SIMULATOR FOR SERVICE-BASED SOFTWARE SYSTEMS: DESIGN AND IMPLEMENTATION WITH DEVS-SUITE

by

Sungung Kim

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2008

SIMULATOR FOR SERVICE-BASED SOFTWARE SYSTEMS: DESIGN AND IMPLEMENTATION WITH DEVS-SUITE

by

Sungung Kim

has been approved

October 2008

Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Stephen S. Yau
Wei-Tek Tsai

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Simulation modeling offers important and unique capabilities for analysis and design of service-oriented computing systems that must satisfy multiple, competing Qualities of Service (QoS) requirements. In order to aid design of service-based software systems (SBS), it is important to employ a suitable modeling framework that can account for the Service-Oriented Architecture (SOA) concepts. Toward this goal, this thesis develops a simulator that can represent and execute service-based software systems. A novel set of generic SOA-based models are developed based on the Discrete Event System Specification (DEVS) framework. The resulting SOA-based DEVS (SOAD) models can be created in the DEVS-Suite simulation environment, a newly developed extension of the DEVSJAVA Tracking Environment. The SOAD models are implemented and incorporated into the DEVS-Suite simulator which affords animation of simulation executions and visualization of simulation results as time trajectories. To demonstrate the modeling capabilities of SOAD, a hierarchical model of a travel agency services is developed. Simulation models for a Voice Communication System (VCS) are also developed according to a real SOA-based implementation of VCS. Future extensions for the SOAD simulator are proposed to enable modeling and simulation of adaptable service-based software systems.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Hessam Sarjoughian, Department of Computer Science and Engineering, Arizona State University. He has given countless hours of his time towards guiding me in this research as well as mentoring me as a graduate student.

I would also like to thank my other committee members, Dr. Stephen S. Yau and Dr. Wei-Tek Tsai for serving on my thesis committee.

I want to thank my colleagues at the ACIMS, Gary Mayer, Vignesh Elamvazhuthi, Sajjan Sarkar, Muthukumar Ramaswamy, and Mohammed Muqsith. It was a pleasure working with you guys.

I am thankful to the members of the Science of Design NSF project and in particular Dazhi Huang who provided experimental results that helped this thesis. The partial financial support of this research under NSF Grant number CCF-0725340 is gratefully acknowledged.

Special thanks to my father and mother. They always support and encourage me. Without their help, this degree would not have been possible.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

1. Introduction

1.1. A Statement of the Problem

Modeling and Simulation has become a necessity for developing many kinds of complex, large scale systems. A major part of engineering systems is to develop models that can aid analysis and design activities. Models which describe both structural and behavioral specifications can be simulated in the virtual environments. They help to detect requirement and design errors in the early stage of product development cycles. This capability can significantly reduce the cost associated with eliminating errors in system implementation and testing development stages. A model can be written by using a variety of system specification formalisms. For example, the Discrete Event System Specification (DEVS), the Discrete Time System Specification (DTSS), and Differential Equation System Specification (DESS) formalisms can be used to simulate discrete event, discrete-time, and continuous models (B.P. Zeigler et al., 2000). Simulation is commonly used as a technique for better understanding of system/software designs, performance optimization, as well as undertaking the role of traditional experimentation.

Currently the concept of Service Oriented Computing (SOC) paradigm is rapidly being adopted for developing distributed computing systems. The Service Oriented Architecture (SOA) is proposed for building software systems from services (Erl, 2006). This framework affords composition of various types of services for distributed applications built on different platforms. An important consideration in developing SOA-based software systems (SBS) is supporting multiple quality of service (QoS) features, such as timeliness, throughput, accuracy and security (Yau et al., 2008). To achieve this

goal, QoS Monitoring and Adaptation sub-systems, in combination with services, are needed to collect and analyze tradeoffs between multiple QoS features and to adapt the composition of services accordingly. As shown in Figure 1, SBS with the Monitoring and Adaptation sub-systems is collectively referred to as Adaptable SBS (ASBS).

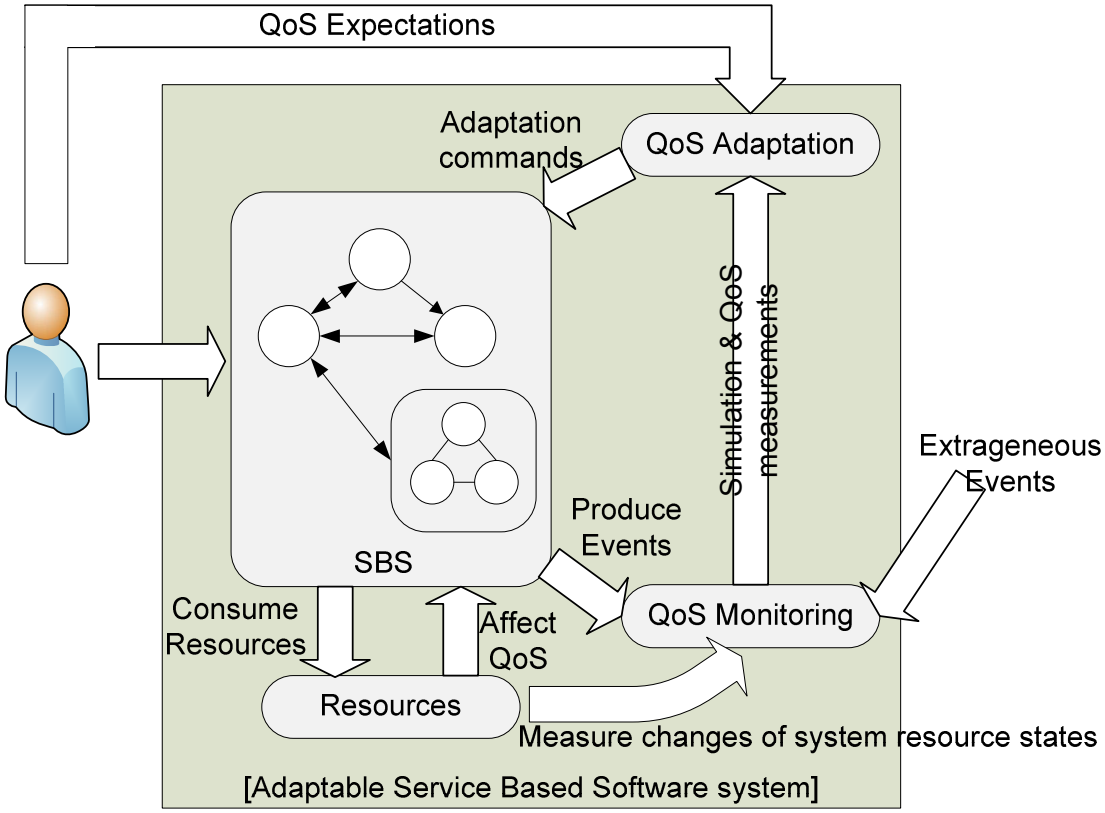


Figure 1. The conceptual view of ASBS (Yau et al., 2008)

Simulation modeling can be used to study multiple QoS attributes of service-based systems and thus determine desirable tradeoffs. In order to verify and validate the design of ASBS, in this research, the DEVS formalism is used to define the characteristics of ASBS. The DEVS framework, similar to other modeling and simulation approaches, supports analysis, design, and development of systems (B. P. Zeigler & Sarjoughian, 2003). This modeling formalism provides a rigorous basis for simulating

service-oriented software systems. A variety of object-oriented Modeling and Simulation (M&S) tools based on the DEVS formalism have been implemented in different programming languages (e.g., C++ and JAVA) and used to simulate many different kinds of systems such as command and communication software systems. In this research, we will use the DEVSJAVA (ACIMS, 2001) and the DEVSJAVA Tracking Environment (DTE) (H.S. Sarjoughian & Singh, 2004; Singh & Sarjoughian, 2003). The former supports animation of hierarchical models and the latter supports specifying and dynamically collecting simulation results as time charts and tabulated data.

The DEVSJAVA shows a view of the entire hierarchy of the simulation model using components-within-components-style and animation of messages moving along the paths of the coupling between components during the simulation (Mather, 2003; B. P. Zeigler & Sarjoughian, 2003). The DEVSJAVA supports injecting inputs into the model during the simulation dynamically so users can easily analyze the dynamics of the simulated models.

The DTE offers a graphical user interface to identify and enable semi-automated experimentations to track the simulation model data sets including states (i.e., Phase, Sigma, Time of the Next Event, Time of the Last Event) and input/output events. The DTE supports the concept of the experimental frame and an implementation of it is integrated with the DEVSJAVA simulator. It allows user flexibility to select and observe the simulation data sets which are tracked dynamically for any number of atomic and coupled models. While simulating a model on the DTE, the simulation data sets from the selected simulation models including user-defined statistical simulation are displayed in a

tabular format which is called the Tracking Log. To increase the usability of the DTE, a plotting tool called TimeView has been implemented. The design and implementation of the TimeView is based on the concept of components and display data in terms of time. The integration of the TimeView into the DTE can support run-time visualization of simulation. However, the TimeView did not account for the concept of time as used in the DTE. Such a capability is necessary for integrating the TimeView into the DTE environment. To aid this research and others (Elamvazhuthi, 2008; H. S. Sarjoughian, in preparation), the DEVS-Suite environment which integrates DEVSJAVA, DTE, and TimeView has been developed (Kim et al., in preparation).

Currently a few approaches have been proposed for SOA-based simulation frameworks in order to help develop service-based software systems (Anderson et al., 2005; Chang et al., 2005; Hiroyuki et al., 2006; John et al., 2006; Srini & Sheila, 2003; Tsai, Fan et al., 2006). These approaches are mainly focused on models that can be simulated for testing purposes of real services. While different modeling and simulation frameworks have been used to simulate service-based software systems, it is desirable to develop an approach where the basic concepts of time as well as software/hardware co-design (Hild et al., 2002; Hu, 2007) can be explicitly modeled and simulated (H. Sarjoughian et al., 2008). Toward this goal, first we need to develop a generic set of SOA-based simulation models including the service broker, service provider, service client, service composition and router SOA components. Therefore, first we need to develop an approach to build an SOA-based DEVS (SOAD) simulator. Second, we need to develop simulated and actual service-based software system examples to examine the

capabilities of the SOAD simulator. The simulator is aimed at supporting simulation-based verification and validation of SBS designs with multiple QoS attributes. Furthermore, the SOA-compliant DEVS framework (H. Sarjoughian et al., 2008) offers a basis for introducing the capability to model and simulate adaptive service-based software systems.

1.2. Thesis Contribution

The overall contribution of this thesis is the design and development of the new SOAD simulation environment which introduces the capability to model and simulate service-based software systems as shown in Figure 2. Generic SOA-based simulation models are designed by introducing SOA modeling capabilities into the object-oriented DEVS-Suite environment. The SOAD environment supports modeling SOA-based primitive and composite services. Example simulation models are developed, executed, and evaluated to demonstrate how the SOAD simulator can support design and analysis of software-based software systems.

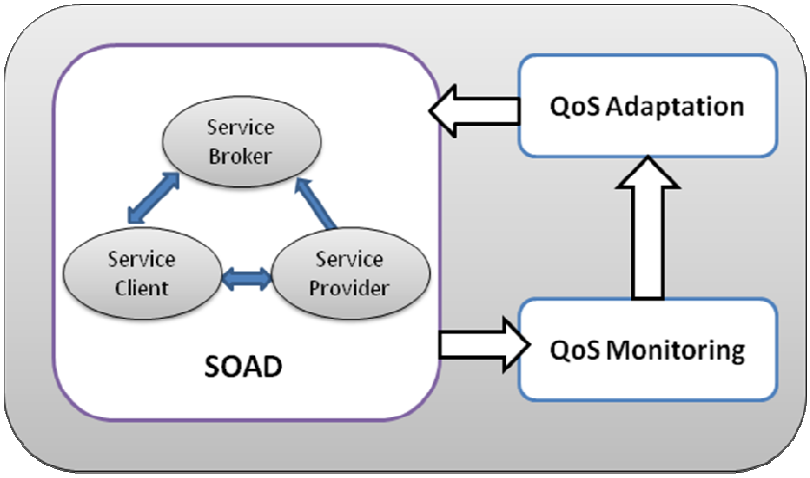


Figure 2. SOAD with M&A sub-systems

1.3. Thesis Organization

The remainder of this thesis is organized as follow. Chapter 2 reviews and discusses related background and research. It includes the detailed descriptions of the DEVSJAVA and DTE as well as the basic modeling and simulation concepts employed in this thesis. It describes the NSF Science of Design project including four critical QoS features and experimental scenarios along with the comparison to other SOA-based simulation approaches. Chapter 3 describes the development of the DEVS-Suite. Chapter 4 presents the SOAD approach and the development of an abstract set of SOA-based simulation models for the service broker, service provider, service client, service composition, and network. In addition to these basic behavioral models, the observational model called transducer is introduced in this chapter in order to simplify simulation data collection for the services and network. Chapter 5 details two example models one of which (i.e., voice communication service) is based on actual software systems that are implemented with SOA and .Net technology. These simulation models are used to validate the abstract SOAD models that are developed against real experimentations. Finally, Chapter 6 presents conclusions and discusses future research.

2. Background and Related Works

This chapter discusses background information about the field of software modeling and simulation including the detailed description of the SimView and DTE and the related works on the SOA-based simulation approaches. Also it includes the introduction to the proposed NSF SOD project including four critical QoS features and experimental scenarios.

2.1. Discrete Event System Specification (DEVS)

Simulation can make many software development process improvements in terms of cost, repeatability, and time. This observation can apply to SBS since it is also based on fundamental concept of components and their interactions. In this research, we use the Discrete Event System Specification (DEVS) formalism to specifying an SOA-based software system. The DEVS formalism provides a method to specify a software system using a time base, input, state, and output, and functions for determining next states and outputs given current states and inputs (B. P. Zeigler & Sarjoughian, 2003). In the DEVS, a system is consisting of two types of models: atomic and coupled models.

An atomic model (B.P. Zeigler et al., 2000) is mathematically represented as,

$$M = (X, Y, S, \delta ext, \delta int, \delta con, \lambda, ta)$$

Where:

- X is the set of input values
- Y is the set of outputs
- S is a set of states
- $\delta int : S \rightarrow S$ is the internal transition function

- $\delta ext : Q \times X^b \rightarrow S$ is the external transition function, where

  $Q = \{(s,e)| s \in S, 0 \leq e \leq ta(s)\}$ is the total state set and e is the time elapsed since last transition

- $\delta con : Q \times X^b \rightarrow S$ is the confluent transition function
- $\lambda: S \rightarrow Y$ is the output function
- $ta: S \rightarrow R^+_{0,\infty}$ is the time advance function

Coupled models in the DEVS can be represented by coupling two or more DEVS atomic models. A coupled model contains the set of components, the set of input ports, and the set of output ports. DEVS employs the concept of input and output ports to represent the connection between each component. The coupled model itself also can be used as a DEVS atomic model to form a larger coupled model (B. P. Zeigler & Sarjoughian, 2003) by coupling an output port of a component with an input port of others. To simulate a DEVS atomic/coupled model, the DEVSJAVA Simulation Viewer which provides animation of messages moving along the paths of the coupling between components and the Tracking Environment which provides a simple graphical user interface to identify and enable semi-automated experimentations to track the simulation model data sets are used.

Mathematical representation of a coupled model (B.P. Zeigler et al., 2000) is described below.

$$DN = (X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\})$$

Where:

- X is the set of external input values

- Y is the set of outputs
- D is a set of components names;
- For each i in D
    - M_i is a component model
    - I_i is the set of influences for i
    - For each j in I_i
    - Z_{ij} is the i-to-j output translation function

2.2. DEVSJAVA Simulation Environment

2.2.1. DEVSJAVA Simulation Viewer

The SimView provides a view of the arbitrary levels of coupled model using boxes-within-boxes-style and animation of messages moving along the paths of the coupling between components during the simulation (Mather, 2003). The interface of the SimView is shown in Figure 3.

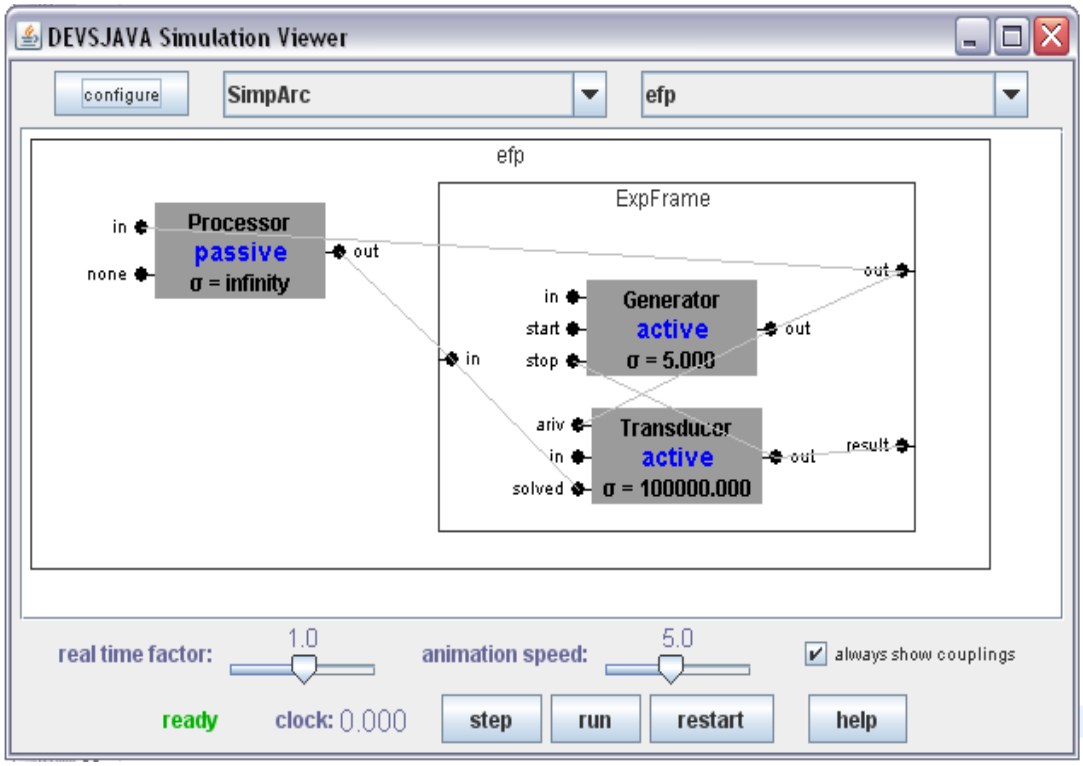


Figure 3. The DEVSJAVA simulation viewer

In addition to the visualization capabilities of the SimView, it allows users to inject input values into Inports of components dynamically during the simulation so that users can easily model and analyze the behavior and hierarchy of simulation model.

Figure 3 shows that the EFP model is currently loaded into the SimView. We are also going to use this model for the DTE as a reference model of the DEVS. The EFP model consists of three atomic model components, as shown in Figure 4, the Generator which generates external events and sends them to the Transducer and Processor, the Processor which processes external events received from the Generator and send the simulation results to the Transducer, and the Transducer which records statistical results

of simulation and request start/stop of simulation to the Generator. The Generator and Transducer are coupled together to form the experimental frame.

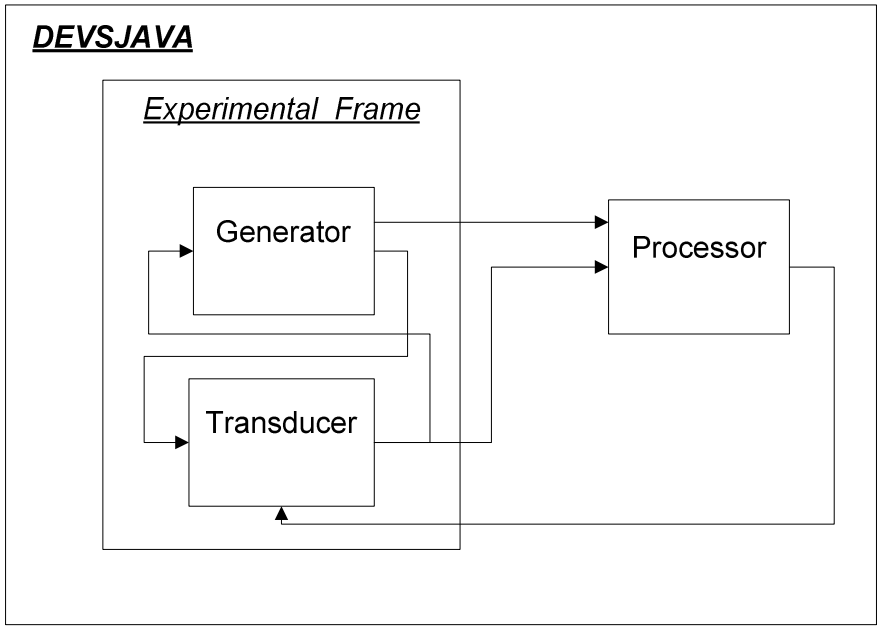


Figure 4. The EFP model in the DEVS

2.3. Tracking Environment with TimeView

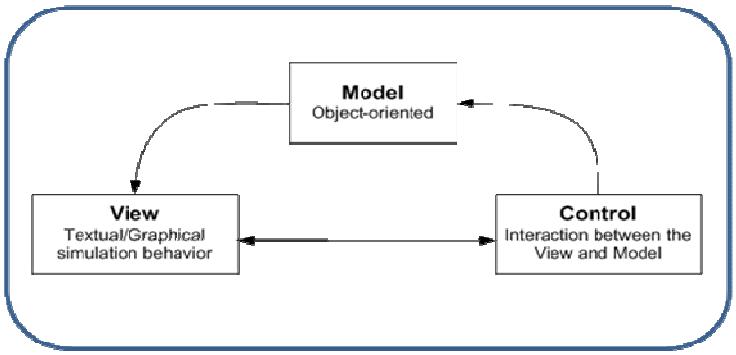
2.3.1. Architecture Design of Tracking Environment

The Tracking Environment was developed based on the software architecture as shown in Figure 5 (b). The illustrated software architecture contains a conceptual interface layer called FAÇADE layer to handle data and control services required by the VIEW and CONTROLLER in conjunction with the classical Model-View-Control (MVC) paradigm as shown Figure 5 (a). As illustrated in Figure 5 (b), only the FACADE layer is allowed to interact with the MODEL and its inner components. In the traditional MVC paradigm, the simulation data sets displayed on the View are obtained directly from the simulation model. This means the View is also allowed to interact with the

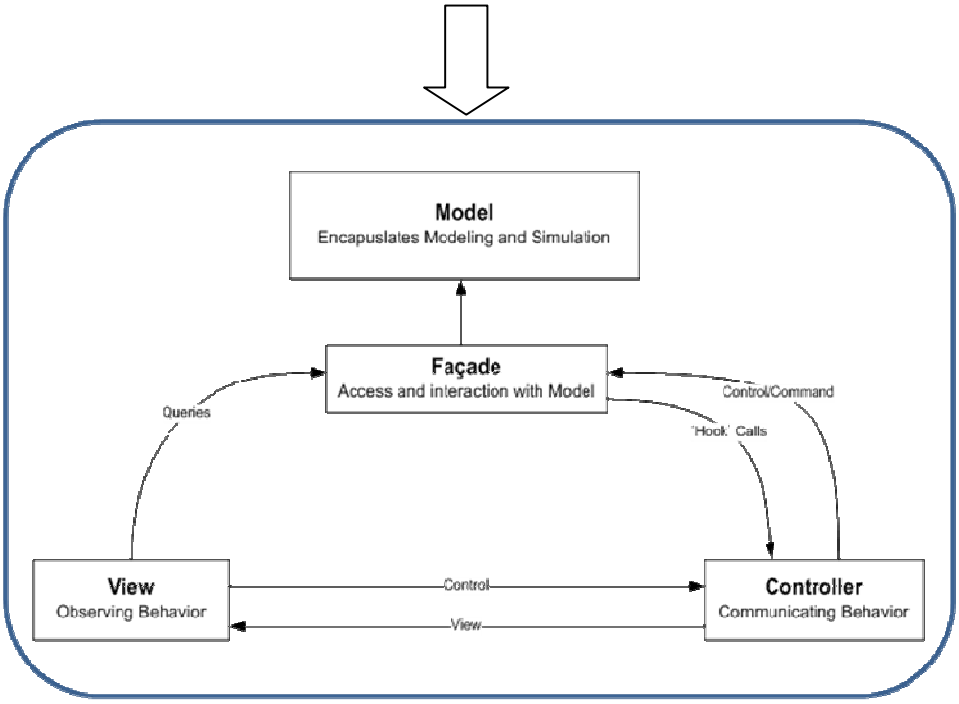
MODEL. As mentioned earlier, the Tracking Environment supports dynamic configuration for monitoring of simulation behavior which implies the View is allowed to retrieve selected data for itself.

By employing the FAÇADE layer into the traditional MVC, the software architecture gives important benefits to the design of the system (H.S. Sarjoughian & Singh, 2004; Singh & Sarjoughian, 2003). They include:

- Enhanced encapsulation
- Modularized development by layering technique.
- Reduced complexity of dependencies between components
- Improved weak coupling problem



(a) Traditional Software Architecture



(b) Software Architecture for the Tracking Environment

Figure 5. Software architecture concept

As shown in Figure 6, the Tracking Environment allows users to select the simulation data sets to be tracked such as state variables and input/output events for any number of atomic and coupled models. Thereafter, during the execution of simulation, the Tracking Environment provides two internal frames, the Tracking Log and the Console, to track and monitor the selected simulation data sets. The output of tracked

simulation data sets with its state is displayed in a tabular format on the Tracking Log frame in addition to the Console frame which records received and sent events.

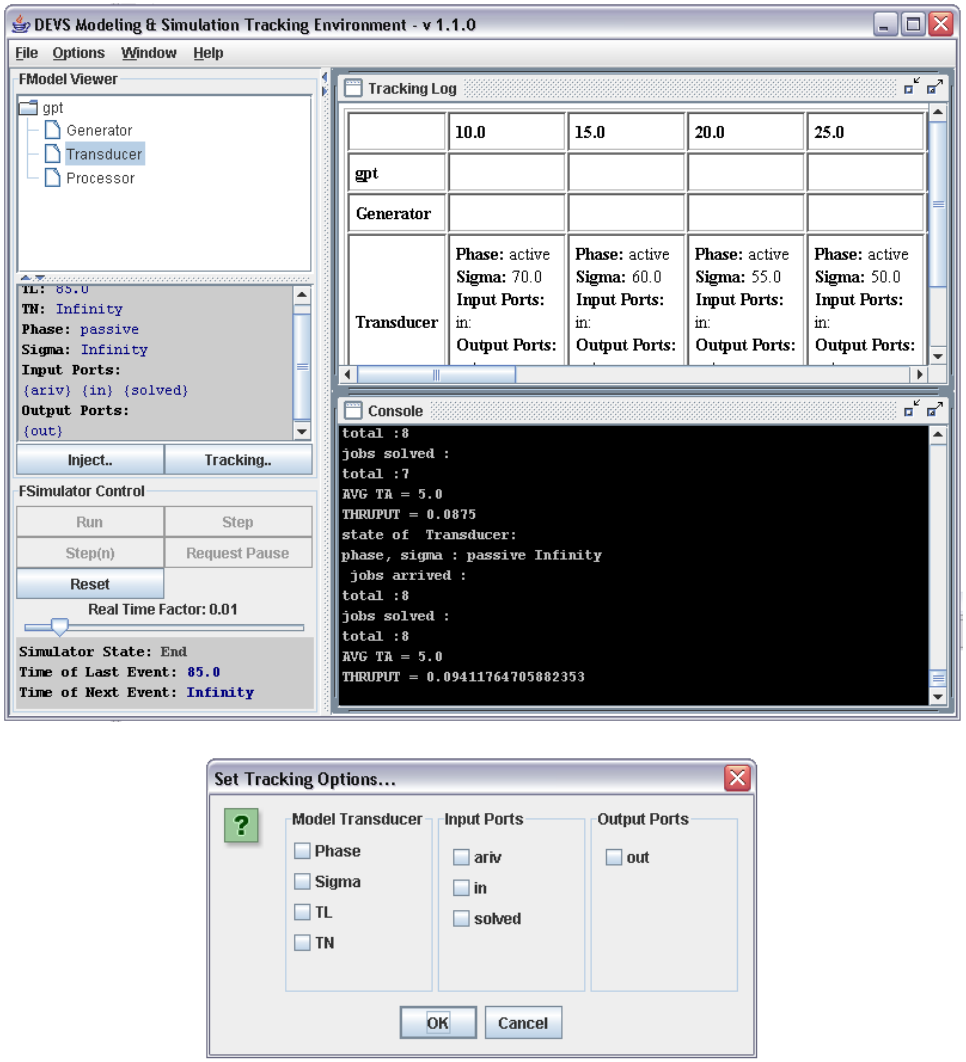


Figure 6. Simulation Tracking Environment

On the left side of the controller, the Tracking Environment provides the FModel Viewer which displays the hierarchy of the simulation model so that users are easily able to understand the structure of the simulation model and help them to select which atomic and coupled models to track and monitor. In general, the simulation data sets to track

need not always to be the same for all models. Hence the tracking options shown in Figure 6, which are for the Transducer atomic model in the GPT coupled model, will be changed when we select another model on the FModel Viewer.

In addition, the Tracking Environment provides a user convenient option called the Real Time Factor which can adjust the scale of simulation logic time in order to get a faster/slower or even soft real-time response, as shown in Figure 6 at the bottom left of the Tracking Environment. For example, when a user adjusts the scale of Real Time Factor as 1, the logic time of the simulator in the Tracking Environment is corresponding to 1 second in the real-time and the simulation is executed under the soft real-time condition.

2.3.2. TimeView

The TimeView, a separate un-timed viewer of data, was designed and implemented by Robert Flasher as part of his undergraduate senior project in the Computer Science and Engineering department at Arizona State University. It supports plotting data sets along the time axis. Source data can be input and output and state changes from atomic or coupled DEVS models. At runtime, data can be fed into the TimeView for plotting. For example, default and user defined data variables such as size of a queue can be automatically plotted as the time trajectory charts until the end of the simulation (see Figure 6). Therefore, users can monitor and track atomic models' input/output and state changes during simulation. Currently the TimeView only accepts the primitive data types (e.g., double and string) for an event to be displayed on the time trajectory chart.

The current version of the TimeView increments each trajectory by a predefined time periods, for example, time is incremented by 10 units of time as shown in Figure 7 and then plots the simulation data sets at the time instances events are received. This environment is similar to an oscilloscope and allows users the flexibility to adjust time period scale for every simulation run. The length of a trajectory that is viewable can also be specified.

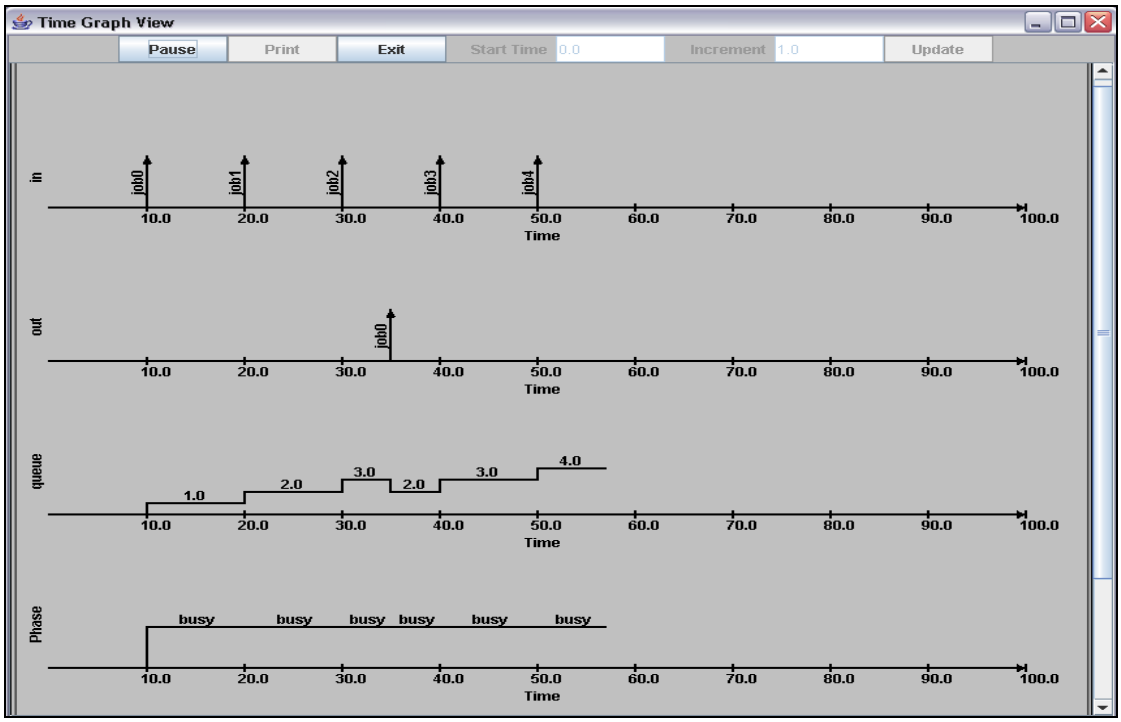


Figure 7. The TimeView UI

2.3.3. Integration of TimeView into the Tracking Environment

2.3.3.1. Tracking options. As stated above, the TimeView does not have the concept of time and nor the capability of control. The TimeView is a simple tool to display a series of (x, y) values on the trend charts like an oscilloscope. Given the limitations, the TimeView is integrated into the Tracking Environment so that its

controller of the Tracking Environment can update the TimeView graph by synchronizing with the simulation time. Currently, the simulation data sets which are tracked and monitored by the Tracking Environment are the primitive data type including the String. Then the data types in the TimeView should also be consistent with the Tracking Environment. The user has the flexibility to select view options for any number of atomic/coupled models as well as the unit of each tracking data and X-axis and increment of X-axis, as shown in Figure 8. The TimeView can be invoked for each atomic/coupled model independently with selected simulation data sets.

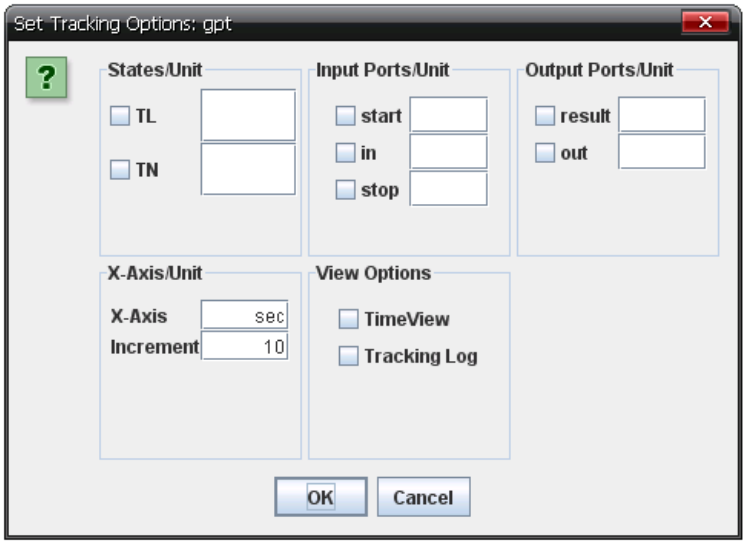


Figure 8. The Tracking options

2.3.3.2. Data flow chart. After selecting one or both of the tracking view options (i.e., TimeView and TrackingLog), the Tracking Environment assigns a tracker for the simulation data sets to be tracked for atomic and coupled models. Figure 9 shows data flow of the DEVS Tracking Environment. Logically, whenever an input/output event occurs during the simulation execution, *ModelTrackingComponent* Class loops through

the tracker to get simulation data sets for the selected models. Originally the *ModelTrackingComponent* Class contains the method to get simulation data sets from the tracker as well as to construct the Tracking Log.

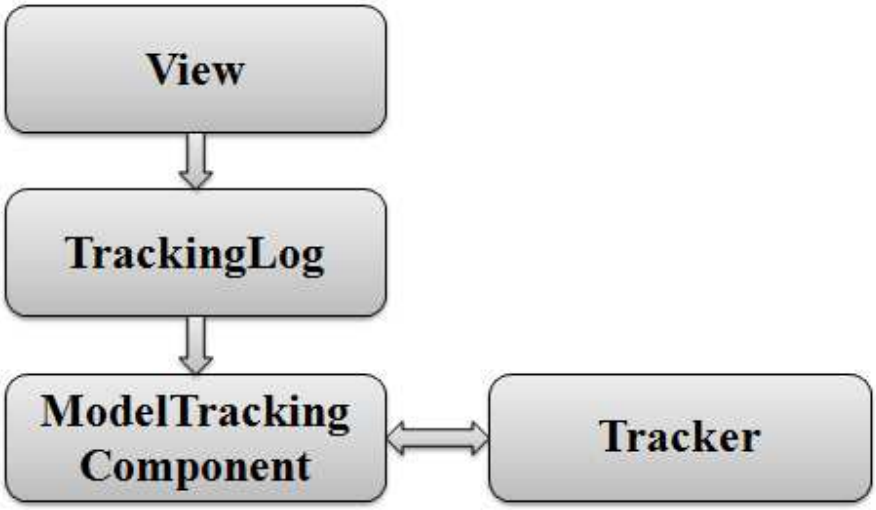


Figure 9. Data flow of the DEVS Tracking Environment

To integrate TimeView into the DEVS Tracking Environment, *TrackingControl* Class is employed as an intermediate Class between *TrackingLog* and *ModelTrackingComponents* Classes (see Figure 10). Then, the logic to get simulation data sets for selected models is moved from *ModelTrackingComponent* Class into the *TrackingControl* Class which sends the data sets to TimeView or/and Tracking Log for runtime viewing as shown in Figure 10. Therefore, the role of *ModelTrackingComponents* Class is limited to construct the Tracking Log only.

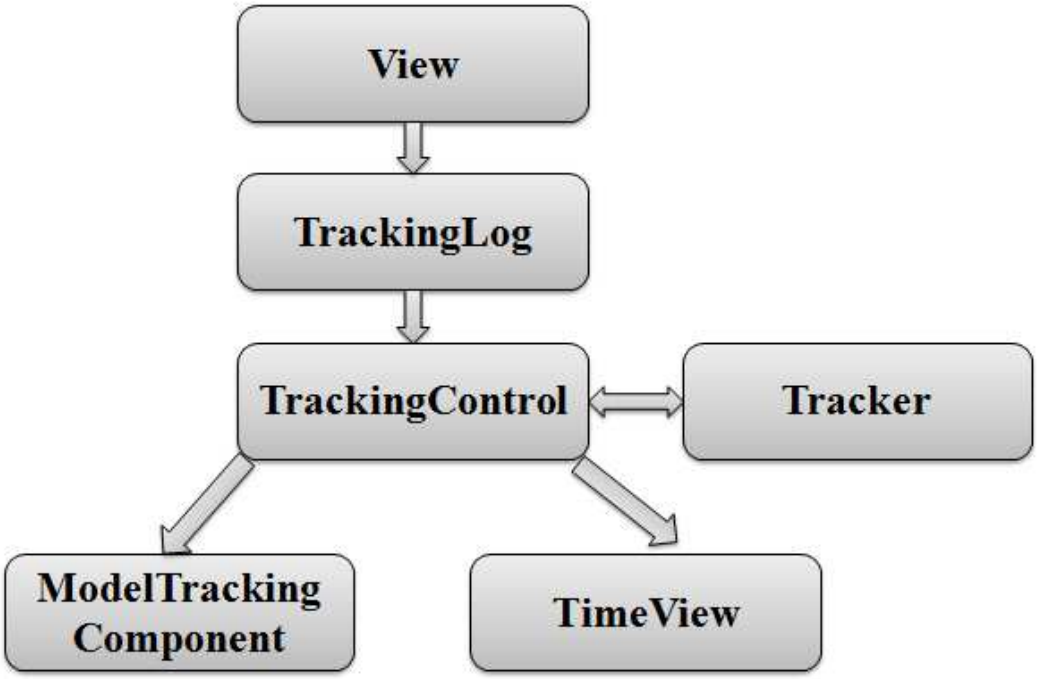


Figure 10. Integration with the TimeView

2.4. Service Oriented Architecture based Software System

2.4.1. Service Oriented Architecture

SOA is a software architecture style aimed at dynamic discovery and use of services over a network. Before understanding SOA, we should understand the definition of each component in SOA clearly. Figure 11 show the conceptual model of SOA. Service provider provides services that may be used by other services. It can publish its service interface and access information to the service broker using Web Services Description Language (WSDL) (WSDL, 2001). A service can be described as (Chen & Tsai, 2008),

- An interface between the producer and the consumer.

- A service is well-defined, self-contained, and does not depend on the context or state of other service.
- Newly developed modules or just wrapped around existing legacy software to give them new interfaces.
- A service is a unit of work done by a service provider to achieve desired end results for a consumer.
- Provides loosely coupled Application Programming Interface (API), with standard interface, so that it can be discovered and called (invoked) by another service.

The services can communicate with one another by exchanging messages. WSDL is an XML based language for describing Web services and how to access them. It includes the location of the service and the methods (called endpoints) that are exposed for other services to use. The service broker is a service repository and registry that stores information about the published services. A common implementation of service broker is the Universal Description, Discovery, and Integration (UDDI) developed by OASIS (OASIS, 2003). A proposed ideal features that a service broker should have are (Chen & Tsai, 2008),

- Service registry
- Service repository
- Service specification and requirement
- Application templates
- GUI templates

- Collaboration protocols and templates
- Policies
- Database and ontology
- Integrated testing and evaluation tools
- Quality of service

Service client lookups the service broker to search a desirable service by a key word or service name defined using WSDL. If a service is found, the service broker sends the service information stored in the repository back to the service client, then binding to service provider to invoke one of its operations available in the service using Simple Object Access Protocol (SOAP, 2003). SOAP is a XML based protocol to allow communication between SOA-based applications on different operation systems, technologies, and programming languages over HTTP (W3C, 2007).

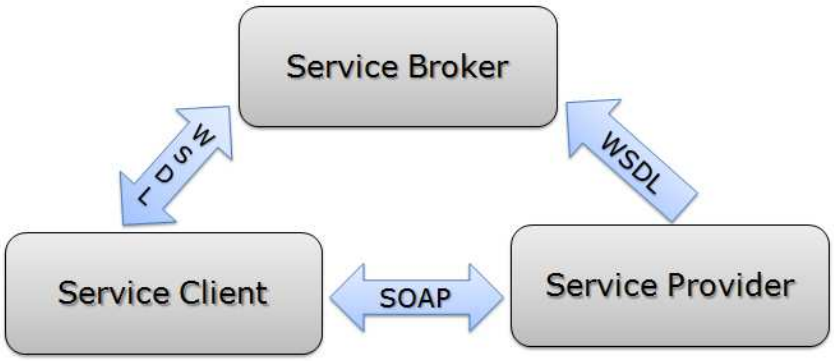


Figure 11. Service oriented architecture

SOA enables service clients to be separated from service providers. Therefore, service clients do no need to know how the services are implemented, which platforms are used, or how they are distributed. It leverages the reusability of services. One of the

most important desired advantages of SOA is to rapidly compose distributed services. For the service composition, two methods may be used. They are

- Orchestration: A central process which can be a service itself takes control over all involved services and coordinates the execution of different operations. BPEL (Business Process Execution Language) supports orchestration.
- Choreography: There is no central coordinator. Each service involved can communicate with any partners. WS-CDL (Web Services Choreography Description Language) is a composition language that supports choreography

2.4.2. Adaptable Service Based Software System

As stated previously, SOA enables dynamic composition of various types of services for distributed applications built on different platforms. Dynamic binding of services makes SBS more flexible and that is one of the most important advantages of using SOA. However, dynamic service composition requires the development of SBS with multiple QoS such as timeliness, throughput, accuracy, security, dependability, survivability, and availability. Service-based software systems need satisfy multiple QoS simultaneously and thus tradeoff among the QoS features is necessary. However, currently we do not have comprehensive understanding of these tradeoffs and relationships so that it is a challenge to satisfactorily manage multiple QoS features simultaneously. To overcome this, as shown in Figure 1, QoS Monitoring and Adaptation sub-systems may be used to collect data concerning QoS which can be analyzed to adapt the composition of services accordingly.

2.4.2.1. Four critical QoS features. Since we cannot study the tradeoffs of all QoS features due to limitation of time and resources, four QoS (i.e., timeliness, throughput, accuracy, and security) are important to be considered (Yau et al, 2008). However, in this work, security QoS feature is not considered.

A set of experiments are devised to collect necessary data to develop a design approach for developing Adaptive Service-Based Software Based Systems. Two types of atomic services, communication intensive (Voice Communication) and computation intensive (Motion Detection) models, are developed (Yau et al, 2008). Voice Communication System (VCS) provides voice streaming service to multiple users simultaneously. For experimenting with this system, sampling rate, number of clients and buffer size are varied. Motion Detection System can also be considered. The Motion Detection (MD) service provides the rate of motion detected for a certain time period. Several motion detection algorithms are used to calculate the user request rates. The composite service can be constructed by combining these atomic services. However, a simple composite simulation service called Travel Agency service, which provides the closest resort place by zip code, is developed. The VC and Travel Agency services are used together to model and simulate a composite service. The MD service is not used since it is under development.

Table 1

Four QoS Metrics Table

QoS Features	Metrics	Experimental Data
Accuracy	Loss Rate	The number of bits lost between two nodes after transmission
	Error Rate	The frequency of erroneous bits between two nodes after transmission
Timeliness	Response Time	The difference between the time of submitting a service request and the time of receiving a service confirmation
	Service Delay	The difference between the time of submitting a service request and the time of receiving the service result
	Jitter	Variation of delay generated by the transmission equipment
Throughput	Data rate	The rate in which data are encoded
	Bandwidth	The data transfer rate measured in bits per second
Security	Security Rating	Initial security configuration Security events detected in runtime

2.5. Related Works

A SOA-based framework using High Level Architecture (HLA) Infrastructure (HLA, 1999) has been proposed to develop and evaluate SOA-based network centric and system-of-systems applications using Process Specification and Modeling Language (PSML) (Tsai, Chun et al., 2006). From existing SOA services, composite services can be

synthesized and executable code generated for the actual application and simulated for testing purposes. The services are geographically distributed and interconnected as web services. The DEVS and PSML models have basic differences such as explicit representation of time, event preemption, and closure under coupling of model components. Another important difference is the mapping from DEVS and PSML to SOA. SOAD is defined in terms of the basic SOA elements (service client, service provider, and service broker) as well as the primitive and composite service composition. More generally, SOAD is grounded in system-theoretic modeling and simulation concepts whereas PSML is based on software modeling targeted for service-based computing systems (WinterSim, 2004).

Some other tools are also proposed to support simulation of SOA-based software systems. A UML simulator is proposed to define interaction among web service by a UML model (Hiroyuki et al., 2006). By using Active Hyper-graph, it supports execution of the extended UML model called BPEL/UML which can support mapping elements of BPEL4WS document onto elements of UML active diagram and WSDL onto elements of UML class diagram. The interfaces of services are defined in order to validate interaction between BPEL/UML models and BPEL/UML models with real services. The Petri Net formalism is used to provide decision procedures for web service simulation, verification, and composition (Srini & Sheila, 2003). By using the DAML-S description of a Web service that is translated in situation calculus, KarmaSim simulator (Srinivas, 1999) automatically generates the Petri Nets in order to perform the desired analysis. These

UML simulator and Petri Net are focusing on supporting workflow design, rather than the individual component.

There are other approaches to web service composition in terms of QoS properties. One research is focusing on the relationship between service chain complexity and QoS for the user (Anderson et al., 2005). Agent based approach is used to model a set of end-users that request service invocations through the network. Users use a catalog that provides the name of a server where a requested service is located to find that service. The TouchGraph library (TouchGraph) is extended and used as a JAVA visualization tool to model service chaining, visualize network traffic and quantify service chain complexity. Simulation based Web service composition based on their QoS properties, such as performance, reliability, and availability, is proposed (Chang et al., 2005). Users can specify the service composition with QoS concerns by using the proposed Web Process Composer. Simulation is performed based on user composition and the simulation results are sent to the QoS Monitor to analyze and evaluate QoS of the web process. The evaluation results are feedback to the Web Process Composer to repeat the simulation until the desired QoS is achieved. This approach is somewhat similar to our ASBS approaches in terms of monitoring and adaptation capabilities. However, the ASBS consider multiple QoS features and their relationship (i.e., their satisfaction tradeoffs) with both hardware and software aspects rather than simple QoS features of web process. A performance engineering method for service composition is proposed (John et al., 2006). This approach is to apply performance test-bed generation techniques to software system based on SOA. Service composition can be described at a high level

using Business Process Modeling Notation (BPMN) or own ViTABaL-WS Web service composition notation. These high level service compositions can be extended with a lower level service composition model at the detailed service interface level in MaramaMTE which is JAVA based performance test-bed generation tool. The test-beds are executed for the service composition and results are provided to the engineer.

Table 2 shows the comparisons between the approaches briefly reviewed in relation to SOAD. The explicit use of time (discrete values) in services is crucial in developing verifiably correct simulation models of dynamical real services (WinterSim, 2004). The dynamic simulation model with an explicit representation of real time can be used instead of a real service and the time based QoS features such as throughput for the service can be collected. In Table 2, the Petri Nets formalism support for representing time, but situation calculus description translated from the DAML-S ontology to the Petri Nets does not explicitly represent use of time. Moreover, compare to the proposed SOAD, currently none of approaches described above can support dynamic changes of service composition and no concept for the separation of modeling SOA-based software system in terms of hardware and software are presented.

Table 2

Comparisons of Approaches in Terms of M&S Concept

Approach	Formalism	Components	Timing	Hierarchy	Seq. / Parallel
Service Chain	Y	Y	Y	-	Y/Y
MaramaMTE	Y	Y	-	Y	Y/Y
Petri-Net	Y	Y	Y	Y	Y/Y
Activity Hypergraph	Y	Y	-	Y	Y/Y
PSML-S	Y	Y	Y	Y	Y/Y
SOAD	Y	Y	Y	Y	Y/Y

3. Extension of Tracking Environment with SimView

This chapter describes the integration process of the SimView into the DTE. Although these two simulation tools are built on the same DEVS formalism, the objectives of the simulation environments are different from each other and the integration of them into one environment is required to incorporate with Monitoring and Adaptation capabilities in our ASBS. In this chapter, in order to integrate two simulation environments into one consolidated simulation environment, we describe the decision process of selecting an architectural design pattern, a type of simulation model, and interface of the new simulation environment.

3.1. Analysis on the SimView and DTE

The brief descriptions of SimView and DTE are placed in Chapter 2. In this section, we need to analyze more details of these simulation environments in terms of architecture design pattern, simulation model types, mechanism to load a model, and simulation control logic in order to make a right decision while integration process. For the purpose of validating our selection, the comparison between SimView and DTE for each category described above is performed and provided below.

3.1.1. Architectural Design Pattern

As described in the Chapter 2, the DTE was developed based on the traditional software architectural pattern called Model-View-Controller (MVC) as shown in the Figure 12. The traditional definition of each component is described below (Wikipedia).

- Model: The domain-specific representation of the information on which the application operates. Domain logic, DEVS formalism for the case of our

simulation, adds meaning to raw data. In our simulation, we have a set of well defined JAVA based APIs to represent these models (ex, atomic model and coupled model).

- View: The view renders the contents of a model. Multiple views can exist for an application.
- Controller: Processes and responds to events, typically user actions, and may invoke changes on the model.

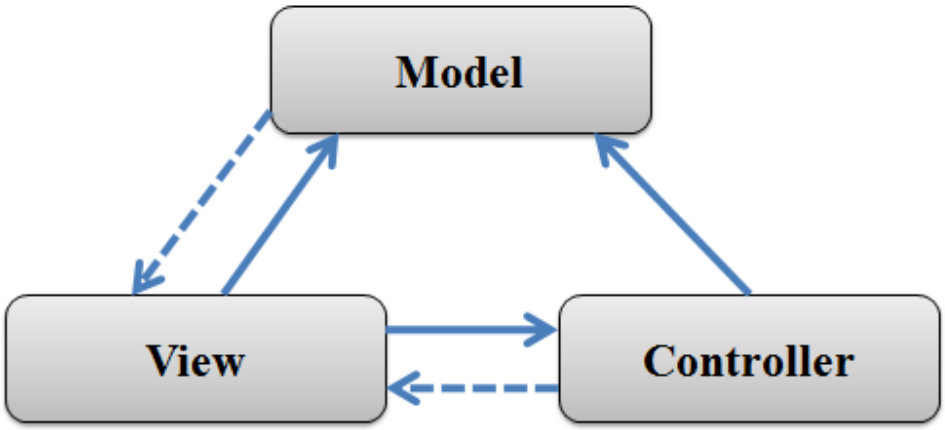


Figure 12. Model-View-Controller pattern

As shown in Figure 12, the solid lines indicate a direct association and the dashed lines indicate an indirect association. The separation of model and view allows users to create multiple views for the same model and increase reusability of models. This is one of the main reasons why we adopt the FACADE design pattern later. It also is easier for the developer to implement and maintain models for the application.

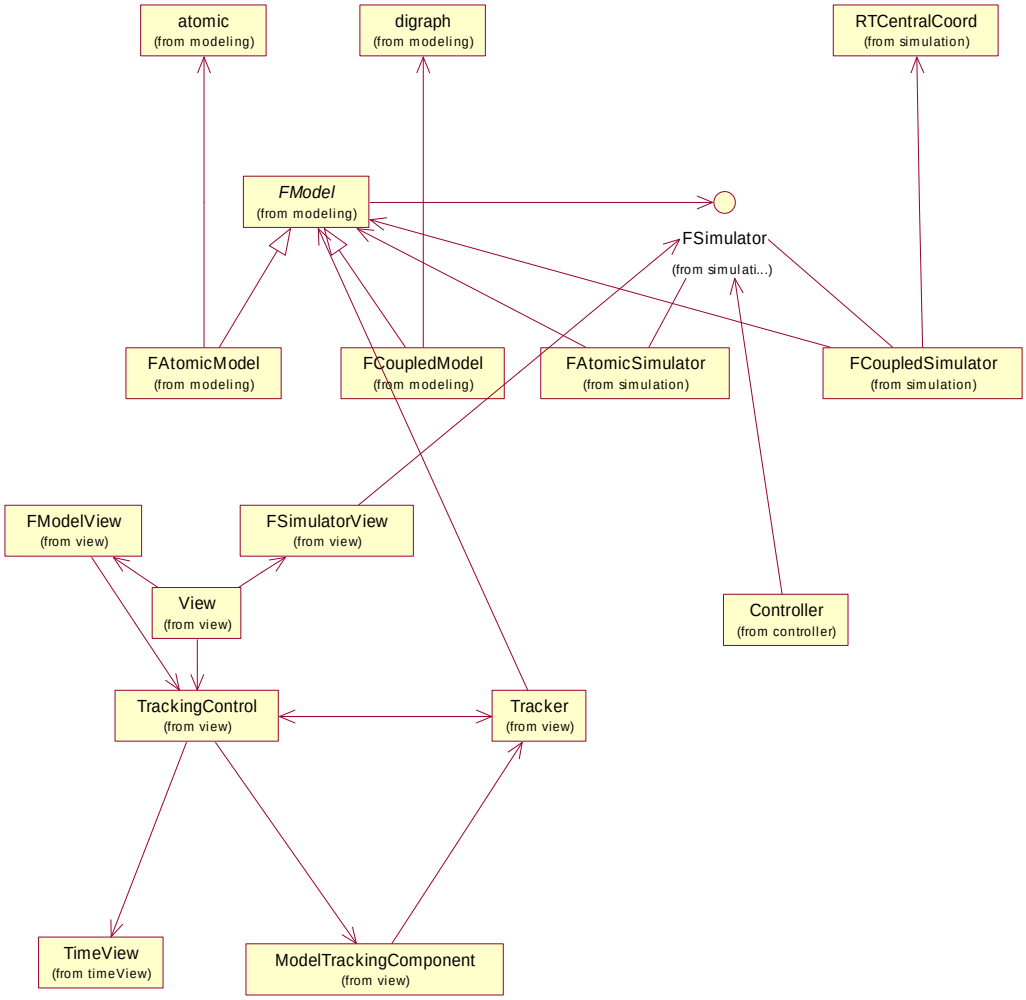


Figure 13. MFVC framework for the Tracking Environment

As discussed in the Chapter 2, based on the MVC framework, the DTE adopts the FAÇADE layer between Model layer and View-Controller layer. This is because DTE has multiple view options and we want to control and synchronize these view options by using one controller as well as other advantages described in the Chapter 2. The MVC framework with FACADE layer is referred as MFVC (Model-Façade-View-Controller).

In the traditional MVC paradigm, the simulation data sets displayed on the View are obtained directly from the simulation model which means the View is also allowed to interact with the Model. However, as shown in Figure 13, the Façade layer can only access to the Model and get a single set of simulation data to store. As mentioned earlier, the DTE supports dynamic configuration for monitoring of simulation behavior which implies the View is allowed to retrieve selected data for itself by getting data from the Façade layer. In addition, as a result of integrating the TimeView into the DTE, there are currently two view options, TimeView (*TimeView* class) and Tracking Log (*ModelTrackingComponent* class).

Unlike the DTE architecture which has a solid architectural software design pattern, the SimView is not constructed by using a MVC design pattern. View and Control are integrated onto the one JAVA file (*SimView.java*) so that it is hard to maintain and update the software when there are modifications. Figure 14 displays the simple UML diagram to show how the *SimView.java* file is implemented as one application. We can easily recognize that all components are strongly tight and depending on each other so that it is not a good approach to build a robust software system in terms of modularity, reusability, and complexity of the software. Consequently, we decided to take the architectural approach of the DTE which is the MVC framework adapting FAÇADE layer and use the SimView as one of simulation view options that users can select in the DTE like existing TimeView and Tracking Log.

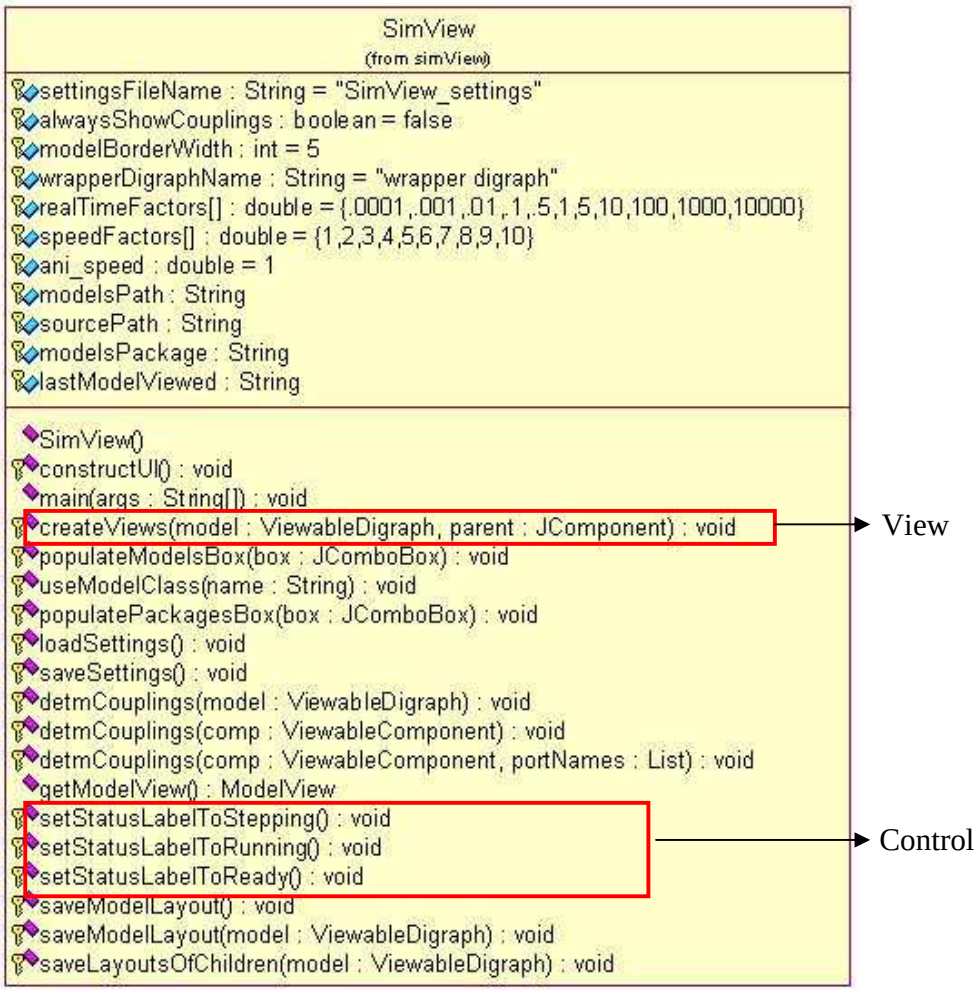


Figure 14. UML diagram for the SimView

3.1.2. Simulation Model Type

As we stated in the Chapter 2, the objectives of these simulation environments are different from each other. SimView provides animation of simulation models and enables the modelers to specify models directly in the DEVS terms. On the other hand, the DTE which is built on the DEVSJAVA Simulation Environment without visualization parts provides visual user specified data selection and automated simulation data gathering along with the trend chart capability. Therefore, they require separate simulation model

types for execution, for example, *ViewableAtomic* and *ViewableDigraph* models for the SimView and *atomic* and *digraph* models for the DTE. Figure 15 shows hierarchy of these simulation models in the DEVSJAVA.

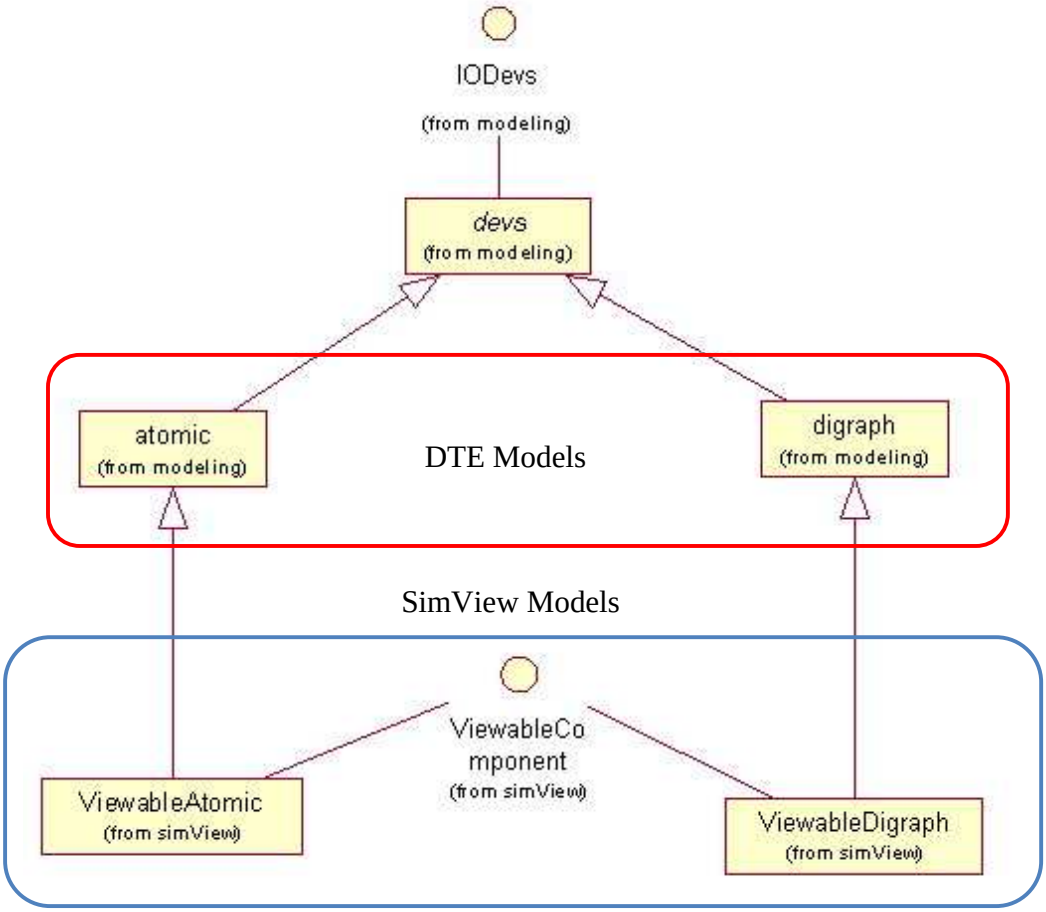


Figure 15. The hierarchy of simulation models

As we can see, both models are built based on the DEVS formalism. Unlike simulation models for the DTE, SimView provides a view of the arbitrary levels of coupled model using boxes-within-boxes-style and the animation of messages moving along the paths of the coupling between components during the simulation (Mather, 2003). Therefore, *ViewableAtomic* and *ViewableDigraph* are extended from the basic

DEVS model, *atomic* and *digraph* model, to support these capabilities. As a result, *ViewableAtomic* and *ViewableDigraph* models are adopted as default simulation models for the new simulation environment since they can provide not only behaviors and input/output data of the simulation model for the Tracking Environment, but also animation of the simulation model for the SimView without any modification. The Viewable models need to incorporate with the FAÇADE layer adopted as the architectural design pattern in the section 3.1.1. More details are provided later section.

3.1.3. Model Loading Mechanism

Two types of model loading mechanisms are used for the existing simulation environments. The first method is used by DTE, as shown in Figure 16.

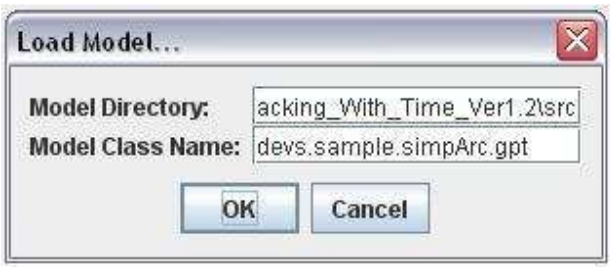


Figure 16. Tracking Environment loading mechanism

In this method, a user must specify the name of model root directory and the path to the model class from that directory. Typically a user does not change the model directory often, the main problem of this approach is that a user must specify the entire path to the model whenever a different model is loaded into the simulation environment. Moreover, if a model is located in the different folder or different level of the folder structure, it is hard for a user to specify the entire path to the model at once.

SimView uses the second mechanism currently as shown in Figure 17. A user must configure the path to packages of model classes and source files as well as model package names. After the configuration, a user must select a package name at the top of the SimView as rounded with red line, and then SimView will automatically display the list of available models in the selected package for model selection on the right scroll box.

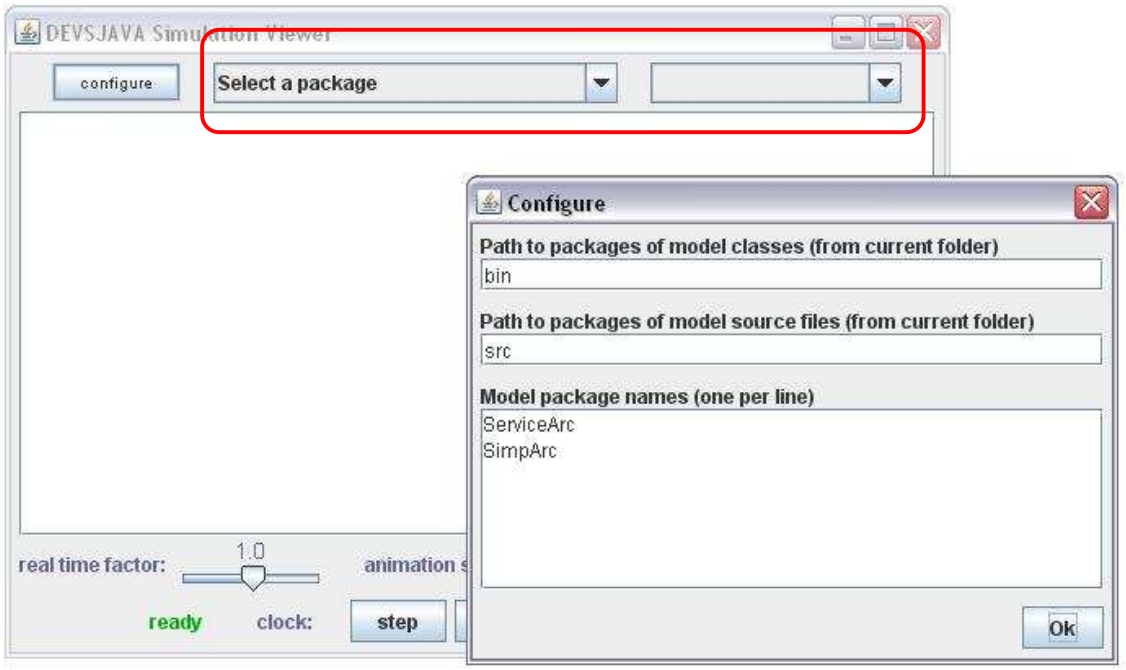


Figure 17. SimView loading mechanism

Consequently, the second mechanism is more convenient and logical for users to select a model to be simulated since it provides automation of the displaying the list of available models in the package. In addition, at the level of model selection, a user will have the option to choose view options, such as SimView and/or DTE, to be displayed on the consolidated environment.

3.1.4. Simulation Control Logics

Two simulation environments use the same type of control logic for the simulation. However, there are some slight differences in the purpose of each simulation environment. Figure 18 display the list of controls each simulation environment provides and how they are differ from each other.

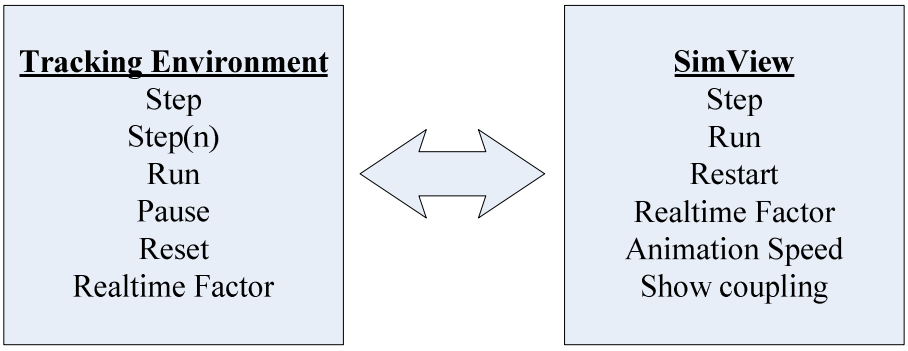


Figure 18. The list of Control Logics in the simulation environment

Step(n) and Pause controls are presented in the DTE, but not in the SimView. Alternatively, Animation Speed, which controls the speed of message moving, and Show coupling, which shows coupling between models, controls are only specialized for animation capabilities of the SimView. For the case of integration control logics, the new integrated simulation environment must support both capabilities of DTE and SimView so that all control logics including those specialized control logics must be presented.

3.2. Integration of SimView into DTE

In the Chapter 3.1, the details on the SimView and DTE are analyzed in terms of the architectural design pattern, simulation model type, model loading mechanism, and simulation control logic. Based on the analysis performed in that section, the integration

process, as well as the final form of the consolidated simulation environment, is presented in this section. As analyzed in section 3.1, the DTE is built on the robust software architecture pattern called MVC design pattern. Therefore, the DTE becomes the base architecture of the new simulation environment. Subsequently the SimView is used as one of view options in the DTE since the DTE has adopted the FAÇADE layer to control multiple view options by a universal controller. In addition, as discussed in the section 3.1.2, *ViewableAtomic* and *ViewableDigraph* models become basic simulation models for the DEVS-Suite.

3.2.1. Interface Integration

Since the model loading mechanism of the SimView provides more convenience and automation of displaying available models to the user, that mechanism adopted into the DEVS-Suite. In addition, for the user convenience, the DEVS-Suite provides user flexibility in that a user can select view options, SimView and/or Tracking at the level of model selection as shown in Figure 19. On the other hand, as discussed in Chapter 2, the user can select TimeView and/or Tracking Log at the level of model tracking option. A user must configure a path to the source packages and names of the packages which contain the model the user want to load. Consequently, after a user select a package, the DEVS-Suite searches available and validated simulation models in that selected package to display for selection by the user. Figure 20 displays the sequence diagram showing how the DEVS-Suite works for loading a model.

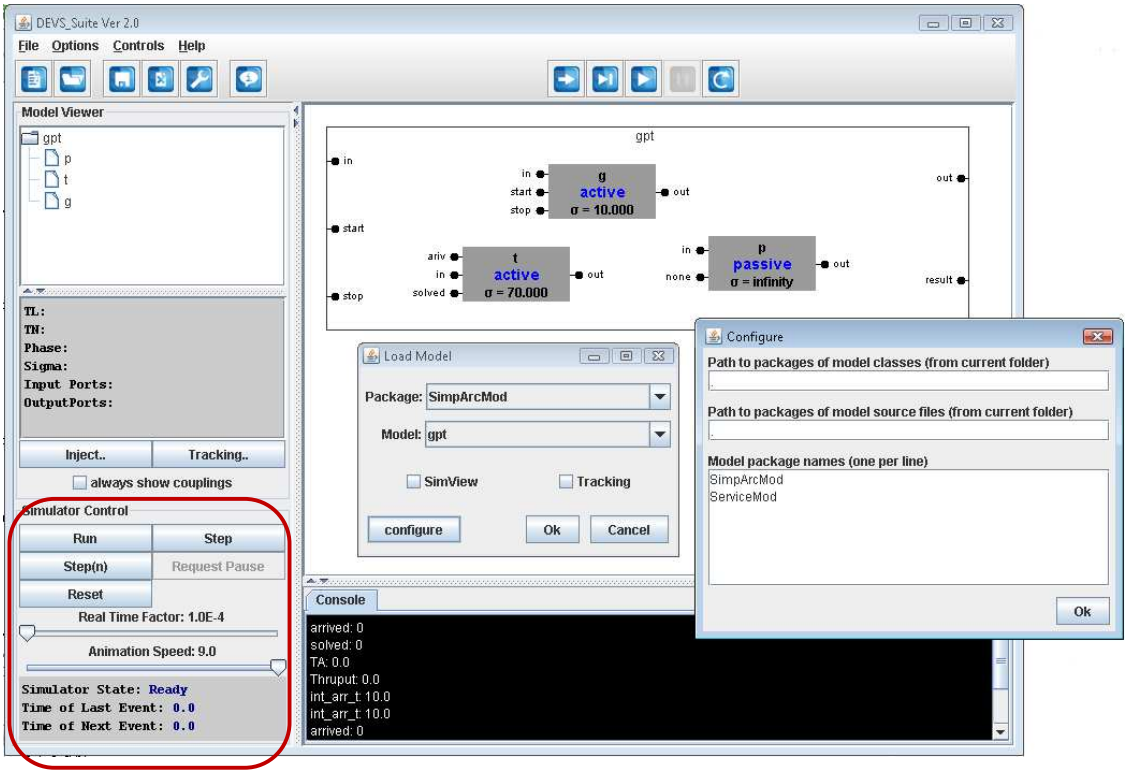


Figure 19. Interface of the DEVS-Suite

Since the SimView is integrated into the Tracking Environment, the specialized control logics for the SimView such as Animation Speed and Show coupling must be presented in the DEVS-Suite controller as shown in the red circled area in Figure 19. Figure 19 also displays the SimView with the *GPT* model is loaded onto the DEVS-Suite.

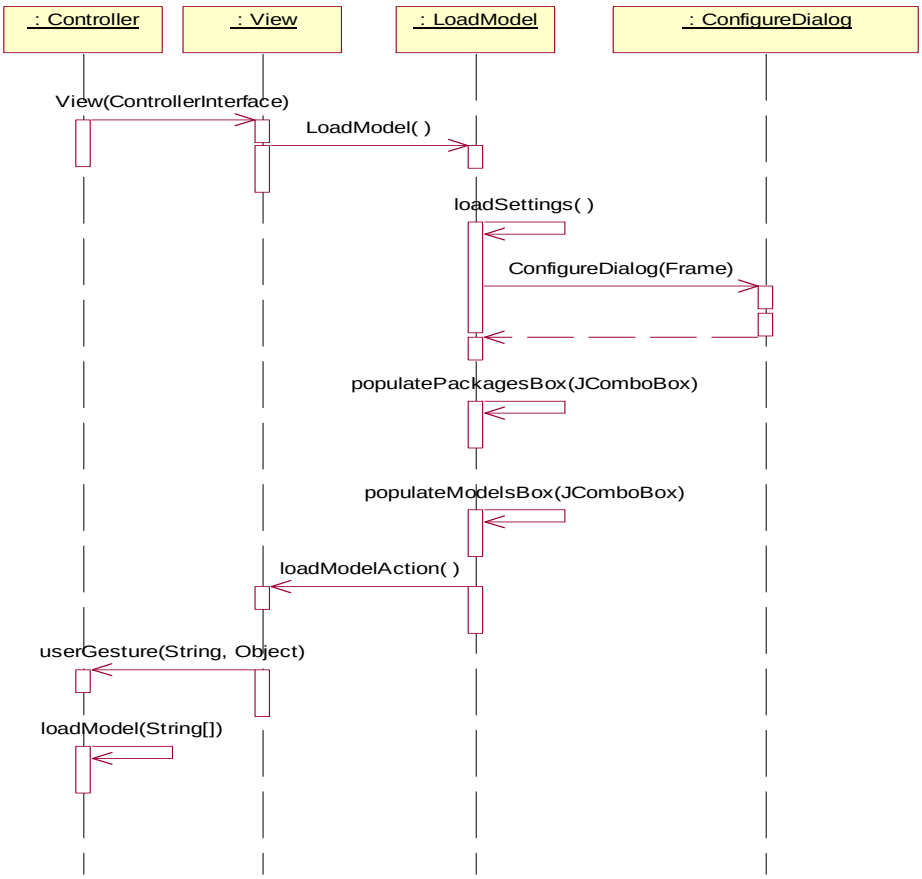


Figure 20. The sequence diagram for model loading mechanism

3.2.2. Architecture Integration

Figure 21 shows the new MFVC framework for the DEVS-Suite. This class diagram contains only important classes for the purpose of simplification. The entire class hierarchy diagram is provided at the end of this chapter. Basically the implementation of FACADE layer does not have any changes, but the connection to the Model is altered to Viewable models and simulators as shown below.

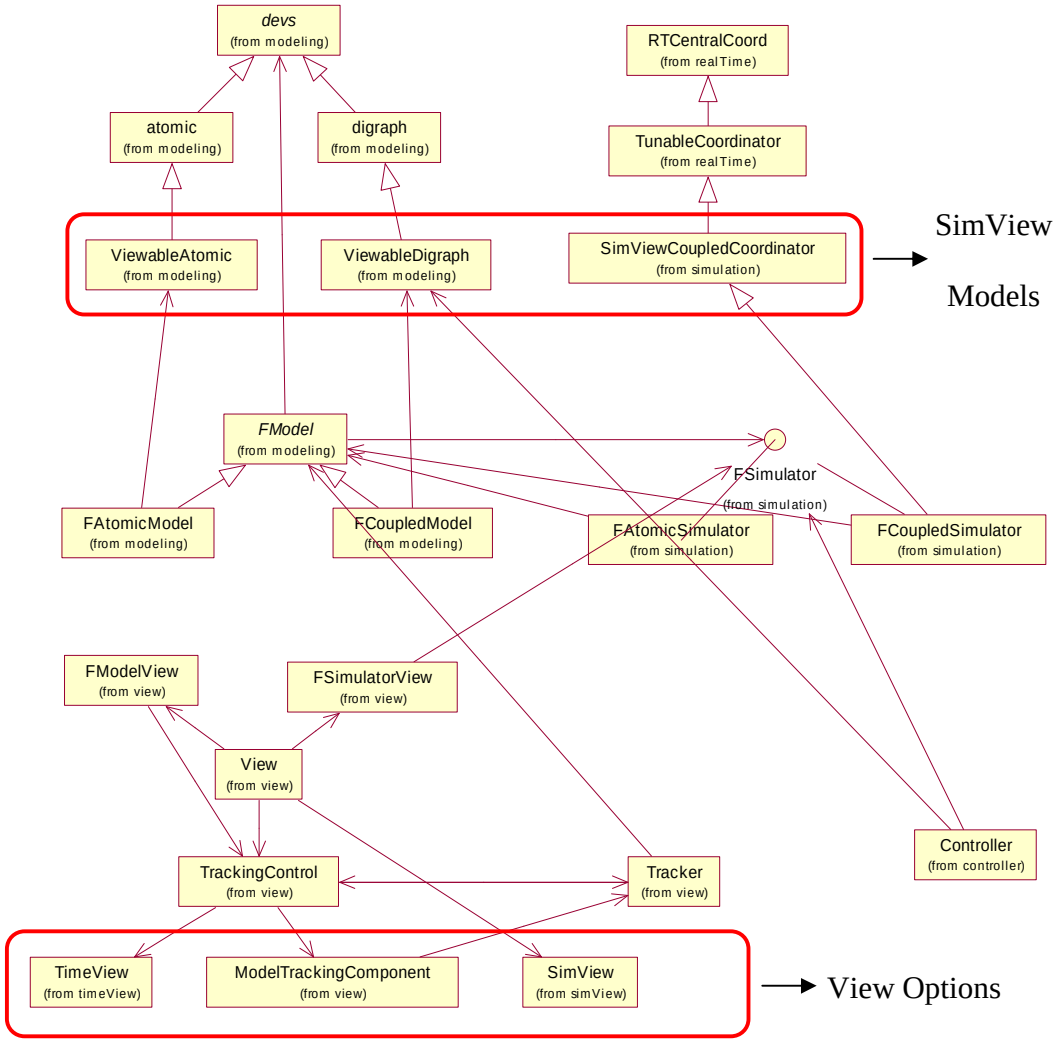


Figure 21. The MFVC framework for the DEVS-Suite

Since the Viewable models and simulators are extended from the original DTE models and simulators, semantically this connection is satisfied with the requirements for the DTE as well as the SimView itself.

Finally, the SimView classes are integrated into the Model as a view option for the DTE. As shown in Figure 21, at the *View* class, the simulation data getting from the

Façade layer are sent to two view option classes, *SimView* (SimView) and *Tracking Control* (Tracking), based on the user selection. Figure 22 displays the simplified MFVC class diagram for the DEVS-Suite. Classes are grouped into one of the following packages, Model, Façade, View, and Control and the interrelationships between these packages and classes are presented in Figure 23.

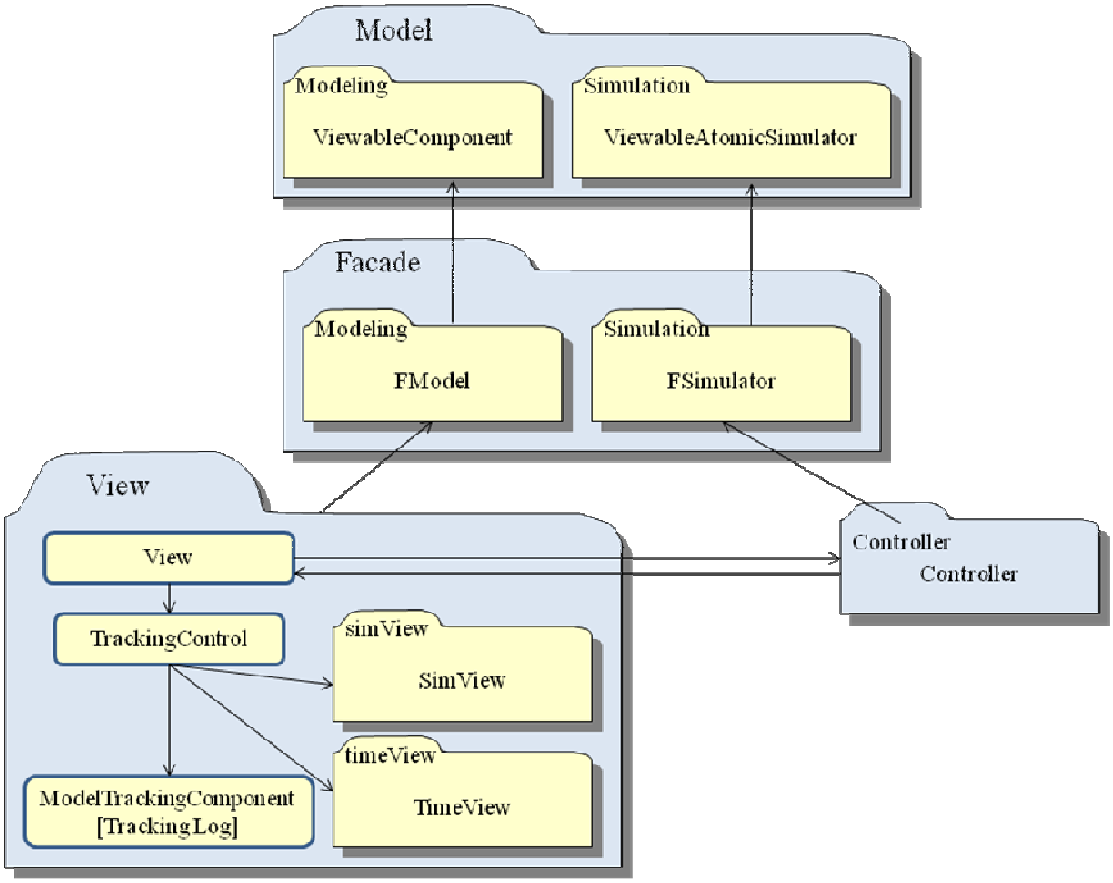


Figure 22. The simplified MFVC framework

In this chapter, Integration process of the SimView into the DTE is discussed for the purpose of supporting M&A capabilities in ASBS. The DEVS-Suite is now capable to provide simulation data so the Monitoring sub-system can analyze the service composition and adopt the control form the Adaptation sub-system which reflects the

dynamic binding of services by changing coupling between simulation models. Now the new simulation environment for the SOA-based simulation is ready. The next step is to develop a set of SOA-based simulation models to support desirable quality of service for ASBS.

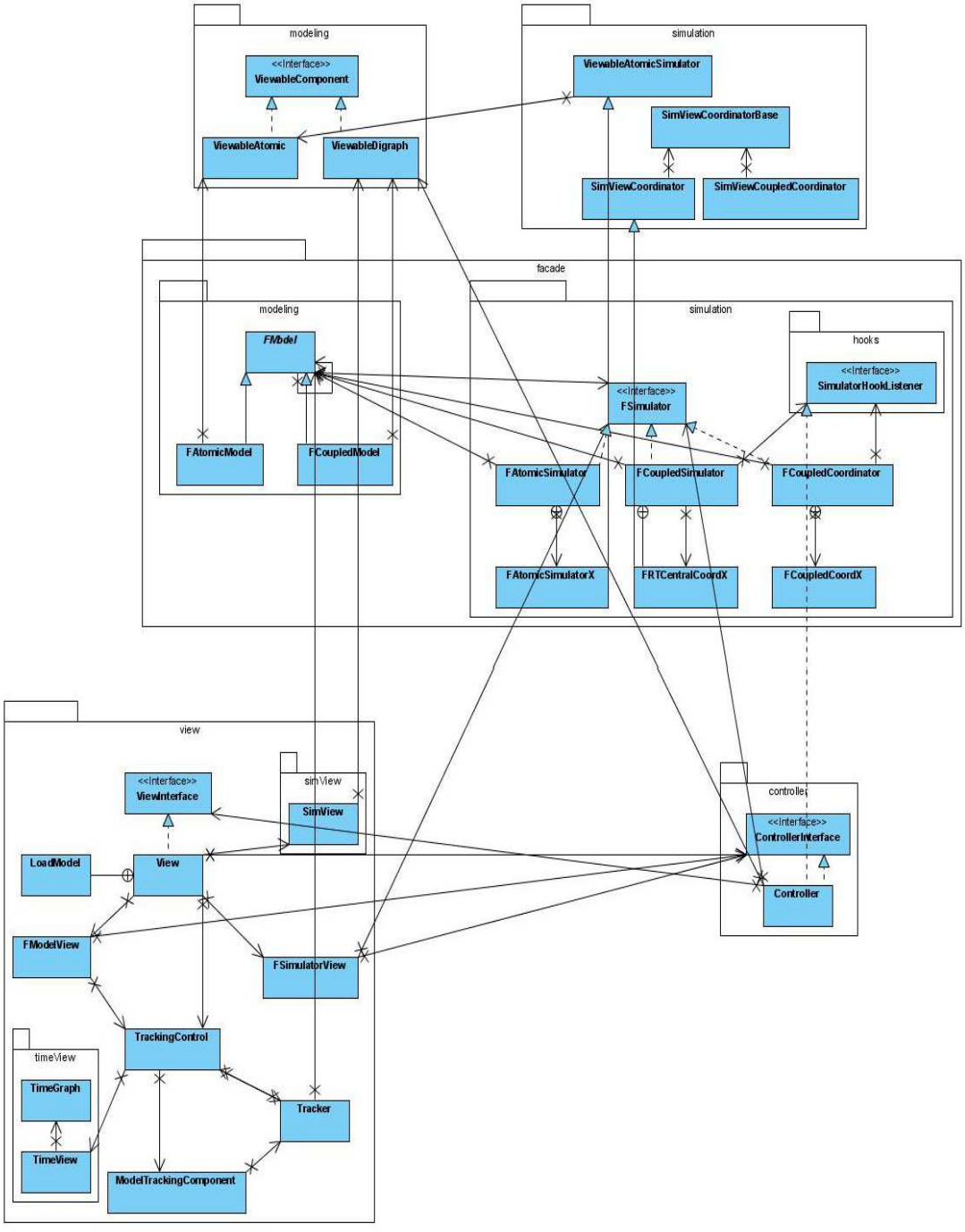


Figure 23. MFVC component/package specification for DEVS-Suite

4. DEVELOPMENT OF SOA BASED SIMULATION MODELS

This chapter describes the SOA DEVS (SOAD) approach for modeling and simulating service-based software systems. Generic SOA-based simulation models are developed for service broker, service provider, service client and service composition as well as a simple network for simulating computer network traffic. In addition, a set of transducer models are developed to automate collection of simulation data sets for service and network models.

4.1. SOAD Framework

To support simulation modeling of SOA-based software systems, our approach is to introduce SOA concept and capabilities into the DEVS framework (H. Sarjoughian et al., 2008). The extended DEVS framework with the SOA called SOA DEVS framework is developed in order to enable simulation based-design of service oriented computing. The approach provides a basis for verifying and validating the design of Monitoring and Adaptation sub-systems that conceptualized for Adaptive Service-based Software Systems. SOAD is designed and implemented using DEVS-Suite. In SOAD, both software and hardware components of service-based software systems are modeled. This is useful in order to model and simulate the role network (e.g., router) plays in the overall dynamics of system under consideration. As discussed in Chapter 2, there exists no simulator that is grounded in a system-theoretic modeling and simulation framework such as DEVS. By incorporating the SOA concept into the DEVS simulation models and accounting for hardware aspect of service-based software systems, SOA-compliant DEVS simulator can be developed (H. Sarjoughian et al., 2008).

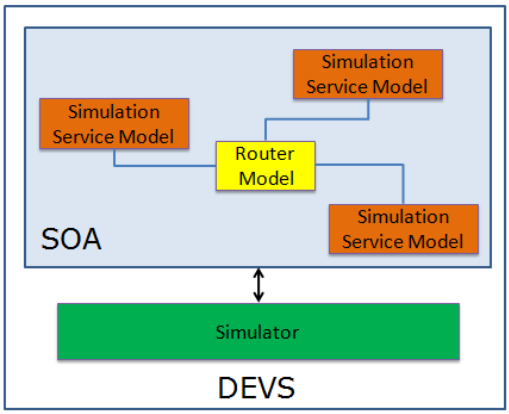


Figure 24. SOA-based DEVS Modeling Approach

4.1.1. Comparisons between the SOA and DEVS

Before extending the DEVS framework with the SOA concept and capabilities, it is important to compare these frameworks in terms of their concepts and capabilities since SOA and DEVS are used to develop real and simulated SBSs, respectively. The comparisons between the SOA and DEVS framework are described below and shown in Table 3.

Table 3

An Association between the DEVS and SOA Frameworks (H. Sarjoughian et al., 2008)

SOA	DEVS
autonomous	atomic and coupled models modularity
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

- The concept of autonomous services corresponds to the concept of modularity of atomic and coupled models. DEVS models are defined in terms of generic functions (δ_{ext}, δ_{int}, δ_{conf}, λ) and time (ta).
- 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 services promotes loose coupling of composite services. The functions of a service can be arbitrary defined. Atomic and coupled model components require state information which includes time t ($t \in S$) in order to allow synchronization of events produced and consumed. The time-based dynamics of DEVS model components has a central role in simulation.

Based on the analysis in Table 3, we can notice that one fundamental difference between the DEVS and SOA is the use of the broker concept. In the SOA, all services must publish its service to the broker service in order to be discovered and composed with other services. Therefore, the connection between service providers and service client is only established by the broker service only.

In DEVS, however, the broker concept is not accounted for and thus the DEVS atomic and coupled models are not SOA compliant even thought these models have important similarities to those of primitive and composite services. In fact, we can model a SOA-like software system by using the DEVS atomic model and coupled model and applying the concept of publish/subscribe ports and dynamic structure (Ramaswamy, 2008). However, this approach to modeling service-based software systems is not SOA-compliant since there is no model for the broker service.

4.1.2. Mapping SOA Elements to the DEVS Elements

As stated previously, the SOA elements have similarities and differences with those of DEVS. We need to map these SOA elements into the DEVS models in order to develop the SOAD simulator. Below, Table 4 shows the correspondences between the SOA and DEVS elements. The SOA-compliant DEVS framework is characterized in terms of primitive, composite, and broker services (H. Sarjoughian et al., 2008) which in this thesis are referred to as service provider, service client, and service broker, respectively.

Table 4

Correspondences between the DEVS and SOA Elements (H. Sarjoughian et al., 2008)

SOA Model Elements	SOAD Model Elements
services (service provider, service client, service broker)	atomic models (service provider, service client, service broker)
service description	entity (service-information)
messages	entity (service-lookup and service-call)
messaging framework	ports and couplings
service registry and discovery	executive model
composition of services	coupled models (service providers)

In Table 4, the service provider, service client, and service broker are mapped to DEVS atomic models. Similarly, composite service is mapped to a DEVS coupled model. In addition, the messages and their exchanges in the DEVS can be extended to represent service description and messages. DEVS model communications via messages, ports, and coupling can be used to represent the SOA publish/subscribe concept.

4.2. Software Models

To realize the SOA-compliant DEVS framework, we need to develop the SOA-based DEVS service provider, service client, and service broker models. In addition to

the primitive SOA-based service models, it is also necessary to develop a composite service model.

4.2.1. SOA-Compliant DEVS Models

Based on the relationship defined between SOA and DEVS frameworks (see Table 4) the service provider, service client, and service broker are primitive services in the SOAD framework. As shown in Figure 25, the service provider and service client are defined to have specific ports for requesting and publishing services (H. Sarjoughian et al., 2008). Similarly, the service broker is defined to identify, publish, and found ports given its role with the service provider and client. The coupling relationships among the primitive service provider and service client with one broker are shown in Figure 25. Each of these SOA-based DEVS models are extended from the DEVS atomic model and are defined to have their unique structures and behaviors as described in Section 4.3. For example, a service provider publishes its service to the service broker and performs its own functionality as requested. A service client looks up the service information through the service broker and may subscribe to the published service. The composition of services is represented by a coupled model. A particular realization of the composite service is defined to be a composite service which contains at least two services. The composite service publishes its service as well as each of the services it contains to the service broker. The service broker stores the service definitions and sends them to the clients if the desired services are available.

Three types of messages are defined for the SOAD simulator. They are service-info, service-lookup, and service-call message, as shown in Table 4. Service-info

message which contain the description of the service is used between the service broker and the service provider and the service client. A service-lookup message is used by the service client in order to ascertain whether or not some desired services are available or not from the broker. The service-call message is used between the service client and the service provider. From the service client to the service provider, the service-call message contains a data for the specific method (endpoint) in the service and it contains the serviced result from the service provider to the service client.

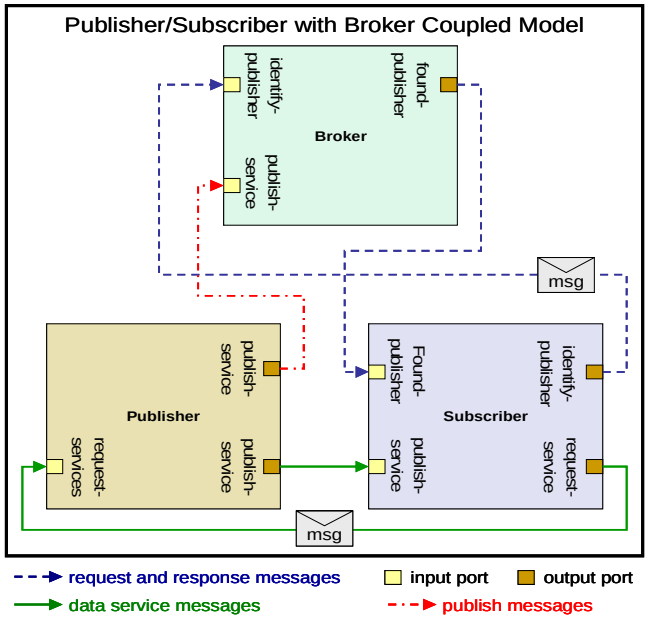


Figure 25. SOA-compliant DEVS model

4.2.2. A Simple Network Model

We need to have a simple network model to represent hardware aspects of the SOAD framework as shown in Figure 26 (H. Sarjoughian et al., 2008). Basically the

network model has capabilities of FIFO message queue, transmission delay, and traffic bandwidth.

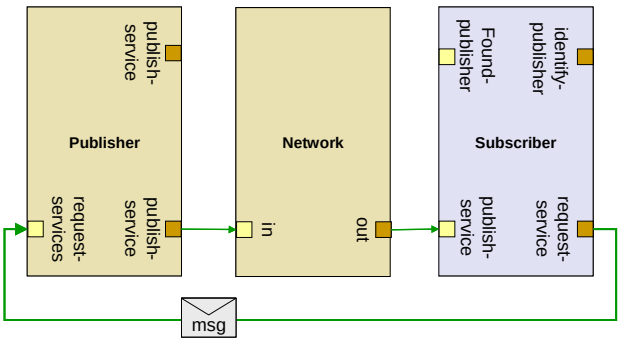


Figure 26. Communication of messages

4.3. Modeling of SOA-Compliant DEVS models

As stated in Chapter 2, each SOA component, service broker, service client, service provider, and composite service, has its own features and behaviors that should be modeled. However, it is impractical to model all possible features and behaviors of the component so that critical features that can represent the characteristics of the model are selected to be modeled.

4.3.1. Service Broker Simulation Model

There are many desirable features for the service broker as discussed in Chapter 2, but service broker simulation model is modeled with service registry feature only for simplicity as shown Figure 27. Basically service provider can publish its service description including endpoints information to the service broker and service client can find the service based on the desired endpoint along with the service name. Service broker store the received service-info messages as it is and return it to the service client

when the service-lookup message matches one of service-info message in the repository. Otherwise, it notices the service client that there is no information available.

Figure 27. Service broker simulation model

4.3.2. Service Client Simulation Model

Service client simulation model is modeled with two behaviors such as looking up the service broker and invoking service provides. First, service client simulation model looks up the service broker using a desired endpoint along with the service name. The list of service providers that a service client simulation model wants to subscribe is constructed when a service client simulation model is defined. If the service client simulation model receives the service-info message from the service broker, then invoke the service provider; otherwise it may continue to look up the service broker for a given number.

Figure 28. Service client simulation model

4.3.3. Service Provider Simulation Model

Service provider simulation model is modeled with its own performService() function that fulfills a set of specific services, as depicted in Figure 29. The service provider simulation model publishes its input ports as endpoints at the given time. It

should be able to handle multiple requests and service them simultaneously. Accessing information is supported by coupling and ports.

Figure 29. Service provider simulation model

4.3.4. Composite Service Simulation Model

To model the composite service simulation model, orchestration is used as a simple service composition in SOAD as shown in Figure 30. There are number of ways to composite services as we discussed in Chapter 2. However, in this research, sequential service composition is implemented to narrow down the scope of this research. Service composition information should be defined in the service-information message model as binding information. Each primitive service provider in the composite model does not know the order of invocation.

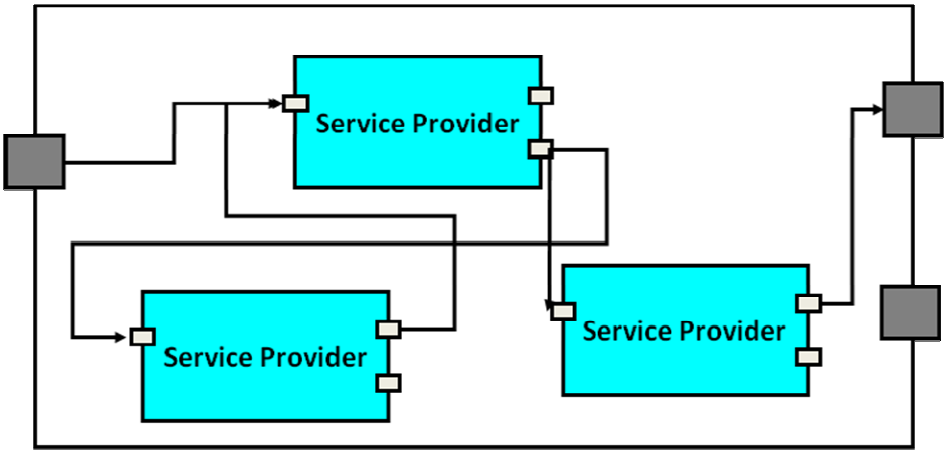


Figure 30. Composite service simulation model

4.4. Implementation of SOA-Compliant DEVS models

As noted previously, a set of generic DEVS models, *GenService*, that represents static and dynamic software aspects of SOA capabilities are developed. In this section, we implement the structural specification of each proposed model and how they correspond with the DEVS specification. The *GenService* API contains several pre-defined behavioral SOA-based simulation models that can be categorized into three types based on its characteristics: generic messages, primitive services, and composite service. The DEVS-Suite is used to simulate the specific model created by the generic *GenService* API, called Application Model.

4. 4.1. Generic Messages

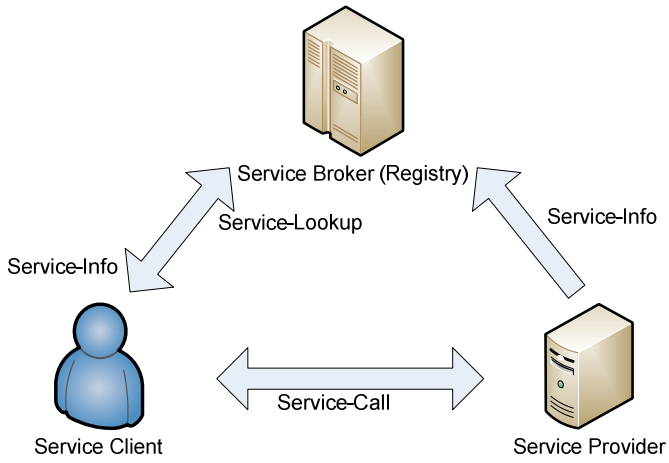


Figure 31. The message types in GenService

As shown in Figure 31, there are three principal usages, publication, lookup, and subscription, of message between services in the SOA. These three different usages require three discrete types of messages due to varying data requirements in each message. Consequently, three different types of messages are employed in the

GenService API. They are derived from the Entity class in the DEVSJAVA API as shown in Figure 32.

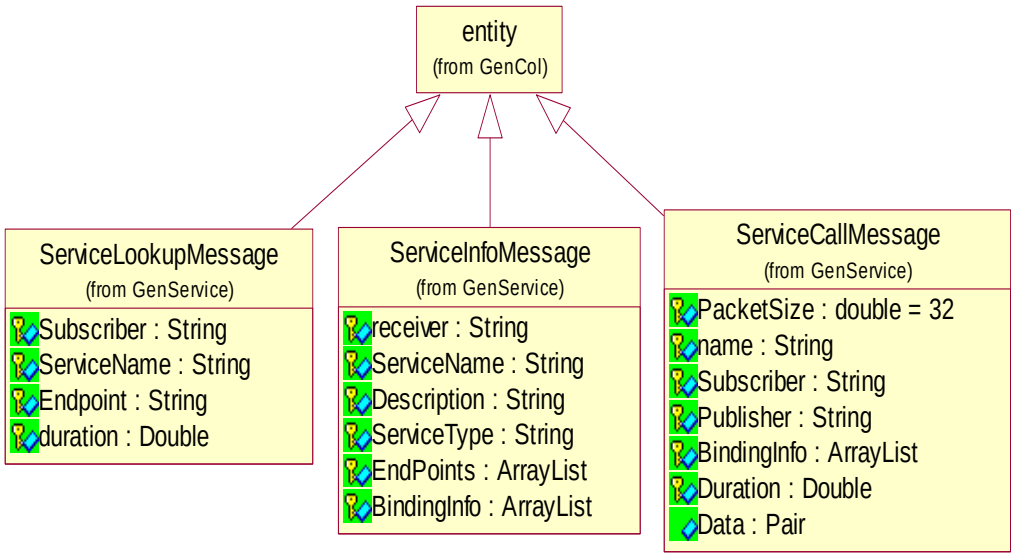


Figure 32. Messages in the SOAD

4.4.1.1. ServiceInfo and ServiceLookup messages. The WSDL is used in the real environment between services and service brokers. In the simulation environment, among the three message types, *ServiceInfo* and *ServiceLookup* represent characteristics of the WSDL (see Table 5).

These messages are needed for publishing services and their discovery. The *ServiceInfo* message type is used to publish the service to the service broker. It contains the service definition given a service name, description, service type (atomic or composite), endpoints, and binding information as shown in Figure 32. The endpoint consists of two parts: exposed method name and argument type for the method. Currently the method is limited to accept only one argument to perform its functionality and later it

will need to be extended. Binding information contains the list of services with an endpoint. Logically the order of services in the list represents the order of service in the composition. This binding information feature is implemented partially for now, but it will be resolved in the next research to support dynamic service compost ion. The *ServiceInfo* message type is stored into the service broker class directly in order to lookup.

Table 5

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

ServiceLookup message type contains the subscription information, the name of the service provider, the endpoint to service client, the data type to be sent, and the time frame to subscribe service. The name of the service provider and an endpoint in that service provider are used as key value to find the desired service information in the Service Broker. In reality, the service client can use a service description or a specific combination of service information to lookup the broker to locate a service. However, this capability is limited to use of service name with an endpoint for the simulation.

4.4.1.2. ServiceCall message. The SOAP, XML based communication protocol, is used over HTTP in the communication between services. In the simulation environment,

ServiceCall message type corresponding to the SOAP properties is employed for exchanging messages between services with the required data. Figure 32 shows the structure of ServiceCall message. The size of ServiceCall message depends on the size of service data plus the default size of packet, 32 Bytes.

4.4.2. Primitive Services

The primitive services such as service provider, service client, and service broker as a DEVS atomic model are proposed as shown in the Figure 33. The default behavioral specification of the *ViewableAtomic* model is presented in the (B. P. Zeigler & Sarjoughian, 2003). These simulation services have a one-to-one correspondence with the SOA service. Services in the SOA can be considered as components in the component-based system. Unlike a component, a service is fully self-contained and loosely coupled.

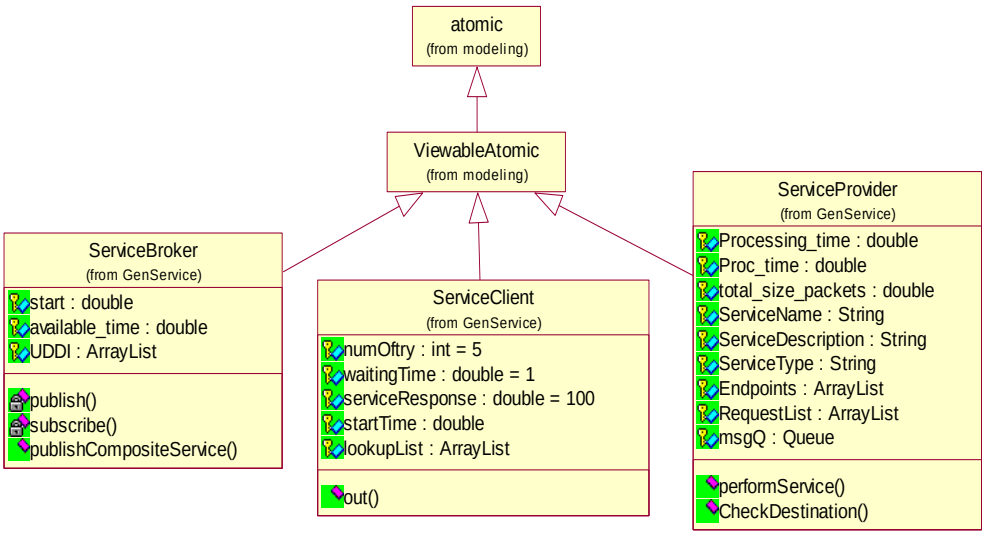


Figure 33. Primitive services in the SOAD

4.4.2.1. ServiceBroker model. ServiceBroker model has a container (UDDI) to store *ServiceInfo* messages as a service description. The desired service can be discovered by looking up an endpoint from the *ServiceClient* as a key. Figure 33 shows two important methods as characteristics of SOA,

- Publish: Store the published service information as a *ServiceInfo* message into the UDDI.
- Subscribe: Return the index of the matched service in the list. An endpoint from the service client is used to lookup the services. If no service is found, then a negative value is returned. Service Broker sends the matched service information (*ServiceInfo*) or "No Found" message to the client.

4.4.2.2. ServiceClient model. ServiceClient model defines a service client in the SOA. A service client can be defined with the list of service that the service client wants to subscribe sequentially. At the beginning, a client with a given start time begins to look up the service broker to search whether the desired service is currently available or not. If the endpoint is not available or even if the service broker itself is not available yet, the service client attempts to lookup the service broker again after a set amount of time units until the specified number of attempts, which is currently set at 5 times as shown in Figure 33. If the endpoint is found and gets the service information, then the service client sends a message with a required data for the endpoint and then waits for the response from the service for the given response time, 100 time units. After completion of a service subscription, if there are more services remaining in the subscription list, then

the service client looks up the broker again and subscribes the service until no more services are in the list.

4.4.2.3. ServiceProvider model. ServiceProvider model defines behavior of its specific service with a *performService* method. The *performService* receives a data from the service client as an argument and performs its specified service depending on the subscribed port (endpoint) using that data. Currently, we do not consider a service which contains multiple methods in it which means a service has only one endpoint to be subscribed upon request. As an initial behavior, all service providers need to publish their services to the Service Broker at the given time. Figure 33 shows the specifications of *ServiceProvider*.

Unlike other simulation models, the *ServiceProvider* model has two time logics, Processing Time and Service Duration, for a queue and a list, msgQ and RequestList respectively, as shown below.

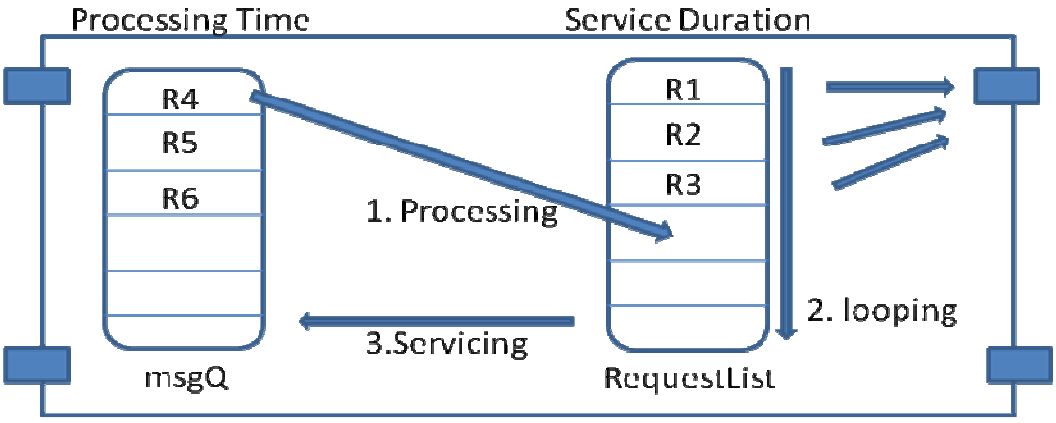


Figure 34. Internal Event function in the ServiceProvider

Processing Time is the required time for a request to be processed before servicing. In other words, a request needs to wait in the msgQ for the Processing Time. For example, if the Processing Time is 5 time units, then R4 in the Figure 34 has waited for 5 simulation time units before it is stored into the RequestList to be served for the requested Service Duration. All service requests, R1 to R4, in the RequestList are handled by the *performService* method for each request at a time.

The functionality of the RequestList is to handle multiple user requests for the same endpoint or service simultaneously. As shown in Figure 35 (a), multiple service clients can subscribe the same endpoint at a time. In that case, at the programming level, endpoint objects from the same endpoint are created for and assigned to each request, as shown in Figure 34 (b). Therefore, it looks like only a service client subscribes this endpoint at a time. This capability is implemented by the RequestList. Multiple requests are stored in the RequestList and they are serviced simultaneously by iterating the entire list at one time. Then the simulation time is advanced.

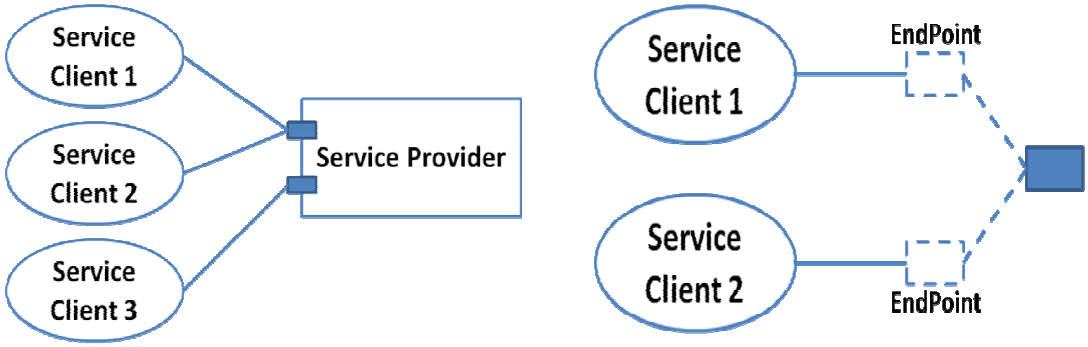


Figure 35. Connection between service clients and an endpoint

The internal event function as a DEVS atomic model loops itself by changing three states, “processing”, “looping”, and “servicing” sequentially as shown in Figure 34 until msgQ and RequestList become empty. At the first “processing” state, if the Processing Time becomes zero, then the top request is pulled out from the msgQ and added into the RequestList. In the “looping” state, the service provider loops the RequestList to serve each request if the requested Service Duration for the request is not equal to zero and then send output messages to each corresponding service client by changing the state to “servicing”. If Service Duration for a request is zero, then it skips to the next request. At the “servicing” state, after all requests in the RequestList are handled, then it removes requests which have zero Service Duration from the RequestList. Finally, the state is changed to “processing” again for another loop.

4.4.3. Composite Service Model

The composite service model contains at least two service providers (either primitive or composite service) models to represent a composite service. The flow of service invocations needs to be specified at the service model design stage. Figure 36 shows how the real services are composed using BPEL. This is a basic capability for hierarchical service provider composition which has to be extended to support different kinds of workflow patterns (Russell et al., 2006).

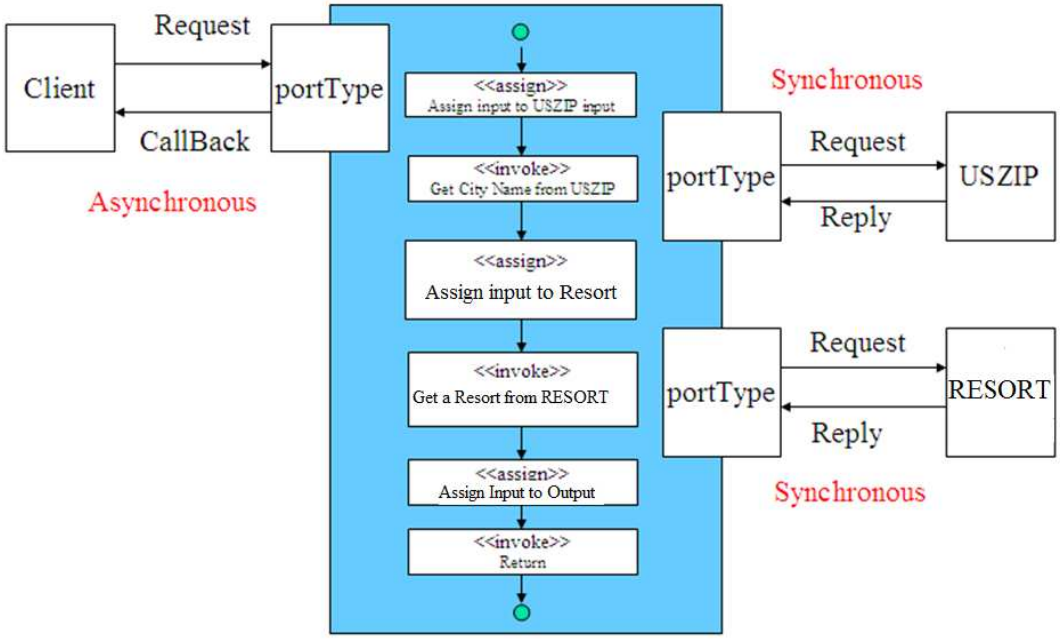


Figure 36. Business Process Execution Language

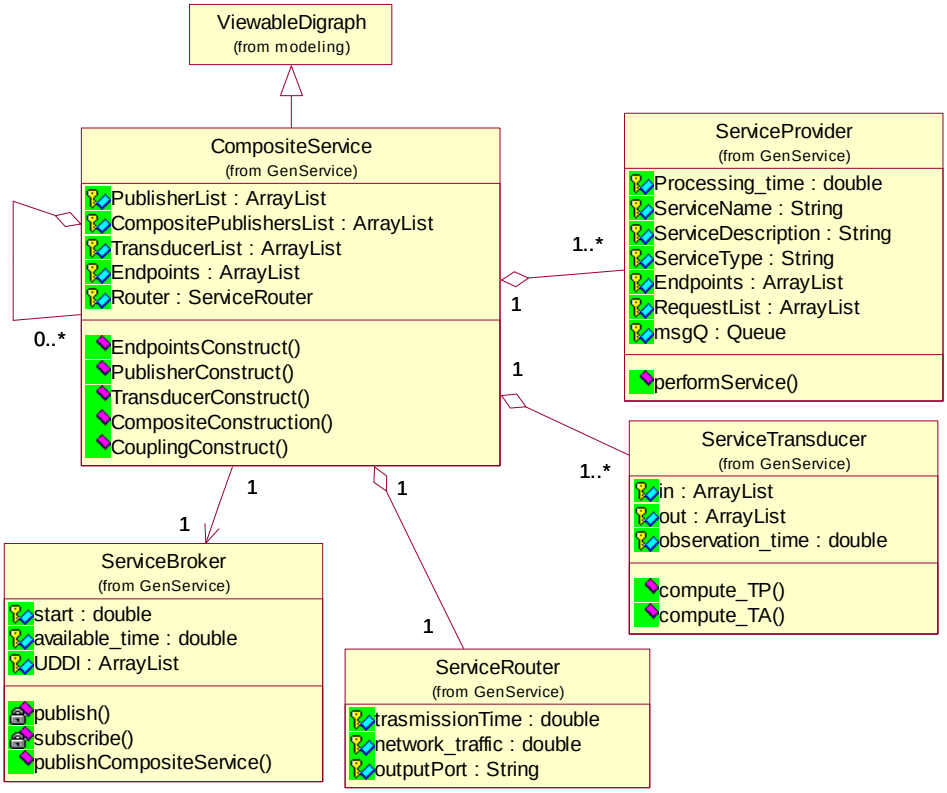


Figure 37. Composite service model

4.4.4. Application Composition

As shown previously, a SOA-compliant DEVS model consists of a set of service provider and service client with a service broker. Default couplings between these primitive services are permanent. Therefore, we employ the *ApplicationComposition* model that constructs default coupling between the service models as shown in the Figure 38.

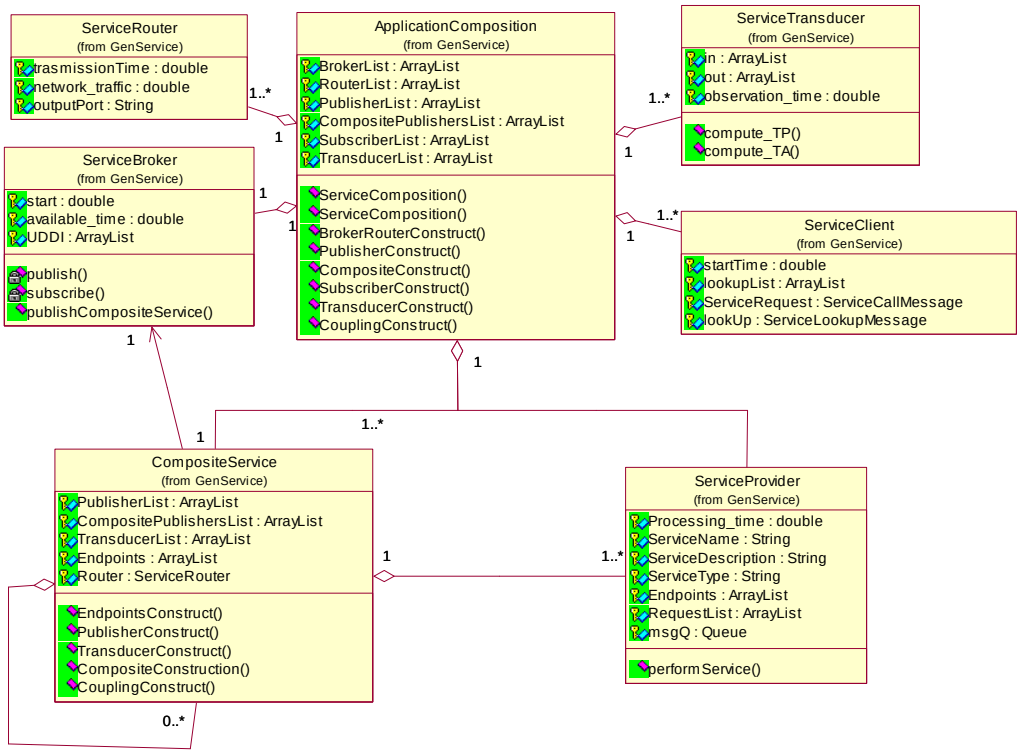


Figure 38. ApplicationComposition model

First of all, this generic composition model contains five empty construction methods which allow users to construct specification of each component in SOAD. As stated earlier, at least one service provider (either primitive or composite), one service client, and one service broker are required to compose a SOA-compliant DEVS application model. The cardinalities in Figure 38 represent that constraint. At the last, *CouplingConstruct* method which is used for coupling between each DEVS component in the five lists is predefined.

4.4.5. ServiceTransducer Model

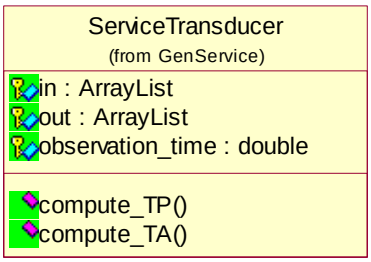


Figure 39. ServiceTransducer model

Transducer, an observational component, collects a set of simulation data for the service and it is attached to each service model, service provider, service client, and service broker. Transducer keeps track of all arrivals and departures for a given observation interval then turnaround time and throughput for the model can be computed as shown in Figure 39.

- Turnaround time: the length of time between its arrival and its departure from the attached service.
- Throughput: the average rate of message departures from the service, estimated by the number of requests processed during the observation interval, divided by the length of the interval.

5. Simulation Experiments

This chapter demonstrates the experiments of a set of SOAD simulation models discussed in Chapter 4. Using the simulation models, a set of the experimental scenarios such as the Voice Communication Service and Travel Agency Service are developed and simulated on the DEVS-Suite environment in order to verify and validate that the simulation models are suitable to represent the SOA concepts and capabilities.

5.1. Service Composition and Configurations

The service composition can be defined with four configurations as shown in Table 6. These configurations are based on the number of service clients and providers, not on the number of service broker since there is only one broker in the service composition. The service composition must be able to support these configurations in terms of SOA concept and capabilities.

Table 6

Service Composition Configurations

	Client	Provider	Broker
Configuration 1	1	1	1
Configuration 2	n	1	
Configuration 3	1	n	
Configuration 4	n	n	

5.2. Experimental Scenarios

Two experimental scenarios, such as the Voice Communication Service (VCS) System and the Travel Agency Service (TAS) System are developed in order to validate the SOAD simulation models.

5.2.1. Real Voice Communication Service System

A real experiment for the VCS system is developed to capture four critical QoS features, such as timeliness, throughput, accuracy, and security. The experiment is a simple network intensive service, where multiple service clients can use the VCS simultaneously to receive real-time voice data streams with various qualities of voice configured by a user-specified sampling rate as shown in Figure 40.

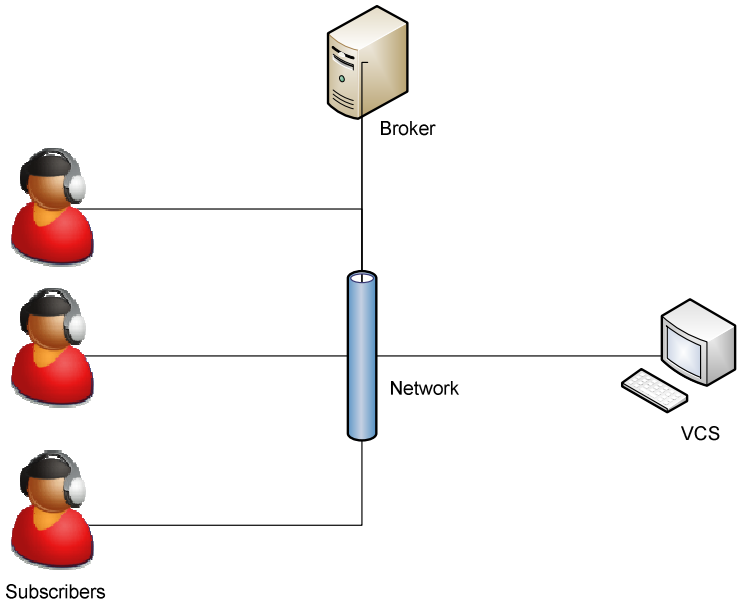


Figure 40. Voice Communication service

The real experiment is developed using C# in .Net and deployed on the .NET development server. Since the VCS is a network intensive service, throughput is mainly examined as a main QoS feature. Processor, Memory, Physical Disk, System, IP, UDP, TCP, Server, and Web Services are collected as the experimental data using Windows Performance Object. There are three experiment control variables, such as the sampling rate for recording the voice data stream, the number of service clients, and the buffer size

for storing the voice data before network. Table 7 shows the different setting for those control variables.

Table 7

The Experimental Control Variables Settings

Sampling Rate (KHz)	44.1, 88.2, 132.3, 176.4, 220.5
# of Service clients	1, 2, 3, 4, 5
Buffer Size (KBytes)	16, 32, 48, 64, 80

The real experiment has been run under 125 experimental conditions, and each experimental condition has 5 independent runs to collect 5 replicates of data set. For each individual replicate, 60 data observations are recorded. By comparing these data sets with simulation data sets, we can verify and validate our SOAD simulation models.

In the real experiment, the actual broker service is not presented since we assumed that the VCS has already been published to the service broker and service clients have looked up the broker to find the VCS, meaning that service clients already know the VCS information such as the URL of the VCS which are described in WSDL. However, in the SOAD framework, the broker must be implemented so that the SOA concepts and capabilities are represented by the generic simulation models.

5.2.2. Travel Agency Service System

A simple simulation experiment called Travel Agency Service (TAS) system is designed to show the service composition with a composite service. For this service composition, we developed two primitive simulation services, such as the USZIP service

which provides the city name by a given zip code and the RESORT service which displays the closest resort place by a given city name. Then these primitive services are used to construct the TAS.

Figure 41 depicts a simple experimental scenario of using the TAS. A service client invokes the TAS with a zip code (85281). The TAS sends the 85281 to the USZIP service and then the USZIP service provides a city name (Tempe) generated by the zip code to the RESORT service. The RESORT service then produce the closet resort name (the Phoenix Resort) by the city name and The TAS returns the result to the client.

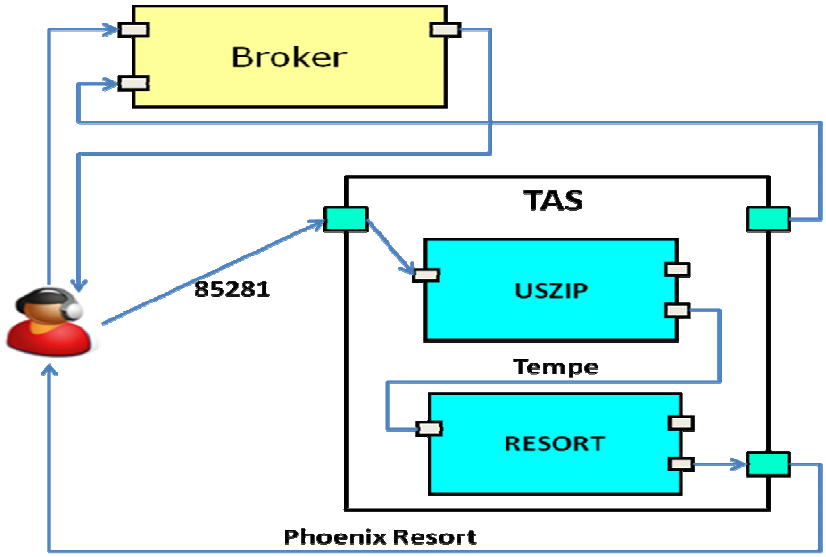


Figure 41. Travel Agency service composition

5.3. Service Composition with Primitive Services

Based on the service configurations stated in Table 6, we can create several simulation scenarios for the VCS simulation. First of all, we need to construct instances of service providers including specifications of endpoints, service clients and a service broker as well as the hardware component, network, for a service composition.

5.3.1. Composition for the VCS Model with Configuration 1

Since we already have the *ApplicationComposition* class defined in Chapter 4, the composition of primitive services as well as the network for the VCS can be derived from the *ApplicationComposition*. In the *ApplicationComposition* class, as shown in Figure 42, six independent service lists are created to store instances of each component. For the VCS composition, we need to construct all participants at each construction function call and store them into the corresponding list. The order of each construction function call does not matter. However, the coupling construction function must be called at the last to automatically construct coupling between all components stored in the six service lists.

```
public ServiceComposition(String nm){
    super(nm);

    BrokerList              = new ArrayList <ServiceBroker>           ();
    RouterList              = new ArrayList <ServiceRouter>           ();
    PublisherList           = new ArrayList <ServicePublisher>        ();
    CoupledPublishersList   = new ArrayList <ServiceCoupledPublishers>();
    SubscriberList          = new ArrayList <ServiceSubscriber>       ();
    TransducerList          = new ArrayList <ServiceTransducer>       ();

    //This function call construct the list of brokers and routers
    BrokerRouterConstruct();

    //This function call construct the list of subscribers
    SubscriberConstruct();

    //This function call construct the list of publishers
    PublisherConstruct();

    //This function call construct the list of composite services
    CompositeConstruct();

    //This function call construct the list of transducers
    TransducerConstruct();

    //This function call construct the coupling of each component
    CouplingConstruct();
}
```

Figure 42. The Service Composition class

5.3.1.1. Service broker and Network. Since we need only one broker and one network link in the configuration 1 for the VCS simulation, we construct a service broker with a service start time and a length of available time and a network link with a network bandwidth and store them into corresponding lists, as shown in Figure 43.

```
public void BrokerRouterConstruct(){

    //attributes
    double available = 60;     //available time for broker
    double startTime = 0.5;
    double bandwidth = 10;      //bandwidth for the network or router

    //Create unique components
    Broker = new ServiceBroker("Broker", available, startTime);
    Router = new ServiceRouter("Router Link", bandwidth);

    BrokerList.add(Broker);
    RouterList.add(Router);
}
```

Figure 43. Broker and Network construction

5.3.1.2. Service provider. As shown in Figure 44, *qRate* is defined as an endpoint which requires an argument in double data type to subscribe and store into the list of endpoints for the VCS. Then, the VCS is constructed with the service name, service description, service type, list of endpoints, and processing time for the request.

```
public void PublisherConstruct(){

    ArrayList <Pair> Endpoints = new ArrayList <Pair> ();
    Endpoints.add(new Pair("qRate", "Double"));

    VoiceComm Service1 =
        new VoiceComm("VoiceComm", "Voice Communication", "Atomic", Endpoints, 1);

    //Construct the publisher list
    PublisherList.add(Service1);
}
```

Figure 44. Service provider construction

The basic behaviors and structures of the provider are already defined in the generic *SerivceProvider* class as stated in Chapter 4. Therefore, to construct the VCS, we need to define service specifications for *qRate* in the *performService* method as shown in Figure 45.

```
public class VoiceComm extends ServicePublisher{

    public VoiceComm(String name,       //Service Name
                     String descpt,     //Service Description
                     String svType,     //Service Type atomic or composite
                     ArrayList <Pair> endpts, //The list of endpoint
                     double processingTime){  //processing time to handle a request
        super(name, descpt, svType, endpts, processingTime);
    }

    public Pair performService(Pair data){
        double buffersize = 16;   //buffersize (Kbps)  - 32 48 and ....user choice
        double avgNumOfDatagram = 260;  //average number of datagram
        double sizeOfmsgs = (avgNumOfDatagram * buffersize);
        ServiceReturn.setSize(sizeOfmsgs);
        return data;
    }
}
```

Figure 45. qRate specification

Since the VCS returns the voice data streams with the requested sampling rate to the client, the size of returning messages vary based on the control variable setting discussed in Table 7. For the current setting, the buffer size for the network communication is set to 16 Kbytes and the average number of datagrams received by the service client is set to 260. These settings are adopted from the real experiment. Therefore, the size of each message from the VCS to the service client is the average number of messages multiple by the buffer size, that is, 4160 Kbytes. We ignored the datagrams header size, 32 Bytes, since it is a constant and negligible in size.

5.3.1.3. Service client. With the configuration 1, as shown in Figure 46, the VCS simulation has only one service client which is defined with the name of the client, the list of *ServiceLookup* messages that contains the client with an endpoint, the sampling rate, and the duration of time to subscribe to the service, and the time to look up the Service Broker. This Service Client is added into the list of clients to be coupled with other components in the simulation. The default structure and behavior of the client is defined in the generic *ServiceClient* class in Chapter 4.

```
public void SubscriberConstruct(){

    //Construct ServiceLookup information: The list of service to subscribe
    ArrayList <ServiceLookup> lookupList = new ArrayList <ServiceLookup> ();
    lookupList.add(new ServiceLookup("VoiceComm", "qRate", new Pair("Hz", 220500), 60));

    //Construct Subscriber with the list of service lookup message
    ServiceSubscriber Subscriber1 = new ServiceSubscriber("Subscriber1", lookupList, 0.1);

    //Add the subscriber into the list
    SubscriberList.add(Subscriber1);

}
```

Figure 46. Service client construction

5.3.1.4. Transducer. The list of transducers for each primitive service as well as the network is constructed to collect a set of simulation data as shown in Figure 47. These data sets are compared with the data sets collected from the real experiment in order to validate the SOAD simulation models. Transducers observe each component for a given length of time and then measure the performance metrics defined in each transducer model. For example, the transducer for the network link can measure the average transmission delay, total size of messages transmitted, and network utilizations as shown in Figure 48.

```
public void TransducerConstruct(){
    BrokerTransd BroTrans       =  new BrokerTransd("BrokerTransd", observation);
    RouterTransd NecTrans       =  new RouterTransd("RouterTransd", observation);
    SubscriberTransd SubTrans1  =  new SubscriberTransd("Sub1Transd", observation);
    PublisherTransd PubTrans1   =  new PublisherTransd("VoiceCommTransd", observation);

    //Always the same order: Broker >> Network >> Subscriber >> Publisher
    TransducerList.add(BroTrans);
    TransducerList.add(NecTrans);
    TransducerList.add(SubTrans1);
    TransducerList.add(PubTrans1);
}
```

Figure 47. Transducer construction

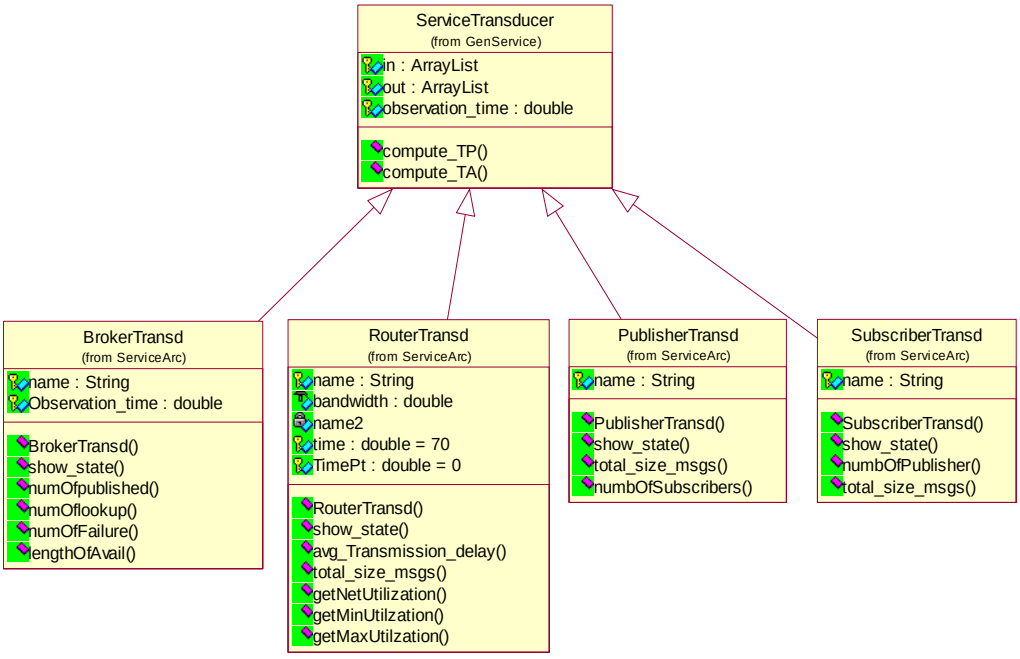


Figure 48. Transducers in the VCS

5.3.1.5. Coupling of services. The user takes a role of constructing a service broker, a network, the list of service clients, and the list of service providers while coupling between these components is automatically completed. Figure 49 shows the service composition for the VCS simulation with the configuration 1.

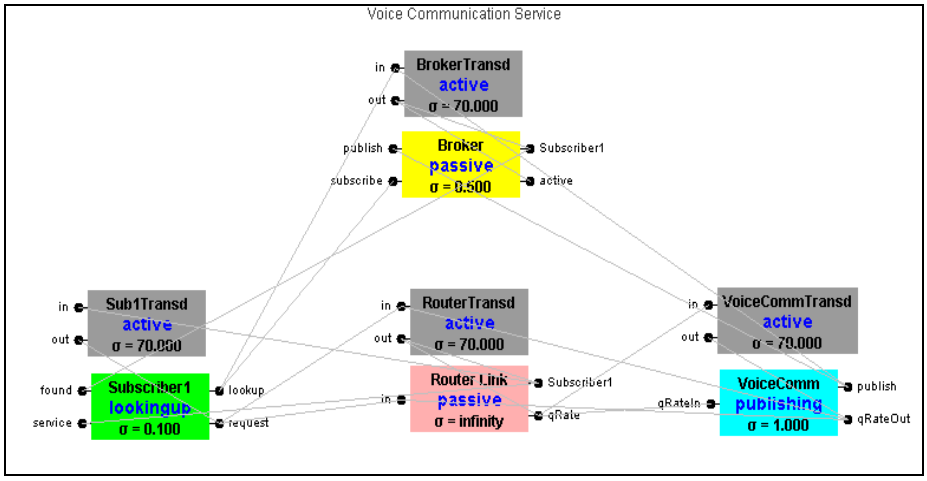


Figure 49. Voice Communication service with Configuration 1

5.3.2. Composition for the VCS Simulation with Configuration 2

Configuration 2 represents multiple clients with one provider and that is easily constructed since the generic provider class has the capability to handle multiple requests simultaneously as described in Chapter 4. Since the list of clients and the list of providers are maintained independently and coupling between these primitive services are already established, we only need to construct more clients to subscribe the VCS as shown in Figure 50.

There are no changes with the service broker and the VCS. Therefore, the VCS with multiple clients as mentioned as configuration 2 are constructed as depicted in the Figure 51. As shown, the entire VCS simulation is not that different from the Figure 49, which only has one client, except for the number of client. However, the VCS can receive multiple requests and return a voice data stream with requested sampling rate back to clients simultaneously.

```
public void SubscriberConstruct(){
    //Construct ServiceLookup information: The list of service to subscribe
    ArrayList <ServiceLookup> lookupList = new ArrayList <ServiceLookup> ();
    //Construct subscriber with name, service lookup info, start time
    lookupList.add(new ServiceLookup("VoiceComm", "qRate", new Pair("Hz", 220500), 60));

    ServiceSubscriber Subscriber1 = new ServiceSubscriber("Subscriber1", lookupList, 0.1);

    lookupList = new ArrayList <ServiceLookup> ();
    lookupList.add(new ServiceLookup("VoiceComm", "qRate", new Pair("Hz", 220500), 60));
    ServiceSubscriber Subscriber2 = new ServiceSubscriber("Subscriber2", lookupList, 0.1);

    lookupList = new ArrayList <ServiceLookup> ();
    lookupList.add(new ServiceLookup("VoiceComm", "qRate", new Pair("Hz", 220500), 60));
    ServiceSubscriber Subscriber3 = new ServiceSubscriber("Subscriber3", lookupList, 0.1);

    //Construct the subscriber list
    SubscriberList.add(Subscriber1);
    SubscriberList.add(Subscriber2);
    SubscriberList.add(Subscriber3);

}
```

Figure 50. Service client construction with Configuration 2

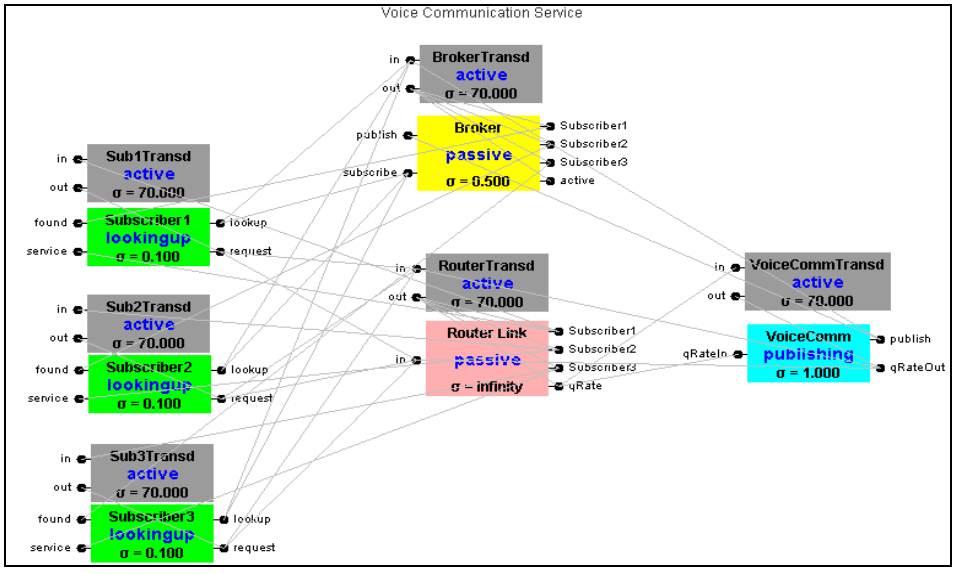


Figure 51. The VCS simulation with Configuration 2

5.3.3. Validation on the SOAD Simulation Models.

Since configuration 3 and 4 use the simulated service, comparisons between the simulation data sets from the transducers and the real experimental data sets from the Window Performance Objects are conducted for the VCS system with configuration 1

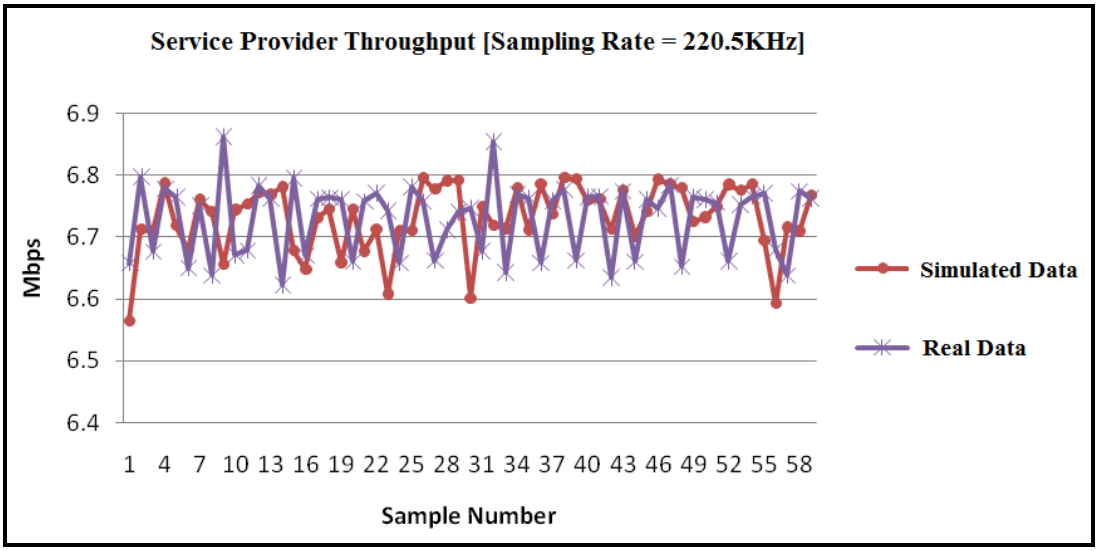
and 2 so that the SOAD simulation models can be validated in terms of four critical QoS features, mainly throughput for the VCS system. Since it is more important for the aspect of service provider, we measured throughputs of the real VCS and simulated VCS with configuration 1 and 2 60 times. The simulation control variable settings for each case are shown in Table 8.

Table 8

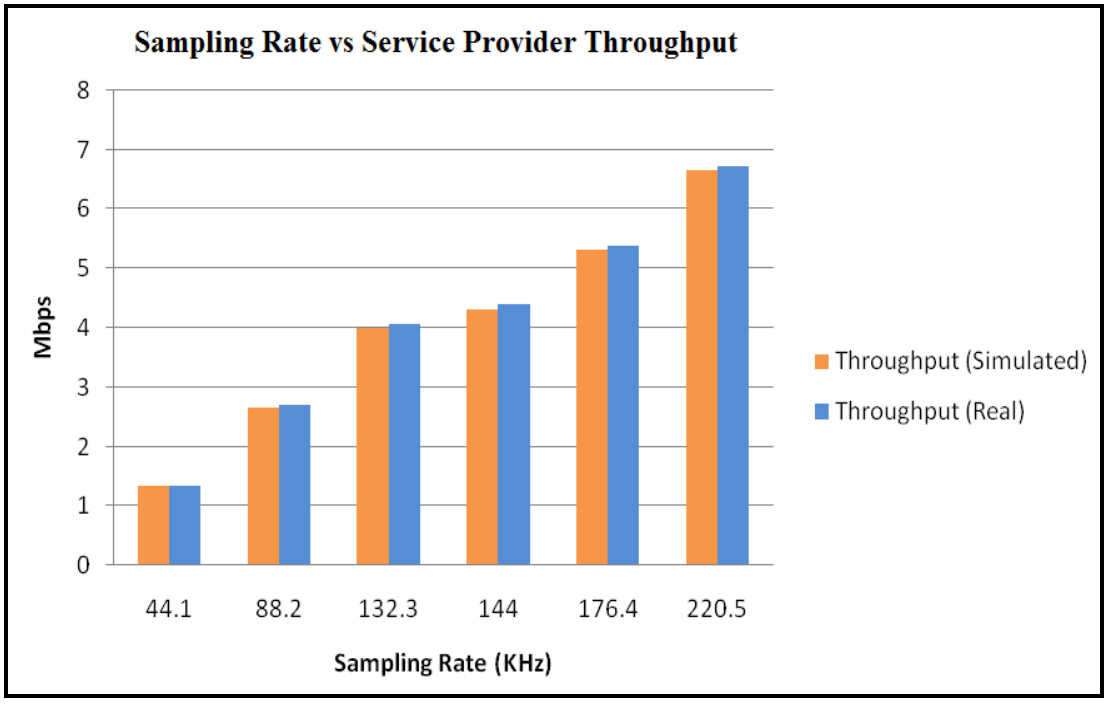
The Simulation Control Variable Setting

	Configuration 1	Configuration 2
Sampling Rate	44.1 - 220.5 KHz	44.1 KHz
# of Service clients	1	1-5
Buffer Size	16 Kbytes	

With configuration 1, we collected two sets of data for both the real and simulated VCS (Roontiva et al., in preparation). First, we measure service provider throughputs with a fixed sampling rate 220.5 KHz for 60 times. Second, we adjusted the sampling rate from 44.1 KHz to 220.5 KHz and collected service provider throughputs. The results of the real and simulated VCS with configuration 1 are shown in Figure 52.



(a) The VCS Throughput by Sample Number



(b) Sampling Rate vs. Service Provider Throughput

Figure 52. The measurements of the VCS throughput with Configuration 1

For the case of configuration 2, we measure throughputs with a fixed sampling rate 44.1 KHz for the both the real and simulated VCS by adjusting the number of service clients from 1 to 5. The actual measurements are presented in Table 9 and comparisons are shown in Figure 52.

Table 9

Throughputs for the Real and Simulated VCS by Number of Service Clients

Number of Service Clients	Throughput (Simulated)	Throughput (Real)
1	1.332300356	1.3459957
2	2.670600711	2.693091272
4	4.333476156	4.396452098
5	5.311201423	5.388197829

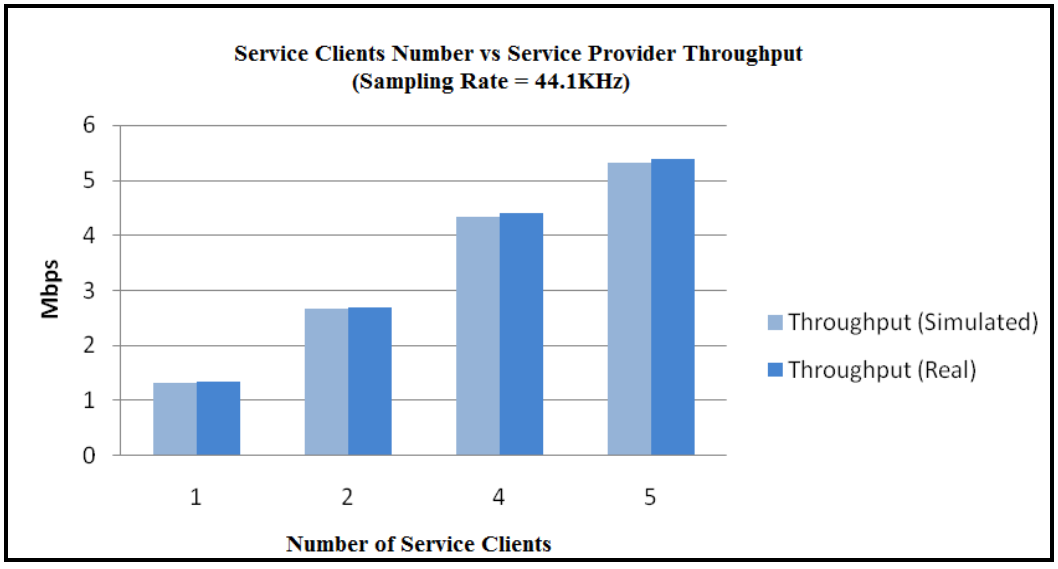


Figure 53. Comparison the throughputs between the real and simulated VCS

Based on the comparisons between the simulation data and experimental data, there are differences between them; however the differences are negligible as shown in Figure 52 and 53 meaning that we can validate that the SOAD simulation models are suitable to represent the static software aspect of the SOA capabilities.

5.3.4. Composition for the VCS Simulation with Configuration 3

Since the SOAD simulation models have been validated, we can extend the service composition with configuration 3. In this configuration, a service client subscribes multiple service providers. For this case, we can use two more primitive services developed for the TAS simulation, which are, USZIP service and RESORT service as shown in Figure 54 and add them into the service provider list for coupling.

```
public void PublisherConstruct(){

    ArrayList <Pair> Endpoints = new ArrayList <Pair> ();

    Endpoints.add(new Pair("qRate", "Double"));
    VoiceComm Service1 =
        new VoiceComm("VoiceComm", "Voice Communication", "Atomic", Endpoints, 1);
    Service1.setBackgroundColor(Color.CYAN);
    PublisherList.add(Service1);

    Endpoints = new ArrayList <Pair> ();
    Endpoints.add(new Pair("CityByZip", "Double"));
    USZipService Service2 =
        new USZipService("USZip", "City by Zip Service", "Atomic", Endpoints, 1);
    Service2.setBackgroundColor(Color.CYAN);
    PublisherList.add(Service2);

    Endpoints = new ArrayList <Pair> ();
    Endpoints.add(new Pair("ResortByCity", "String"));
    ResortService Service3 =
        new ResortService("Resort", "Resort by City Service", "Atomic", Endpoints, 1);
    Service3.setBackgroundColor(Color.CYAN);
    PublisherList.add(Service3);
}
```

Figure 54. Service Provider construction with Configuration 3

Since we do not need complicated services, the specifications of endpoints in these services are really simple as shown in Figure 55.

```
public Pair performService(Pair data){
    double sizeOfmsgs = 32;
    Double doubleVal;
    Pair returnVal = new Pair();

    doubleVal =
        Double.parseDouble(data.value.toString());

    //if the zip code is 85281, then return tempe
    if(doubleVal == 85281){
        returnVal.key = "String";
        returnVal.value = "tempe";
    }
    else{
        returnVal.key = "String";
        returnVal.value = "No Found";
    }
    ServiceReturn.setSize(sizeOfmsgs);
    return returnVal;
}
```

(a) USZIP

```
public Pair performService(Pair data){
    double sizeOfmsgs = 32;
    Pair returnVal = new Pair();

    //if argument is tempe,
    //then return pheonix resort
    if(data.value.toString().equals("tempe")){
        returnVal.key = "String";
        returnVal.value = "Pheonix Resort";
    }
    else{
        returnVal.key = "String";
        returnVal.value = "No Found";
    }

    ServiceReturn.setSize(sizeOfmsgs);
    return returnVal;
}
```

(b) RESORT

Figure 55. Specifications of endpoints in USZIP and RESORT services

For the case of a service client, the lookup table is maintained to store the lookup messages for subscription. A service client looks up the broker at first to subscribe the VCS. After completion of the VCS, a service client looks up the broker again if there are more services that the service client wants to subscribe in the lookup list. The order of subscriptions needs to be specified as shown in the Figure 56.

```
public void SubscriberConstruct(){
    //Construct ServiceLookup information: The list of service to subscribe
    ArrayList <ServiceLookup> lookupList = new ArrayList <ServiceLookup> ();

    //construct the subscription list
    lookupList.add(new ServiceLookup("VoiceComm", "qRate", new Pair("Hz", 220500), 60));
    lookupList.add(new ServiceLookup("USZip", "CityByZip", new Pair("double", 85281), 1));
    lookupList.add(new ServiceLookup("Resort", "ResortByCity", new Pair("String", "Tempe"), 1))

    ServiceSubscriber Subscriber1 = new ServiceSubscriber("Subscriber1", lookupList, 0.1);

    //Construct the subscriber list
    SubscriberList.add(Subscriber1);

}
```

Figure 56. Service clients construction with Configuration 3

Finally, Figure 57 shows the service composition for the configuration 3.

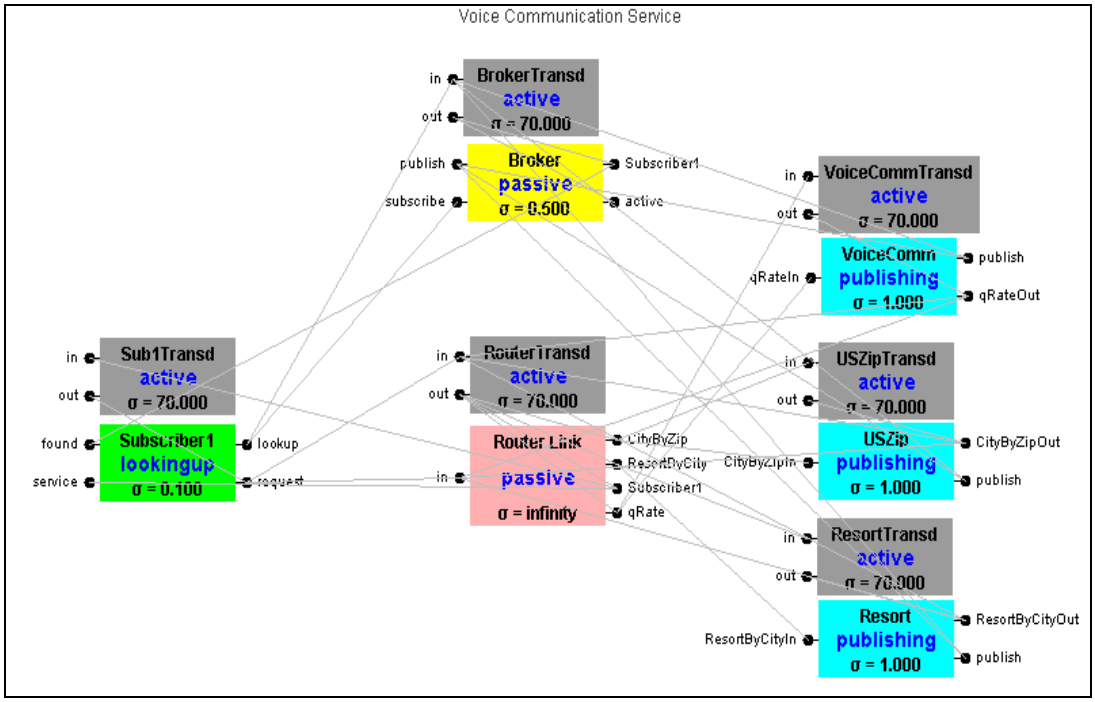


Figure 57. The VCS simulation with Configuration 3

5.3.5. Composition for the VCS Simulation with Configuration 4

Configuration 4 is a consolidation of configuration 2 and 3 so that we need to construct multiple service clients as well as service providers. For the construction of service providers, we use the VCS, USZIP, and RESORT services. We modified little bit for client construction so that each service client subscribes to a different service provider. Figure 58 shows the service client construction, where Subscriber 1 subscribes the USZIP service, Subscriber 2 subscribes the VCS, and Subscriber 3 subscribes the RESORT service. Figure 59 is the service composition with configuration 4.

```
public void SubscriberConstruct(){

    //Construct ServiceLookup information: The list of service to subscribe
    ArrayList <ServiceLookup> lookupList = new ArrayList <ServiceLookup> ();
    lookupList.add(new ServiceLookup("USZip", "CityByZip", new Pair("double", 85281), 1));
    ServiceSubscriber Subscriber1 = new ServiceSubscriber("Subscriber1", lookupList, 0.1);

    lookupList = new ArrayList <ServiceLookup> ();
    lookupList.add(new ServiceLookup("VoiceComm", "qRate", new Pair("Hz", 220500), 60));
    ServiceSubscriber Subscriber2 = new ServiceSubscriber("Subscriber2", lookupList, 0.1);

    lookupList = new ArrayList <ServiceLookup> ();
    lookupList.add(new ServiceLookup("Resort", "ResortByCity", new Pair("String", "Tempe"), 60)
    ServiceSubscriber Subscriber3 = new ServiceSubscriber("Subscriber3", lookupList, 0.1);

    Subscriber1.setBackgroundColor(Color.GREEN);
    Subscriber2.setBackgroundColor(Color.GREEN);
    Subscriber3.setBackgroundColor(Color.GREEN);

    //Construct the subscriber list
    SubscriberList.add(Subscriber1);
    SubscriberList.add(Subscriber2);
    SubscriberList.add(Subscriber3);
}
```

Figure 58. Service client construction with Configuration 4

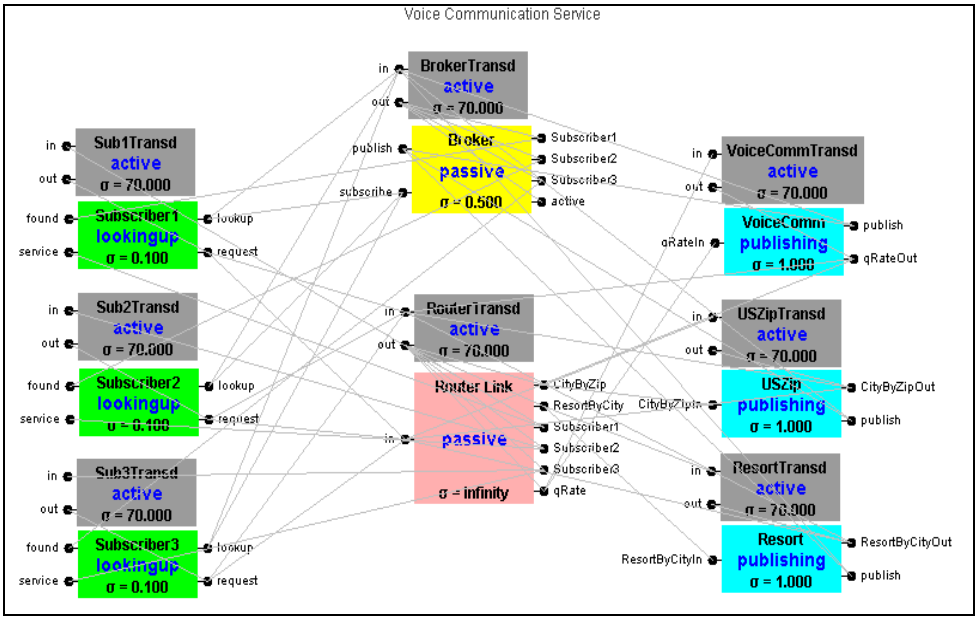


Figure 59. Service composition with Configuration 4

5.4. Service Composition with Composite Service

Since we have constructed two primitive services, USZIP and RESORT, and the USZIP service can provide a required data, the city name, for the RESORT service, we can construct a composite service, the TAS, by using them.

5.4.1. Composition for the TAS simulation with configuration 4.

The service composition is shown in Figure 60. The services in the composite service class can be either a service provider or a service client in terms of a SOA concept. In addition, a service broker is not involved in the composite service class since only one universal service broker at the top architectural level is used as a repository for the composite service.

```
public ServiceCoupledPublishers(String nm){
    super(nm);

    //Router construction
    Router = new ServiceRouter("Router", bandwidth);

    //Endpoints for this composite servcie
    Endpoints               =  new ArrayList <Pair>                          ();

    //Publisher construction
    PublisherList           = new ArrayList <ServicePublisher>               ();
    CoupledPublishersList   = new ArrayList <CopyOfServiceCoupledPublishers>();
    TransducerList          = new ArrayList <ServiceTransducer>              ();

    // This function call construct the list of endpoints
    EndpointsConstruct();

    // This function call construct the list of publishers
    PublisherConstruct();

    // This function call construct the list of transducers
    TransducerConstruct();

    // This function call construct the list of composite services
    CompositeConstruction();

    // This function call construct the coupling of each component
    CouplingConstruct();
}
```

Figure 60. Composite service composition

5.4.1.1. Endpoints construction. Since the TAS itself is a composite service, the TAS must have at least one endpoint to publish its functionality to the service broker. Figure 61 shows the endpoint construction for the TAS.

```
public void EndpointsConstruct(){

    Endpoints.add(new Pair("ResortByZip", "Double"));

}
```

Figure 61. Endpoints construction for the RBZ

5.4.1.2. Service provider construction. The specifications of endpoints in the USZIP and RESORT services are already defined previously. Therefore, we need to construct the primitive client list for the TAS using them as shown in Figure 62.

```
public void PublisherConstruct(){

    ArrayList <Pair> Endpoints = new ArrayList <Pair> ();
    Endpoints.add(new Pair("CityByZip", "Double"));
    USZipService Service1 =
        new USZipService("USZip", "City by Zip Service", "Atomic", Endpoints, 1);
    Service1.setBackgroundColor(Color.CYAN);
    //Construct the publisher list
    PublisherList.add(Service1);

    Endpoints = new ArrayList <Pair> ();
    Endpoints.add(new Pair("ResortByCity", "String"));
    ResortService Service2 =
        new ResortService("Resort", "Resort by City Service", "Atomic", Endpoints, 1);
    Service2.setBackgroundColor(Color.CYAN);
    PublisherList.add(Service2);

}
```

Figure 62. Service client construction for the RBZ

5.4.1.3. Service composition with the VCS and TAS. Now we have a primitive service, the VCS, and a composite service, the TAS and then we construct the service with configure 4 using VCS, TAS, and three clients, where two clients subscribe the VCS and one client subscribes the TAS. Since we have constructed all services except the TAS, we need to construct the TAS in the service composition as shown below.

```
public void CompositeConstruct(){
    ArrayList <Pair> Endpoints = new ArrayList <Pair> ();
    Endpoints.add(new Pair("ResortByZip", "Double"));

    ResortByZipServices Service2 = new ResortByZipServices();
    CoupledPublishersList.add(Service2);

    ServiceInfo CompositeService = new ServiceInfo("ResortByZip",
                                                   "Find a resort by zip",
                                                   "composite", Endpoints);

    //Set Binding Info (Service, endpoint)
    CompositeService.setBindingInfo(new Pair("ResortByZip", "ResortByZip"));
    CompositeService.setBindingInfo(new Pair("USZip", "CityByZip"));
    CompositeService.setBindingInfo(new Pair("Resort", "ResortByCity"));

    Broker.publishCompositeService(CompositeService);

}
```

Figure 63. Composite service construction

Since there are no automatic ways to composite services at a run time currently, we need to specify the service binding information, or rather the order of primitive and/or composite services in the composite service so that the message contains the information concerning which service is the next receiver. This composite service must be published manually into the broker to be subscribed. Figure 63 shows the entire composite service composition with the VCS and the TAS.

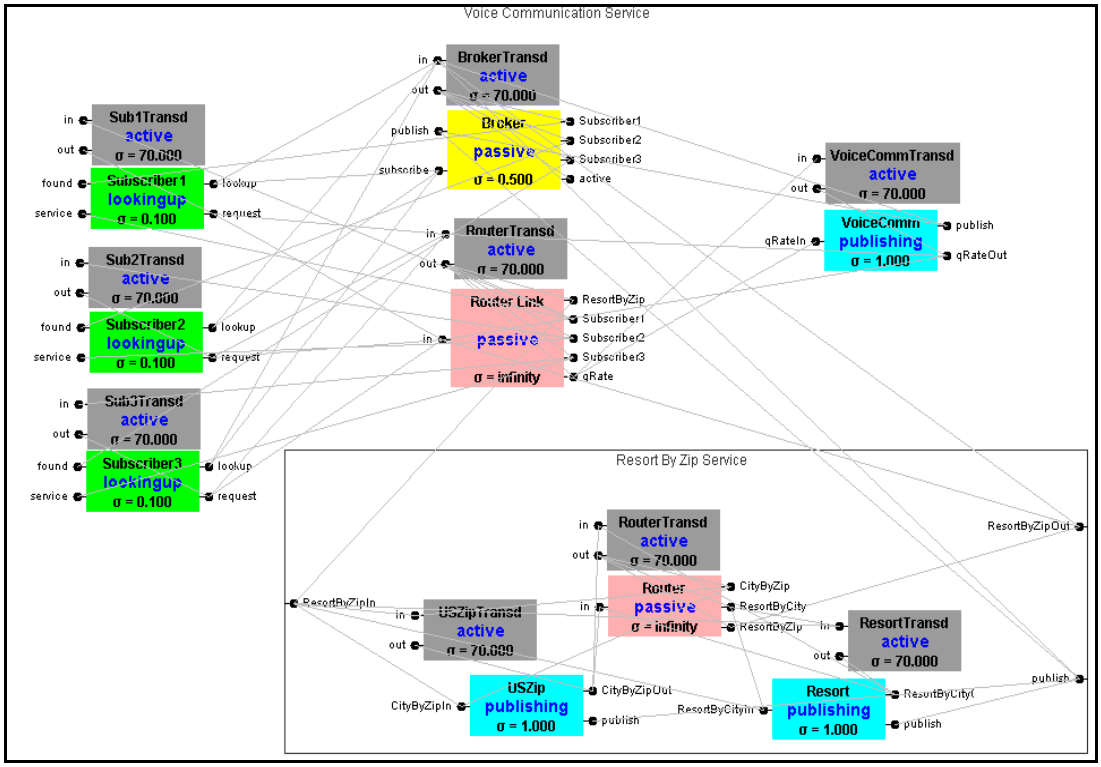


Figure 64. The Composite service composition with the VCS and the TAS

5.5 Scaling SOAD Models with the DEVS-Suite

In addition to the simulation experiments that were developed for the configurations listed in Table 6, it is useful to determine how large of a system can be simulated in the DEVS-Suite for a given hardware computational resource. The Voice Communication System with Configuration 2 is used to examine the scalability of the SOA and the DEVS-Suite. A representative set of simulation models having 20 to 7000 model components were devised and simulated for the VCS system. For the convenience, the SimView is turned off and any set of data is tracked. Figure 65 shows how the number of service clients impacts the wall-clock simulation time given a desktop machine with Core 2 Duo 2.66 GHz CPU and 4GB RAM. As expected, the wall-clock

simulation time increases proportional to the number of service clients. The largest simulation executed contained 3500 service client models and 3500 transducer models. The total number of models in DEVS-Suite can be increased provided that more powerful hardware (more cache and virtual memory as well as higher speed single or multi-core processors). In particular, the default setting of the Java Virtual Machine can be changed to allow more virtual memory which is needed to load and execute larger number of objects. The DEVS-Suite, therefore, supports conducting relatively large-scale simulation on single machines and thus supports the scalability needs of the simulating service-based software systems.

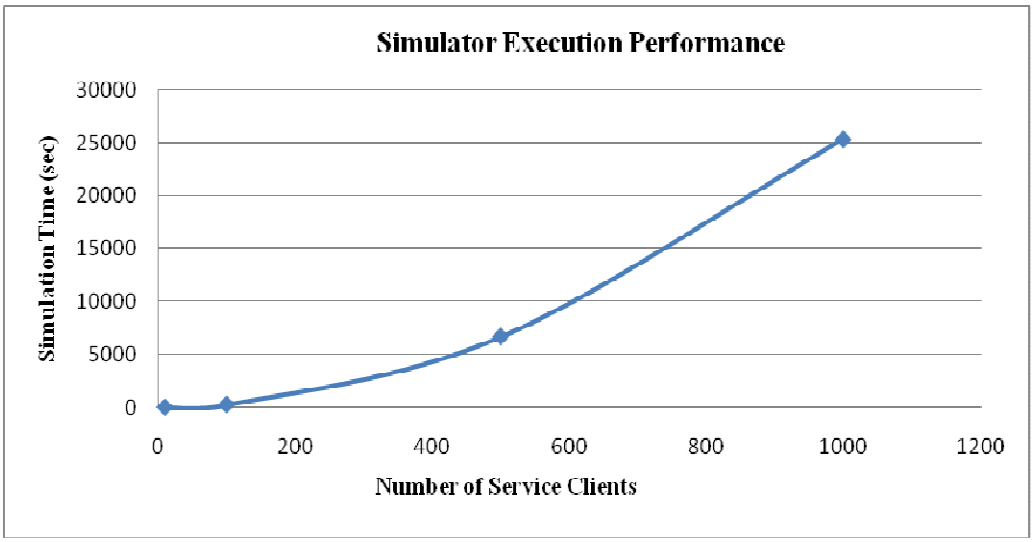


Figure 65. Execution scalability of the DEVS-Suite simulator

6. Conclusion and Future Research

6.1. Conclusion

As stated previously, SOA-based software design for distributed computing systems poses new challenges to existing simulation tools. SOA enables dynamic composition of different types of services as needed. Dynamic service composition requires the development of high quality SBS that can simultaneously satisfy multiple QoS features. To achieve this goal, QoS Monitoring and Adaptation sub-systems are needed to collect and analyze tradeoffs between multiple QoS features and adapt the composition of services accordingly.

To develop the ASBS framework and support design, implementation, and testing of its Monitoring and Adaptation sub-systems, a suitable SOA-based simulation framework referred to as SOAD has been developed. We developed a set of service abstractions – service broker, service client, service provider, and their relationships, such as the service provider must publish its service to the service broker before being subscribed and the client only can find out the service provider via service broker. In addition to these SOA elements and relationships, a set of message types corresponding to WSDL and SOAP in the real SOA are also developed to comply with the SOA framework. Since SOAD simulator should account for the both hardware and software aspects of SOA, simple hardware components such as a network link is modeled. In addition a set of transducer models have been developed to collect data on services and network links.

Since current existing simulation environments are not aimed to support early creation of designs for ASBS, we developed the new DEVS-Suite simulation

environment which extends the Tracking Environment and DEVSJAVA simulators. The capability to track, animate, and plot time-based simulation data sets helps analyze the dynamics of adaptive service-based software systems.

Models were developed for a voice communication system and a travel agency system. A real voice communication system was used to develop their simulated counterparts. The VCS simulation models demonstrated the ability to develop alternative design configurations and evaluating their dynamics using DEVS-Suite. The resulting SOAD simulator demonstrated creation and validation of different simulation models and scenarios which is key for evaluating alternative adaptive service-based software systems in terms of their quality of service attributes.

6.2. Future Research

The current SOA-based simulation models do not support service-based software systems where services can be added or removed at run-time. Since the adaptation system in the ASBS requires dynamic service composition at run time, it is important for the SOAD simulation model to change its structure dynamically by adding or removing service models. Dynamic Structure DEVS modeling (Barros, 1997) is suitable to be incorporated into the SOAD simulator. As we stated, the SOAD should account for the both the hardware and software aspects of SOA and the DEVS/DOC, a software/hardware co-design approach has been developed (Hild et al., 2002; Hu, 2007). The SOAD should be extended with the DEVS/DOC capabilities so that details of hardware components can be modeled and simulated which in turn can provide a richer basis for the Monitoring sub-system. Furthermore, it is important for the SOAD simulator

to be integrated with the Monitoring and Adaptation sub-systems in order to have a testbed that can support ASBS design and simulation-based testing.

References

ACIMS. (2001). Arizona Center for Integrative Modeling and Simulation. 2007, from http://www.acims.arizona.edu/SOFTWARE

Anderson, C., Rothermich, J. A., & Bonabeau, E. (2005). *Modeling, quantifying and testing complex aggregate service chains.* Proceedings of the 2005 IEEE International Conference on Web Services, Orlando, Florida, USA.

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

Chang, H., Song, H., Kim, W., Lee, K., Park, H., Kwon, S., et al. (2005). Simulation-Based Web Service Composition: Framework and Performance Analysis. In *Systems Modeling and Simulation: Theory and Applications,* pp. 352-360.

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

DEVS-Suite (2008), Computer Science and Engineering Department, Arizona State University, from http:// acims1.eas.asu.edu/WebStarts.

Elamvazhuthi, V. (2008). *Visual Component-Based System Modeling with Automated Simulation Data Collection and Observation.* Unpublished master's thesis. Arizona State University, Tempe, AZ.

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

Hild, D. R., Sarjoughian, H. S., & Zeigler, B. P. (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., Taku, F., Toshiyuki, M., & Sadatoshi, K. (2006). *A UML Simulator for Behavioral Validation of Systems Based on SOA.* International Conference on Next Generation Web Services Practices.

HLA. (1999). High Level Architecture. http://hla.dmso.mil [cited 2006]: Defense Modeling and Simulation Office.

Hu, W. (2007). *Visual and Persistent Co-Design Modeling for Network Systems.* Doctoral Dissertation. Arizona State University, Tempe, AZ.

John, G., John, H., Lei, L., & Na, L. (2006). *Performance engineering of service compositions.* 2006 international workshop on Service-oriented software engineering, Shanghai, China.

Kim, S., Sarjoughian, H. S., Flasher, R., & Elamvazhuthi, V. (in preparation). *DEVS-Suite: A Component-based Simulation Tool for Rapid Experimentation and Evaluation.*

Mather, J. (2003). *The DEVSJAVA Simulation Viewer: A modular GUI that visualizes the structure and behavior of hierarchical DEVS models.* University of Arizona, Tucson, AZ.

OASIS. (2003). OASIS UDDI Specifications TC. http://www.oasis-open.org/specs/#uddiv3.0.2.

Ramaswamy, M. (2008). *System Theory Based Modeling and Simulation of SOA-based Software Systems.* Unpublished master's thesis. Arizona State University, Tempe, AZ.

Roontiva, A., Huang, D., Xu, X., Ye, N., & Yau, S. S. (in preparation). *QoS Performance Models of Cause-Effect Dynamics for QoS of Voice Communication Service: Towards Adaptive Service-based Systems*.

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

Sarjoughian, H., Kim, S., Ramaswamy, M., & Yau, S. (2008). *A Simulation Framework for Service-Oritented Computing Systems.* Proceedings of the 2008 Winter Simulation Conference.

Sarjoughian, H. S. (in preparation). *A Unified Logical, Visual, and Persistent Component-based Modeling Framework.*

Sarjoughian, H. S., & Singh, K. R. (2004). *Building Simulation Modeling Environments Using Systems Theory and Software Architecture Principles.* the Advanced Simulation Technology Symposium, Washington DC, USA.

Singh, R., & Sarjoughian, H. S. (2003). Software Architecture for Object-Oriented Simulation Modeling and Simulation Environments: Case Study and Approach.

SOAP. (2003). Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/: W3C.

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

Srinivas, N. (1999). *Reasoning About Actions in Narrative Understanding*. Paper presented at the Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence.

TouchGraph. from http://www.touchgraph.com/

Tsai, W. T., Chun, F., Yinong, C., & Paul, R. (2006). *DDSOS: a dynamic distributed service-oriented simulation framework.* Simulation Symposium, 2006. 39th Annual.

Tsai, W. T., Fan, C., & Chen, Y. (2006). *DDSOS: a Dynamic Distributed Service-Oriented Simulation Framework.* 39th Annual Simulation Symposium, Huntsville, AL, USA.

W3C. (2007). SOAP Version 1.2 Part 1: Messaging Framework. from http://www.w3.org/TR/soap12-part1/

Wikipedia. (2006). Department of Defense Architecture Framework. Retrieved April 20, 2008, from http://en.wikipedia.org/wiki/Department_of_Defense_Architecture_Framework

WinterSim. (2004). Future of Simulation. 2004 Winter Simulation Conference, Washington DC, USA

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

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

Zeigler, B. P., & Sarjoughian, H. S. (2003). *Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models.* from http://www.acims.arizona.edu/PUBLICATIONS/publications.shtml.