We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

22

Formalizing and Validating UML Architecture Description of Service-Oriented Applications

Zhijiang Dong[1], Yujian Fu[2], Xudong He[3] and Yue Fu[4]
[1]*Department of Computer Science, Middle Tennessee State University*
[2]*Department of Computer Science, Alabama A&M University*
[3]*School of Computer Science, Florida International University*
[4]*Department of Technology, Fuyuan High Technology Co.,Ltd.*
[1,2,3]*USA*
[4]*P.R. China*

1. Introduction

Service-oriented applications, especially web systems, are self-descriptive software components which can automatically be discovered and engaged, together with other web components, to complete tasks over the Internet. The importance of service-oriented application architecture descriptions has been widely recognized in recently year. One of the main perceived benefits of a service-oriented application architecture description is that such a description facilitates system property analysis and thus can detect and prevent web design errors in an earlier stage, which are critical for service-oriented applications. Software architecture description and modeling of a service-oriented application plays a key role in providing the high level perspective, triggering the right refinement to the implementation, controlling the quality of services of products and offering large and general system properties. While several established and emerging standards bodies (e.g., [5, 4, 3, 1, 2] etc.) are rapidly laying out the foundations that the industry will be built upon, there are many research challenges behind service-oriented application architecture description languages that are less well-defined and understood [33] for the large number of web service application design and development.

On the other hand, Unified Modeling Language (UML), a widely accepted object-oriented system modeling and design language, has been adapted for software architecture descriptions in recent years. Several research groups have used UML extension to describe the service-oriented application's architecture ([7, 29]). However, it is hard to detect the system problems, such as correctness, consistency [30] etc., of the integration of Web services without a formal semantics of web services architecture. Currently, although a software architecture description using UML extension contains multiple viewpoints such as those proposed in the SEI model [39], the ANSI/IEEE P1471 standard, and the Siemens [31]. The component and connector (C&C) viewpoint [42], which addresses the dynamic system behavioral aspect, is essential and necessary for system property analysis.

Source: Petri Net,Theory and Applications, Book edited by: Vedran Kordic, ISBN 978-3-902613-12-7, pp. 534, February 2008, I-Tech Education and Publishing, Vienna, Austria

To bridge the gap between service-oriented application architecture research and practice, several researchers explored the ideas of integrating architecture description languages (ADLs) and UML [8, 13, 14, 35]. Most of these integration approaches attempted to describe elements of ADLs in terms of UML such that software architectures described in ADLs can be easily translated to extensions of UML. There are several problems of the above approach that hinder their adoption. First, there are multiple ways to describe ADLs in terms of UML [24], each of which has advantages and disadvantages; thus the decision on which extension of UML to use is not unique. Second, modifications on UML models are difficult to be reflected in the original ADL models since the reverse mapping is in general impossible. Finally, the software developers are required to learn and use specific ADL to model software architecture and use the specific extension of UML, which is exactly the major cause of preventing the wide use of ADLs. Currently, there is less work involved to apply these methodologies to the service-oriented applications.

In this paper, we present an approach opposite to the one mentioned above and apply our approach to the web applications, i.e. we translate a UML architecture description into a formal architecture model for formal analysis. Using this approach, we can combine the potential benefits of UML's easy comprehensibility and applicability with a formal ADL's analyzability. Moreover, this approach is used to formally analyze the integration of web services. The formal architecture model used in this research is named SO-SAM, an extended version of SAM [27], which is based on Petri nets and temporal logic; and supports the analysis of a variety of functional and non-functional properties [28]. Finally, we validate this approach by using model checking techniques. This approach presents an effective way of the Service-Oriented Architecture (SOA) in a logical format so that stake holders can better use artifacts to leverage Unified Modeling Language (UML) components in their architecture and design efforts.

The remainder of this paper is organized as follows. In section 2, we review SO-SAM with predicate transition nets and temporal logic for high-level design. After that, we presented our approach in section 3 and the validation of the approach is demonstrated in section 4. Finally, we draw conclusions and describe future work in section 6.

2. Preliminaries

2.1 Overview of SO-SAM

SO-SAM [20] is an extended version of SAM [44] with the web service oriented features. SO-SAM [20] is a general formal framework for specifying and analyzing service-oriented architecture with two formalisms – Petri Net model and temporal logic, which is inherited from SAM.

In addition, SO-SAM extended the net inscriptions with *servicesorts* and net structure with *initial* and *final* ports that carry service triggering information. Furthermore, SO-SAM restricted SAM connector without hierarchical architecture. Finally, SO-SAM component is described by WSDL or XML. Also, the message in ports is defined by XML message. For the more information, please refer to [20]. In this paper, we choose algebraic high level nets [17] and linear time first order temporal logic as the underlying complementary formalisms of SAM. Thus, next we simply introduce the algebraic high level nets used in our approach.

2.2 SAM

SAM [44] is an architectural description model based on Petri nets [37], which are well-suited for modeling distributed systems. SAM [44] has dual formalisms underlying – Petri nets and Temporal logic. Petri nets are used to describe behavioral models of components and connectors while temporal logic is used to specify system properties of components and connectors.

SAM architecture model is hierarchically defined as follows. A set of compositions $C = \{C_1, C_2, \ldots, C_k\}$ represents different design levels or subsystems. A set of component C_{mi} and connectors C_{ni} are specified within each composition C_i as well as a set of composition constraints C_{si} , e.g. $C_i = \{C_{mi}, C_{ni}, C_{si}\}$. In addition, each component or connector is composed of two elements, a behavioral model and a property specification, e.g. $C_{ij} = (S_{ij}, B_{ij})$. Each behavioral model is described by a Petri net, while a property specification by a temporal logical formula. The atomic proposition used in the first order temporal logic formula is the ports of each component or connector. Thus each behavioral model can be connected with its property specification. A component C_{mi} or a connector C_{ni} can be refined to a low level composition C_l by a mapping relation h, e.g. $h(C_{mi})$ or $h(C_{mi}) = C_l$.

Figure 1 shows a graphical view of a simple SAM architecture model. The formal analysis and design strategy of the SAM model on the software architecture is given in work [27].

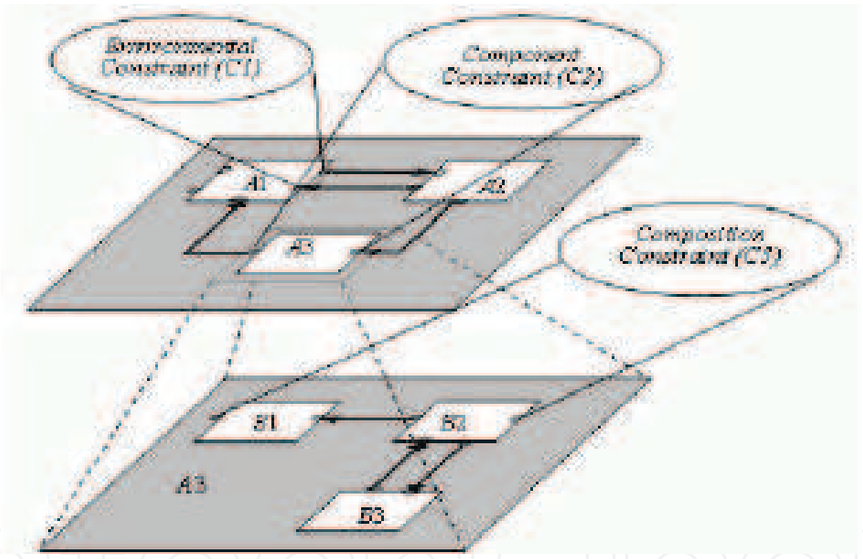


Fig. 1. A SAM Architecture Model

SAM gives the flexibility to choose any variant of Petri nets and temporal logics to specify behavior and constraints according to system characteristics. In our case, Predicate Transition (PrT) net [25] and linear temporal logic (LTL) are chosen.

In summary, although our work was strongly influenced by SAM, we have enhanced the state of the art by supporting modern software engineering philosophies equipped with component-based and object-oriented notations and applied to web services-oriented systems, as well as integrated with WSDL and XML.

2.3 Algebraic high-level nets

An algebraic high-level net [17] integrates Petri net with inscription of an algebraic specification defining the data types and operations. Instead of specifying a single system

model, an algebraic Petri net represents a class of models that often differ only in a few parameters. Such a compact parameterized description is unavoidable for modular specification and economic verification of net models in the dependable system design.
Generally speaking, an algebraic high-level (AHL) nets $N = (SPEC, A, X, P, T, W^+, W^-, cond, type)$ consists of following parts:

- An algebraic specification $SPEC = (S, OP, E)$, where $SIG = (S, OP)$ is a signature, and E is a set of equations over SIG;
- A is an $SPEC$ algebra;
- X is an family of S-sorted variables;
- P is a set of places;
- T is a set of transitionssuch that $P \cap T = \emptyset$;
- Two functions W^+, W^- assigning to each $t \in T$ an element of the free commutative monoid[1] over the cartesian of P and terms of $SPEC$ with variables in X.
- $cond$ is a function assigning to each $t \in T$ a finite set of equations over signature SIG.
- $type$ is a function assigning to each place a sort in S.

Fig. 2 shows an algebraic high-level net of sender-receiver. Its algebraic specification is defined as following:

```
SPEC =
    sorts: nat, bool, data, queue
    opns: err: → data
          nil: → data
          inq: data × queue → queue
          deq: queue → queue
          first: queue → data
          empty: queue → bool
          length: queue → nat
    eqns: deq(nil) = nil
          deq(inq(x,nil)) = nil
          deq(inq(x,inq(y,q))) = inq(x,deq(y,q))
          first(nil) = err
          first(inq(x,nil)) = x
          first(inq(x,inq(y,q))) = first(inq(y,q))
          empty(nil) = true
          empty(inq(x,q)) = false
          length(nil) = 0
          length(inq(x,q)) = length(q) + 1
```

From the figure, we can see transition *send* is enabled if place $p1$ contains a data and the queue in place p has space. As a result of the firing of *send*, the data is added to the queue. Whenever place $p4$ has a token and the queue in p is not empty, transition *receive* is enabled. When it fires, the first data in the queue is output to place $p3$ and the first data in the queue is removed.

[1] A set M with an associative operation _ and an identity element for that operation is called a monoid. A commutative monoid is a monoid in which the operation is commutative. A commutative monoid is a free commutative monoid if every element of M can be written in one and only one way as a product (in the sense of _) of elements of subset P M.

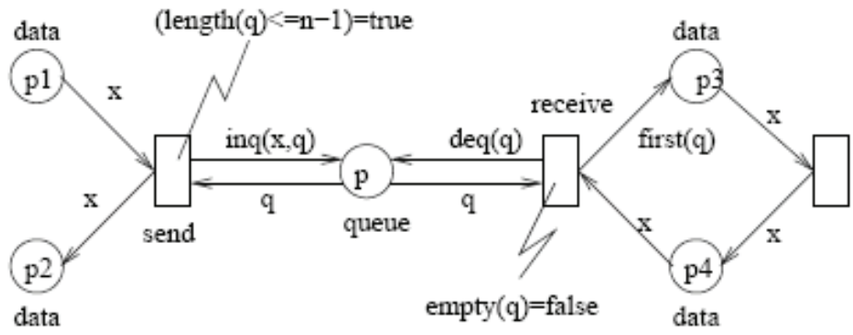


Fig. 2. Sender-Receiver Model by Algebraic High-Level Net

2.4 Linear temporal logic

Temporal formulas are built from elementary formulas using logical connectives $\neg$ and $\wedge$ (and derived logical connective $\vee$, $\Rightarrow$, and $\Leftrightarrow$, universal quantifier $\forall$ (and derived existential quantifier $\exists$), and temporal operators always $\Box$, future $\Diamond$, and until $\mathcal{U}$.

The semantics of temporal logic is defined on behaviors (infinite sequences of states). The behaviors are obtained from the execution sequences of petri nets where the last marking of a finite execution sequence is repeated infinitely many times at the end of execution sequence. For example, for an execution sequence M_0, „ Mn, the following behavior $\sigma = << M_0, \ldots, Mn, Mn, \ldots >>$ is obtained, where M_i is a marking of the Petri net.

Let $\sigma = << M_0, M_1, \ldots >>$ be the behavior, where each state M_i provides an interpretation for the variables mentioned in predicates. The semantics of a temporal formula p in behavior σ and position j is denoted by (σ, j) j= p. We define:

- For a state formula p, $(\sigma; j) \models p \Leftrightarrow M_j \models p$;
- $(\sigma; j) \vDash \neg p \Leftrightarrow (\sigma, j) \nvDash p$;
- $(\sigma; j) \vDash p \vee q \Leftrightarrow (\sigma, j) \vDash p$ *or* $(\sigma, j) \vDash q$;
- $(\sigma; j) \vDash \Box p$, $(\sigma, i) \vDash p$ for all $i \geq j$;
- $(\sigma; j) \vDash \Diamond p$, $(\sigma, i) \vDash p$ for some $i \geq j$;
- $(\sigma; j) \vDash p \mathcal{U} q \Leftrightarrow \exists i \geq j : (\sigma, i) \vDash q$, *and* $\forall j \leq k < l$, $(\sigma; k) \vDash p$.

2.5 Component and connector view

Component and connector view was one of the four views proposed in [31, 32], which is described as an extension of UML. The component and connector view describes architecture in terms of application domain elements. In this view, "the functionality of the system is mapped to architecture elements called components, with coordination and data exchange handled by elements called connectors." [31]

In the component and connector view, components, connectors, ports, roles and protocols are modelled as UML stereotyped classes. Each of them is represented by a special type of graphical symbol, as summarized in Fig. 3. A component communicates with another component of the same level only through a connector by connections, which connect relevant ports of components and roles of connectors that obey a compatible protocol. In addition to the connections between components and connectors, ports (roles, resp.) of a component (connector, resp.) can be bound to the ports (roles, resp.) of the enclosing

component (connector, resp.).
In order to present our approach we use an image processing example used in the distributed web application that was adapted from [31]. Fig. 4, 5, 6 from [31] shows a concrete and complete component and connector view, which is the running example of this paper. Fig. 4(a) is a configuration of *ImageProcessing* component. Fig. 4(b) shows another aspect of the configuration. Both of them are UML class diagrams and model different aspects of the system. From these two figures, we can see the component *ImageProcessing* contains two components: *Packetizer* and *ImagePipline*, and one connector *Packet-Pipe*. The ports of component *ImageProcessing*, *raw-DataInput*, *acqControl*, and *framedOutput* are bound to ports *rawDataInput* of component *Packetizer*, *acqControl* and *framedOutput* of component *ImagePipeline* respectively. Component *Packetizer* communicates with component *ImagePipeline* through connector *PacketPipe*. Component *Packetizer* and connector *PacketPipe* is connected by a connection between port *PacketOut* and role *source*, which obey (conjugate) protocol *DataPacket*. Component *ImagePipeline* and connector *PacketPipe* is connected by a connection between port *PacketIn* and role *dest*, which obey (conjugate) protocol *RequestDataPacket*. Being a conjugate means that the ordering of the messages is reversed so that incoming messages are now outgoing and vice versa.

Element/Relation	UML Notation	Graphical Symbol
component	Class <<*component*>>	
connector	Class <<*connector*>>	
role	Class <<*role*>>	
port	Class <<*port*>>	
protocol	Class <<*protocol*>>	
composition	Composition	
binding	Association	
connection	Association	
obeys	Association	obeys
obeys conjugate	Association	obeys conjugate

Fig. 3. UML Extension for component and connector View

Fig. 4. alone is not enough to illustrate component and connector view since only components and connectors of the system and corresponding connections among them are demonstrated. Additional diagrams are needed to define protocols and functional behavior of components and connectors. A protocol, represented by a stereotyped class, is defined as

a set of incoming message types, a set of outgoing message types and the valid message exchange sequences. The valid message exchange sequence is represented by a sequence diagram. Fig. 5 shows the definition of *RawData*, *DataPacket*, and *RequestDataPacket* protocols. From Fig. 5(c), we can see protocol *RequestDataPacket* has one incoming message: *packet(pd)*, and three outgoing messages: *subscribe(c)*, *desubscribe(c)*, and *requestPacket(c)* where *c*; *pd* are parameters of messages. In order to communicate with object *B* based on protocol *RequestDataPacket*, object *A* first sends object *B* a message *subscribe(c)* where *c* indicates the sender *A*. Then a message *requestPacket* is sent to *B* to request a packet. Later, object *A* may receive a packet *pd* from *B*. The symbol "*" in the figure indicates that the pair of message *requestPacket* and *packet(pd)* may occur many times. Finally, object *A* sends a message *desubscribe(c)* to *B* to stop requesting packet.

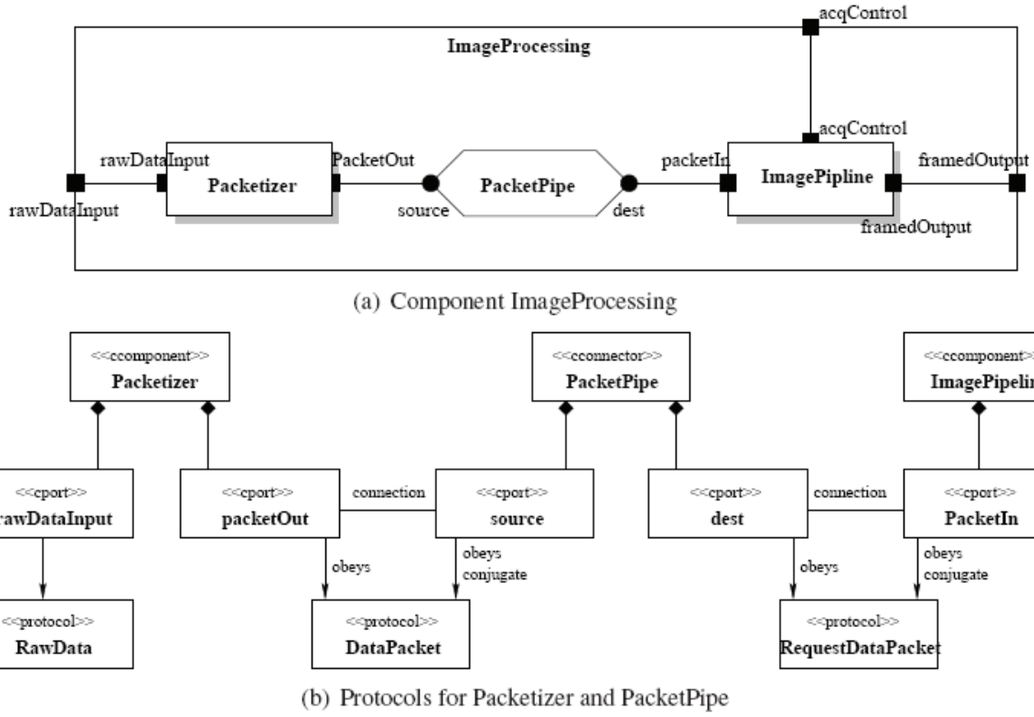


Fig. 4. Structural Aspect of Component and Connector View

The behavior of components/connectors may be described formally by UML statechart diagrams, for example the behavior of component *Packetizer* and connector *PacketPipe* in Fig. 6. Statechart diagrams describe the dynamic behaviors of objects of individual classes through a set of conditions, called states, a set of transitions that are triggered by event instances, and a set of actions that can be performed either in the states or during the firing of transitions. From Fig. 6(b), we can see the statechart diagram of connector *PacketPipe* contains two states: *waiting* and "*assign packet to ready client*" and seven transitions. When connector *PacketPipe* receives an event *subscribe(c)*, it invokes its operation *AddClient(c)* although we do not know exactly the functionality of this operation. When the connector receives an event *packet(pd)*, it saves the packet *pd*. And the response of connector *PacketPipe* to an event *requestPacket(c)* is up to the condition: the client *c* has read current packet or not. If yes, the connector treats it as a request for next packet; otherwise it sends current packet to client *c* through an event *c.packet*(*pd*). If all clients have read current packets, the connector updates its packet queue and enter state *"assign packet to ready*

client" in which the connector sends current packet to clients that has submitted their requests. If all requests are processed, the connector returns to state *waiting*. As we can see from this figure, connectors and components mainly handle incoming messages of protocols they obey (conjugate).

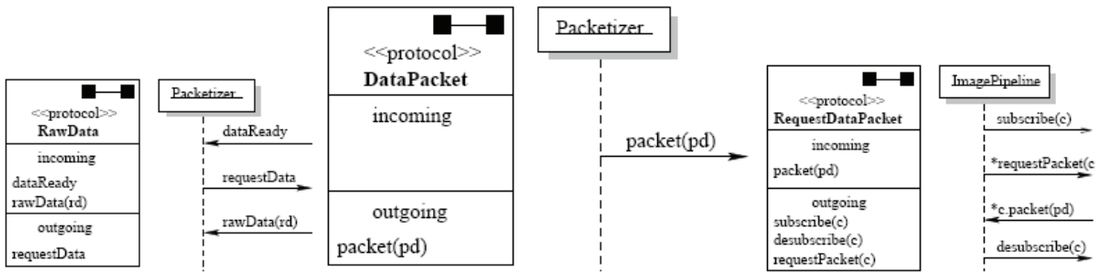


Fig. 5. Protocols in Component ImageProcessing

2.6 Service oriented software architecture model

Definition 1 (Web Service) *A web service is defined by a service component (composition of sub-services) as a tuple such that* $S\ N =< S_{ID}, f, Pt, ST >$ *where*

- S_{ID} *is the service identification. When a service component is a composition of sub-services, each sub-service has its service id and service component's id.*
- *f is SO-SAM structure mapping function.*
- *and* $Pt_{ini} \cup Ptf_{nl} = Pt$. *A service net does not have internal ports compared to service component, i.e.,* $Pt_{internal} = \varnothing$.
- *S T is a set of service constraints.*

The behavior of each web service S_i in SO-SAM is defined by a service net SN, which must starts when the initial ports Pt_{ini} has messages and ends when the final ports receive messages. The properties are defined using a set of temporal logic formulae ST. The relation between service net and the behavior model of a service component can be summarized as

- A service net SN is a subset of the behavior model of a service component $S\ N_{Cm}$, i.e., $S\ N\ \_\ S\ N_{Cm}$. A service net can be a single activity of a service component that describes only a sub-service of that service component, or a composite service of all subservices of that service component. The relation between a service net and other service nets of a service component can be publishing, binding, discovery, integration, etc.

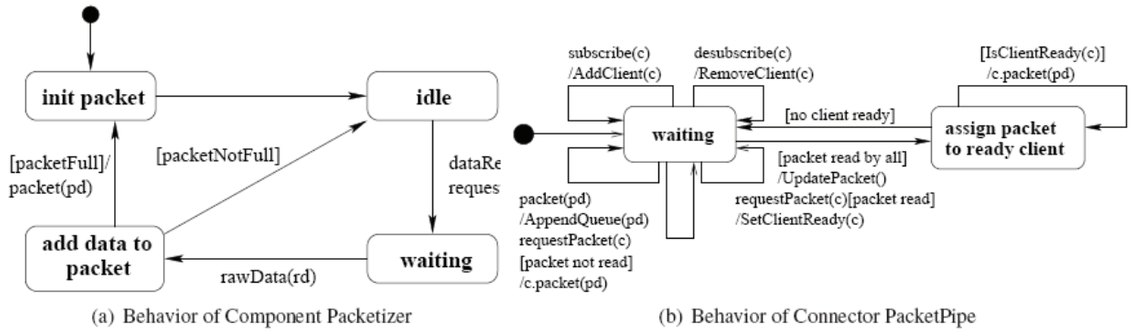


Fig. 6. State Chart Diagram of Elements

- Ports of a service net is a subset of ports of its service component, i.e., $Pt \in Pt_{Cm}$.
- If a service net SN is the behavior model of a service component SN_{Cm}, then the service component does not have internal ports, i.e., $Pt_{internal} = \varnothing$.
- Set of service constraints of a service net ST is a subset of service constraints of its service component, i.e., $ST \subseteq ST_{Cm}$.

Definition 2 (SO-SAM) *Service-Oriented Software Architecture Model, SO-SAM, is an extended version on the SAM, and defined by a set of service components* C_{Sm} *, a set of connectors* C_n*, a set of service constraints* C_{Ss} *and a structure mapping function f , i.e., SO-SAM ,* $< C_{Sm}, C_n, C_{Ss} >$.

2.7 Net specification

In the SO-SAM model we define three different group of sorts for three purposes: service description and publishing (SDP), service communication and binding (SCB), and service discovery or finding (SDF). In the service description and publishing (SDP) group, we identify four sorts in the net specification of a PrT net as portSpec, msg-Parameter, connection, and operation. Message is to specify the data identification flow through a port. Operation is to describe the operation can be imposed on messages. PortSpec is used to specify the parameters and functions of the service output from a port. Connection is to describe the protocol that used for the data flow through the port (which can be described by SOAP or HTTP). This group of sorts can be mapped to WSDL specification in the Table 1.

SDP in SO-SAM	WSDL	Description
portSpec	portType	The functions performed by the web services
msgParameter	message	The messages used by the web services
connection	binding	The communication protocols used by the web services
operation	operation	The operations can be performed on the message

Table 1. Mapping Relation between SDP (in SO-SAM) and WSDL

In the service discovery or finding (SDF) group, we concern two participants – service provider (SP) and service registry and broker (SR). Service provider has to have identification, contact info, category, service description, and so on. Service registry provides access point, communication protocol, and information about the company itself, including contact information, industry categories, business identifiers, and a list of services. Moreover, the binding process can be defined on the above specification. Let us use symbol $S\ DF =< SP, SR >$ denote all possible sorts using for the SDF group. After identifying these sorts, we can map a sort to a tag in the UDDI specification. However, reverse mapping from UDDI tags to SDF sorts is impossible. Because some tags such as *bindingTemplate* need functional description instead of signatures.

The behavior model in a SO-SAM refers to a Petri net, in this paper we use PrT net. The binding can be formally specified by the constraint function R. Checking the sat isfiability of R is to checking the each data in a message, data type matching, protocol conformation, and so on between web services from requestor and provider. Since a web service may have multiple binding templates, a mutual exclusion choice occurs.

The group of service communication and binding (SCB) is more related to the communication protocol (SOAP/HTTP). SOAP is a simple XML based protocol to let applications exchange information over HTTP. The communication protocol constructs the connection between parties. The specific sorts for the protocol signature can be message

definition of header, body and fault, encoding style etc..
All the above three groups are called service sorts $S_S = \{S_{SDP}, S_{SCB}, S_{SDF}\}$. For instance, *SName*, *SDesc*, *portS pec*, *message*,*URL* are service sorts, where *S Name* is the name (identification) of the service, *S Desc* is the service description, *URL* is the Uniform Resource Locator. We use $OP_{S\ s}$ to denote the operations on the service sorts. We call the sorts defined in SAM model data sorts S_D.
The extension on the signature of the sorts and operations is very convenient for the specification and modeling of the web services architecture, binding, substitution, and the composition of sub-services and their integration of legacy code.
Each architectural component is either statically or dynamically realized by a web services component. Architectural components are connected to each other via XML-based message passing through Simple Object Access Protocol (SOAP) [3]. The behavior of the connection is specified by SAM architectural connectors. The message passing mediates the interactions between architectural components via the rules that regulate the component interactions. In our model, connectors carry the tasks of service compositions. Thus our model supports both executable and dynamic web service compositions.
Ports in each component are either input ports or output ports. In the extension to web applications, ports are used to transfer messages among services, same as in SAM model. However, we regulate messages as a tuple with the information of service name, service description, location, URL, etc., so that the message carries service information.
A component is composed of the above ports that carry service information, behavior description and property specification. The behavior of a component is defined by a Petri net, which is called a service net. In the service net, tokens in a place has to have specific sort to be consistent with the above port and message definition. A basic component is one that does not have sub-components and non-empty connectors, otherwise, it is a composition. A composition is a composite service. The relation between a composition and its subcomponents and connectors is defined by a mapping function f . Mapping function f is a set of maplets from super component(connector)'s identities to sub-components'(connectors').
Service integration and composition can be done through connectors. Connectors have the same definition as in SAM. The Petri net for a connector is a regular Petri net that describe the integration and composition of services. A connector cannot be a composition.

2.8 Net structure

There is a relation between the architecture elements of a component/connector and the elements in its behavior model (a PrT net). For each port of a component/connector, we have a corresponding one place defined in its PrT net. The sort of the port is same as the sort of the corresponding place. The relation between port and place can be defined as a port-place mapping function ξ as follows.
Definition 3 (Port-Place Mapping Function) *The behavior mapping function _ is a mapping relation from the set of ports of a component or a connector C_i to the set of places of its behavior model, a PrT net N_i. Let Pt be the set of ports, and P be the set of places in the behavior model, we simply use Pt and P to represent the set of identifications of ports (places), we have, ξ: $Pt \rightarrow P$,*

1. $$\xi = \begin{cases} \exists p_i \in P: \ \forall pt_i \in Pt \\ \exists \phi(p): \ \forall \phi(pt_i) \in \phi(Pt) \end{cases}$$

2. If there is a message sent out from a port (pt_i), $M'(p_i) < M(p_i)$, where $f(pt_i) = p_i$; if there is a message received in a port (pt_i), $M'(p_i) > M(p_i)$, where $f(pt_i) = p_i$.

Obviously, this port-place mapping relation is also suitable for the SAM model. The port-place mapping function ξ connects a component/connector with its behavior model explicitly.

The behavior model of a component/connector of a SOSAM model, a PrT net, is called a service net. A service net is different from the behavior model of SAM in following aspects:

1. We identify the initial places and final places for a service net.
2. The set of sorts includes both service sorts and data sorts, i.e., $S = S_S, S_D$.
3. The behavior mapping function ξ defined a relation between a components/connector and its service net and is reversible. In other words, function ξ^{-1} exists.

Definition 4 (Port) *Ports in SO-SAM are communication interfaces of each service and graphically represented by semi-circles. Messages in ports are modeled by tokens. The sort of each token is defined by two parts, service sort S_S and data sort S_D. Data sort are the sorts defined in SAM [26].*

Thus we have sort S of a port is defined by $S \triangleq S_S, S_D$

A port may have a PrT net that associated to it by a function $\beta : pt_i \rightarrow N_{pt_i}$, where N_{pt_i} is a PrT net that used to describe the operations that can be performed on the sorts of the port pt_i.

Service sort S_S is a service description, query or binding requestor, which can include a service name, operation, description, URL, etc.. Service sort must be carried by all tokens in the service net. A token may be described or represented by a PrT net since it carries service information and some service information includes both messages and functions. If the port has an associated net N, the net is actually used to describe some of the tokens or some sort of the port. The net can be a service net. A service net is a PrT net that carries service characters, and is defined as follows.

Definition 5 (Service Net) *A service net is a Petri net defined by 8-tuple, $SN \triangleq < P, T, F, \phi, R, L, M_o, M_p >$, where*

- *P, T are finite set of places and transitions; F is flow relation: $P \times T \cup T \times P$.*
- *ϕ is sort assignment: $P \rightarrow \wp(S)$ ([26]), but sorts S are extended to carry service information.*
- *The final ports Pt_{fnl} of a service net communicate with a set of initial ports Pt_{ini} of another service net through a connector.*
- *R (guard function), L (label function), and M_0 (initial markings) follow the definitions in the paper [26].*
- *M_P is place mapping function, $M_P : P_t \rightarrow P$*

where each sort must contain S_S . Mapping function M_P associates each port to a place in the service net. It is a one-to-one mapping function, because in the SAM model the place and ports share the same name, and two ports between upper level and lower level share the same name. It is also an onto mapping function because all places that are identified as communicators to the other service nets will not be increased for a basic service component.

2.9 Architecture structure

We first give the definition of the architecture structure mapping function ζ as follows.

Definition 6 (Architecture Structure Mapping Function) *The architecture structure mapping function is defined as a relation from a composition to the element set of its subelements. We use C_i to denote a composition, Cm_i and Cn_i denote the component and connector, Pt_{C_i} , Pt_{Cm_i} , and Pt_{Cn_i} denotes the set of ports in a composition, component, and connector respectively. structurally, we have*

1. $\zeta = \begin{cases} Cm_i \cup Cn_i, & \text{if } Cm_i, Cn_i \in C_i \\ Pt^{Cmi} \cup Pt^{Cni}, & \text{if } Pt^{Cmi}, Pt^{Cni} \in Cm_i \cup Cn_i \wedge Cm_i, Cn_i \in C_i \end{cases}$

2. $\phi(Pt^{C_i}) \subseteq (Pt^{Cm_i} \cup Pt^{Cn_i})$.

The architecture structure mapping function ζ defines the structure relation between composition and its subcomponents and connectors. The function has two parts, one regulates the components and connectors in the composition, and another maps the ports of the composition with those of components and connectors. The constraints mapping can be considered in the behavior mapping. Since the behavior description of SAM architecture model is available in the bottom level of the hierarchy, we inherit this character from SAM directly without any update. Some service sorts Ss can be more abstract in the higher level abstraction. The result services after discovery and matching of these service descriptions can be satisfied with the service from requestor if there is more detailed information provided and discovered.

Definition 7 (SO-SAM Structure Mapping Function) *The mapping function f defined for SO-SAM model between two levels is the composition of behavior mapping function and architecture structure mapping function, i.e.,*

$f = \xi \circ \zeta$.

Considering the fact that behavior description is only available in the bottom level (which is inherited from SAM), this structure mapping function is also suitable for the mapping relation of SAM model.

Definition 8 (Service Component) *Each component C_{mi} in SO-SAM is defined by a tuple, component name Cm_iID, mapping function f , set of ports Pt that is composed of the set of input ports Pt_I and the set of output ports Pt_O, the set of initial ports $Pt_{ini} \in Pt$, the set of finial ports $Pt_{fnl} \in Pt$, a service net SN, and a set of temporal logic formulae ST, e.g., $C_{im} \triangleq < C_{im}ID, f, Pt_{ini}, Pt_{fnl}, Pt_{internal}, SN, ST >$.*

Initial ports are represented by dash line bold half circles, and final ports are represented by solid line bold half circles. Each set of initial ports in a service component must connect to a set of final ports in another service component through a connector, and vice versa.

In the component, each service must be started from one set of initial ports, but can be ended at multiple finial ports separately. This is because a service can reach different final states but starts at the same condition.

Connector of SO-SAM model is used to but not limited to describe the following activities:

1. Service publishing. This is an advertisement process of a service provider. The descriptions of a service or update of a service description is disclosed to possible requestors. A locating of possible matching service can be done afterwards.
2. Service discovery. We consider service discovery and finding to be the process of locating candidate service providers. Service repository maintains lists of service providers categorized according to proprietary classification schemes. Service requestor located the service based on the request and service description provided. Temporal and spatial availability for all requests is demanding for a service. Request refinement does not belong to this process.
3. Service binding. Service binding is a process that based on the discovery a connection between provider and requestor is established. A protocol for the negotiation is used after discovery.
4. Service substitution. Substitution uses accurate service descriptions to allow rational

optimization of sub-services within a composition. Assume we have two services A and B, Service A may be an electronic new report and service B an electronic weather report. If we try to outsource them then difficulties arise. A may only be offered in the USA and B in Chile. Composition of them becomes useless if you live in iceland; and pretty useless too if A is available on weekdays and B only on weekends. This raises the notion of *substitutability* in the context. The subsititution one function F by another G can be done if G has weaker preconditions and stronger postconditions. Some rules can be established for service substitution by weakening preconditions.

5. Service integration. The majority of businesses today are in an extremely dis-integrated state. Years of piecemeal IT development and investment have led to a situation common in all but the smallest organizations where there are numerous non-compatible systems. Many have been developed on different platforms using different languages. Thus organizations have created numerous barriers within their own IT infrastructures. Web services define a transport method for processes defined in XML. At the core of the Web service revolution is their ability to allow code to speak to code without human intervention. In the SO-SAM model, the connector provides the formal specification that connects service components with different interfaces. The constraint function *R* of the transitions in a connector defines the required messages in the input ports and describes the messages flow to the output ports.
6. Service composition. Compositions produces tightly-coupled integration between sub-services to ensure that value is added over the sum of the individual service. The question is: if we have two trusted services *A* and *B*, after composition of *A* and *B*, we have a service *C*, is this service *C* trustable? Simply, the question is the composition of sub-services can still hold the properties of its sub-services or not. Connectors in the SO-SAM model can formally describe the composition of subservices, thus it is possible for the formal verification of composed service against service properties.

3. Transformation from component and connector view to SO-SAM

Component and connector (C&C) view [42] has been the main underlying design principle of most ADLs [36], which is also a major view type in several software architecture documentation models supporting multiple architecture views such as SEI [39] and Siemens [31]. C&C view is essential and necessary for system dependability analysis since it captures a system's dynamic behavioral aspect. SO-SAM model and component and connector view share a set of common terms such as components, connectors, and ports. Therefore it is straightforward to map them to the counterparts in SO-SAM. However, due to the meaning difference and various formal methods to describe elements' behavior, the concrete mapping procedure is not that easy. This section shows a method to construct a complete and executable SO-SAM model from a component and connector view.

A component (connector, resp.) in component and connector view is mapped to a service component (connector, resp.) in SO-SAM model. It is easy to understand from structural aspects. However, the behavior mapping is complex since different formal methods are used to model behavior. UML statechart diagrams are used to model behavior in component and connector view, contrasting with Petri net model in SO-SAM. Fortunately, our previous work [12] showed that it is possible to transform statechart diagrams to Petri net models. In UML statechart diagrams, method invocations and relationships between variables are implicit in the elements' structure. For example, in Fig. 6(a), the conditions

PacketNotFull and *PacketFull*, and relationship between variable *rd* and *pd* is not illustrated explicitly. However, such information has to be expressed explicitly in order to obtain a complete and executable Petri net. In order to bridge the gap, we utilize algebraic high level nets [17], a variant of high level Petri nets, to model behavior of elements. This method is possible because SO-SAM model does not specify a particular Petri net model as its formal foundation. We use algebraic specifications [15] to capture structures
of elements obtained from UML statechart diagrams because algebraic specifications are abstract enough that no additional information about implementation detail is assumed, and they are also powerful enough to represent implied information about components or connectors. Although the work [12] is for SAM architecture model, we can still use it and adapt it to the SO-SAM model since they share the same net structure. The main differences exist in the service sorts in the net specification, initial and final ports in the net specification and net inscription. The following rule gives us a general idea to derive components or connectors in SO-SAM.

Rule 1 (Component and Connector) *A Component (connector, resp.) in component and connector view is mapped to a service component (connector, resp.) in SO-SAM according to following steps:*

Step 1 *An algebraic specification, which specifies the abstract interface of the component (connector, resp.), is generated from a UML statechart diagram. The idea to construct algebraic specification is described later.*

Step 2 *Construct a complete and executable algebraic high level net from the UML statechart diagram according to the approach in [12] and the generated algebraic specification. There is a special place in the generated algebraic high level nets that contains element information and provides necessary information for transitions.*

Step 3 *A component (connector, resp.) with a UML statechart diagram in component and connector view is mapped to a component (connector, resp.) with an algebraic high level net in SO-SAM.*

Step 4 *A composited connector in the component and connector view is flattened and mapped to a connector in the SO-SAM model.*

While it is inherently impossible to prove the correctness of the transformation, we have carefully validated the completeness and consistency of our transformation rule. First, from structure point of view, concepts of components or connectors in component and connector view and SO-SAM are the same. Both of them support component composition, binding with enclosing element, and they can have their own behavior and communication channels–ports or roles. Therefore, the main functionalities of components or connectors in component and connector view are presented in SO-SAM counterparts. Second, algebraic specification can be used to specify modular, more specific classes [16]. Therefore, the implied information of statechart diagrams, i.e. the operations and their properties can be correct and fully specified by algebraic specifications. Since functions of algebraic specifications only define what to be done, no additional implementation information not implied in statechart diagrams is introduced. Finally, our previous work [12] and others work [41] have shown that the behavior described by statechart diagrams can be fully captured by corresponding Petri nets.

The idea to obtain algebraic specifications from UML statechart diagrams is as follows:

- For each element (component and connector), its algebraic specification defines a sort, called *element sort*, like *packetizer* for component *Packetizer* and *packetpipe* for connector *PacketPipe*. If a data type of parameters is not defined by a primitive algebraic specification, a new algebraic specification is imported. Such a imported algebraic

specification generally defines only one sort, like *Packet* for component *Packetizer*. Each action of transitions in a UML statechart diagram is considered as a function such that: *action name* : *element sort* × *parameters sort list* → *element sort*. Here *parameters sort list* includes service sorts as well. For a guard condition of a transition, a function from *element sort* (with necessary parameter sorts) to boolean is added.

- For each variable that is defined in the element (i.e. the variable is not defined in the related events), a function like *GetVariableTypeName* : *element sort* → *element sort* × *VariableType*[2] is specified. The properties of these functions can be constructed as equations if they are implied in the UML statechart diagrams, like guards *PacketNotFull* and *PacketFull* cannot hold at the same time.

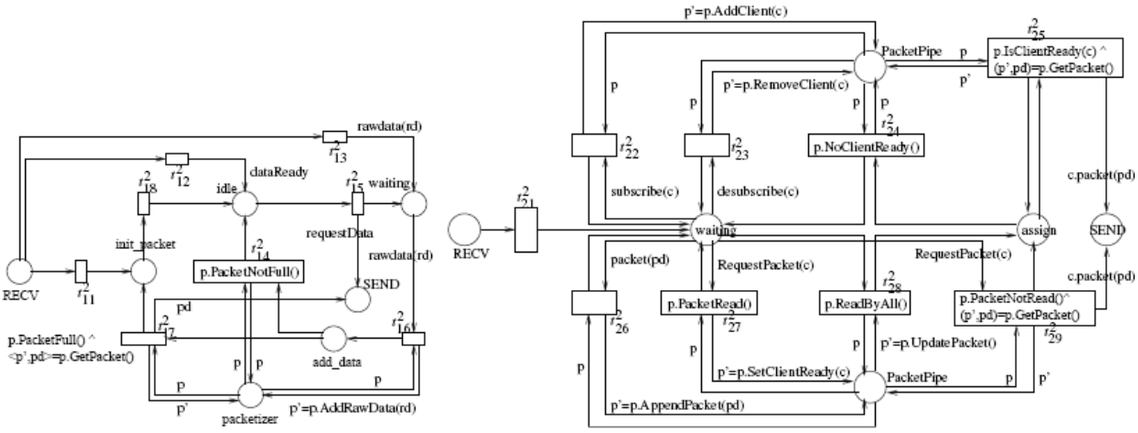
From the above idea, we know that the algebraic specification for component ***Packetizer*** contains four functions, two of them correspond to guard conditions_.***PacketNotFull***(), _.***PacketFull***() : ***packetizer*** → ***bool***; and one is obtained from actions: _.***AddRawData***( ) : ***packetizer*** × ***rawdata*** → ***packetizer***, and one from undefined variable: _.***GetPacket***() : ***packetizer*** → ***packetizer***× ***packet***. In these functions, "_" is used to indicate a variable placeholder, ***bool*** is a sort defined in primitive algebraic specification Bool [15], and sorts ***rawdata*** and ***packet*** are defined in imported algebraic specifications ***Packet*** and ***RawData*** respectively, which are defined by users and normally only one sort (***rawdata***, ***packet*** resp.) is specified. Appendix A gives complete algebraic specifications of component ***Packetizer*** and connector ***Packet-Pipe*** obtained from Fig. 6.

With these algebra specifications, we can generate corresponding algebraic high level nets according to Rule 1. Fig. 7 shows the generated Petri nets from UML statechart diagrams in Fig. 6. Each generated Petri net has three special places: RECV containing messages from environment, SEND temporarily storing messages generated for its environment, and the place whose name is the same as its element's name (here, ***Packetizer*** and ***PacketPipe*** resp.), holding the abstract structural information of the element. In additional to these three places, there is a corresponding place indicating current status for each state in statechart diagrams, for example, places ***idle***,***waiting***, ***init packet*** and ***add data*** for the same name states. A special token in these places indicates if the corresponding state is active. Place RECV sends events from external environment to places that are interested in the event. In Fig. 7(a) place ***idle*** and ***waiting*** are interested in event ***dataReady*** and ***rawdata(rd)*** respectively. If state ***idle*** is active and an event ***dataReady*** is available, transition t_{15} is fired. As a result, an event ***requestData*** is added to place SEND, and place ***waiting*** becomes active. State ***add data*** becomes active if state ***waiting*** is active and an event ***rawdata(rd)*** is available. At the same time, the token in place ***packetizer*** is changed to another one through operation _.***AddRawData***(_) of algebraic specification ***Packetizer***.

Components and connectors in component and connector view are connected through a connection if they are enclosed directly by the same element and the corresponding ports and roles obey (conjugate) a compatible protocol. Therefore, the mapping from ports or roles in component and connector view to ports in SO-SAM is actually the mapping from relevant protocols describing behavior of ports or roles to ports of SO-SAM components/connectors. However, ports in SO-SAM models have their own characteristics.

[2] This is actually the abbreviation of two functions: *GetVariable* : *element sort* → *VariableTypes* and *UpdateElement* : *element sort* → *element sort* since these two functions are invoked sequently in our example.

A port in SO-SAM model is a place that has either no incoming arcs or no outgoing arcs. In other words, the communication between ports is unidirectional. Therefore, a protocol in component and connector view, which consists of a set of incoming message types, a set of outgoing message types and the valid message exchange sequences, is mapped into a set of interface places (To avoid confusion, we use interface places to refer to ports in SOSAM model). The type of tokens in an interface place is $OID \times OID \times MESSAGE\_TYPE$, where OID is a set of unique identification number for each instance of the element, which specifies sender and receiver of a message, and $MESSAGE\_TYPE$ is the set of message types of the protocol (Here we ignore the parameters of messages for brief). Rule 2 specifies how to map a port/role in component and connector view to interface places in SO-SAM.



(a) Petri net for Packetizer (b) Petri net for PacketPipe

Fig. 7. Behavior Model of Elements

Rule 2 (Ports and Roles) *A port (role, resp.) of a component (connector, resp.) in component and connector view is mapped to a set of interface places of the corresponding component (connector, resp.) specified by Rule 1: For each protocol that the port (role, resp.) obeys (conjugate), each kind of incoming messages is mapped to an incoming (outgoing, resp.) interface place of the component (connector, resp.) with the name of the message type; and each kind of outgoing messages is mapped to an outgoing (incoming resp.) interface place of the component (connector, resp.).*
Initial and final ports can not be obtained from the UML architecture description directly. We provided two possible solutions:

- *One is extending C&C view with new UML stereotypes initialPort and finalPort. This would bring a direct transformation from C&C architecture to SOSAM. The problem is this also brings more complexity into UML architecture description.*
- *Another is manually adding the specification for these ports according to the system architecture description. For instance, we can say dataReady and RawData as initial port and frameout and final port in our case.*

A port (role, resp.) of an element is actually –roughly speaking– a "channel" that forwards messages of specified types either from element itself to environment, or from environment to element. In Rule 2, a token represents an occurrence of an message of specified type, and the direction of a message is specified by the place containing the token – incoming or outgoing. Therefore, the mapping in Rule 2 conserves the main structural features of ports/roles and related protocols, and the reverse mapping exists, which ensures the

correctness of the rule.

The behavior of a protocol, defined by UML sequence diagrams to demonstrate valid message exchange sequences, actually specifies possible sequences of relevant messages along time axle. A sequence of protocol messages illustrates their occurrence order, which can be specified by a set of temporal constraints, the basic predicates of which are the names of interface places obtained through Rule 2. For example, from Rule 2, we know port ***RawData*** of ***Packetizer*** is represented by two incoming places ***dataReady*** and ***rawData***, and one outgoing place ***requestData***. We use predicate ***dataReady***(<***sid***, ***rid***,$m_{dataReady}$>) to describe if place ***dataReady*** contains a token representing an event ***dataReady*** that is sent to ***rid*** by ***sid***.

In order to construct temporal constraints, we consider two elements communicating with each other through a protocol, for example ***RawData***. First we only consider a pair of adjacent events, for example ***DataReady*** and ***requestData***. For this pair of events, it means if an event ***DataReady*** occurs, then an event ***requestData*** must occur some time later, which is described by a temporal formula:

$$\forall <sid,rid,md>, \Box(dataReady(<sid, rid,md>) \rightarrow \Diamond requestData(<sid, rid, mr>)) \quad (1)$$

However, this temporal formula cannot reflect the situation implied in the sequence diagram of the protocol: no other events of the protocol can occur between events ***dataReady*** and ***requestData***. In order to describe this implied property, we have a reasonable assumption at architecture level that the communication media is reliable, no message is lost and no need to resend a message. Therefore, another temporal formula is introduced to address this missing situation:

$$\Diamond \forall <sid,rid,m_d>, \Box(dataReady(<sid,rid,m_d>) \rightarrow \neg((\bigcirc dataReady(<sid,rid,m_d>)) \vee$$

$$requestData(<rid,sid,m_r>) \vee rawData(<sid; rid;m_{rd}>)) \mathcal{U} requestData(<sid,rid,m_r>)) \quad (2)$$

This temporal formula means if an event of ***dataReady*** occurs, no other events such as ***dataReady***, ***requestData*** and ***rawData*** can occur before the first event of ***request-Data***. Predicate $\bigcirc$***dataReady***(<***sid***, ***rid***,m_d>) is used to guarantee that the temporal formula is satisfied at the time the event ***dataReady*** occurs. Therefore, given a sequence diagram of a protocol with ***n*** messages, we can obtain (***n*** - 1) × 2 temporal formulas.

In addition to the consideration of one session of a protocol, we have to inspect the relationship of two adjacent sessions of the same protocol between two objects, i.e. one session can start only after the previous session ends. Such a relationship is specified by a temporal constraint:

$$\forall <sid,rid,mrd>, \Box(rawData(<sid,rid,mrd>) \rightarrow \neg(dataReady(<sid,rid,md>) \vee requestData$$

$$(<rid,sid,mr>) \vee (\bigcirc rawData(<sid,rid,mrd>))) \mathcal{U} dataReady(<sid, rid,md>)) \quad (3)$$

Although we think the above generated constraints are strong enough, there is still one more case we ignored: the first session of a protocol in a running system may starts with any messages but the first message. For example, a session of protocol ***RawData*** starts with message ***dataReady***, and then obeys relevant part of the sequence diagram. We can see this session satisfies the above temporal formulas, but conflicts with the behavior of the protocol. Such a case can be avoided in three different ways, and the choice of them is up to users. One is to introduce a temporal predicate ***basetime*** that holds only at the time "zero", and a

new temporal formula:

$$\Box(basetime \rightarrow \neg(dataReady(<sid, rid, md>) \vee requestData(<rid, sid, mr>) \vee rawData(<sid, rid, mrd>)) \, U \, dataReady(<sid, rid, md>)) \quad (4)$$

The second method is to introduce a past time operator such as "eventually in the past". The final way is to prove that system structure guarantees that such case cannot happen.

Thus, from the above discussion, a sequence diagram for a protocol is mapped to a set of temporal constraints. Appendix B shows the full property constraints derived from the sequence diagrams of protocols *RawData*, *DataPacket* and *RequestDataPacket*.

The following rule is used to construct a set of constraints for components or connectors according to the above discussion.

Rule 3 (Constraint) *For each protocol that a port (role, resp.) obeys (conjugate), a set of constraints, generated from the corresponding sequence diagram according to the above discussion, is added to the property specification of corresponding components (connectors, resp.) . When a constraint is added to a component (connector, resp.), sid or rid in tokens (the choice is up to the direction of corresponding message) is substituted by the actual identification number of the component (connector, resp.) since the component (connector, resp.) can only receive messages sent to itself.*

A sequence diagram of a protocol specifies possible message communication sequences. However, it is impossible to limit the firing sequences of transitions in Petri nets to meet specified occurrence sequences of tokens in places. Although we cannot specify the firing sequences of transitions, but we can prove that if each possible firing sequence meets the behavior of a protocol. From the above discussion, we can see the generated set of temporal formulas exactly realizes the behavior of a protocol – the message sequences. By adding these temporal formulas as property specifications to components/connectors obeying the protocol, inconsistencies between behavior of elements and protocols can be easily detected. Since the behavior mapping in Rule 1 is complete and consistent, we know the detected inconsistencies also exist in the original model, i.e. Rule 3 is complete and consistent.

We may obtain a component (connector, resp.) with a behavioral model, and related ports and constraints according to Rules 1, 2 and 3 respectively. Next task is to get a complete component or connector, i.e. ports of a component or connector has to be integrated with its behavior model. Rule 4 is used to guide such a procedure, and Rule 5 establishes the connection between components and connectors.

Rule 4 (Integration) *The interface places, i.e. ports of a component (connector, resp.) in SO-SAM are integrated into its behavior model with the previous generated algebraic high level nets according to the following steps:*

Step 1 *Each incoming interface place is connected to place RECV through a transition, firing of which transmits tokens in the incoming place that are sent to the instance of component or connector to place RECV unconditionally.*

Step 2 *Each outgoing interface place is connected to place SEND through a transition, which forwards tokens of a special type in place SEND to the outgoing place.*

Rule 5 (Connection) *From Rules 2 and 4, if there is a connection between ports of a component and a role of a connector, then generated behavior models of the component and connector share a set of places that corresponds to the protocol they obey (conjugate). Therefore, to establish the connection between a component and a connector in SO-SAM, we merge these shared interface places because an incoming (outgoing, resp.) interface place in the component has an outgoing (incoming, resp.) counterpart in the connector such that they contain messages of the same type, and vice versa.*

In component and connector view, relationships between ports and behaviors are not specified explicitly. A port forwards incoming messages to the queue of the component/ connector, which provide events for its behavior – the statechart diagram. The statechart diagram sends messages to its environment through a port. In Rule 4, place SEND serves as output queue and place RECV is input queue. The forward action is represented by the firing of transitions connecting place SEND, RECV and other interface places. Therefore Rule 4 captures the communication between ports and the corresponding behaviors in component and connector view.

Due to the space limitation, we cannot specify the transformation of binding and multiplicity. However, such transformations are similar and straightforward. Fig. 8 shows the final result of generated SO-SAM model from the running example. In order to give a concise description, algebraic specifications and internal parts of behavioral models are omitted.

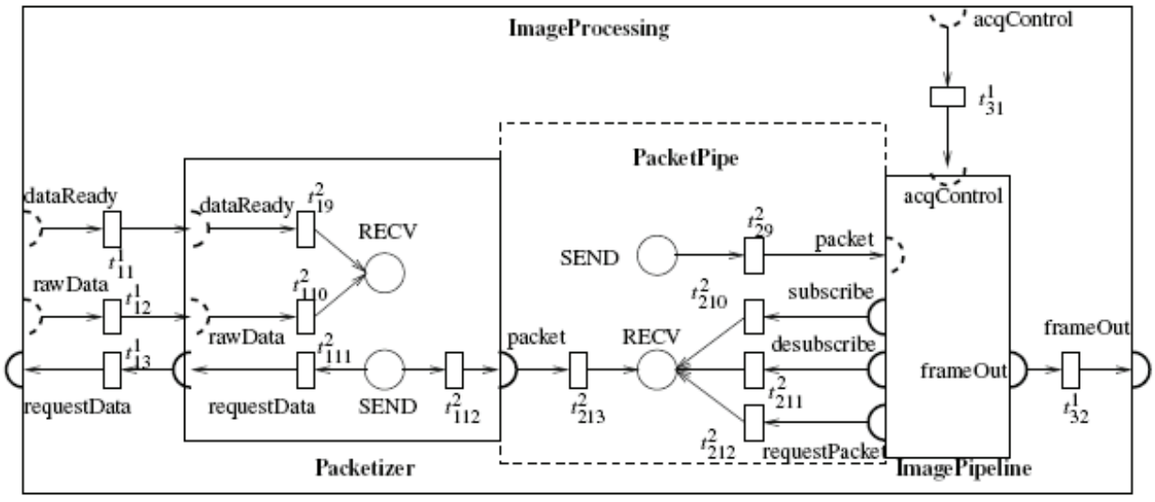


Fig. 8. ImageProcessing in SO-SAM

In Fig. 8, components, for example component *Packetizer* and component *ImageProcessing* are represented by solid rectangles, while connectors such as *PacketPipe* are represented by dashed rectangles. The Petri nets enclosed by rectangles are the behavior models of components or connectors. Semicircles on the edge of rectangles are places that represent ports derived from relevant protocols. An inside semicircle indicates an incoming place that only receives tokens from environment, while an outside semicircle indicates an outgoing place that only sends tokens to environment. For example, component *Packetizer* has two incoming places *dataReady*, *rawData*, and one outgoing place *requestData*. These three places are derived from protocol *RawData* according to Rule 2. Component *Packetizer* and connector *PacketPipe* is connected through Rule 5. The binding between components and its enclosing component is implemented as a transition between corresponding places, which only forwards tokens from one place to another according to types of places (i.e. incoming or outgoing). For example, transition t_{11} forwards tokens in place *dataReady* of *ImageProcessing* to place *dataReady* of *Packetizer*, while transition t_{13} forwards tokens in place *requestData* of *Packetizer* to place *requestData* of *ImageProcessing*.

Finally, we give an execution path of component *ImageProcessing*. Let component *Packetizer* be in state *idle*, connector *PacketPipe* in state *waiting*, and place *ImageProcessing.dataReady* contains a token representing message *dataReady*. This initial

condition can be represented by the initial marking (ImageProcessing.dataReady, Packetizer.idle, PacketPipe.waiting). Here we only list related places (not including places such as ***packetizer*** and ***PacketPipe***) that contain tokens, and ignore concrete token values that can be derived from context. We also assume that a packet consists of only one raw data, i.e. operation ***PacketFull***() will be true if ***AddRawData***() is invoked once. Table 2 shows the execution of communication based on protocols ***RawData*** and ***DataPacket***. This example demonstrates the application of our method.

4. Validation of the approach

The SO-SAM model allows formal validation of a service net against system constraints and property specified on its abstraction represented by a component or connector. Here, validation means that the developer can animate the specification by providing initial markings and checking if the responses meet the expected results. Validation of SO-SAM is based on the precise syntax and semantics of Petri net formal language and temporal logic. The validation will cover the topology and dynamic behavior of the Petri net as well as temporal logic formulae. Here we simply introduce how to translate SO-SAM model to the Maude [9] language. For the details, please refer to the work [22].

Step	Marking of Component *ImageProcessing*	Fired Transition
1	idle, ImageProcessing.dataReady, PacketPipe.waiting	t^1_{11}
2	idle, Packetizer.dataReady, PacketPipe.waiting	t^2_{19}
3	idle, Packetizer.RECV, acketPipe.waiting	t^2_{12}
4	idle, PacketPipe.waiting	t^2_{15}
5	Packetizer.waiting, Packetize.SEND, PacketPipe.waiting	t^2_{111}
6	Packetizer.waiting, Packetizer.requestData, PacketPipe.waiting	t^1_{13}
7	Packetizer. waiting, Image Processing. request Data, PacketPipe. waiting	unspecified transition
8	Packetizer.waiting, PacketPipe.waiting	unspecified transition
9	Packetizer.waiting, ImageProcessing.rawData, PacketPipe.waiting	t^1_{12}
10	Packetizer.waiting, Packetizer.rawData, PacketPipe.waiting	t^2_{110}
11	Packetizer.waiting, Packetizer.RECV, PacketPipe.waiting	t^2_{13}
12	Packetizer.waiting, PacketPipe.waiting	t^2_{16}
13	Packetizer.add data, PacketPipe.waiting	t^2_{17}
14	Packetizer.initial packet, Packetizer.SEND, PacketPipe.waiting,	t^2_{18}
15	Packetizer.idle, Packetizer.SEND, Packetpipe.Waiting,	t^2_{112}
16	Packetizer.idle, Packetizer.packet, PacketPipe.waiting	t^2_{213}
17	Packetizer.idle, PackePipe.RECV, PacketPipe.waiting	t^2_{21}
18	Packetizer.idle, PacketPipe.waiting	t^2_{26}
	Packetizer.idle, PacketPipe.waiting	

Table 2. A Path of Executing Protocols RawData and DataPacket

4.1 Translation from SO-SAM to Maude

First, we presented a stepwised translation algorithm from SO-SAM model to Maude programming language. After that, the experimental results are illustrated.

Step 1. Translation to the functional module: generate the sorts operators used in the functional modules for the model signatures. This step translates each place, sorts, markings in a Petri net into the corresponding part in Maude's functional module.

Step 2. Translation to the system modules: there are three types of system modules, one is for the model signature that corresponds to the architecture structure and dynamic behavior of the model, one is for the mapping to the predicates, and one is for the model checking, which includes the property specification.

1. Each basic component and connector are defined as a system module (SysID) with the declaration of variables and necessary rules and operators. Each composition is specified as a system module that including its sub-components and connector that are predefined as a module. All guard conditions in a transition are a (un)conditional rule.
2. Each place is mapped to an operator in the predicate system module (SysID-PREDS). The connection between operators and predicate is established by an equation.
3. Model checking module (SysID-CHECK) is mainly for the initial marking and property specification.

In our translation, system signature such as sorts and operators are declared in the functional module. This translates the places/ports, sorts into algebra in Maude that will be used in the system modules. The dynamic semantics of Petri net can be mapped to the rewriting rules used in Maude. Computationally, the meaning of rewriting rules is to specify local concurrent transitions that can take place in a system if the pattern in the rule's lefthand side matches a fragment of the system state and the rule's condition is satisfied. In that case, the transition specified by the rule can take place, and the matched fragment of the state is transformed into the corresponding instance of the righthand side. Thus we can see an amazing match between semantics of Petri net and rewriting logic. These are theoretic aspect of the above translation algorithm.

4.2 Results

The basic requirements for the image processing in the distributed web applications are correctness, robustness and reliability. We use model checker of Maude [9] to validate the SO-SAM model obtained from UML architecture description against system properties. After studying models and the errors discovered during the model validation, two main property categories have been selected:

1. Structural properties: this kind of properties is closely related to the topology of the model. These properties can be directly verified on the SO-SAM model without animating the transactions. These properties are necessary conditions that ensure the feasibility of the state transitions. If one of them is not fulfilled, we can assert firmly that the communication between ports in UML description cannot happen.
2. Behavioral properties: the dynamic feature of these properties means that they are related to state changing of the system. The evaluation of the dynamic properties are based on the behavior description – Petri nets. Its verification is achieved on a set of places describing a possible evolution of the system. All four properties in section 3 fall in this group. The results output from Maude are true for all the above formulae. Most the above formulae are safety properties.

The results can be obtained within 10ms. It is worth to notice that the model checking technique used for the verification of system properties are only available for propositional formula. For the first order formula, it is still a challenge research topic in this area.

5. Related work

We can identify in the literature two categories of works that are mostly related to our research. The first one concerns works that modeling service oriented architecture descriptions using UML. The second one is composed of the works of formalizing the semantics of SOA in different aspects.

5.1 UML description of SOA

In the first category most use UML profiles to describe the service oriented architecture. [11] proposed UML profiles to specify functional aspects in SOA, which are defined based on the XML schema of Web Service Description Language (WSDL) [5]. The profile provides a set of stereotypes and tagged values that correspond to elements in WSDL, such as *Service*, *Port*, *Messages* and *Binding*. There is no consideration of nonfunctional aspects of web services. In work [34] a case study is presented on the investigation of the UML profile specification of SOA.

Compared to work [34] and [11], [6] proposes a UML profile to describe both functional and non-functional aspects in SOA. This work provides generic stereotypes to specify a wide range of applications. However, the semantics of this profile tend to be ambiguous. For example, several stereotypes for nonfunctional aspects (<<policy>>, <<permission>> and <<obligation>>) are intended to specify the responsibility of a service. There is no precise definition of how developers specify web applications with these stereotypes.

[29] proposes a UML profile to facilitate dynamic service discovery in SOA. This profile provides a set of stereotypes (e.g., <<uses>>, <<requires>> and <<satisfies>>) to specify relationships among service implementations, service interfaces and functional requirements. For examples, users can specify relationships in which a service uses other services, and a service requires other services that satisfy certain functional requirements. These relationship specifications are intended to effectively aid dynamic discovery of services.

[23] and [10] define UML profiles to specify service orchestration in UML and map it to Business Process Execution Language (BPEL) [1]. These profiles provide a limited support of non-functional aspects in message transmission, such as messaging synchrony. The proposed profile does not focus on service orchestration, but a comprehensive support of non-functional aspects in message transmission, message processing and service deployment.

[43] describes a UML profile for data integration in SOA. It provides data structures to specify messages so that users can build data dictionaries that maintain message data used in existing systems and new applications. The non-functional aspect of data integration is separated from functional one in this profile. Data integration can be enabled in an implementation independent manner.

There is less work on the service architecture description using UML architecture model. [40] specifies a series of service architecture patterns using UML service component. For instance, interaction service pattern describes capabilities and functions to deliver content

and data using a portal, or other related Web technologies, to consumers or users. This work infuses the component design with service building block to facilitate large scale system design. However, there is no formal reasoning of these patterns and how develop workers to use these patterns.

5.2 Formalizing SOA

In this paper we have briefly shown how UML architecture description model can be formalized using a biformalism SAM extension – SO-SAM, and the benefits that can be obtained from such formalization, namely the definition of integration and composition verifications between services, and the architecture reasoning that can bridge the differences between a priori incompatible Web services. Thus we have shown how existing formal methods can be successfully applied in the context of Web services, providing useful and practical advantages.

In addition, the formal specification of service properties using temporal logic provides us with a tool for expressing other complicated safety and liveness properties (apart from those already mentioned). In fact, any property expressed as a temporal logic formula can be considered as a sub-system specification, and therefore, checking that property on a certain web service component, would consist in reasoning the service-oriented architecture. On the other hand, having a simple formal description to describe web service architecture and integrations will allow us the application of model-checking techniques to construct (or extend) existing validation tools, as made in [19] with Promela.

Two major approaches for describing web service applications can be categorized: (a) the application oriented view of the service oriented applications or web systems (built only on the individual WSDL descriptions of the constituent web services); (b) the platform independent, architecture oriented view of service-oriented applications, which consists of different (simple) "global model" that describes how such independently defined service integration and compostion in high level abstraction.

BPEL4WS, WSFL and WSCDL are notations that use the application oriented view approach, whilst UML profile, service components, and web component are examples of the architecture oriented view approach. Application oriented view notations are in general more adaptable to each particular situation and system, but are not as amenable to web service reuse as architecture view descriptions are. Although the web service community is currently divided trying to decide which is the best approach, we argue that they can be considered as complementary tactics, rather than rivals.

The way to marry both approaches can be achieved by integrating and infusing the results from different categories, similarly like what we have discussed in this paper, mapping the UML architecture description to SO-SAM model and simply checking that the system properties defined over its constituent web services that can be replaced (in our sense), integrated or composed by their individual constituents (can be defined using an application oriented view approach). In this way, both approaches could easily co-exist.

Apart from the previous work of the authors [20, 22], there is a large amount of proposals in the literature dealing with composition, interoperation and adaptation issues in the field of Component-Based Software Engineering (CBSE), and in protocol verification in general [19]. Some of these works have been also applied to web service architecture reasoning. In cite [18], building on previous work in the field of Software Architecture by the same authors, a model-based approach is proposed for verifying Web service composition, using Message

Sequence Charts (MSCs) and BPEL4WS. In [38], and from a semantic Web point of view, a first-order logical language and Petri Nets are proposed for checking the composition of Web services. In [19], model-checking using Promela and SPIN is proposed for analysing the composability of choreographies written in WSFL. All these works deal with the (either manual or automated) simulation and analysis of Web service composites, been able to detect mismatch between their choreographies.

6. Conclusion

In this paper, we proposed a method to use SO-SAM to formally specify service-oriented application architectures modeled by an extension of UML – component and connector view. By doing so, we combine the benefit of UML – easy to comprehend and extensive tools support, and the analyzability of SO-SAM.

The cost of our methods mainly comes from three parts: the construction of algebraic specifications, the generation of algebraic high-level nets from statechart diagrams, and the creation of temporal formulas from sequence diagrams. Since an algebraic specification is used to model the implied information of statechart diagrams, generally speaking we can generate operation and sort definitions of an algebraic specification automatically, but not for the relationships among these operations. The size of a generated algebraic specification is "linear" to the size of implied information. From our previous work [12], we know the generation of Petri nets from a statechart diagram can be fulfilled automatically for most cases, and a Petri net and the corresponding statechart diagram are at the same size. The generation of temporal logic formulas from sequence diagrams can be largely automated since the generation is very simple and straightforward.

There are at least three immediate extensions to the work we have presented here. First, we intend to integrate the translation from UML architecture to SO-SAM with the mapping from SO-SAM to Maude so that some existing tool we have developed can be used for the model checking of system properties. And second, we intend to make effective use of the tools currently available for SAM model [21] to reason about the web specifications during the runtime. Finally, the translation into SO-SAM presented here must be extended in order to consider full application oriented view approach such as WSCI [4]; in particular, dealing with constructs such as correlations, transactions, properties and others, that have been omitted in this work. This extension would allow the analyzing on the more application oriented view approach using UML architecture descriptions.

7. Acknowledgments

This work is supported in part by Alabam A&M University.

8. References

[1] Business Process Execution Language for Web Services (BPEL4WS). Available from http://www.ibm.com/developerworks/library/wsbpel.
[2] DAML-S and OWL-S. Available from http://www.daml.org/services/owl-s/.
[3] Simple Object Access Protocol (SOAP), W3C Note 08. Available from http://www.w3.org/TR/SOAP/.
[4] Web Service Choreography Interface (WSCI) 1.0. Available from

http://www.w3.org/TR/2002/NOTEwsci-20020808/.
[5] Web Services Description Language (WSDL) 1.1. Available from http://www.w3.org/TR/wsdl.
[6] R. Amir and A. Zeid. A UML profile for service oriented architectures. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 192–193, New York, NY, USA, 2004. ACM Press.
[7] H. S. Bhatt, V. H. Patel, and A. K. Aggarwal. Web enabled client-server model for development environment of distributed image processing. In *Proceedings of the First IEEE/ACM InternationalWorkshop on Grid Computing (GRID'00)*, pages 135–145, London, UK, 2000. Springer-Verlag.
[8] S.-W. Cheng and D. Garlan. Mapping Architectural Concepts to UML-RT. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, June 2001.
[9] M. Clavel, F. Dur´an, S. Eker, P. Lincoln, N. Mart´ı-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
[10] DeveloperWorks. UML 1.4 Profile for Software Services with a Mapping to BPEL 1.0, July 2004.
[11] DeveloperWorks. UML 2.0 Profile for Software Services, April 2005.
[12] Z. Dong, Y. Fu, and X. He. Deriving Hierarchical Predicate/Transition Nets from Statechart Diagrams. In *Proceedings of The 15th International Conference on Software Engineering and Knowledge Engineering (SEKE2005)*, 2003.
[13] A. Egyed. Automating Architectural View Integration in UML. Technical Report USCCSE-99511, Center for Software Engineering, University of Southern California,Los Angeles, CA, 1999.
[14] A. Egyed and N. Medvidovic. Extending Architectural Representation in UML with View Integration. In *Proceedings of the 2nd International Conference on the Unified Modeling Language*, pages 2–16, October 1999.
[15] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.
[16] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer-Verlag, 1990.
[17] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic High-Level Nets: Petri Nets Revisited. In *Proceedings of Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types Joint with the 4th COMPASS Workshop*, volume 785 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 1994.
[18] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, volume 00, page 152, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
[19] X. Fu, T. Bultan, and J. Su. Formal verification of e-services and workflows. In *Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web (CAiSE'02/WES'02)*, pages 188–202, London, UK, 2002. Springer-Verlag.
[20] Y. Fu, Z. Dong, and X. He. An Approach to Web Services Oriented Modeling and Validation. In *Proceedings of the 28th ICSE workshop on Service Oriented*

Software Engineering (SOSE2006), 2006.

[21] Y. Fu, Z. Dong, and X. He. A method for realizing software architecture design. In *Proceedings of the Sixth International Conference on Quality Software(QSIC'06)*, pages 57–64, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Y. Fu, Z. Dong, and X. He. Modeling, Validating and Automating Composition of Web Services. In *Proceedings of The Sixth International Conference on Web Engineering (ICWE 2006)*, 2006.

[23] T. Gardner. UML Modeling of Automated Business Processes with a Mapping to BPEL4WS. In *ECOOP Workshop on OO and Web Services*, July 2003.

[24] D. Garlan, S.-W. Cheng, and A. J. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computer Programming*, 44(1):23–49, July 2002.

[25] H. J. Genrich. Predicate/Transition Nets. *Lecture Notes in Computer Science*, 254, 1987.

[26] X. He. A formal definition of hierarchical predicate transition nets. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 212–229, London, UK, 1996. Springer-Verlag.

[27] X. He and Y. Deng. A Framework for Specifying and Verifying Software Architecture Specifications in SAM. volume 45 of *The Computer Journal*, pages 111–128, 2002.

[28] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng. Formally analyzing software architectural specifications using sam. *Journal of Systems and Software*, 71(1-2):11–29, 2004.

[29] R. Heckel, M. Lohmann, and S. Th¨one. Towards a UML Profile for Service-Oriented Architectures. In *Workshop on Model Driven Architecture: Foundations and Applications*, 2003.

[30] R. Heckel, H. Voigt, J. K¨uster, and S. Th¨one. Towards Consistency of Web Service Architectures. Available from http://www.upb.de/cs/agengels/Papers/2003/HeckelVoigtKuesterThoene-CI03.pdf.

[31] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 2000.

[32] C. Hofmeister, R. L. Nord, and D. Soni. Describing Software Architecture with UML. In *Proceedings of the TC2 1st Working IFIP Conference on Software Architecture (WICSA1)*, pages 145 – 160, 1999.

[33] R. Hull, M. Benedikt, V. Christophides, and J. Su. Eservices: A look behind the curtain. In *Proceedings of the International Symposium on Principles of Database Systems (PODS)*. ACM Press, June 2003.

[34] E. Marcos, V. de Castro, and B. Vela. Representing web services with UML: A case study. *International Conference on Service Oriented Computing*, 2003.

[35] N. Medvidovic, A. Egyed, and D. S. Rosenblum. Round-Trip Software Engineering Using UML:From Architecture to Design and Back. In *Proceedings of the 2nd Workshop on Object-Oriented Reengineering*, pages 1–8, September 1999.

[36] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[37] T. Murata. Petri Nets: Properties, Analysis and Applications. *In Proceedings of the IEEE*, 77(4):541–580, 1989.

[38] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition

of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM Press.

[39] J. S. Paul Clements, Len Bass. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, January 2003.

[40] I. Prithvi Rao, Certified IT Architect. Using uml service components to represent the SOA architecture pattern. Available from http://www.ibm.com/developerworks/architecture/library/arlogsoa/.

[41] J. Saldhana, S. M. Shatz, and Z. Hu. Formalization of Object Behavior and Interactions From UML Models. *International Journal of Software Engineering and Knowledge Engineering*, pages 643–673, 2001.

[42] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[43] M. Vok´ac and J. M. Glattetre. Using a domainspecific language and custom tools to model a multitier service-oriented application—experiences and challenges. In C. W. Lionel Briand, editor, *Models 2005, Montego Bay, Jamaica October 2-7*, LNCS 3713, pages 492–506, Heidelberg, 2005. Springer-Verlag GmbH.

[44] J. Wang, X. He, and Y. Deng. Introducing Software Architecture Specification and Analysis in SAM through an Example. *Information and Software Technology*, 41(7):451–467, 1999. Appendix

A Algebraic specifications for *Packetizer* and *PacketPipe*

Packetizer(Packet, RawData) = Bool + Packet + RawData

sorts: packetizer

opns: _.PacketNotFull(): packetizer → bool
_.PacketFull(): packetizer → bool
_.AddRawData( ): packetizer × rawdata → packetizer
_.GetPacket(): packetizer → packetizer × packet

eqns: c ∈ packetizer
c.PacketNotFull() = ¬ c.PacketFull()

PacketPipe(Client) := Bool + Client

sorts: packetpipe

opns: ReadByAll(): packetpipe → bool
_.PacketRead(): packetpipe → bool
_.PacketNotRead(): packetpipe → bool
_.IsClientReady( ): packetpipe × client → bool
_.NoClientReady(): packetpipe → bool
_.AddClient( ): packetpipe × client → packetpipe
_.RemoveClient( ): packetpipe × client → packetpipe
_.AppendPacket( ): packetpipe × packet → packetpipe
_.SetClientReady( ):packetpipe × client → packetpipe
_.GetPacket(): packetpipe → packetpipe × packet
_.UpdatePacket(): packetpipe → packetpipe

eqns: pp ∈ packetpipe, c ∈ client, p ∈ packet
pp.IsClientReady(c) = true ⇒ pp.NoClientReady()=false

(pp.SetClientReady(c)).IsClientReady(c) = true
(pp.AddClient(c)).IsClientReady(c) = false

B Temporal constraints obtained from UML sequence diagrams

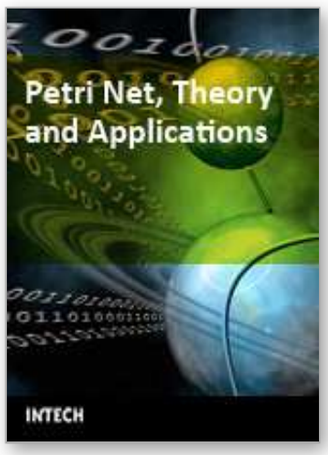
Temporal formulas for protocol *RawData*:
$\forall sid, rid,$
$\Box$ (*dataReady*(<*sid, rid, md*>) $\Rightarrow \Diamond$*requestData*(<*sid, rid, mr*>))
$\Box$(*dataReady*(< *sid, rid, md* >) $\Rightarrow \neg((\bigcirc$*dataReady*(< *sid, rid,md* >)) $\vee$ *requestData*(< *rid, sid, mr* >) $\vee$ *rawData*(< *sid, rid, mrd* >))U*requestData*(<*sid, rid, mr*>))
$\Box$ (*requestData*(<*sid, rid, mr*>) $\Rightarrow \Diamond$*rawData*(<*sid, rid,mrd*>))
$\Box$ (*requestData*(<*sid, rid, mr* >) $\Rightarrow \Diamond$(*dataReady*(< *rid, sid, md* >) $\vee$ ($\bigcirc$*requestData*(<*sid, rid, mr* >)) $\vee$
rawData(< *rid, sid, mrd* >))U*rawData*(<*rid, sid, mrd*>))
$\Box$ (*rawData*(< *sid, rid, mrd* >) $\Rightarrow \Diamond$(*dataReady*(< *sid, rid, md* >) $\vee$ *requestData*(< *rid, sid, mr* >) $\vee$ ($\bigcirc$*rawData*(< *sid, rid, mrd* >)))U*dataReady*(<*sid, rid, md*>))

Temporal formulas for protocol *DataPacket*:

$\forall$(<*sid, rid, mp*>), $\Box$ (*packet*(<*sid, rid, mp*>) $\Rightarrow$ *true*)

Temporal formulas for protocol *RequestDataPacket*:

$\forall sid, rid,$
$\Box$ (*subscribe*(<*sid, rid, ms*>) $\Rightarrow \Diamond$*requestPacket*(<*sid, rid, mr*>))
$\Box$ (*subscribe*(<*sid, rid, ms* >) $\Rightarrow \neg((\bigcirc$*subscribe*(<*sid, rid, ms* >)) $\times$ *requestPacket*(<*sid, rid, mr*>) $\times$ *packet*(<*rid, sid, mp*>) $\times$
desubscribe(<*sid, rid, md*>))U*requestPacket*(<*sid, rid, mr*>))
$\Box$ (*requestPacket*(<*sid, rid, mr*>) $\Rightarrow \Diamond$*packet*(<*rid, sid, mp*>))
$\Box$ (*requestPacket*(< *sid, rid, mr* >) $\Rightarrow \neg$(*subscribe*(<*sid, rid, ms* >) $\times$ ($\bigcirc$*requestPacket*(<*sid, rid, mr* >)) $\times$ *packet*(<*rid, sid, mp* >) $\times$
desubscribe(<*sid, rid, md*>))U*packet*(<*rid, sid, mp*>))
$\Box$ (*packet*(< *sid, rid, mp* >) $\Rightarrow \Diamond$(*desubscribe*(< *rid, sid,md* >) $\times$ *requestPacket*(<*rid, sid, mr*>)))
$\Box$ (*packet*(<*sid, rid, mp* >) $\Rightarrow \neg$(*subscribe*(<*rid, sid, ms* >) $\times$ *requestPacket*(<*rid, sid, mr*>) $\times$ ($\bigcirc$*packet*(<*sid, rid, mp*>)) $\times$
desubscribe(< *rid, sid, md* >))U (*desubscribe*(<*rid, sid, md* >) $\times$ *requestPacket*(<*rid, sid, mr*>)))
$\Box$ (*desubscribe*(<*sid, rid, mp* >) $\Rightarrow \neg$(*subscribe*(<*sid, rid, ms*>) $\times$ *requestPacket*(<*sid, rid, mr* >) $\times$ *packet*(<*rid, sid, mp* >) $\times$
($\bigcirc$*desubscribe*(<*sid, rid, md*>)))U*subscribe*(<*sid, rid, md*>))



Petri Net, Theory and Applications

Edited by Vedran Kordic

ISBN 978-3-902613-12-7

Hard cover, 534 pages

Publisher I-Tech Education and Publishing

Published online 01, February, 2008

Published in print edition February, 2008

Although many other models of concurrent and distributed systems have been de- veloped since the introduction in 1964 Petri nets are still an essential model for concurrent systems with respect to both the theory and the applications. The main attraction of Petri nets is the way in which the basic aspects of concurrent systems are captured both conceptually and mathematically. The intuitively appealing graphical notation makes Petri nets the model of choice in many applications. The natural way in which Petri nets allow one to formally capture many of the basic notions and issues of concurrent systems has contributed greatly to the development of a rich theory of concurrent systems based on Petri nets. This book brings together reputable researchers from all over the world in order to provide a comprehensive coverage of advanced and modern topics not yet reflected by other books. The book consists of 23 chapters written by 53 authors from 12 different countries.

InTech Europe

University Campus STeP Ri

Slavka Krautzeka 83/A

51000 Rijeka, Croatia

Phone: +385 (51) 770 447

Fax: +385 (51) 686 166

www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai

No.65, Yan An Road (West), Shanghai, 200040, China

中国上海市延安西路65号上海国际贵都大饭店办公楼405单元

Phone: +86-21-62489820

Fax: +86-21-62489821

IntechOpen