A Petri Net-Based Specification Model towards Verifiable Service Computing

Jia Zhang
Department of Computer Science
Northern Illinois University
DeKalb, IL 60115, USA
(312)718-2468
jiazhang@cs.niu.edu

Carl K. Chang
Deapartment of Computer Science
Iowa State University
Ames, IA 50011, USA
(515)294-4377
chang@cs.iastate.edu

Seong W. Kim
Samsung Advanced Institute of Technology
P.O. Box 111, Suwon 440-600 Korea
seongwoon.kim@samsung.com

A Petri Net-Based Specification Model towards Verifiable Service Computing

ABSTRACT

The emerging paradigm of Web services opens a new way of engineering enterprise Web applications via rapidly developing and deploying Web applications, by composing independently published Web service components to conduct new business transactions. However, how to formally validate and reason about the properties of an enterprise system composed of Web service components remains a challenge. This chapter introduces an advanced topic of enterprise service computing – formal verification and validation of enterprise Web services. The authors introduce a Web Services Net (WS-Net), which is an executable architectural description language incorporating the semantics of Colored Petri Nets with the style and understandability of the Object-Oriented concept and Web services concept. As an architectural model that formalizes the architectural topology and behaviors of each Web service component as well as the entire system, WS-Net facilitates the simulation, verification, and automated composition of Web services.

The layout of the chapter is as follows:

- Introduction
- Challenges of formal verification of Web services composition
- Related work
- Alternative tools and methodologies
- Introduction to WS-Net approach
- Conclusions and future trends

KEYWORDS

Very High-Level Languages, Modeling Languages, Petri Nets, Software Architecture, Internet-Based Technology, Web Technologies

INTRODUCTION

The emerging paradigm of Web services opens a new way of engineering enterprise Web applications. The key concept is to rapidly develop and deploy Web applications by composing independently published Web service components to conduct new business transactions. Accordingly, a Web services-oriented system refers to a Web application system that contains one ore more Web service components. In theory, a system containing Web service components may also contain some parts that are not Web services; however, in reality, every component in a Web services-oriented system is wrapped as a Web service. Thus, for the rest of this chapter, we will use the terms *service components*, *services,* and *components* interchangeably.

The existing Web services model favors the creation, registration, discovery, and composition of

distributed Web services. In this model, Web services basically adopt a triangular provider/broker/requester operational model, or so called Services Oriented Architecture (SOA), as shown in Figure 1. A service provider publishes services at a public service registry using Universal Description, Discovery, and Integration (UDDI) (UDDI 2004). The public interfaces and binding information of the registered services are clearly defined in a standard Web Service Description Language (WSDL) (WSDL 2004). Such a public service registry generally provides two interfaces: a registry interface serving service providers, and a query interface serving service requesters. As illustrated in Figure 1, published Web services are hosted by the service providers. A service requester queries the service registry for interested services registered, and obtains the binding information of the corresponding service provider. Then the service requester binds to the service provider, and remotely invokes the services from the service provider through a lightweight messaging protocol - the Simple Object Access Protocol (SOAP) (SOAP 2003).

Although the paradigm of Web services has been extensively considered as the model of the next generation of distributed computing and Internet computing; how to formally validate and reason about the properties of an enterprise system composed of Web service components remains a challenge. As a matter of fact, the actual adoption of Web services in industry is quite slow, mainly because there lacks an established way of formally testing Web services-oriented systems (Zhang, 2005). As a research aiming at facilitating Web services composition and verification, Web Services Net (WS-Net) is an executable architectural description language, which incorporates the semantics of Colored Petri Nets with the style and understandability of the Object-Oriented concept and Web

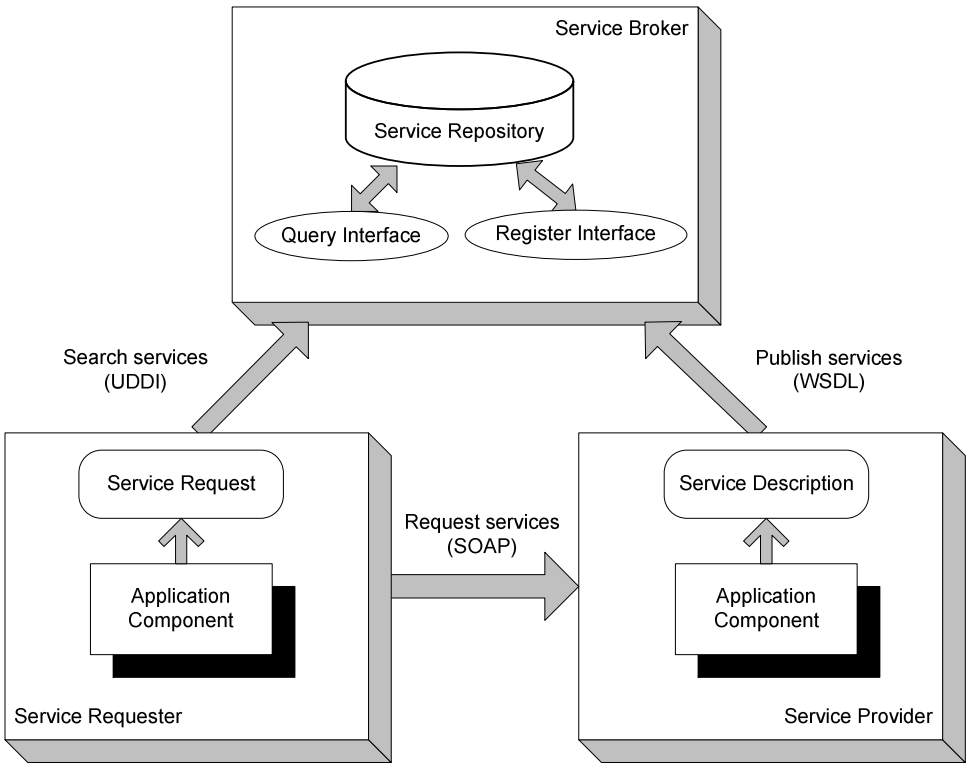

Figure 1. Service-Oriented Architecture

services concept. WS-Net describes each system component in three hierarchical layers: (1) An interface net declares the set of services that the component provides; (2) An interconnection net specifies the set of services that the component requires to accomplish its mission; and (3) An interoperation net describes the internal operational behaviors of the component. Each component maybe either an independent Web service or a composition of multiple Web services; and the whole system can be considered as the highest level component. Thus, WS-Net can be used to validate the intra-component and hierarchical behaviors of the whole system. As an architectural model that formalizes the architectural topology and behaviors of each Web service component as well as the entire system, WS-Net facilitates the simulation, verification, and automated composition of Web services. To our best knowledge, our WS-Net is the first attempt to comprehensively map Web services elements to Colored Petri nets, so that the latter can be used to facilitate the simulation and formal verification and validation of Web services composition.

The remainder of the chapter is organized as follows. First, we will introduce the state of the art of Web services composition towards services-oriented engineering. Second, we will discuss the challenges of formal Web services-oriented verification. Third, we will compare options and make selections. Fourth, we will discuss related work. Fifth, we will introduce our WS-Net approach. Finally, we will make conclusions and discuss future work.

STATE OF ART OF WEB SERVICES COMPOSITION AND CHALLENGES

State of Art of Web Services Composition

In this section, we will briefly introduce the state of the art of Web services composition.

Concept of Web Services Composition

The SOA model discussed in the section of introduction describes how to obtain a single Web service. In reality, however, a service requester typically needs to synergistically coordinate and organize multiple Web services into business processes. Web services composition thus refers to the construction process of composite services from Web services, as shown in Figure 2. It generally contains two procedures: selecting and constructing. The selecting procedure focuses on selecting qualified services, while the constructing procedure focuses on dynamically building flow structures over selected services.

Figure 2 illustrates a simplified Web services composition. The composition is conducted by a composer residing at the final environment. The composer decides to integrate four existing Web services, namely, S_1, S_2, S_3, and S_4. As shown in Figure 2, there are temporal relationships between the four services. S_1 will run first. Then depending on the output of S_1, either S_2 or S_3 will be executed. If S_2 is chosen, then S_4 will be executed afterwards. As shown in Figure 2, the four service components will be implemented by four Web services, which belong to different service providers and reside on different sites. When the composer runs the composite service in the final environment, it will remotely invoke the corresponding Web services according to the predefined scenario.

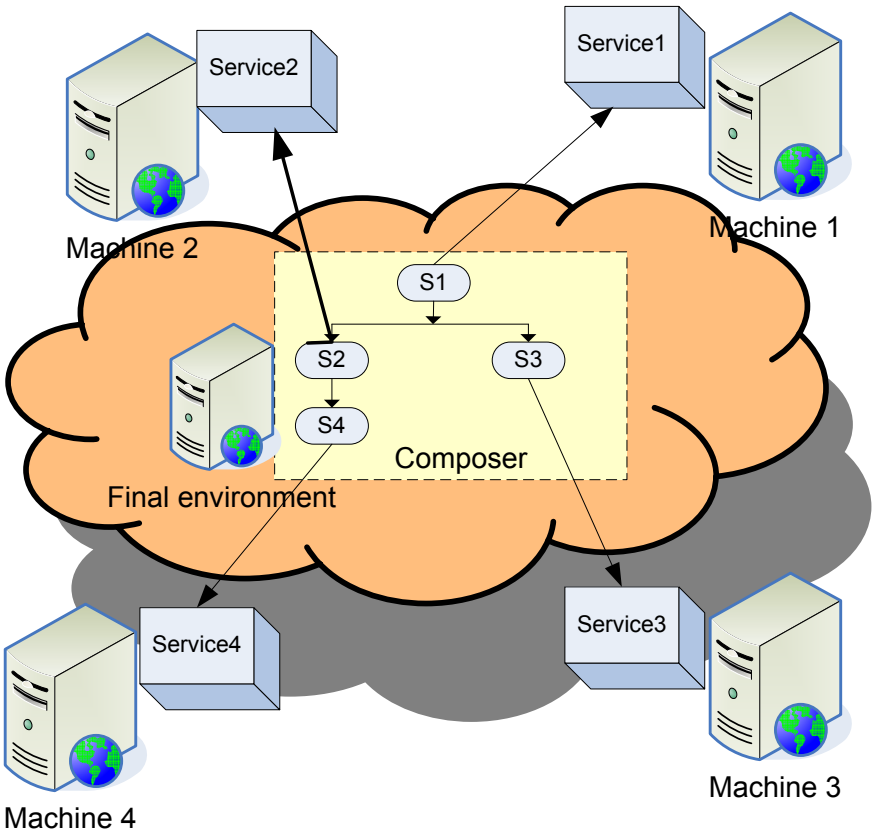

Figure 2. Web services composition

The construction of a composite Web service can be modeled by specifying a structure of service components using a service flow language, such as the BPEL4WS (BPEL4WS 2003) and Microsoft BizTalk Server (Microsoft 2003). The structure defines an e-Business process model; an invocation of the composite service is treated as an instance of the process model. Examples of composition model are eFlow (Casati, Ilnicki et al. 2000), the scenario-based service composition by Kiwata et al (Kiwata, Nakano et al. 2001), the quality-driven model by Zeng et al (Zeng, Benatallah et al. 2003; Zeng, Benatallah et al. 2004), the constraint-driven composition by Aggarwal et al (Aggarwal, Verma et al. 2004), etc.

With the rapid increase of the number of Web services published on the Internet on the daily basis, the demand for integrating heterogeneous services in an automatic or semi-automatic way becomes urgent. Efforts have been made to automate the service composition process by employing discovery agents. According to the information provided in a service request, these discovery agents generate a structure of service operations based on some registered services. Commonly used approaches for agents to make decisions are rule-based systems (Ponnekanti and Fox 2002) and ontology-based approaches (Arpinar, Aleman-Meza et al. 2004). However, to date fully automated approaches sometimes involve unavoidable unrealistic assumptions, e.g., rule-system-based approaches assume that the service requester knows the exact input and output interfaces of a desired composite service. Liang et al. thus utilized a semi-automatic approach to assist service composition (Liang, Chakarapani et al. 2004).

As shown in Figure 2, Web services composition differs from traditional application component composition in several significant ways. First, unlike in a traditional component composition where component parts are deployed in the same final environment, Web services composition is a virtual composition in the sense that all participating services will never be physically deployed into the final environment. This is because Web services are hosted by corresponding service providers and can only be used through remote invocations. Second, Web services composition implies uncertainties. Since Web services are hosted and managed by their own service providers, their availabilities may autonomously change (Zhang 2005). An available Web service on a specific day may not be available in the next day. Furthermore, the Quality of Service (QoS) and even functionalities of Web services can also change over time, due to changes of adopted technologies or business models from the corresponding service providers. Therefore, Web services composition may have to be conducted upon per-usage basis. Third, Web services composition needs to serve more dynamic changes of user requirements. This is not a new issue; however, users adopt Web services for higher flexibility and adaptability thus they hold higher expectations than before.

Architecture of Web Services-oriented Systems

Just as an architectural diagram is an essential guideline for architectural constructions, software architecture is critical for the success of a Web services-oriented system and Web services composition. According to the American Heritage Dictionary, architecture is "the art and science of designing and erecting buildings, a structure of structures collectively, a style and method of design and construction." (Mifflin 2000) In other words, software architecture plays an essential role in providing the right insights, triggering the right questions, and offering general tools for thoughts. Thus, Architecture Description Languages (ADLs) (Medvidovic and Taylor 2000) are commonly created to formally specify and define the architectural model of a software system as a guideline before construction.

As shown in Figure 3, Web services composition is a process of composing multiple Web services as components into a composite Web services-oriented system. In order to facilitate services composition, the architecture specification (e.g., an ADL) that represents a complex Web services-oriented system should capture two dimensions of design information: service dimension and architecture dimension. The service dimension focuses on specifying low-level interconnections between Web services, e.g., how to glue different service components together, how to communicate between services, how to orchestrate services, and how to choreograph services. The architecture dimension, on the other hand, focuses on specifying high-level compositional view or architectural view of a final composite service, e.g., hierarchical relationships among service components.

In more detail, a Web services-oriented ADL needs to catch the following three categories of design information: (1) the structural properties and Web services component interactions, (2) the behavioral functionality of each major high-level Web services component, and (3) the behavioral functionality of the entire system. In recent years, researchers from both industry and academia have been developing a number of Web services-oriented ADLs, typically, Web Services Description Language (WSDL) (WSDL 2004), Web Services Flow Language (WSFL) (Leymann 2001), Business Process Execution Language for Web Services (BPEL4WS) (BPEL4WS 2003), Web Service Choreography

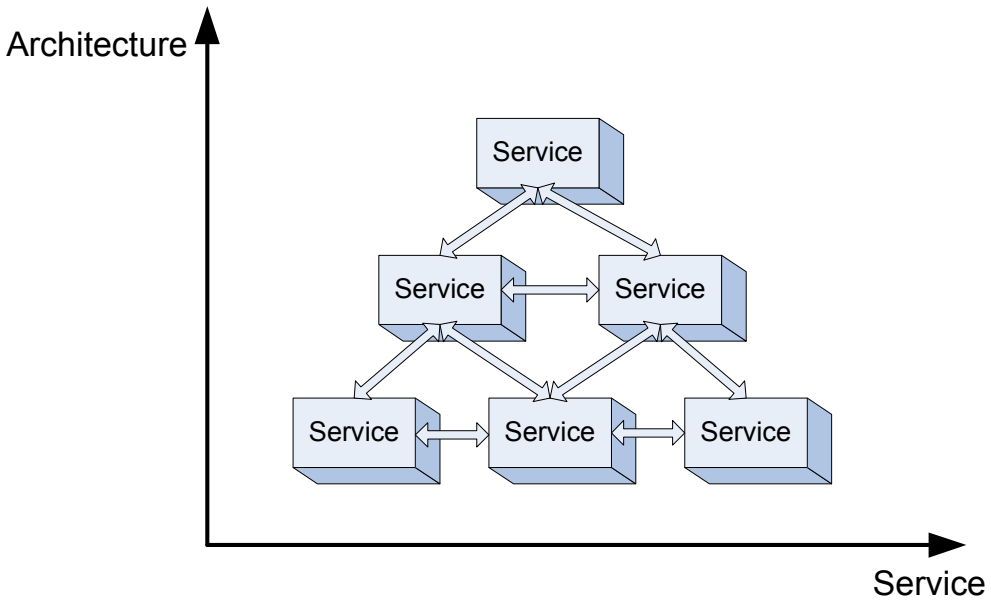

Figure 3. Two dimensions of Web services composition

Interface (WSCI) (WSCI 2002), XLANG (Thatte 2001), etc.

CHALLENGES OF FORMAL VERIFICATION AND VALIDATION OF WEB SERVICES COMPOSITION

Although Web services composition exhibits a boom to enterprise software engineering, it is not clear that this new model guarantees various Web service components can be composed and integrated seamlessly and properly (Zhang 2005). In detail, the flexibility of Web services-centered computing is not without penalty since the value added by this new paradigm can be largely defeated if, to name a few, selected Web service components do not thoroughly fulfill the requirements (i.e., functional or nonfunctional), selected Web service components act errantly in the composed environment, or if selected service components cannot collaborate harmoniously. Therefore, it is necessary to fully test a Web services-oriented system under various input conditions, and to logically verify certain maintenance and QoS conditions associated with Web services. Without ensuring the trustworthiness of Web services composition, it is difficult for Web services to be adopted in mission critical applications. In short, formally verifying and validating the correctness of logic inside of a composition of Web services-oriented systems become critical.

Nevertheless, formal verification of Web services composition is not a trivial task. The last fifty years of software development history has witnessed the establishment of an independent research branch as *software testing*. Software testing contains a wealth of theories, technologies, methodologies, and tools to guide the verification process of a component-based software product. However, formal validation and verification over Web services composition poses new challenges due to the unique features of Web services composition.

We cannot simply apply the traditional software testing technologies to formally measure and test Web services-oriented system. First, such a testing needs to be highly efficient. There are times when service components cannot be decided until run time. Testing and making decisions at runtime require efficient strategies and techniques. Second, how to test a Web services-oriented system with limited information is challenging. Current Web services interfaces expose limited information. Web services are Web components only accessible via interfaces published in standard Web services-specific interface definition languages (e.g., WSDL) and accessible via standard network protocols (e.g., SOAP). Third, such a testing may have to be performed on the per-usage basis. Service components are hosted by service providers and invoked remotely; thus, it may be questionable to assume that the services stay at the same quality over time. Fourth, Web services-oriented testing has to be highly effective. The fundamental hypothesis of the existing testing methods is that exhaustive test cases can be conducted upon the testing software product if necessary. It is neither feasible nor practical to apply this assumption to Web services testing. Unlike traditional software products that are deployed into the target environment, Web services require remote accessing. Thus, conducting a significant amount of tests on a Web service implies expensive maintenance of network connections and network transmissions, let alone unpredictable Web conditions such as traffic and safety. Finally, since it is difficult to obtain precise function descriptions of a Web service, most of the time the only feasible and practical way of testing Web services-oriented system is through simulation.

In summary, due to the specific distributed nature of Web services, these existing software testing models and methodologies deserve re-inspection in the domain of Web services composition.

RELATED WORK

Researchers have conducted significant work in the field of Web services composition description and verification.

Web Services-oriented ADLs

In recent years, researchers from both industry and academia have been developing a number of Web services-oriented Architecture Description Languages (ADLs), typically Web Services Description Language (WSDL) (WSDL 2004), Web Services Flow Language (WSFL) (Leymann 2001), Business Process Execution Language for Web Services (BPEL4WS) (BPEL4WS 2003), Web Service Choreography Interface (WSCI) (WSCI 2002), XLANG (Thatte 2001), etc.

The common similarity is that all of these ADLs are built upon eXtensible Markup Language (XML) (XML) technology, which is extensively considered as a universal format for structured documents and data definitions on the Web. Among them, WSDL is the basis of other work. Intending to formally and precisely define a Web service, WSDL from W3C (http://www.w3c.org) is becoming the *ad hoc* standard for Web services publication and specification. However, it has been widely admitted that WSDL can only specify limited information of a Web service, such as function names and limited input and output information. In recognition of this problem, researchers have been developing other description languages to extend the power of WSDL to depict system architecture.

The following are some outstanding examples:

- Web Services Flow Language (WSFL) is a WSDL-based language focusing on describing the interactions between Web service components. WSFL defines the interaction patterns of a collection of Web services, as well as the usage patterns of a collection of Web services in order to achieve a specific business goal.

- The Business Process Execution Language for Web Services (BPEL4WS) specifies an interoperable integration model aiming to facilitate the automatic integration of Web service components. BPEL4WS formally defines a business process and process integration.

- Web Service Choreography Interface (WSCI) utilizes flow messages to define the relationships and interactions between Web service components. According to WSCI, a Web service component exposes both a static interface and a dynamic interface when it participates in a message exchange with other Web service components.

- XLANG considers the behaviors of a system as an aggregation of the behaviors of each Web service component. Therefore, XLANG specifies the behaviors of each Web service component independently. The interactions between Web services are conducted via message passing, which is expressed as the operations in WSDL.

However, these ADLs either merely focus on static functional descriptions of Web service components as a whole (e.g., WSDL), or concentrate only on the behavioral integrations between Web service components (e.g., BPEL4WS). In addition, these ADLs focus on topological descriptions and concentrate on describing interactions between Web service components. They lack the capability to describe the hierarchical functionality of the components. Moreover, there is little concern about expressing dynamic behaviors of the defined system. Furthermore, none of these current ADLs support dynamic verification and monitoring of the system integrated. Contrast to their work, our research focuses on supporting both static and dynamic Web services-based composition.

Chang and Kim proposed I^3 (Chang and Kim 1999), which is a layered executable architectural model defining component-based systems. However, I^3 is based upon the Structural Analysis and Design Technology (SADT) (Ross 1984), which is a traditional functional decomposition- and data flow-centered methodology. In contrast with I^3, our WS-Net aims at integrating Colored Petri Nets (CPN) with the style and understandability of the Object-Oriented paradigm. In addition, I^3 intends to present a generic specification model oriented to generic component-based software systems. WS-Net, on the other hand, focuses on Web services-oriented system architecture and seamlessly integrates with WSDL and XML technology. In other words, although WS-Net was strongly influenced by EDDA (Trattnig and Kerner 1980) and I^3, we have enhanced the state of the art by supporting modern software engineering philosophy equipped with Object-Oriented and component-based notations. Furthermore, WS-Net is applied to Web services-oriented systems, as well as integrated with the *ad hoc* Web services standards, such as WSDL and XML.

Formal Verification Work on Services-oriented Systems

Narayanan and Mcllraith proposed to use Petri nets as tools to simulate, verify, and automate Web services composition (Narayanan and McIlraith 2002). Their Web services composition refers to the composition of programs into a Web service, and does not consider the composition of Web services into new Web services. Contrast with their work, our research covers both inter-Web services and intra-Web services composition. In detail, our Interconnection Net simulates and validates the interactions and composition among Web services. The Interoperation Net simulates and validates the composition inside of a Web service, which may or may not contain other Web services. Moreover, contrast to their work that does not provide detailed mapping between Web service elements and Petri nets elements, our approach provides a direct mapping between the two methodologies. Thus, our approach can be used as a guidance to construct Petri nets for Web services composition. Furthermore, unlike related work, WS-Net itself is an architectural description language that can facilitate hierarchical Web services composition description and definition, in addition to simulating Web services composition.

Pi-Calculus, one form of process algebra, was the theoretical basis of the precursors of the *ad hoc* services composition definition language BPEL4WS (Pi-Calculus). The fundamental entity in Pi-Calculus is a process, which can be an empty process, an I/O process, a parallel composition; a recursive definition; or a recursive invocation. Describing services in such an abstract way facilitates reasoning about the composition's correctness through reduction. Pi-Calculus enables verification of liveness and behavioral properties. Salaun et al. adopted process algebraic notations to describe Web services and the inter-services interactions at an abstract level (Salaun, Bordeaux et al. 2004). Then they performed reasoning about the correctness over the services composition through simulation and property verification. They also explored the links between abstract descriptions and concrete descriptions. From the experience out of a sanitary agency case study, they found that process algebras are adequate to describe and reason about services composition, especially to ensure composition correctness. Furthermore, Ferrara defined a two-way mapping between abstract specifications written in process algebraic notations and executable Web services written in BPEL4WS (Ferrara 2004). In addition to temporal logic model checking, Bordeaux et al. adopted bisimulation analysis to verify the equivalent behaviors between two Web services (Bordeaux, Salaün et al. 2004). Contrast with their work based upon process algebra, our WS-Net roots in Petri nets that are more suitable to simulate and reason about large-scale Web services composition and verification. In addition, our WS-Net is an architectural description language that supports hierarchical description and definition.

Some researchers focus on adopting finite state automata to verify Web services composition. Bultan et al. uses finite state automata to model the conversation specification thus to verify Web services composition (Bultan, Fu et al. 2003; Fu, Bultan et al. 2005). Their approach models each service component as a Mealy machine, which is a Finite State Machine (FSM) with input and output. Service components communicate to each other through asynchronous messages. A conversation between service components is modeled as a sequence of messages. Each service component has a queue to hold messages; while a centralized global "watcher" keeps track of all messages in the whole composite system. Berardi et al. proposed to describe a Web service's behaviors as an execution tree and then translates it into an FSM (Berardi, Calvanese et al. 2003). An algorithm was also presented to check whether a possible composition exists, i.e., a composition will finish in finite number of steps. Contrast with their work based upon finite state automata, our WS-Net is based upon Petri nets that are more suitable to verify large-scale Web services composition. In addition, our WS-Net is an

architectural description language that supports hierarchical definition of large-scale Web services composition and early-stage validation.

ALTERNATIVE TOOLS AND METHODOLOGIES

In this section, we will first discuss briefly alternative tools and methodologies, together with how they can be used to model Web services composition. By providing their comparisons, we will discuss our selection on Petri nets technology together with reasons.

Currently there are generally three alternative techniques to formally verify Web services composition: (1) Petri nets, (2) process algebras, and (3) finite state automata.

Petri Nets

Petri nets is a well-established abstract language to formally model and study system composition (Jensen 1990). In general, a Petri net is a directed, connected, and bipartite graph with two kinds of node types: places and transitions. The nodes are connected via directed arcs; and tokens occupy places. When all the places pointing into a transition contain an adequate number of tokens, the transition is enabled and may fire; thus the transition removes all of its input tokens and deposits a new set of tokens in its output places.

Web services can be modeled as Petri nets by assigning transitions to services and places to inputs/outputs. Each Web service has its own Petri net, which describes service behaviors. Each service may own either one or two places based upon the nature of the service: (1) one input place only, (2) one output place only, or (3) one input place and one output place. At any given time, a service can be in one of the following states: not initiated, ready, running, suspended, or completed. After defining a net for each service, services can be composed by applying various types of composition operators between nets: sequence, choice, iteration, parallel, etc. These composition operators will orchestrate nets in different execution patterns.

After modeling Web services composition with Petri nets, one can investigate the generated Petri nets to verify system properties, such as deadlocks or nondeterministic status.

Process Algebras

Process algebras are formal description techniques that use temporal logic model to specify and verify component-based software systems, especially their concurrency and communication. Process algebra is known for its ability of describing dynamic behaviors, compositional modeling, and operational semantics, as well as its ability of behavioral reasoning via model checking and process equivalences. The central unit of process algebras is *process.* A process is an encapsulated entity that contains *state*. Different processes communicate with each other via *interactions.* An *action* (e.g., shipping an order) initiates an *interaction.* A set of processes can form a larger system.

Process algebras can be adopted to model Web services composition. Each Web service can be

modeled as a process. By studying algebraic services composition, one can verify composition properties such as safety, liveness, and resource management. As a matter of fact, Pi-Calculus, one form of process algebra, was the theoretical basis of Business Process Modeling Language (BPML) and XLANG, two precursors of the *ad hoc* services composition definition language BPEL4WS (Pi-Calculus).

In addition to temporal logic model checking, process algebras contains bisimulation analysis that can be used to verify whether two processes have equivalent behaviors, i.e., whether one service includes behaviors of another (Bordeaux, Salaün et al. 2004). Thus, bisimulation analysis can be used to decide whether two Web services are replaceable, as well as the redundancy of services.

Finite State Automata

A Finite State Machine (FSM) or so-called finite automaton is a well-established model of behaviors composed of states, transitions, and actions. A state stores information; a transition indicates a state change and is guarded by a condition that is required to be fulfilled; an action is a description of an activity to be performed at a given time point. Generally, four types of actions can be identified: entry action, exit action, input action, and transition action.

Finite State Machines can be adopted to represent the aspects of a global composition process. In detail, Web services composition specification can be described using temporal logic; then the FSMs model can be traversed and checked to verify whether the workflow specification holds. Specially, this approach can be used to verify data consistency, unsafe state avoidance (deadlock), and business-constraint satisfaction. Current research efforts along this direction can be further categorized into two groups: conversation specification and automatic services composition (Milanovic and Malek 2004). Research on the former one focuses on using Mealy machines to model asynchronous messages between service components, thus verify the realizability of a services composition specification. Research on the latter one focuses on modeling service behaviors as an execution tree, and then translating it into an FSM. The generated FSM then can be checked to verify whether a possible composition exists, i.e., whether it can be finished in finite number of steps.

Comparisons

Each of the three alternative techniques exhibits advantages and disadvantages over simulating and verifying Web services composition, as shown in Figure 4.

Petri nets is a modeling approach that utilizes graphical representation to specify and simulate events and state changes. It has been widely used as a graphical formal modeling tool for systems involving communication, concurrency, synchronization, and resource sharing. However, Petri nets also has disadvantages. First, Petri nets are more expressive than Finite State Machines (Peterson 1981). Second, such a kind of graph is easy to understand when the graph contains small amount of elements. However, if the modeled system is large-scale and complex, the net will quickly grow large, and the graph will become too complicated to understand and extract useful information from it. High-level hierarchical Petri nets provide scalable support for modeling large-scale systems, such as Hierarchical CP-nets (HCP-nets). Third, there is no direct and explicit mapping between Petri nets constructs and

	PN	PA	FSM
Formal	H	H	H
Theoretical foundation	H	H	H
Analysis ability	H	H	H
Handle complexity	H	-	-
Model static properties	H	H	H
Model dynamic properties	H	-	-
Graphical ability	H	-	-
Simulation	H	-	-
Tools	H	-	-
User effort	L	H	H
Concurrency	H	-	-
Deadlock detection	H	-	-
(a)synchronization	H	-	-
Process concept	I	E	E

H: High
L: Low
E: Enabled
I: invalid

Figure 4. Comparisons among PN/PA/FSM

Web services composition. For example, there is no concept of delimited process in Petri nets. Thus, one needs to find an appropriate implicit mapping between Petri nets constructs and Web services concepts, e.g., each Web service can be mapped to a *transition* in Petri nets, and its input and output can be mapped to *places* in Petri nets.

Process algebras can model services composition in a simple and straightforward description. Each Web service component can be naturally modeled as a process, which is the fundamental entity of process algebras. Process algebra facilitates formal specification of message exchanges between Web service components. By studying the interactions between modeled processes, one can verify the services composition in a natural way. However, there exists no tool to automatically investigate and simulate generated algebraic services composition. Manual analysis through algebraic reduction can be both time consuming and error prone.

The same issue exists for finite state automata, although FSMs are natural to model services composition through state modeling and transitions. Enduring the same situation as for process algebras, there is no tool existed to automatically investigate and simulate generated FSMs for services composition. This approach requires users to capture composition properties with temporal

logic formula. Manual analysis is impractical for dynamic and efficient verification, which is a compulsory requirement for services composition verification.

In summary, as shown in Figure 4, all Petri nets, process algebras, and finite state machines can perform formal verification and possess analysis ability, and they all root in theoretical foundations. With the Hierarchical CP-nets (HCP-nets), high-level Petri nets can support more complexity. All Petri nets, process algebras, and finite state machines provide powerful mechanisms to model static system properties. However, only Petri nets support dynamic properties modeling. In addition, Petri nets provide available graphical tools for simulation, while the other two approaches do not. These tools provide powerful facility to analyze concurrency, deadlock, and asynchronous behaviors. Thus, user efforts involved can be relatively low, comparing to those involved in process algebras and finite state machines. However, Petri nets do not specify explicit concept of process; thus, modeling process-centric Web services composition requires special mapping mechanisms between Petri nets concepts and Web services concepts.

Our Selection

In our research, we chose to adopt Petri nets due to its combination of (1) rich computational semantics, (2) ability to formally model systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and stochastic, (3) ability to conduct quantitative performance analysis, and (4) availability of graphical simulation tools. Petri nets also have natural representation of changes and concurrency, which can be used to establish a distributed and executable operational semantics of Web services. In addition, Petri nets can address offline analysis tasks such as Web services static composition, as well as online execution tasks such as deadlock determination and resource satisfaction. Furthermore, Petri nets possess natural way of addressing resource sharing and transportation, which is imperative for the Web services paradigm. Moreover, Petri nets have a set of available graphical simulation tools, represented by Design/CPN (Meta Software Corporation 1993).

INTRODUCTION TO WS-NET APPROACH

In this section, we will introduce our Web Services Net (WS-Net). First, we will provide a brief overview of Petri nets, focusing on their notions for high-level architectural design. We will also briefly describe a couple of high-level Petri nets techniques that will be used in our approach, such as Colored Petri Nets (CPN) and Hierarchical CP-nets (HCP-nets).

Overview of Petri Nets

The concept of Petri nets was coined by Carl Adam Petri in his Ph.D. thesis in 1962 (Petri 1962). Rooted in a strong mathematical foundation, Petri nets is a well-known abstract language to formally model and study systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic (Jensen 1990). In general, a Petri net is a bipartite graph with two kinds of node types: places and transitions. The nodes are connected via directed arcs, and tokens occupy places. Graphically, circles represent places; rectangles represent transitions. All places

holding zero or more tokens together exhibit a specific state of the system at a moment. Transitions can change the state of the system: when all the places pointing into a transition contain an adequate number of tokens, the transition is enabled and may fire; thus the transition removes all of its input tokens and deposits a new set of tokens in its output places.

In other words, transitions are active components. They model activities that can occur (e.g., the transition *fires*), thus changing the state of the system (e.g., the marking of the Petri net). Transitions are only allowed to fire if they are *enabled*, meaning that all the preconditions of the activity must be fulfilled (e.g., there are enough tokens available in the input places). When the transition fires, it removes tokens from its input places and adds some at all of its output places. The number of tokens removed/added depends on the cardinality of each arc. The interactive firing of transitions in subsequent markings is called token game.

Formally, a Petri net is a triple PN = (P, T, F) composed of:

1. P is a finite set of places, $P = \{p_1, p_2, \ldots, p_m\}$
2. T is a finite set of transitions, T = a.
3. F is a set of arcs that represent the flow relation between places and transitions. F contains both input relations that map each transition from a set of places, and output relations that map each transition to a set of places, $F = (P\mathrm{X}T) \bigcup (T\mathrm{X}P)$.

If we use Petri nets to model a system, the transitions model the active part of the system, the places model the passive parts, and the markings describe the system states. The arcs of a graph are classified into three categories: input arcs, output arcs, and inhibitor arcs. Input arcs are arrow-headed arcs from places to transitions; output arcs are arrow-headed arcs from transitions to places; inhibitor arcs are circle-headed arcs from places to transitions.

Colored Petri Nets (CPN) (Jensen 1990) extend the Petri nets to model both the static and dynamic properties of a system. The notation of CPN introduces the notion of token types; namely tokens are differentiated by colors that may be arbitrary data values. Each place has an associated type, which determines the kind of data that the place may contain. The graphical part of CPN depicts the static architectural structure of a system. Combined with other powerful elements such as colored tokens and simulation rules, CPN is highly powerful in modeling dynamic behaviors of a system.
Formally, a CPN is a tuple CPN = ($\sum$, P, T, A, N, C, G, E, I) composed of:

1. $\sum$ is a finite set of non-empty types, called color sets.
2. P is a finite set of places.
3. T is a finite set of transitions.
4. A is a finite set of arcs such that:
   $P \cap T = P \cap A = T \cap A = \emptyset$.
5. N is a node function. It is defined from A into $(P\mathrm{X}T) \bigcup (T\mathrm{X}P)$.
6. C is a color function, which is defined from P into $\sum$.
7. G is a guard function, which is defined from T into expressions such that:
   $\forall t \in T, [Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \sum$
8. E is an arc expression function, which is defined from A into expressions such that:
   $\forall a \in A, [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \sum$
   where p(a) is the place of N(a).

9. I is an initialization function, which is defined from P into closed expressions such that: $\forall p \in P, [Type(I(p)) = C(p)_{MS}]$

In order to handle large-scale and complex systems, several variants of Petri nets emerged. The Hierarchical CP-nets (HCP-nets) were created to manage the complexity of large-scale systems by providing facility to specify the hierarchical relationships between nets. Hierarchical Colored Petri Nets (HCPN) introduce a facility for building Petri nets from subnets or modules. The HCPN theory intends to allow the construction of a large-scale model by using a number of small Petri nets, which are related to each other in a well-defined and well-organized manner.

Earlier researchers have conducted a large amount of work to utilize CPN to model system architecture. EDDA (Trattnig and Kerner 1980) combines Petri nets and SADT technology for high-level system specifications. Although EDDA successfully combines the semantics of Petri-nets with the syntax of SADT, it lacks the ability to specify modern software systems, as EDDA does not embody the Object-Oriented paradigm and the component-engineering concept (Chang and Kim 1999). Pinci and Shapiro presented an automatic mechanism to translate SADT diagrams into Hierarchical CP-nets (HCP-nets), and in turn to convert HCP-nets into Standard ML-executable code (Pinci and Shapiro 1990). This SADT-like Petri nets-based system specification suffers the same problems faced by EDDA, due to the rigid structural nature of SADT and its lack of Object-Oriented concepts.

WS-NET

As we discussed in the previous sections, formal verification of Web services composition is highly in demand, and we decided to adopt the Petri nets technology for the specification and reasoning. Meanwhile, Architectural Description Languages are essential to facilitate services composition design. Therefore, we combine these two demands – we introduce an ADL that enables hierarchical formal verification of Web services-oriented system composition.

In detail, our goal is to enable formal defining and automated reasoning technology to describe, simulate, test, and verify Web services composition. In other words, we intend to establish a formal modeling and specification framework for Web service composition. Our approach is to introduce a Web Service Net (WS-Net), which is a Petri nets-based executable architectural specification language.

WS-Net specifies a Web services-oriented software system as a set of connected architectural components described as nets. The architectural components correspond to the functional units in the system, and one architectural component may in turn be composed of multiple smaller architectural components. The entire system can be viewed as the highest-level architectural component. Each architectural component is either statically or dynamically realized by a Web service component. Architectural components are connected to each other via XML message passing through Simple Object Access Protocol (SOAP), the *ad hoc* transportation standard in the realm of Web services. The message passing mechanism mediates the interactions between architectural components via the rules, which regulate the component interactions. In our model, we will use the concept of connector (Shaw, DeLine et al. 1995) in CPN to model message passing.

As shown in Figure 5, WS-Net defines each architectural component in a three-aspect specification: (1) interface net, (2) interconnection net, and (3) interoperation net. The interface net declares the services to be provided by each Web service component; the interconnection net specifies the Web services to be acquired from other Web service components in order to accomplish its own mission; the interoperation net describes the functionality of each Web service component and the entire system in terms of control flow and data flow.

Each component must have one interface net definition and it can be accessed only via the defined interface. The definition of the interface net follows that of WSDL. The interconnection net specifies the operations to be acquired from other Web services to perform its execution. It is possible for a service component not to have an interconnection net, in which case the service component is self-sufficient and does not need support from other services to conduct its mission. In the interconnection net, each operation required is further specified by a set of foreign transitions, which represents the operations of other components. In other words, the interface net identifies each component in the system as a unique functional object, and the interconnection net specifies the relationships between components. As a result, we can visualize the entire topological view of a system by interconnecting each of the interconnection nets according to our unique component-interconnection technique, which will be discussed in later sections. The interoperation net describes the dynamic behaviors of a service component by focusing on its internal operational nature. The goal of the interoperation net is to dissect each operation into fundamental process units, which, taken together, define the required functional content of the operation. Each transition representing an operation of the component is decomposed into sub-transitions representing fundamental process units. Control flow and data flow

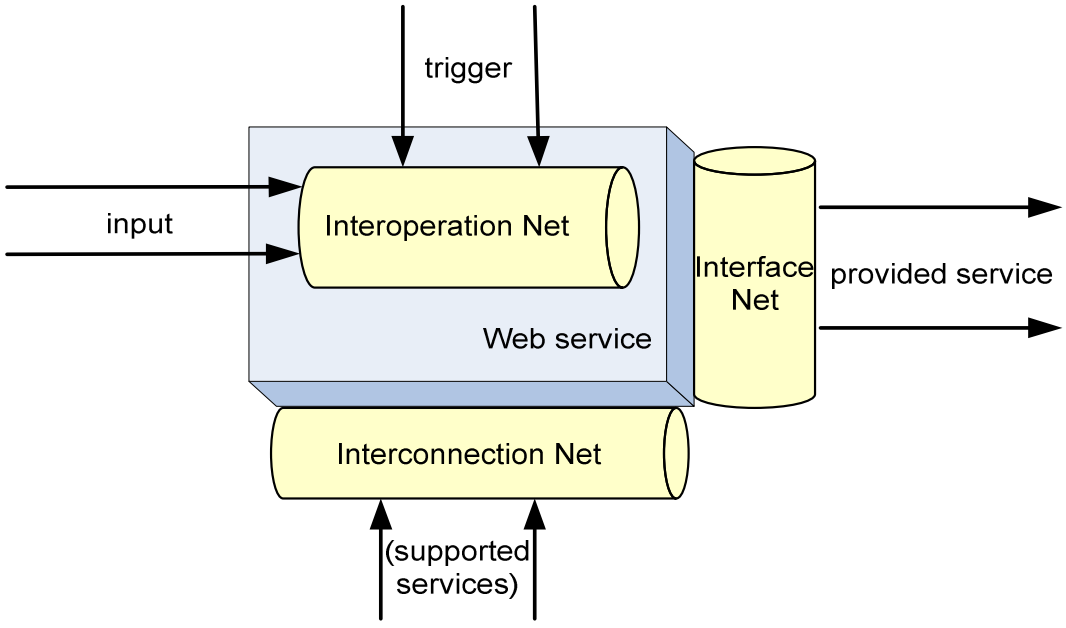

Figure 5. WS-Net

are used for describing inter-communications between process units.

As illustrated in Figure 5, WS-Net covers various aspects of a Web service. The interface net publishes a Web service to outer world. The interconnection net specifies necessary support. The interoperation net not only handles the internal workflow of the Web service, but also handles incoming requests. As shown in Figure 5, a Web service typically adopts a passive invocation mode; meaning that the Web service will stay idle until triggered (i.e., requested). When a Web service is triggered (i.e., invoked), it will also accept incoming input arguments to start the interoperation net. In summary, the three nets of WS-Net together can represent a Web service in a large-scale system and system composition.

To facilitate our discussions, throughout the rest of the chapter, we will use a simplified typical Web services messaging processing model as an example to elucidate the fundamental idea of WS-Net. As shown in Figure 6, this example illuminates a messaging system that provides a messaging service for users. The system is centered around a central storage place that acts as a message queue. All users submit messages to the message queue; and retrieve messages from the message queue. Three distributed Web service components are identified in the system: the message queue, a commit engine, and a message composer. As shown in Figure 6, if a message sender wants to publish a message, he will send the message to the commit engine that stores the message to the message queue. When a user wants to retrieve messages, the message composer will search the message queue on behalf of the user according to his profile, fetch messages, and compose them into a package before sending back to the user. In this example, different components interact with each other via SOAP.

Interface Net

Constructing a WS-Net specification starts from identifying the architectural components from the

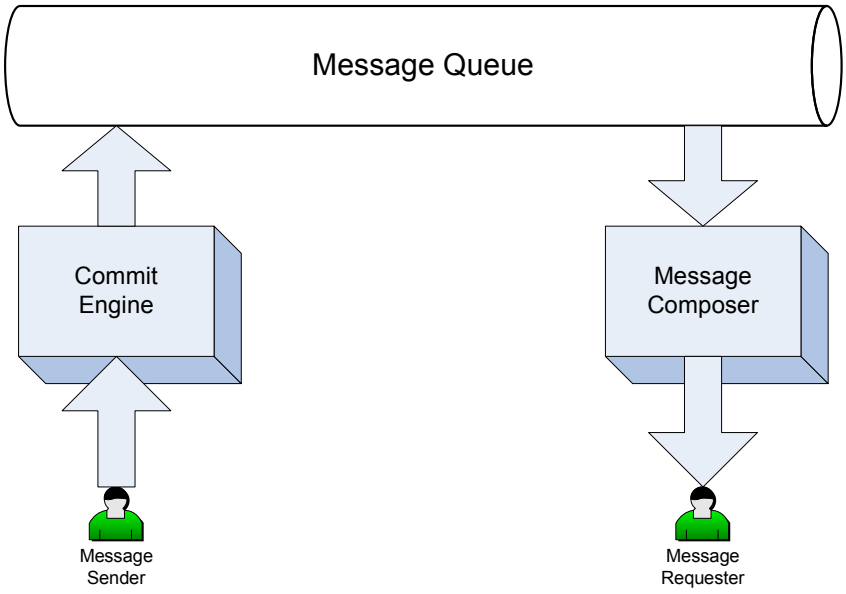

Figure 6. Simplified message queue example

system specification using a top-down approach. We use the term architectural component here instead of a service component, meaning that an architectural component may refer to the whole system, a sub-system, or a single service component. This generic feature enables the interface net to be applied to various levels of system composition. The interface net of a Web service defines expected responsibilities, or features, of each architectural component by specifying its interface as a set of semantically related services (i.e., operations) provided. The interface here denotes the names of the services provided and their signature information for invoking the services. A Web service can be accessed only through its interface. In an interface net, each service is modeled as a transition in Petri nets notation. Therefore, a transition is called a service transition in WS-Net. The name of a service transition refers to the service to be provided by the corresponding service component. Each service transition has an input place called input port where the service receives invocations; and an output place called output port where the result of a service is placed before being returned to the service requester. In other words, a Web service adopts an asynchronous communication mode. The interface net is complied with WSDL.

A WS-Net specification of an architectural component j can be denoted as C_i. $C_i \in C$, where C is the set of all Web service components identified in the software architecture. The interface net of the component C_i can be represented as:

$$C_i = \bigcup_j S_{ij}, S_{ij} \in S_i$$

where S_i is a set of services provided by C_i. Each service $S_{ij} \in S_i$ is represented by a tuple

$S_{ij} = (PI_{ij}, PO_{ij}, T_{ij}, A_{ij}, c)$

where PI_{ij} and PO_{ij} are the input port and the output port of S_{ij}, respectively.

Here we use *port* instead of *place* for compatibility with WSDL specifications, so as to facilitate automatic translation from WSDL specification to WS-Net. T_{ij} represents the service S_{ij} as a transition; A_{ij} represents the input and output arcs of the transition T_{ij}, and c is a color function for the place. The color inscriptions of the place represent the signature information of the operations as used in CPN.

The goal of the interface net is to define the provided services that are provided by the service component. The names attached to the service transition inscription represent the names of the services. Note that the names of the places and transitions are the labels to identify the places and transitions, and they are not considered as semantic inscriptions of Petri nets (Meta Software Corporation 1993). Instead, as defined in Colored Petri Nets (Jensen 1990), they are used to help designers understand the specifications and to support hierarchical composition of pages, such as transition substitution and place fusion. The signature information of each service can be described by color inscription on input and output ports (i.e., CPN places). However, it is not important to specify the detailed data structure at this stage of the design. The main purpose of coloring places is to help people understand the usage of a service component at the architectural level in the whole system. Therefore, the only imperative information to be specified is what kind of information needs to be provided to invoke the service of the service component.

Using our message queue example, the interface net specification of the *message queue* component is illustrated in Figure 7. The *message queue* provides two services: *enqueue service* and *dequeue service*, which accept user submitted messages and retrieve user message requests respectively. In order for the services to be invoked, the corresponding input ports of the services must receive proper tokens. *Enqueue service* receives an *item* token and returns a Boolean *result* as either success or fail; *Dequeue service* receives a *request* as a unit token and returns a *dq_rslt* in case of success, or *err* in case of failure in case of an empty queue. The unit color set has no predefined service, but is very useful as a placeholder (Jensen 1990).

In addition to the inscription for places, transitions, and arcs as used in CPN, WS-Net provides additional inscriptions for both service components and each service provided. In detail, for the service component inscription, service component name is provided to uniquely identify the service component. For a service inscription, service name and connector type information are provided. Connector type implies the possible protocols to be used when the service is invoked. In our example, both *enqueue* and *dequeue* services are invoked via the SOAP protocol. If multiple protocols can be used, the connector type can have multiple entries. Similar to the name inscription for places and transitions, these inscriptions for service components and services are not considered as Petri nets inscriptions. Instead, these inscriptions are used to interconnect architectural service components, which will be discussed in the interconnection net in the next section.

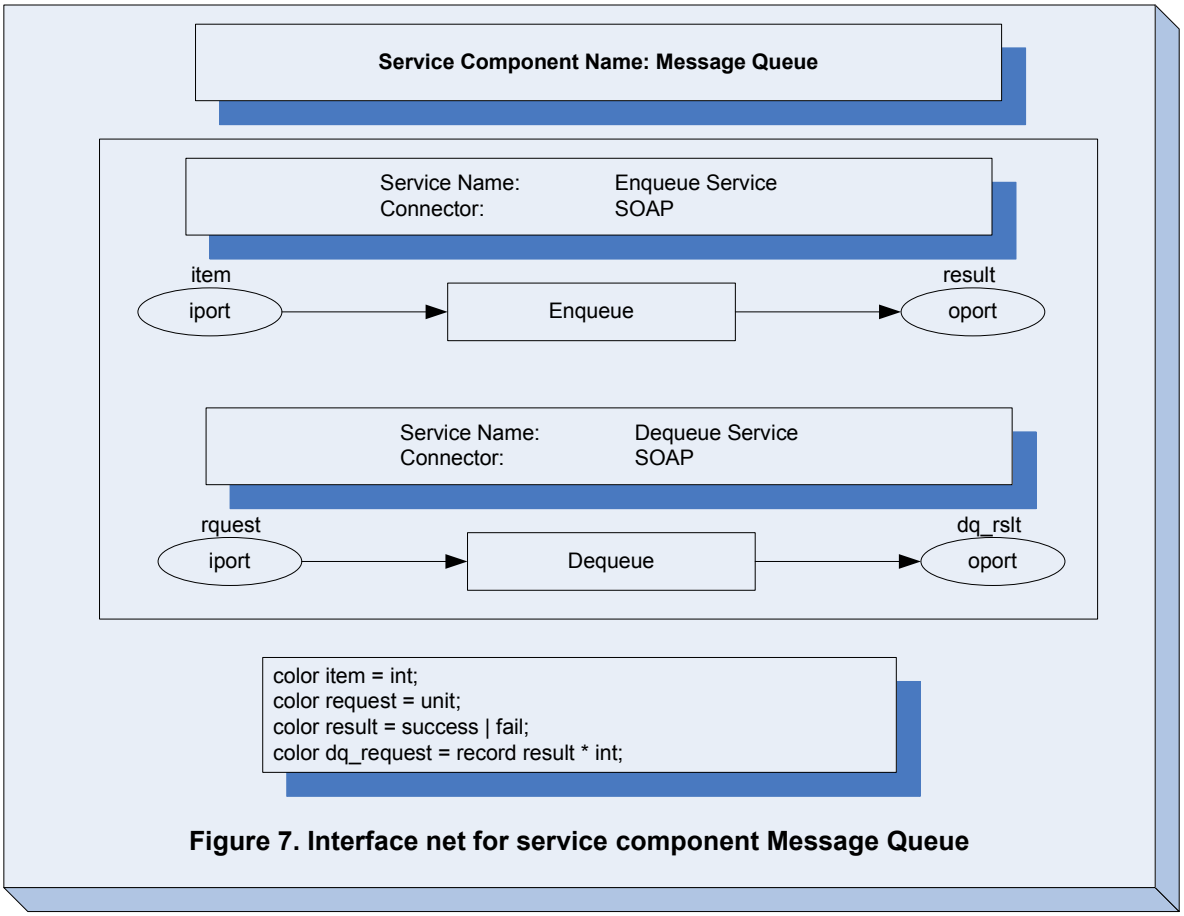

Figure 7. Interface net for service component Message Queue

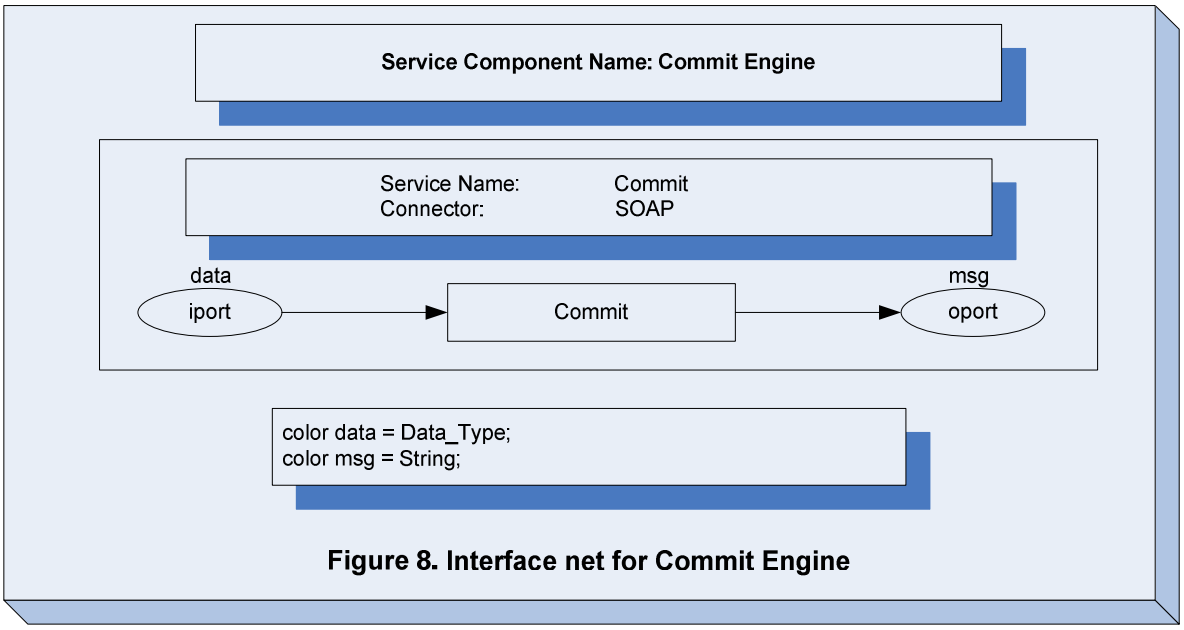

Figure 8. Interface net for Commit Engine

In our example, the two service components *commit engine* and *composer* (as shown in Figure 6) use the *enqueue* and *dequeue* services provided by the *message queue* service component. In other words, the *commit engine* and *composer* are client components for the *message queue* service component. Figure 8 shows the interface net of the *commit engine*, which provides one service called *commit*. The *commit engine* sends *data* as parameters to the *message queue* service component and receives an acknowledgement *msg*. Similarly, as shown in Figure 9, the *composer* provides one service called *retrieve,* and sends a request *rqst* to the *message queue* service component and receives *data* as reply. As we explained previously, the color of token is used here. We assume that the SOAP protocol is used when the *commit engine* and the *composer* service components communicate with the *message queue* service component. Note that an interface net specifies only the services provided by the

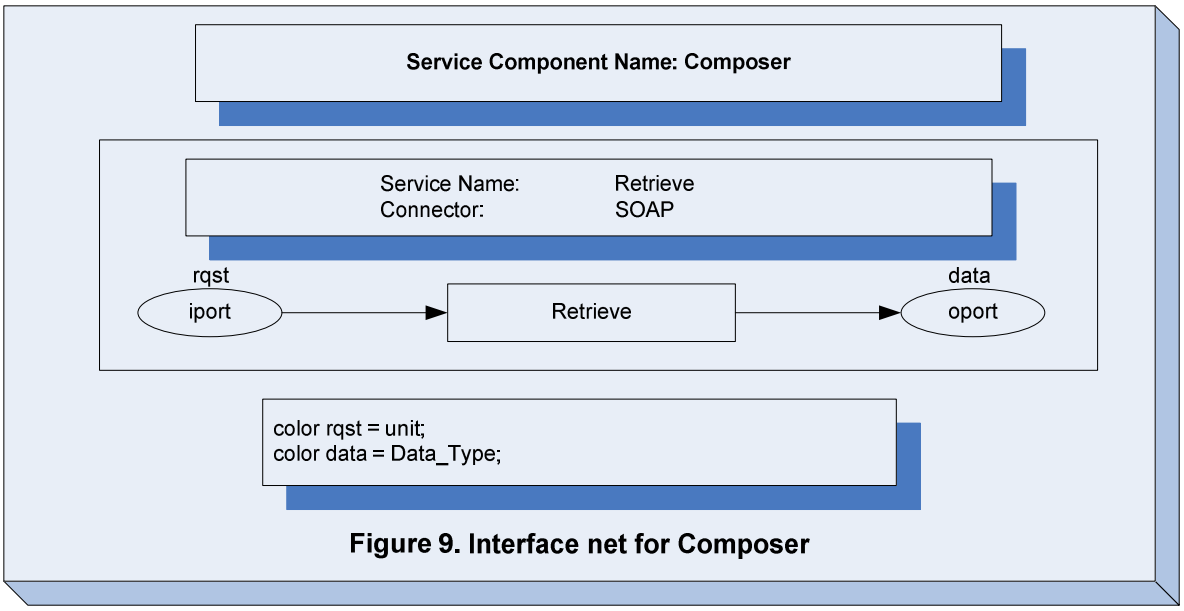

Figure 9. Interface net for Composer

corres

ponding service component; thus, it does not specify connections between components.

Interconnection Net

In order to describe the relationships between architectural components in a services-oriented system, we need to specify both the provided services (i.e. via the interface net specification) and required services of the service components. In other words, a service component may require supporting services in order to provide promised services. By specifying all required services of a service component, the interconnection net describes all the possible dependencies upon other service components in the system. The interconnection net is not mandatory however; instead, it is imperative only when a service component requires foreign services to perform its own commission. For instance, in our example, the *message queue* service component does not require any other services as support; therefore, there is no interconnection net associated with the *message queue* component. The interconnection net depicts a client/server relationship between components. If service component C_i requests a service from service component C_j, C_i is called a client service component, and C_j is called a server service component. A service component can act as a client service component at some time, and as a server service component at other times.

WS-Net chooses to define required services as foreign services since the services need to be performed by other service components. Conforming to the definitions in CPN, in the interconnection net, the required services are specified as a special kind of transition called foreign transition. As in the interface net, the interconnection net specifies an architectural service component as a set of provided services. Each provided service containing foreign transitions is in turn decomposed into a set of required foreign services. A foreign transition is therefore an abstract view of the service provided by another service component. To differentiate with local service component, the input and output places of a foreign transition are called input plugins and output plugins, respectively.

In order to link together a service component and its supporting service components, we introduce two extra transitions for each service component, namely, a start transition and an end transition. As shown in Figure 10, the input port (i.e., place) of the start transition is always the input port of the service component; the output port (i.e., place) of the end transition is always the output port of the service component. A service component may require support from multiple foreign service components before it can perform its execution. Figure 10 illustrates a service that requires n foreign transitions. With the introduction of start transition and end transition, this multiple support relationship can be denoted using WS-Net specifications. Using the notation we introduced above, if a service requires supporting service from multiple foreign services, it will have a set of input plugins and a set of corresponding output plugins.

For services that do not require any foreign transitions, the service specification of the interface net will suffice. Foreign transitions also have inscriptions similar to the provided services of a service component. However, as shown in Figure 10, inscriptions of foreign transitions contain names of the remote service components, names of the services required from the service components, and the type of connectors to be used to invoke each foreign transition. These service names and component names are used to identify the services represented by foreign transitions. In addition to the inscriptions, the color set of the plugins of the foreign services in the client (i.e., local) service component and its

corresponding color set for the services of the remote server service component must be compatible.

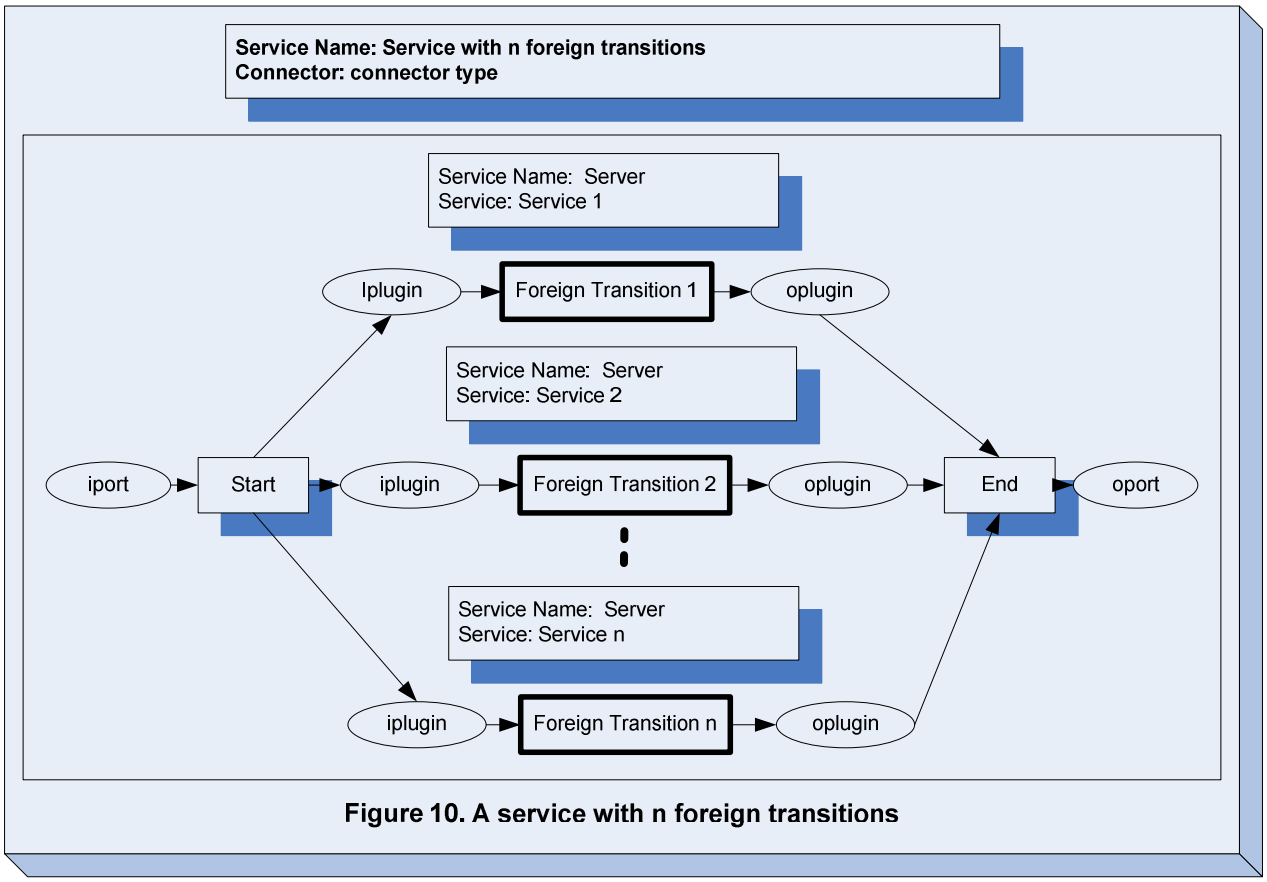

Figure 10. A service with n foreign transitions

Thus, a service S_{ij} requiring foreign services is represented as a tuple:

$$S_{ij} = (PI_{ij}, PO_{ij}, QI_{ij}, QO_{ij}, TS_{ij}, TE_{ij}, TF_{ij}, A_{ij}, c),$$

where PI_{ij} and PO_{ij} are the input port and the output port of S_{ij} respectively, as in the interface net. TS_{ij} and TE_{ij} represent the start transition and the end transition of the service component S_{ij}. The input place of TS_{ij} is the input port PI_{ij}, and the output place of TE_{ij} is the output port PO_{ij}. TF_{ij} is a set of foreign transitions. QI_{ij} is a set of input plugins, and QO_{ij} is a set of output plugins. A_{ij} is a set of input and output arcs for the transitions. As in the interface net, c represents a color function.

Figure 11 and Figure 12 show the interconnection nets of the service component *commit engine* and *composer,* respectively. The *commit* engine service component needs to invoke the *enqueue* service from the component *method queue,* and the *composer* service component needs to invoke the *dequeue* service. Therefore, *enqueue* and *dequeue* services are represented as foreign transitions. Inscriptions for the foreign transitions show that they are calling *enqueue* and *dequeue* services from the service component *message queue* via SOAP.

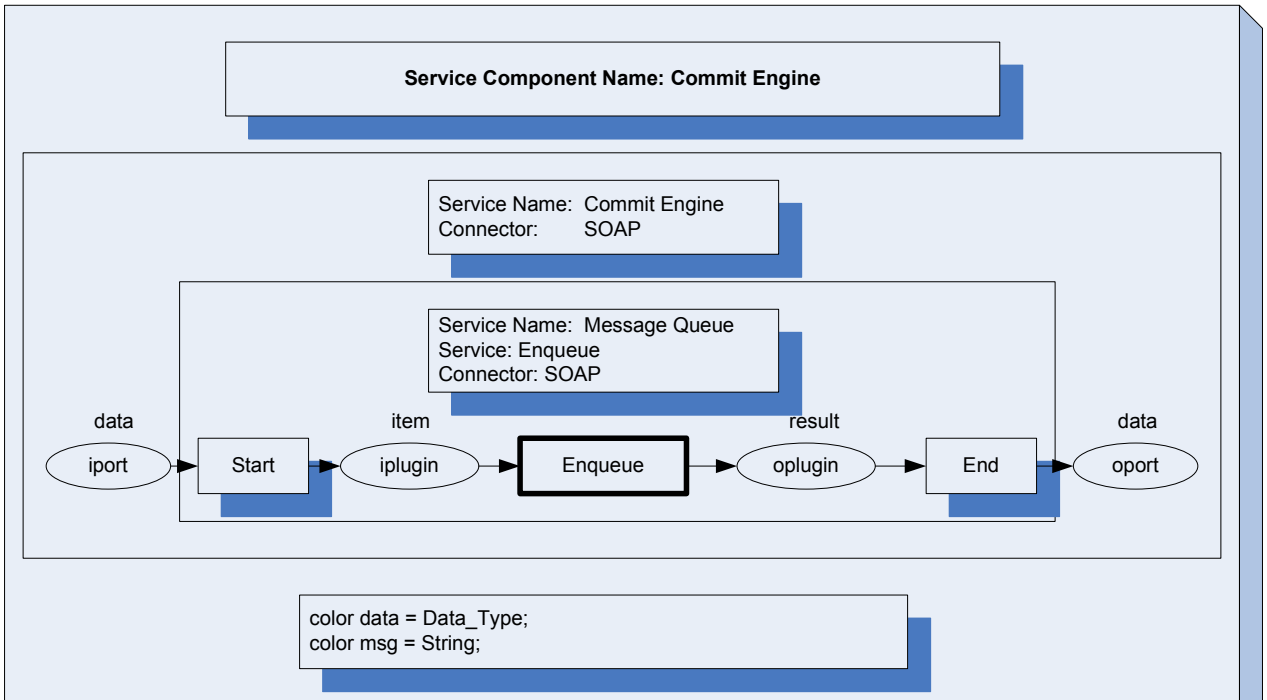

Figure 11. Interconnection net for C*ommit Engine*

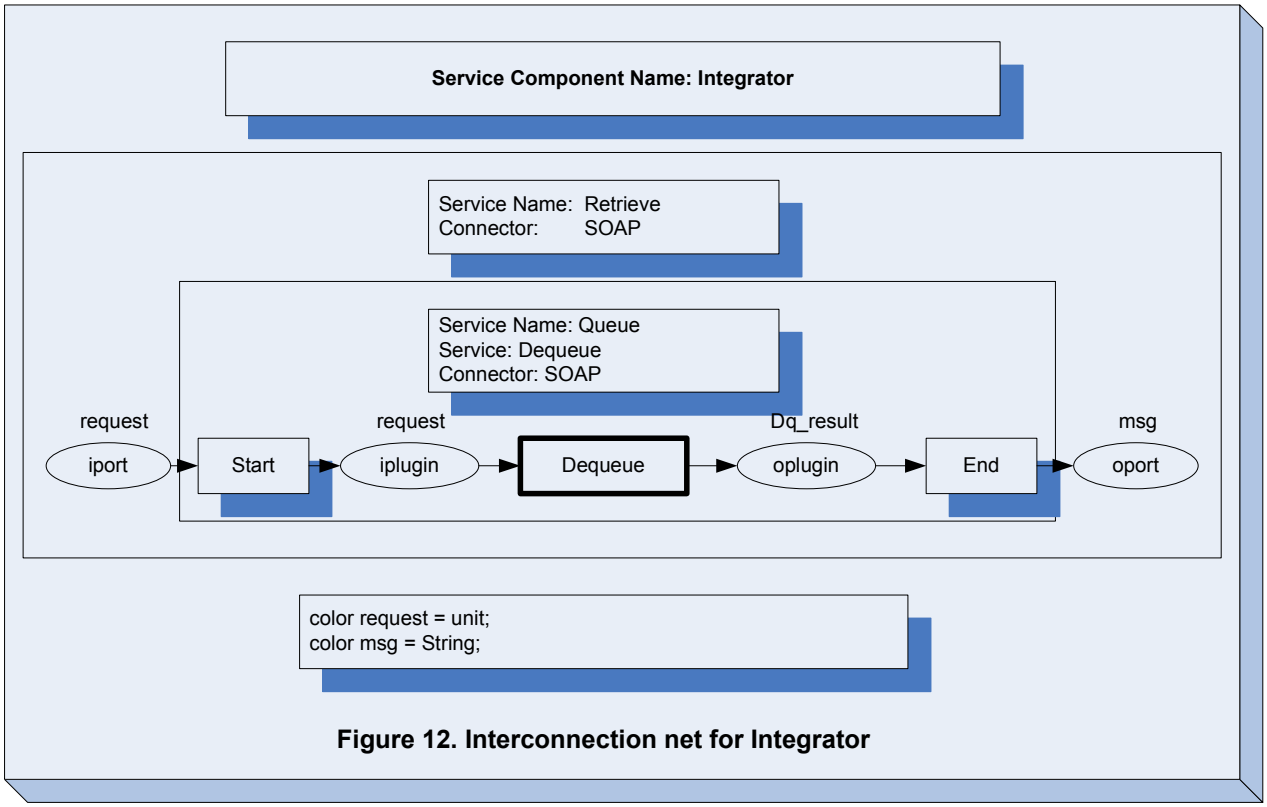

Figure 12. Interconnection net for Integrator

After specifying individual service components in terms of the interface nets and the interconnection nets, we are ready to visualize the entire topological view of a system by interconnecting all of these WS-Net components. Firing a foreign transition means executing the corresponding service transition of the server component. Therefore, connecting WS-Net components can be achieved by merging the ports of the client service components with the ports of the server service component, after removing foreign transitions from the client service components. In our WS-Net, we thus introduce a special kind of transition aiming at connecting ports. This transition is called a connector transition, and it is named by a connector type. Figure 13 shows the connected interconnection net that describes the entire information-communication model by interconnecting the *commit engine* and the *composer* with the *message queue* using SOAP connectors.

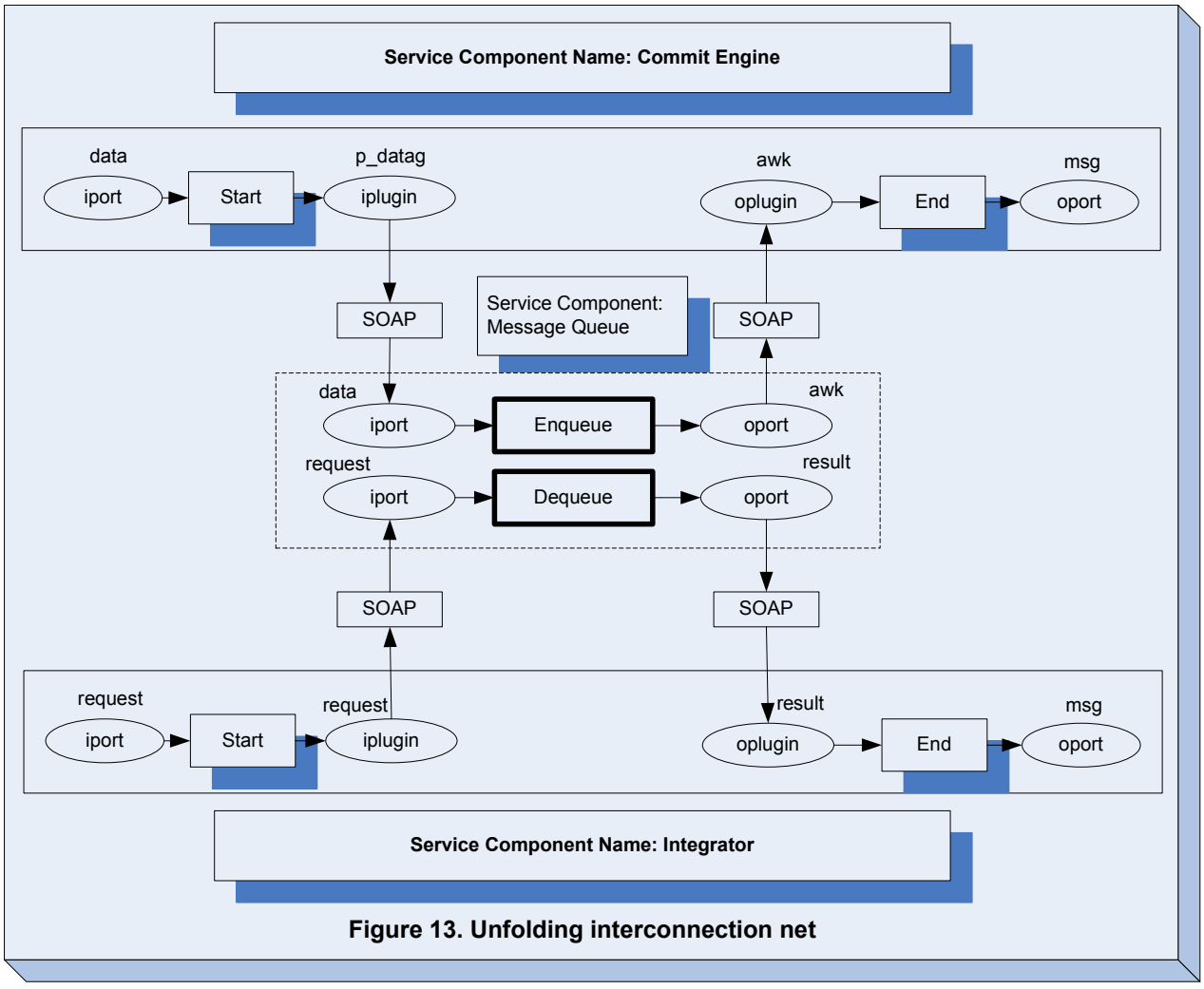

Figure 13. Unfolding interconnection net

In summary, such an interconnection mechanism can be applied across different levels of service component diagrams. In detail, interconnections can be visualized in two levels: (1) interoperation nets of client/server service components, and (2) the folding/unfolding of interface nets of service components. This is an important feature to visualize very large systems. By applying such visual

abstractions, such as replacing large interoperation nets with simpler interconnection nets or even with interface nets, complicated nets can be effectively visualized at various levels of abstraction.

Interoperation Net

The interoperation net describes the dynamic behaviors of a service component by focusing on its operational nature. The goal of the interoperation net is to dissect each service into fundamental process units which, taken together, define the required functional contents of the service. This is similar to the SADT functional decomposition, where each transition representing the operations of a component is decomposed into sub-transitions to represent fundamental operational state. One of the most important differences between the decomposition in our interoperation net and SADT is that, the interoperation net uses the decomposition as a means of expressing the behaviors of the services provided by an architectural service component, rather than functional decomposition for modularization used in SADT. As in SADT, the control flow and data flow are used to describe the interactions between process units. Note that it is important to distinguish foreign transitions from detailed processes. The foreign transitions along with plugin places are used to interconnect the interoperation nets to form the entire system view. Like other Petri nets-based high-level design representations, places are used to represent the control or data; and transitions are used to represent processes.

Chang and Kim found that the straightforward techniques converting functional data flow to Petri nets have a potential problem in repeated (persistent) simulations of the nets (Chang and Kim 1999). To solve this problem, in WS-Net, we distinguish persistent data from transient data. Persistent places are represented as boldface circles. Persistent data items are similar to the data attributes of a class in the Object-Oriented paradigm. These persistent data items represent the state of a service component, and they exist throughout the lifetime of the service component. On the other hand, transient data items are produced by one process and are immediately consumed by another process. Therefore, transient data items are created only when they are needed and destroyed upon the completion of the service.

A service $S_{ij} \in S_i$ of service component C_i can be denoted as follows:

$$S_{ij} = (PI_{ij}, PO_{ij}, PT_{ij}, PP_{ij}, QI_{ij}, QO_{ij}, TL_{ij}, TF_{ij}, A_{ij}, c, G, E, IN),$$

where PI_{ij} and PO_{ij} are the input and output ports; PT_{ij} and PP_{ij} are a set of transient data places and a set of persistent data places respectively. TL_{ij} is a set of local transitions; and TF_{ij} is a set of foreign transitions. QI_{ij} is a set of input plugin places serving as input places for the foreign transitions; and QO_{ij} is a set of output plugin places serving as output places for the foreign transitions. A_{ij} is a set of input and output arcs of the transitions. To describe the functional behaviors of a component, we can use all the inscriptions used in CPN (Jensen 1990). As before, c is a color function to represent the color sets for the places. G is a guard function for the transitions. E is an arc expression function, and IN is an initialization function for the tokens.

In our example, the *message queue* service component has *enqueue* and *dequeue* services. The control and data are represented by places; and processes are represented by transitions. Figure 14 shows the first phase of the interoperation description of the *message queue* component. *Count* and *storage*

places are defined as persistent data and represented with boldface circles. Since the persistent data may exist throughout the lifetime of a service component, we need to initialize the persistent places with proper tokens for later simulations. Tokens in the transient places are produced as a result of firing transitions. It is common for persistent data items to be shared by other services in the same component. If different services use the same persistent data, they need to be merged using the place-fusion technique defined in high-level Petri nets. As shown in Figure 14, *count* and *storage* are persistent places of both the *enqueue* and *dequeue* services. By merging the persistent places of the two services, the interoperation net for the *message queue* component can be completed.

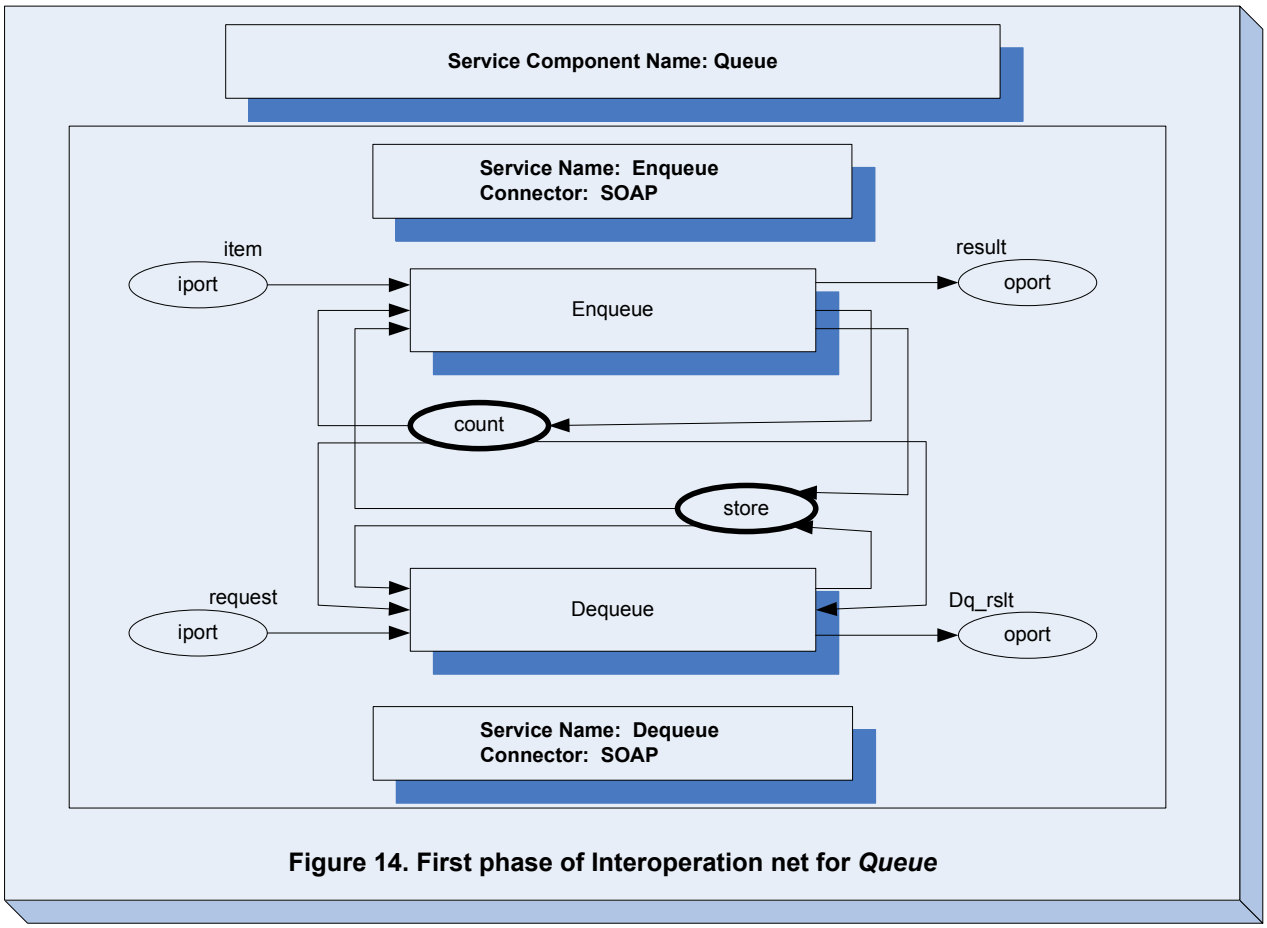

Figure 14. First phase of Interoperation net for *Queue*

As we further decompose the functional behaviors of each service, we can get a more complex interoperation net. Figure 15 shows a more detailed interoperation net of the service *enqueue* of the *message queue* service component. First, the service receives a request to insert a message into the *message queue.* Both the *content* places and *store* places are checked for full before an item can be inserted. If the store place is not full, the message can be inserted into the queue. Then the service updates the full flag after the insertion of the message. A Petri net can be constructed by mapping each functional process into a transition, and input and output into places. After all the interoperation nets of the architectural service components are specified, we can again visualize the entire system topology by connecting the plugins of the client service components with the ports of the server components using the connector transitions.

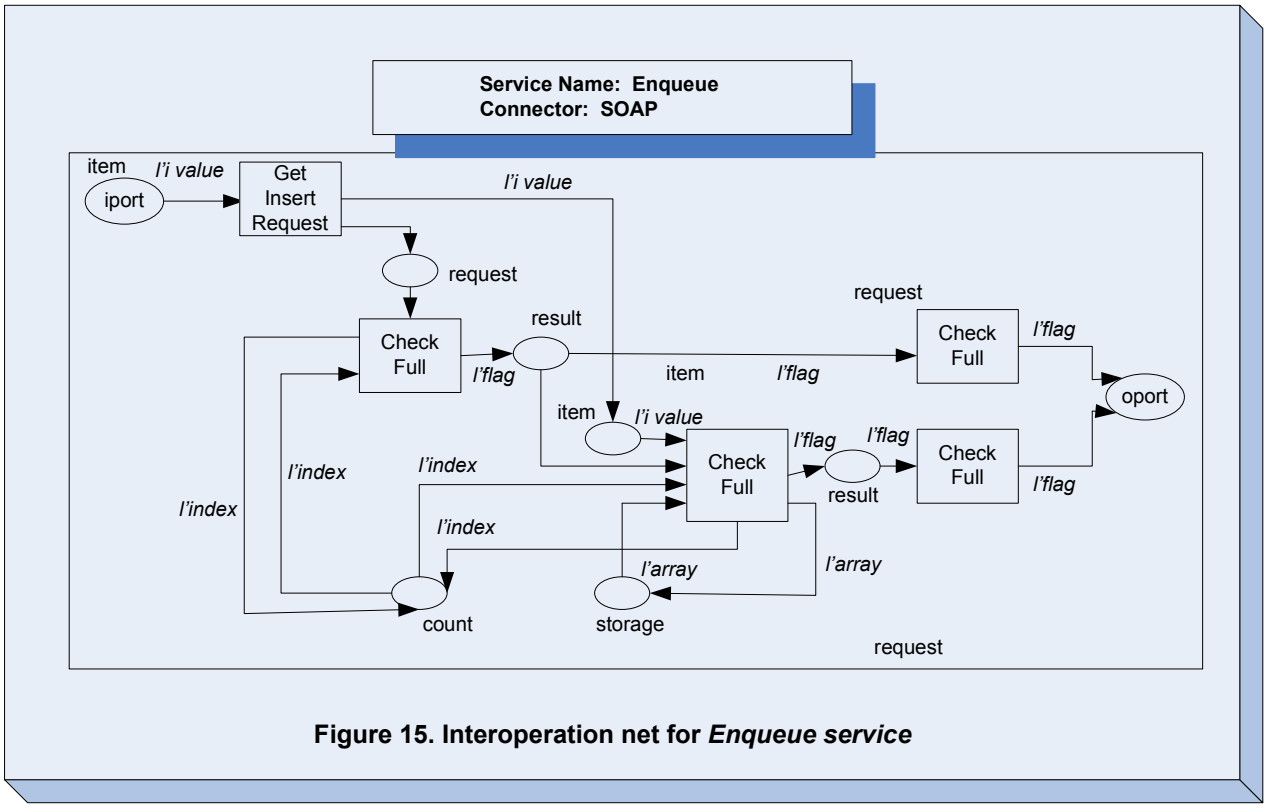

Figure 15. Interoperation net for *Enqueue service*

Connected interoperation nets can be executed under different input scenarios to simulate the behaviors of a services-oriented system. The execution proceeds by assigning initial tokens to the input ports. The execution traces can be visualized to analyze the runtime quality attributes and to enhance communications with user communities by providing an executable model of the system early in the development process.

WS-NET-BASED FORMAL VERIFICATION

We have introduced the basic concepts and specifications of WS-Net. By using a typical Web services message processing as an example, we illustrate how to model a Web services-oriented system into a hierarchical WS-Net. How to model a software system using Petri nets has already been extensively discussed in many other publications, in this section, we will introduce the mappings between Web services concepts and Petri nets specifications, as general guidance of modeling Web services-oriented systems using Petri nets. Figure 16 summarizes the mappings that are critical due to the fact that Petri nets do not explicitly support the concept of process.

As shown in Figure 16, Web services (i.e., service components and services provided) are modeled using transitions. The input and output of a service are modeled by two kinds of places, namely, input places and output places, respectively. Messages exchanging between Web services are modeled as connectors in Petri nets. In order to enhance readability, labels are used to identify Web services by names. If a Web service requires other services as support, foreign transitions are used to model supporting Web services. Thus, the message interactions between Web services are modeled as data

WS concepts	PN concepts
Messages	Connector
Service	Transition
Input	Input place
Output	Output place
Name	Label
Required service	Foreign transition
Interaction	Input/output places via connector
Data	Token
Signature	Color
Data sharing	Place fusion
Hierarchy	Transition substitute
Persistent data	Persistent place
Transient data	Transient place

Figure 16. Mapping between Web services concepts and Petri nets specifications

flow between input places and output places via connectors. In order to handle complexity, transition substitution is used to fold/unfold hierarchical services composition into modularized nets, according to architectural structure of a system. In WS-Net, data are modeled by tokens. The concept of color is used to specify the signature and data types of data. Persistent data and transient data are differentiated using persistent places and transient places. Data sharing between Web services are modeled by the place fusion concept in Petri nets.

With the mapping mechanisms established, we can turn a Web services-oriented system into a simulatable Petri nets-based WS-Net. With this simulation, we can detect and identify services composition errors using the analysis mechanisms provided by Petri nets. The simulation of the executable system thus can be used to verify the correctness of the system. The interconnection mechanism of WS-Net enables analyzing complicated system composition at different granularities. Associated with the interoperation net, WS-Net provides a structured way to zoom in and out to analyze architectural system composition at various levels.

Using WS-Net, formal verification can be conducted at design time of system composition, in order to detect potential composition errors at early stage thus to correct errors as early as possible. Particularly, WS-Net focuses on analyzing important composition criteria, such as reachability, boundedness, and liveness. By examining dead markings, we can verify the reachability of a certain WS-Net thus verify whether certain composition protocols (i.e., rules) are enforced and conformed. The state space analysis can be carried out to detect whether a deadlock possibly exists in the design

of the services composition. The visualization of the composition and interactions between Web service components help engineers verify compositional message exchanges and synchronizations. WS-Net analysis and simulation can start with an initial marking inputted into a WS-Net model. Running the simulations, we can check whether the service composition will execute as expected, and whether the service composition confirms with the conversational protocols between service components. Furthermore, different markings can be used to feed the constructed WS-Net to verify system behaviors under various situations.

CONCLUSIONS AND FUTURE TRENDS

In this chapter, we introduced an advanced topic of service computing: how to formally verify Web services-oriented system composition. We first introduced the basic concepts of services composition in the context of Web services technologies. Then we surveyed possible solutions and existing efforts for formally verifying Web services-oriented system composition. After comparing various options, we introduced Web Services Net (WS-Net), an executable Petri nets-based architectural description language to formally describe and verify the architecture of a Web services-oriented system. The behaviors of such a model can be simulated to detect errors and allow corrections and further refinements. As a result, WS-Net helps enhance the reliability of Web services-oriented applications. Furthermore, it is compatible with the Object-Oriented paradigm and component-based concepts. Supporting modern software engineering philosophies oriented to services computing, WS-Net provides an approach to verify and monitor the dynamic integration of a Web services-oriented software system. Specification formalism in WS-Net is Object-Oriented, executable, expressive, comprehensive, and yet easy to use. A wide body of theories available for Petri nets is thus available for analyzing a system design. To our best knowledge, our WS-Net is the first paper to comprehensively map Web services elements to CPN so that the latter can be used to facilitate the simulation and formal verification and validation of Web services composition.

However, manually transferring WSDL specifications into the WS-Net specifications is not a trivial job. That is why currently we have only built some simple experiments, e.g., the example described in the paper. In order for WS-Net to monitor and verify real-life applications, the translation from WSDL into WS-Net must be automated.

Meanwhile, since all transient tokens are created by local transitions and all persistent tokens are restored before the completion of the service, repeated simulation of the net is possible. Furthermore, in converting functional data flow models to Petri nets, we also face the concurrency and choice problems (Trattnig and Kerner 1980). Those problems need to be addressed properly by system engineers, who build the system architecture by using WS-Net.

Despite of the challenges WS-Net is facing, our preliminary experiences prove its effectiveness and efficiency of formally verifying Web services-oriented system composition. WS-Net uses an iterative and top-down process of investigating and examining services composition using the Petri nets vehicle of technology. Future work will focus on building automatic translation tools from Web services system specification into Petri nets tool-based specifications to automate the simulation of Web services composition.

REFERENCES

Aggarwal, R., Verma, K., Miller, J., & Milnor, W. (2004). Constraint driven Web service composition in METEOR-S. In *Proceedings of IEEE International Conference on Services Computing (SCC)*, Shanghai, China, September 15-18, (pp. 23-30).

Arpinar, B., Aleman-Meza, B., Zhang, R., & Maduko, A. (2004). Ontology-driven Web services composition platform. In *Proceedings of IEEE International Conference on E-Commerce Technology (CEC)*, San Diego, CA, USA, July 7-9, (pp. 146-152).

Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2003). Automatic composition of e-services that export their behavior. In *Proceedings of 1st International Conference on Service-Oriented Computing (ICSOC)*, (pp. 43-58). LNCS 2910, Springer-Verlag.

Bordeaux, L., Salaün, G., Berardi, D., & Mecella, M. (2004). When are 2 Web services compatible? In *Proceedings of VLDB Workshop on Technologies of E-Services (VLDB-TES)*, Toronto, Canada, LNCS 3324, Springer-Verlag, August, (pp. 15-28).

BPEL4WS (2003). Specification: Business Process Execution Language for Web Services Version 1.1. May 5. Retrieved December 19, 2005, from http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Bultan, T., Fu, X., Hull, R., & Su, J. (2003). Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the 12th ACM International World Wide Web Conference (WWW)*, Budapest, Hungary, May 21-23, (pp. 403-410).

Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., & Shan, M. (2000). Adaptive and dynamic service composition in eFlow. March. Retrieved December 19, 2005, from http://www.hpl.hp.com/techreports/2000/HPL-2000-39.pdf.

Chang, C. K., & Kim, S. (1999). I3: a Petri-Net based specification method for architectural components. In *Proceedings of IEEE 23th Annual International Computer Software and Applications Conference (COMPSAC)*, Phoenix, AR, USA, October 25-26, (pp. 396-402).

Meta Software Corporation (1993). *Design/CPN reference manual for X-Windows version 2.0*.

Ferrara, A. (2004). Web services: a process algebra approach. In *Proceedings of the 2nd ACM International Conference on Service Oriented Computing (ICSOC)*, New York, NY, USA, November 15-19, (pp. 242-251).

Fu, X., Bultan, T., & Su, J. (2005). Realizability of conversation protocols with message contents. *International Journal of Web Services Research (JWSR),* October-December, 2(4), 72-97.

Jensen, K. (1990). Coloured Petri Nets: a high level language for system design and analysis. *Lecture Notes in Computer Science, Advances in Petri Nets*.

Kiwata, K., Nakano, A., & Yura, S. (2001). Scenario-based service composition method in the open service environment. In *Proceedings of 5th International Symposium on Autonomous Decentralized Systems*, (pp. 135-140).

Leymann, F. (2001). Web Services Flow Language (WSFL 1.0). IBM Corporation.

Liang, Q., Chakarapani, L. N., Stanley Y.W. Su, Chikkamagalur, R. N., & Lam, H. (2004). A semi-automatic approach to composite Web services discovery, description and invocation. *International Journal of Web Services Research (JWSR),* October-December, 1(4).

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software

architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), 70-93.
Microsoft (2003). BizTalk server 2004: architecture. December. Retrieved December 19, 2005, from http://download.microsoft.com/download/e/6/f/e6fcf394-e03e-4e15-bd80-8c1c127e88e7/BTSArch.doc.
Mifflin, H. (2000). *The American heritage dictionary of the English language*, 4th edition, Houghton Mifflin Company.
Milanovic, N., & Malek, M. (2004). Current solutions for Web service composition. *IEEE Internet Computing,* Nov.-Dec., 8(6), 51-59.
Narayanan, S., & McIlraith, S. A. (2002). Simulation, verification and automated composition of Web services. In *Proceedings of the eleventh ACM International Conference on World Wide Web (WWW)*, Honolulu, Hawaii, USA, May 7-11, (pp. 77-88).
Peterson, J. L. (1981). *Petri Net theory and the modeling of systems*, Prentice-Hall International.
Petri, C. A. (1962). Kommunikation mit automaten. doctoral dissertation, Institut für Instrumentelle Mathematik, Schriften des IIM.
Pi-Calculus. Automata, state, actions, and interactions. Retrieved December 19, 2005, from http://www.ebpml.org/pi-calculus.htm.
Pinci, V., & Shapiro, R. (1990). An integrated software development methodology based on hierarchical Colored Petri Nets. *Lecture Nets in Computer Science, Advances in Petri Nets*: 227-252.
Ponnekanti, S., & Fox, A. (2002). SWORD: a developer toolkit for building composite Web services. In *Proceedings of Alternate Tracks of the ACM 11th International World Wide Web Conference (WWW)*, Honolulu, Hawaii, USA, May 7-11.
Ross, D. (1984). Application and extensions of SADT. *IEEE Computer,* Apr., 18(4), 25-35.
Salaun, G., Bordeaux, L., & Schaerf, M. (2004). Describing and reasoning on Web services using process algebra. In *Proceedings of IEEE International Conference on Web Services (ICWS)*, San Diego, CA, USA, July 6-9, (pp. 43-50).
Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., & Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture,* April, 21(4), 314-335.
SOAP (2003). Simple Object Access Protocol (SOAP) 1.2. May. World Wide Web Consortium (W3C). Retrieved December 19, 2005, from http://www.w3.org/TR/soap12-part1/.
Thatte, S. (2001). XLANG - Web Services for Business Process Design. Microsoft Corporation.
Trattnig, W., & Kerner, H. (1980). EDDA-a very high-level programming and specification language in the style of SADT. In *Proceedings of IEEE Annual InternationalComputer Software and Applications Conference (COMPSAC)*, October, (pp. 436-443).
UDDI (2004). Universal Description, Discovery, and Integration. Retrieved December 19, 2005, from http://www.uddi.org/.
WSCI (2002). Web Service Choreography Interface (WSCI), Version 1.0. Sun Microsystems.
WSDL (2004). Web Services Description Language. Retrieved December 19, 2005, from http://www.w3.org/TR/wsdl.
XML. Retrieved December 19, 2005, from http://www.w3.org/XML.
Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., & Sheng, Q. (2003). Quality driven Web services composition. In *Proceedings of 12th ACM International Conference on World Wide Web (WWW)*, Budapest, Hungary, May 20-24, (pp. 411-421).
Zeng, L., Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnanam, J., & Chang, H. (2004). QoS-aware

middleware for Web services composition. *IEEE Transactions on Software Engineering,* May, 30(5), 311-327.

Zhang, J. (2005). Trustworthy Web services: actions for now. *IEEE IT Professional,* January/February, 32-36.

ABOUT AUTHORS

Jia Zhang, Ph.D., is an Assistant Professor of Department of Computer Science at Northern Illinois University. She is now with BEA Systems Inc.; and also a Guest Scientist of National Institute of Standards and Technology (NIST). Her current research interests center around software trustworthiness in the domain of Web services, with a focus on reliability, integrity, security, and interoperability. Zhang has published over 60 technical papers in journals, book chapters, and conference proceedings. She also has seven years of industrial experience as software technical lead in Web application development. Zhang is an Associate Editor of the International Journal of Web Services Research (JWSR). Zhang received a Ph.D. in computer science from University of Illinois at Chicago in 2000. She is a member of the IEEE and ACM. Contact her at jiazhang@cs.niu.edu.

Carl K. Chang, Ph.D., is professor and chair of the department of computer science at Iowa State University. His research interests include requirements engineering, software architecture, and net-centric computing. He is a founding member of the IEEE International Requirements Engineering Conference (RE), and served as the general chair of ICRE 2000 and RE2003. He also chaired the steering committee for the 2004 IEEE-CS/IPSJ International Symposium on Applications and the Internet (SAINT) after serving as the program chair of SAINT2002 and general chair of SAINT2003. In 2005 he was general chair of IEEE International Conference on Web Services (ICWS). Chang is also active in the educational activities and spearheaded the Computing Curricula 2001 (CC2001) project jointly sponsored by the IEEE Computer Society, ACM, and the National Science Foundation. He served as the Editor-in-Chief for IEEE Software in 1991-94. Chang is a Fellow of IEEE, a Fellow of AAAS, and 2004 President of the IEEE Computer Society. He can be reached at chang@cs.iastate.edu.

Seong W. Kim., Ph.D., is currently with Samsung. Contact him at seongwoon.kim@samsung.com.