages between the publisher and subscriber (shown as solid arrows) can commence after it is known the publisher can provide the requested service. A subscriber atomic model can then send a message to the publisher and request a service. Such logic can control sending of output messages from publisher's output port publish-service to the subscriber's input port publish-service. Clearly, devising this kind of logic in the DEVS "Publisher/Subscriber" model is not desirable. This is because the internal details to be encoded in the "publisher" and "subscriber" are not conceptually viable and impractical for developing SOA-compliant DEVS models. Another difficulty is the concept of a publisher publishing its services to the broker. That is while the abstraction (due to the absence of a broker service) for the arrows 1, 2 and 3 can be modeled, there is no model for the broker service. The publish, request, and response messages shown in Figure 2 are mapped to the identify publisher and found publisher ports between the publisher and subscriber model components shown in Figure 3.

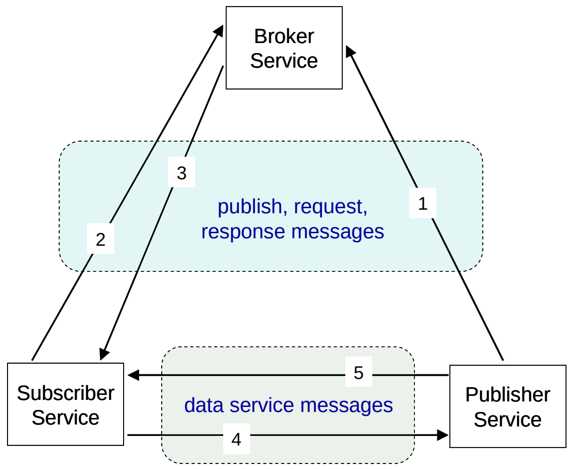

Figure 2. Basic SOA services and messaging patterns

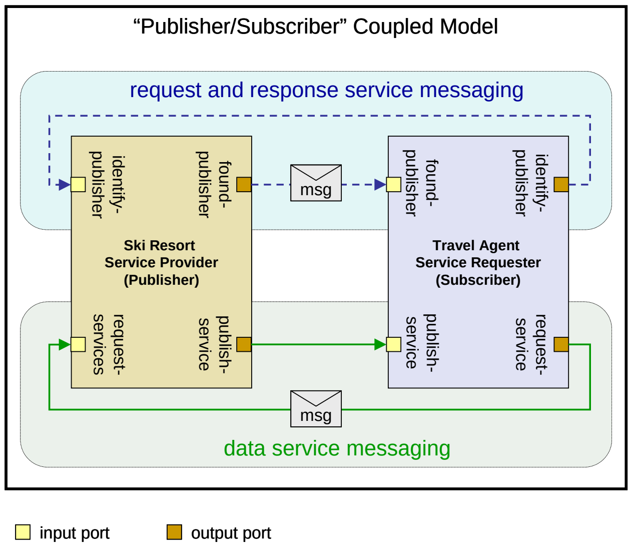

Figure 3. Non SOA-compliant DEVS models

With the SOA services and messages mapped to the DEVS model components, ports, and couplings, there exists no broker service concept. The broker service is not modeled independently – i.e., the dashed arrows shown in Figure 3 are not poor representation of the publish, request, and response relationships between the broker service and the publish and subscribe services. Additionally, a modeler cannot develop a simulation model that is compliant to the SOA specification. This example shows that while the DEVS coupled models can support some of the primitive SOA publish/subscribe concepts including autonomy and abstract logic, it is inadequate. It is better to build simulation models that are SOA-compliant.

5 SOAD SIMULATOR FRAMEWORK

In the previous section, we described that there are basic similarities and differences between the SOA elements and those of DEVS. SOA framework has a higher level of abstraction as compared with DEVS framework. The basic SOA elements shown in Table 2 can be divided into two groups. First, services, service description, and messages represent 'static' part of SOA. Second, communication agreement, messaging framework, and service registry and discovery represent the 'dynamic' part of the SOA. To create the SOADS (i.e., a generic simulated SOA), counterparts of the basic elements of SOA are needed. In Table 2, we have defined a set of DEVS elements that represent the static and dynamic aspects of the SOA. Three DEVS atomic models are proposed. Three of these have a one-to-one correspondence with the SOA services. The generic DEVSJAVA entity type (class) is extended to represent SOA service description. Entity is also extended to represent SOA messages. The input and output ports with couplings are used for messaging framework. An executive atomic model can represent the service registry and discovery and the coupled model can represent composition of (primitive or composite) services.

5.1 Software Models

The publisher, subscriber, and broker services are the basic elements for both service-oriented software systems. The services can be synthesized to form primitive and composite service composition. Next, these two service compositions are described.

5.1.1 Primitive SOAD Models

The generic primitive service composition using DEVS atomic models (publisher, subscriber, and broker) is shown in Figure 4. Messages produced by a service and consumed by another are shown as envelops. As noted above, a message may contain service description or other content consistent with a chosen messaging framework. For example, the message from the Broker to the Subscriber is a service description which contains an abstract definition (an interface for the operation names and their input and output messages) and a concrete definition (consisting of the binding to physical transport protocol, address or endpoint, and service). Another message could be from the Publisher to the Subscriber where the result of the requested service (returned message from the Publisher). The implementation of these messages can be based on SOAP. In the basic SOA framework, the internal operations of atomic services and their interactions are deferred to specific standards and technologies (e.g., .NET (Lenz and Moeller 2003)).

Table 2: DEVS and SOA elements.

SOA Model Elements	SOAD Model Elements
services (publisher, subscriber, broker)	atomic models (publisher, subscriber, broker)
service description	entity (service-information)
messages	entity (service-lookup & service-message)
messaging framework	ports & couplings
service registry & discovery	executive model
service composition	coupled models (primitive and composite)

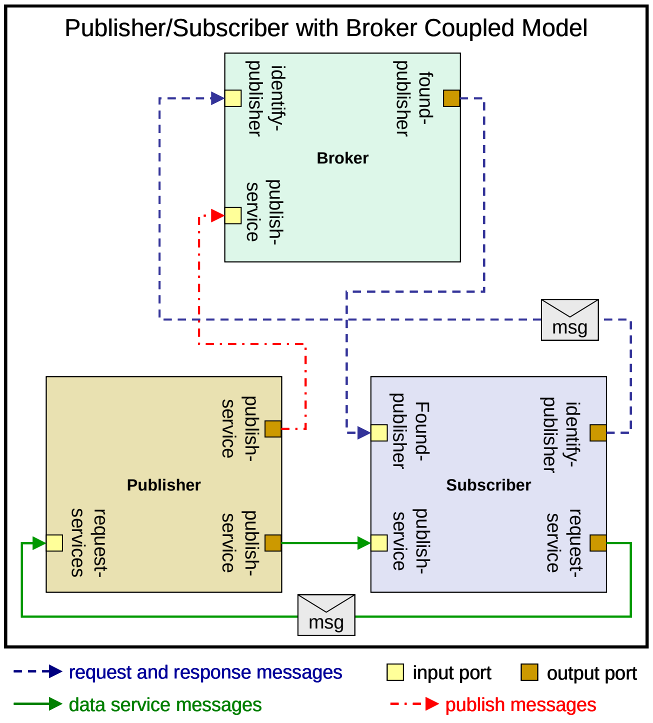

Figure 4: SOA-compliant DEVS models

5.1.2 Composite SOAD Models

An essential capability for simulating service-based software systems is to support modeling of composite service composition. As shown in Figure 5, a composite service composition has publisher or subscriber service which itself is a primitive service composition. Since broker service is required for both primitive and composite service composition, two cases can be considered – i.e., either a single broker service or multiple broker services are used. Both cases can be supported. Use of a single broker service is shown in Figure 4. To avoid cluttering of Figure 5, the brokers shown in the Subscriber and Publisher services are the one that is used for these brokers (this is shown with shaded background for the two brokers and their couplings). The three kinds of couplings provided in coupled DEVS models supports use of a single broker for the primitive service compositions (i.e., Subscriber and Publisher) and their composite (hierarchical) service composition. As can be seen, for example, Publisher1 service has the role of a subscriber with respect to the Subscriber2 which has the role of a publisher. The common concept of DEVS and SOA modularity allows creating composite service composition without restrictions. The DEVS hierarchical coupled modeling naturally supports multiple hierarchical broker services.

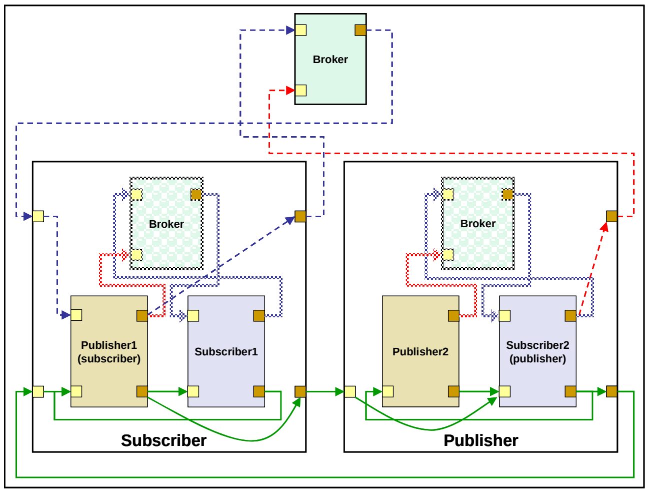

Figure 5. SOA-compliant DEVS model of composite services with a broker service

5.2 Hardware Model

A simple model of a network is used to complement the software aspect of SOA with hardware aspect (see Figure 6). The model is defined as link with finite capacity, transportation delay, and FIFO queuing of messages. This component is not a service – it models the medium through which services send and receive message. For simplicity, direct communications between publishers and subscribers are shown in Figure 6. In general, all messages including publish, request, and response messages to the broker also need to go through the network unless all publisher, subscriber, and broker service are assigned to one processing node.

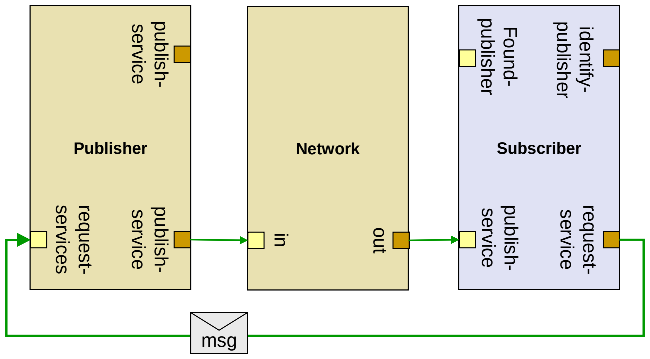

Figure 6: Communication of messages between publisher and subscriber services via a network component

6 SOAD SIMULATION ENVIRONMENT

The above SOAD modeling and simulation approach has been realized using the DEVSJAVA simulation environment (see Figure 7). A set of generic SOAD models are designed and implemented (Kim 2008). They represent static and dynamic aspects of service-oriented software systems. These models are partitioned according to the SOA Models which extend the DEVS Models. The SOAD models are generic in the sense of the generality supported by SOA and DEVS. Specific SOA models (called Application Model) can be used to describe hierarchical service-oriented software systems. A basic hardware model is also developed, but due to lack of space it is not included here. These models are executed using the DEVSJAVA simulation engine.

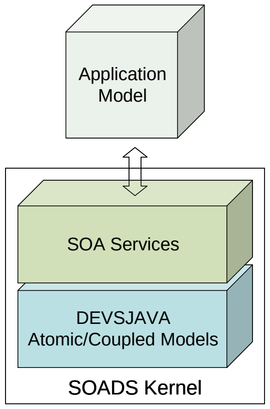

Figure 7: SOADS simulation environment

6.1.1 Messages

Three generic messages are developed. They are abstractions of the WSDL and SOAP specifications (see Figure 8). These message classes are derived from the entity class which belongs to the DESVJAVA API. The *ServiceInfo* and *ServiceLookup* correspond to the WSDL specification (see Table 3). These messages are needed for publishing services and their discovery. The *ServiceInfo* message is used for publishing a service with the broker. It contains a service definition given a service name, service description, service type, the list of endpoints, and binding information. The port and coupling concepts are do not have a one-to-one correspondence the WSDL's port and binding elements – they serve as counterparts to the physical address at which a service can be accessed and the transport technology for message communication. The *ServiceLookup* message is used to find the desired service in the broker using a service name and an endpoint in the message.

Table 3: WSDL and *ServiceInfo* and *ServiceLookup* Messages

WSDL	***ServiceInfo***	***ServiceLookup***
interface	service name, endpoints	service name, endpoint
message	n/a	data
service	n/a (ports and couplings)	n/a (ports and couplings)
binding	binding info	n/a

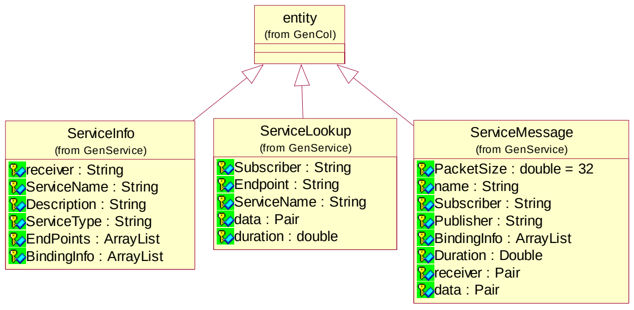

Figure 8: SOA message type models

ServiceMessage message type corresponds to the SOAP specification. It is required to define the data content that is exchanged between a subscriber and a publisher. The differences between the *ServiceMessage* and the other messages are the data is actually used in the subscribed publisher and it must specify the destination of the message.

6.1.2 Primitive Services

The specifications for the primitive SOA publisher, subscriber, and broker service are defined as DEVS atomic models shown in the Figure 9. The *ServiceBroker* has a container (UDDI) to store *ServiceInfo* messages. The *ServiceSubscriber* maintains a list of services to lookup a broker. The *ServicePublisher* defines specific behaviors of its endpoints in the *performService* method. Depending on the subscribed port (i.e., an endpoint), the *performService* can execute different functions and return a Pair which defines data type and value (this is used as data in the *ServiceMessage*). Since multiple users can subscribe to an endpoint at the same time, the *RequestList* in the *ServicePublisher* is devised to handle multiple user subscriptions simultaneously. These requests are processed using FIFO scheme. For brevity some methods (see Section 3.2) for the primitive services are not shown in the class diagrams.

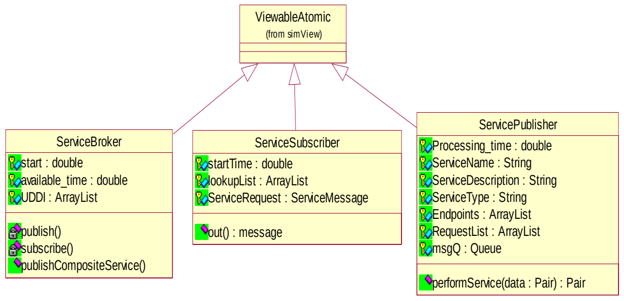

Figure 9: Primitive publisher, subscriber, and broker service models

6.1.3 Primitive Service Composition

As shown in Figure 10, the SOAD has primitive services with a network link and transducers). Based on generic interfaces defined for SOA services, default couplings are defined. Furthermore, defaults couplings are also defined for the network and transducer models. Therefore, to model a primitive service composition, it is necessary to construct the list of subscribers and publishers.

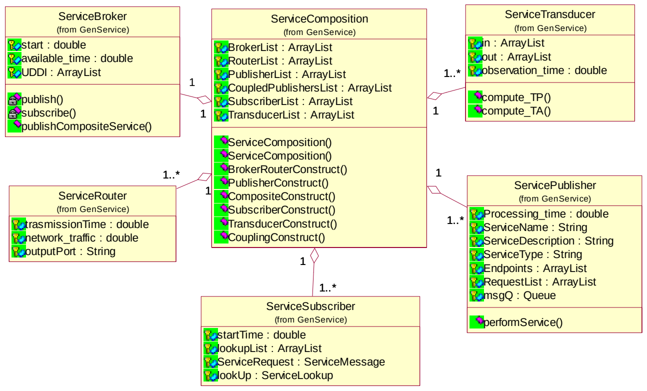

Figure 10: Model of primitive service composition

6.1.4 Composite Service Composition

The composite service composition is similar to the primitive service composition as shown in Figure 11, except there is no list for subscribers since publishers in the composite service composition can be also subscribers. The flow of service invocations needs to be specified given the specifics of the service-based software systems that are being modeled. This is a basic capability for hierarchical service composition which has to be extended to support different kinds of workflow patterns (Russell et al. 2006).

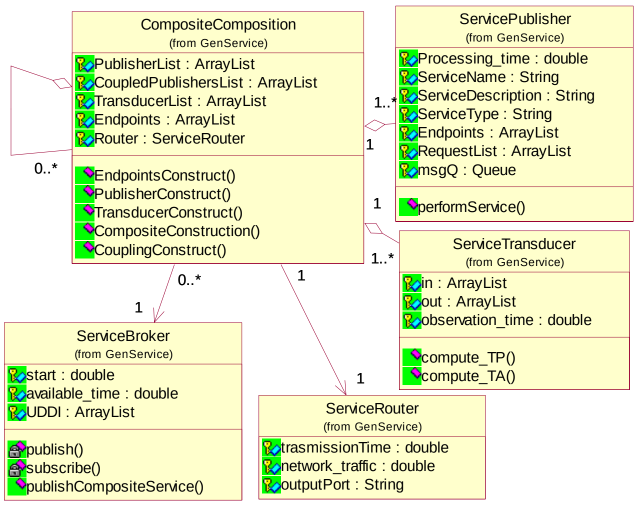

Figure 11: Model of composite service composition

6.2 Example Simulation Model

Models for the primitive and composite service compositions are developed in the SOAD Simulator (Kim 2008). The model shown in Figure 12 has 4 software components (one subscriber (Travel Agent), two publishers (USZip, and Ski Resort), and one broker (Broker)) and one simple hardware component (Router Link). The model also includes five transducers for each of the software and the hardware components. Another simulation model is for a real Voice Communication Service (Yau et al. 2008) which was used to validate the design and implementation of the SOADS environment. The Travel Agent Service and Voice Communication Service are used to show the primitive and composite service compositions.

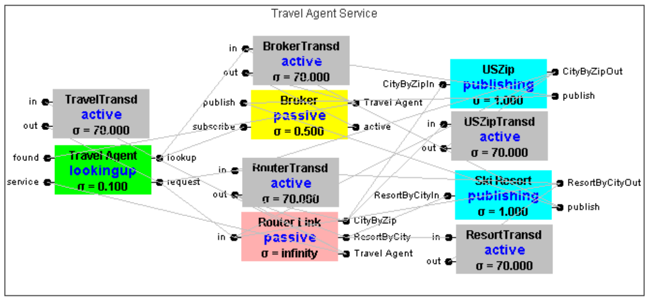

Figure 12: Travel Agent Service primitive composite model

The simple Travel Agent Service can be used to illustrate modeling the basic throughput, timeliness, and accuracy quality attributes of SOA-compliant software based systems. For example, the dynamics of the Travel Agent can be observed in terms of the events it generates and consumes. The output events are defined for the service lookup, the service lookup retry, and the publisher service request. The scheduling of these events is defined in δ_{int} and δ_{ext}. The output events times relative to time instances at which they can be generated are defined to be 0.5, 0.0, and 1.0 second, respectively. The first event is scheduled by the internal transition function. The second and third events are due to the external transition function – i.e., processing of the input events from the Broker. There is also another external transition function for processing the input event it is received from a publisher (either USZip or Ski Resort). The time allocated for δ_{ext} is 1.0 second. The dynamics of the USZip and Ski Resort are the same. Each takes 1.0 second to process a request received from the Router Link and produce an output event. The Router Link takes 0.5 second to deliver a publisher's output event as an input event to a subscriber. The Router Link takes also 0.5 second to deliver a subscriber's output event as an input event to a publisher. The Broker takes 0.0 seconds to respond to the Travel Agent (whether it finds a requested service or not). For simplicity, in this example, the subscriber sends its requests to the publishers sequentially, but simultaneous requests are straightforward to model. Table 4 shows sample quality of service measurements for the Travel Agent Service operating for a period of 71.5 seconds. These generic metrics are captured by the transducers. The SOA generic models defined in Table 2 have stochastic timings, but the results given in Table 4 are based on deterministic timings in order to verify the logical correctness of the generic primitive service composition.

Table 4: Selected metrics for the Travel Agent Service model

Component	Quality of Service Measurements
Travel Agent	Average Turnaround Time (sec): 2.0 Total size of data received (Kbytes): 640.0 Number of subscribed publishers: 2
USZip	Publisher Throughput (msgs/sec): 0.156 Amount of data received (Kbytes): 320.0 Number of subscribers: 1
Ski Resort	Publisher Throughput (msgs/sec): 0.156 Amount of data received (Kbytes): 320.0 Number of subscribers: 1
Router	Average Transmission Time (sec): 0.5 Total size of message received (Kbytes): 1280.0 Utilization for a period of time (%): 1.7073

7 RELATED WORK

Within the simulation community the interest has focused on the use of web services for distributed simulation. For example, the core HLA capabilities (IEEE 2000) can be extended with SOA concepts (e.g., (Chen et al. 2006)) or web services used for distributed simulation (e.g., (Hu et al. 2007; Mittal et al. 2007)). Web services are also proposed to define an ontology with a corresponding software infrastructure for simulation model reuse (Bell et al. 2007).

Researchers interested in web services have proposed the simulation-based approach to assist in analysis, design, and testing of SOA-based software systems. A framework has been developed using HLA to support web services verification and validation (Tsai et al. 2007). Processes, services, and workflows are described using the Process Specification and Modeling Language (PSML). The modeling language used in this framework uses HLA for simulation execution. The PSML and DEVS models have basic differences such as explicit representation of time, event preemption, and closure under coupling. Another important difference is the mapping from DEVS and PSML to SOA. SOAD is defined in terms of the basic SOA elements (subscriber, publisher, and broker) as well as the primitive and composite service composition. From a higher perspective, SOAD is targeted for modeling and simulation of service-based computing systems whereas PSML is targeted for their actual realizations.

Some other approaches have also been proposed to support some software engineering phases of service-oriented systems such as workflow designs. One approach is based on use of Petri Nets formalism (Srini and Sheila 2003). It has been developed to analyze various aspects of web services such as complexity of web services. The DAML-S ontology is used for describing web services that can be simulated using KarmaSIM simulator. An execution scheme based on situation calculus is mapped to Petri Nets modeling elements and thus supports performance analysis, verification, and validation of web services. This approach, however, does not provide a direct mapping from the SOA basic elements to the Petri Nets modeling elements. Agent-based simulation is used to model service chaining (Anderson et al. 2005). The simulator allows macro-level modeling and testing of web services with support for network-like visualization. This simulation focuses on the Web services flow patterns. Another simulator which is a Java-based tool has been proposed for studying performance of service-oriented software systems (John et al. 2006). For validation of service-based software systems, a UML simulator has also been proposed (Hiroyuki et al. 2006). It supports execution of BPEL4WS models described in UML. The simulator is developed for BPEL/UML models where interface of services can be simulated and used in conjunction with real services. The execution of the BPEL/UML models are defined in terms of Activity Hyper-graph and implemented as web services.

Considering the approaches briefly reviewed in relation to SOAD, it is useful to consider support for representing (logical and real) time. The explicit use of time (discrete values) in services is crucial in developing verifiably correct simulation models of dynamical real services. The time-based execution of each model plays an important role in developing dynamical simulations that can be validated. For example, a simulated service where its operations take real time to complete can be used instead of a real service. Direct representation of time, therefore, is necessary for characterizing complex structures and behaviors of services. This, in turn, supports evaluating time-based quality of service attributes such as throughput. Of these, the approach which uses the Petri Nets formalism enjoys explicit use of time. However, the situation calculus for the DAMIL-S supports sequencing of actions (i.e., time is not explicitly accounted for). Furthermore, unlike the proposed SOAD, none of the above approaches are formalized to model and simulate services that may change their structures at run-time and separately modeling service-oriented software systems in terms of their hardware and software layers.

8 CONCLUSION

The basic goal for the proposed SOAD framework is to take advantage of fundamental commonalities between SOA and DEVS. As we have shown, the simulated services share important characteristics with those of real services. This is useful because users interested in simulating service-oriented services can use the SOA principles and the component-based modeling concepts. An important observation for the proposed framework is that the DEVS formalism is well positioned to support modeling of (i) services with dynamic structures and (ii) separately modeling software and hardware aspects of service-based software systems. The extension of the SOAD with the key capabilities is under development. The SOAD framework has the potential to inspire and serve as a basis for community-based development of realistic SOA. A community of researchers and developers, akin to those of ns-2, can introduce important capabilities such as modeling and simulating complex workflow patterns. Development of expressive and robust model libraries are very useful for advancing simulation-based design of service-based software systems where disparate quality of service attributes such as timeliness and accuracy can be evaluated and analyzed systematically and efficiently. Another incentive to use the DEVS approach is the availability of distributed DEVS-based tools that are available freely for research and education purpose and can support large-scale simulations.

ACKNOWLEDGEMENT

This research has been supported by the NSF Grant No. CCF-0725340.

REFERENCES

ACIMS. 2001. *Arizona Center for Integrative Modeling and Simulation*. Available via <http://www.acims.arizona.edu/SOFTWARE> [accessed 2007].

Anderson, C., J. A. Rothermich and E. Bonabeau. 2005. Modeling, quantifying and testing complex aggregate service chains. In *Proceedings of the 2005 IEEE Inter-*

national Conference on Web Services, 274–281. Orlando , Florida , USA.

Barros, F. 1997. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation*, 7(4): 501–515.

Bell, D., S. d. Cesare, M. Lycett, N. Mustafee and S. Taylor. 2007. Semantic Web Service Architecture for Simulation Model Reuse. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, 129–136. Chania, Crete Island, Greece.

Chen, X., W. Cai, S. J. Turner and Y. Wang. 2006. SOArDSGrid: Service-Oriented Architecture for Distributed Simulation on the Grid. In *Workshop on Parallel and Distributed Simulation* 65–73. Washington, DC, USA.

Chen, Y. and W. T. Tsai. 2008. *Distributed Service-Oriented Software Development*,. Kendall/Hunt Publishing.

Erl, T. 2006. *Service-Oriented Architecture Concepts, Technology and Design*, Prentice Hall.

Hild, D. R., H. S. Sarjoughian and B. P. Zeigler 2002. DEVS-DOC: A Modeling and Simulation Environment Enabling Distributed Codesign. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 32(1): 78–92.

Hiroyuki, K., F. Taku, M. Toshiyuki and K. Sadatoshi. 2006. A UML Simulator for Behavioral Validation of Systems Based on SOA. In *Proceeding of the 2006 International Conference on Next Generation Web Services Practices*, 3–10. Seoul, Korea.

Hu, X., B. Zeigler, M. H. Hwang and E. Mak. 2007. DEVS Systems-Theory Framework for Reusable Testing of I/O Behaviors in Service Oriented Architectures. In *Proceeding of the 2007 IEEE International Conference on Information Reuse and Integration*, 394–399. Las Vegas, NV, USA.

IEEE. 2000. *HLA Framework and Rules, Version IEEE 1516-2000*.

John, G., H. John, L. Lei and L. Na. 2006. Performance engineering of service compositions. In *Proceeding of the 2006 international workshop on Service-oriented software engineering*, 26–32. Shanghai, China.

Kim, S. 2008. *Simulation of Service Based System: Modeling and Implementation using the DEVS-SUITE*. MS Thesis, Computer Science and Engineering, Arizona State University, Tempe, Arizona, USA

Lenz, G. and T. Moeller. 2003. *.NET: A Complete Development Cycle*, Addison-Wesley.

Mittal, S., J. L. Risco and B. P. Zeigler. 2007. DEVS-based simulation web services for net-centric T&E. In *Proceedings of the 2007 summer computer simulation conference*, 357–366. San Diego, California, USA.

Møller, A. and M. I. Schwartzbach. 2006. *An Introduction to XML and Web Technologies*, Addison-Wesley.

ns-2. 2002. *Network Simulator* Available via <http://www.isi.edu/nsnam/ns/> [accessed 2008].

Russell, N., A. H. M. t. Hofstede, W. M. P. v. d. Aalst and N. Mulyar 2006. Workflow control-flow patterns: A revised view. *BPM Center Report BPM-06-22*.

Srini, N. and M. Sheila 2003. Analysis and simulation of Web services. *Computer Networks*, 42(5): 675–693.

Tsai, W. T., Z. Cao, X. Wei, R. Paul, Q. Huang and X. Sun 2007. Modeling and Simulation in Service-Oriented Software Development. *SIMULATION Transactions*, 83(1): 7–32.

Yau, S. S., N. Ye, H. S. Sarjoughian and D. Huang. 2008. Developing Service-based Software Systems with QoS Monitoring and Adaptation. In *Proceeding of the 12th IEEE Int'l Workshop on Future Trends of Distributed Computing Systems*. Honolulu, Hawaii, USA.

Zeigler, B. P., H. Praehofer and T. G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press.

Zeigler, B. P. and H. S. Sarjoughian. 1997. Object-Oriented DEVS. In *Proceeding of the 11th SPIE*, 100–111. Orlando, Florida, USA.

AUTHOR BIOGRAPHIES

HESSAM S. SARJOUGHIAN is Assistant Professor of Computer Science & Engineering at ASU and Co-Director of the Arizona Center for Integrative Modeling and Simulation. His research focuses on multi-formalism modeling, collaborative modeling, distributed simulation, and software architecture. He can be contacted at <sarjoughian@asu.edu>.

SUNGUNG KIM is a Master student in the Computer Science and Engineering department at ASU. His research is in the development of SOA based simulation models. He can be contacted at <skim109@asu.edu>.

MUTHUKUMAR RAMASWAMY is a Master student in the Masters of Engineering in Modeling & Simulation program at ASU. His research area is system theory based simulation of SOA-based software systems. He can be contacted at <muthukumar.ramaswamy@asu.edu>.

STEPHEN S. YAU is Professor of Computer Science and Engineering and Director of Information Assurance Center at Arizona State University, Tempe. His current research interests include cyber security, trust, privacy, software engineering, distributed computing systems, and ubiquitous computing. He can be contacted at <yau@asu.edu>.