

# A Runtime Monitoring and Validation Framework for Web Service Interactions

Zheng Li, Yan Jin, Jun Han

*Faculty of ICT, Swinburne University of Technology  
PO Box 218, Hawthorn, Melbourne, Victoria 3122, Australia  
{zli, yjin, jhan}@ict.swin.edu.au*

## Abstract

*Web services are designed for composition and use by third parties through dynamic discovery. As such, the issue of interoperability between services is of great importance to ensure that the services can work together towards the overall application goals. In particular, the interaction protocols of a service need to be implemented and used properly so that the service composition can conduct itself in an orderly fashion. In our previous work, we have proposed a lightweight, pattern/constraint-based approach to specifying service interaction protocols. It has been incorporated into OWL-S for service developers to describe service interaction constraints. In this paper, we present a framework for monitoring the run-time interaction behaviour of Web services and validating the behaviour against their pre-defined interaction constraints. The framework involves interception of service interactions/messages, representation of interaction constraints using finite state automata, and conformance checking of service interactions against interaction constraints. As such, the framework provides a useful tool for validating the implementation and use of services regarding their interaction behaviour.*

## 1. Introduction

The emergence of Web services brings dramatic changes in IT system architectures and application paradigms. It is heavily used in the areas of system integration where Web service technology serves as a bridge between applications on heterogeneous platforms. Enterprises also use Web services to wrap its IT system functions and deploy them on the web, so that clients outside the enterprise system boundary can utilize those functions regardless technical implementation details. The interface description of a

Web service is the first thing a user should obtain and analyse before it invokes the service. Service interface description can be regarded as the contract of interaction with its consumers. In general, such a contract should cover issues more than interface signatures. However, the current Web service description standard – WSDL only specifies the location and operation signature of a service. The research community has widely recognized the need of a rich description framework for Web service to provide semantic information about a service, especially its interaction protocols [5].

In [18], we have proposed an approach to specifying interaction protocols/rules of Web service using interaction constraints (ICs). The approach is rule-based and declarative in nature. It is able to address the limitations of the procedural/imperative programming approach employed by OWL-S when characterizing services with diversified behaviours. One of many advantages of a declarative approach is that it requires much simpler description, needing only one-third to one-sixth of the statements required by the procedural approach when representing the same behaviour [10]. In addition, describing a service in an declarative manner enables a consumer to use the service in ways that the service designer may not foresee [11]. It also gives better support to automatic reasoning-based validation of the composition of multiple services with diversified behaviours [16].

Typically, interaction constraints are specified by service designers and are not expected to be violated. However, whether the actual runtime interaction with a service conforms to its constraints is another question. A run-time service monitoring and validation utility is needed to provide crucial information on service execution traces for identifying reasons when such a violation occurs. In this paper, we present a framework and associated techniques that support automatic validation of run-time service behaviours against its

specified interaction constraints. The framework employs finite state automata (FSA) as semantic representation of interaction constraints, uses message monitor to track messages going in and out of services, utilizes distinctive mechanisms to manage dynamically created FSAs and perform automatic conformance checking of service interaction against ICs. In [18], we focus on the approach to constraint specification and only give a brief introduction on the monitoring and validation framework due to limited space. In this paper, we give a thorough account of the framework architecture, the functionalities and coordination of its major components, and the use of FSAs for run-time validation. We also provide a more scalable solution to IC representation by separating pattern FSAs from scope FSAs.

The rest of the paper is organized as follows. Section 2 presents an overview of our approach to specifying interaction constraints for Web services. Section 3 explores the monitoring and validation framework in detail. The implementation is described in section 4. We then discuss related work in section 5 before concluding the paper in section 6.

## 2. An overview of Web service interaction constraint specification

To motivate, let us consider an auctioneer service that provides auction services on the web. The auctioneer publishes its interface in WSDL and communicates with a number of bidders and sellers by exchanging SOAP messages. The service is able to accept registrations from new bidders/sellers and hold auctions among registered bidders. It provides several operations to allow users to register and un-register themselves to the service, login and logout the service, as well as bid or sell an item. The service also provides an operation for bidders to retract their previous bids. Figure 1 shows an excerpt of the WSDL description of the auctioneer service.

```
<wsdl:definitions
  targetNamespace="http://localhost:8080/axis/services/Auctioneer"
  .....
  <wsdl:portType name="Auctioneer">
    <wsdl:operation name="opRegister" parameterOrder="userInfo">
      <wsdl:input message="impl:opRegisterRequest"
        name="opRegisterRequest"/>
      <wsdl:output message="impl:opRegisterResponse"
        name="opRegisterResponse"/>
    </wsdl:operation>
    <wsdl:operation name="opUnRegister" parameterOrder="userInfo">
      <wsdl:input message="impl:opUnRegisterRequest"
        name="opUnRegisterRequest"/>
      <wsdl:output message="impl:opUnRegisterResponse"
        name="opUnRegisterResponse"/>
    </wsdl:operation>
```

```
<wsdl:operation name="opLogin" parameterOrder="userInfo">
  <wsdl:input message="impl:opLoginRequest" name="opLoginRequest"/>
  <wsdl:output message="impl:opLoginResponse"
    name="opLoginResponse"/>
</wsdl:operation>
<wsdl:operation name="opLogout" parameterOrder="userInfo">
  <wsdl:input message="impl:opLogoutRequest"
    name="opLogoutRequest"/>
  <wsdl:output message="impl:opLogoutResponse"
    name="opLogoutResponse"/>
</wsdl:operation>
<wsdl:operation name="opBid" parameterOrder="userInfo itemNo price">
  <wsdl:input message="impl:opBidRequest" name="opBidRequest"/>
  <wsdl:output message="impl:opBidResponse" name="opBidResponse"/>
</wsdl:operation>
<wsdl:operation name="opRetract" parameterOrder="userInfo bidRefNo">
  <wsdl:input message="impl:opRetractRequest"
    name="opRetractRequest"/>
  <wsdl:output message="impl:opRetractResponse"
    name="opRetractResponse"/>
</wsdl:operation>
<wsdl:operation name="opSell" parameterOrder="userInfo itemInfo">
  <wsdl:input message="impl:opSellRequest" name="opSellRequest"/>
  <wsdl:output message="impl:opSellResponse" name="opSellResponse"/>
</wsdl:operation>
</wsdl:portType>
.....
</wsdl:definitions>
```

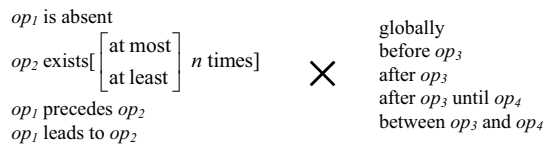
Figure 1: An Excerpt of WSDL 1.1 file for Auctioneer Web Service

The WSDL file above only contains operation signatures of the auctioneer service. Semantic information to enable service behavioural analysis is missing. What is needed is a framework for rich description for Web services, especially interaction protocols, to ensure proper use and interaction with services.

To address this issue, OWL-S has been proposed as a framework to describe the semantics of Web services. The framework is influential and well accepted in the research community. Its Process Model employs a procedural/imperative programming approach, and defines the service interaction protocol as a process [1]. It provides a set of control constructs, such as *sequence*, *split* and *split+join* etc., to specify possible process flows of a service's operations to reach a particular result. Although this procedural approach is suitable to certain situations, it has some limitations in characterizing services with diversified behaviours. For example, instead of offering one fixed process, a service may allow users to customize their own transaction process based on their own needs. In OWL-S, each transaction has to be defined by a composite process, thus the resulting process model

will become too complex as part of the interface description to be easily understood and properly used.

To address such limitations of OWL-S Process Model, we proposed in [18] a rule-based/declarative approach which is better-suited to represent services with diversified behaviours. The approach defines the interaction contract/protocol for a service as a set of interaction constraints. It advocates the use of intuitive pattern and scope operators from the property Specification Pattern System (SPS) proposed by Dwyer et al. in [7-9] for constraint specification. More specifically, it provides operators for specifying the restrictions on both the occurrence of individual operations' invocations and the order (or sequencing) between different operation invocations. The supported pattern and scope operators and their usage are listed in Figure 2, where  $op_1$ ,  $op_2$ ,  $op_3$  and  $op_4$  are distinct operations and  $n$  is a natural.



**Figure 2: Pattern and Scope Operators**

The patterns to restrict the occurrences of service invocations include *absence*, *existence*, and *bounded existence*. In particular, the *absence* pattern requires that invocations to the given operation not to occur (within the given scope, explained later). The *existence* pattern states that invocations to the given operation must appear. The *bounded existence* pattern extends it with lower and upper bounds on the number of invocations. For example, to control the overall system performance, the auction service provider may want to set a limit on the number of bids that bidders can make. This can be stated as “*opBid exists at most 3 times*”, where the upper bound is 3.

The patterns to state sequencing rules of service invocations include *precedence*, *response*, *precedence chain*, and *response chain*. For instance, a precedence property of the auctioneer service “*opRegister precedes opUnRegister*” states that there must be at least one *opRegister* invocation before any *opUnRegister* invocation. One may think *opRegister* enables *opUnRegister*. A response property “*opLogin leads to opLogout*” states that an *opLogin* invocation must eventually be followed by an *opLogout* invocation. In essence, this specifies a cause-effect relationship between *opLogin* and *opLogout*.

To handle more complex interaction constraints, we provide scope operators to state the portions of a

service's interaction history, in which the given pattern must hold. The scopes include *global*, *before*, *after*, *between-and*, and *after-until*. The *global* scope refers to the entire history. The *before* scope refers to the initial portion of the history up to the first occurrence of a call message of the given operation. The *after* scope however states the inverse, *i.e.* the portion after the first occurrence of a reply message of the given operation. In the *between-and* scope, each portion is marked between consecutive occurrences of two messages. The starting message is the reply message of the first given operation, while the ending message is the call message of the second given operation. The *after-until* scope is similar but allows the portion to be open ended. That is, the given pattern continues to take effect after a reply to the first operation, even if the second operation will never be invoked afterwards. In contrast, in the *between-and* scope, the second operation has to be invoked in order for the given pattern to be applicable.

To consider the effect of different parameter values on the service interaction rules, we associate conditions with each constraint specification. For instance, the earlier IC on the upper bound of bids can be fine-tuned to hold for each bidder and each of its login session as:

(1) *opBid exists at most 3 times after opLogin until opLogout*  
**where**  $opBid.userInfo = opLogin.userInfo = opLogout.userInfo$ ;

The following is another example constraint with conditions for auctioneer service:

(2) *opbid precedes opRetract after opLogin until opLogout*  
**where**  $opBid.userInfo = opRetract.userInfo = opLogin.userInfo = opLogout.userInfo$  **and**  $opBid.bidRefNo = opRetract.bidRefNo$ ;

Aiming at providing a common terminology and a standard way to specify ICs for Web services, we defined an Interaction Property Pattern (IPP) ontology for these pattern and scope operators. IPP ontology is defined using OWL and is designed as an add-on to OWL-S Process Model and as an alternative to the *CompositProcess* class. Figure 3 presents all the IPP classes and their relationships. More detailed introduction of IPP ontology and an example IC definition using the ontology can be found in [18].

### 3. Runtime monitoring and validation

The approach presented in the previous section enables explicit specification of Web Service interaction constraints. It helps service designers and service consumers to implement and interact with the

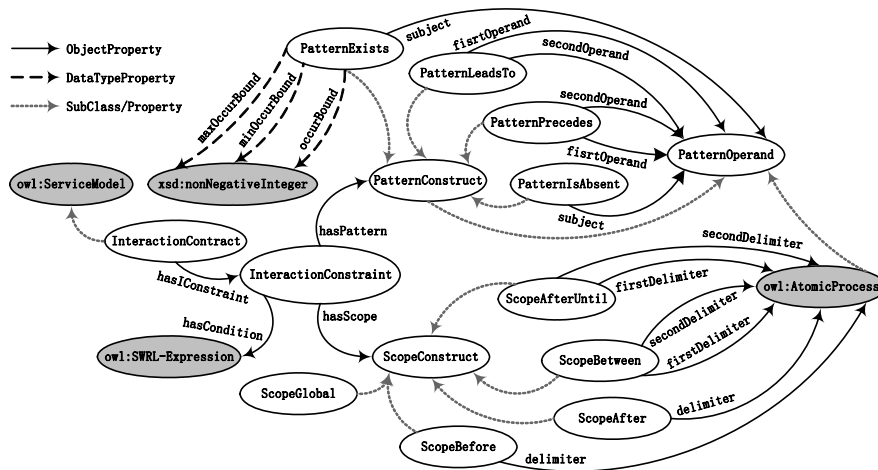


Figure 3: Classes in IPP ontology

service in a proper and predictable way. However, to ensure the service is used correctly and behaves at runtime, we need a tool that can capture the service runtime interaction and check its conformance with the pre-defined interaction constraints. In this section, we introduce a framework and detail the associated techniques for such runtime monitoring and validation.

### 3.1 Framework Architecture

The monitoring and validation framework intercepts and analyses messages exchanged between a Web service and its clients, and validates the message sequence against the IC specifications. This process is fully automated at run-time. Figure 4 illustrates the overall architecture of the framework.

The framework consists of three major functional parts: message monitoring, interaction constraint specification management, and constraint validation. These three parts interact with each other to provide run-time monitoring and validation of ICs for Web Services.

The message monitoring is conducted by Message Monitor (MM). It observes request and response SOAP messages going in and out of Web Services to intercept run-time operation invocations. Upon obtaining a SOAP message, MM will analyse the message body to extract key elements related to the operation invocation, and generate events for IC validation. MM also exploits the message ID generated by Web service execution platform (e.g. the Axis engine) to correlate a response message to its causing request message.

The interaction constraints specified for the Web service being monitored are managed by Constraint Specification Manager (CSM). It includes a parser that extracts a service's IC specifications written in the IPP ontology from the OWL-S file and translates them into an internal format (Cf. section 3.2). CSM manages all parsed ICs and, upon request, is able to return all ICs that concern a given interaction event.

The IC validation is carried out by the collaboration of a Service Level Validation Manager (SLVM) and a number of Constraint Level Validation Managers (CLVMs). SLVM manages and coordinates the validation of all ICs of one service. It works on the service level and communicates with other components in the framework. For instance, it uses the Pattern & Scope FSA Library (PSFL) for FSAs that semantically represent pattern and scope operators, and queries CSM for ICs of concern. SLVM also manages the lifecycle of CLVMs. A CLVM is in charge of the validation of a single constraint. It manages the validation process on the constraint level. Since a service may have multiple specified ICs, there may be a number of CLVMs for one SLVM.

In the subsequent sections, we shall give more details about CSM, pattern and scope FSAs, as well as the validation process. Since MM is platform-dependent, we shall describe it in section 4.

### 3.2 Constraint Specification Management

In developing CSM, we intend to keep it easy to accommodate various input IC specification languages apart from XML. We thus define a structured format to internally represent the syntax of ICs so that the

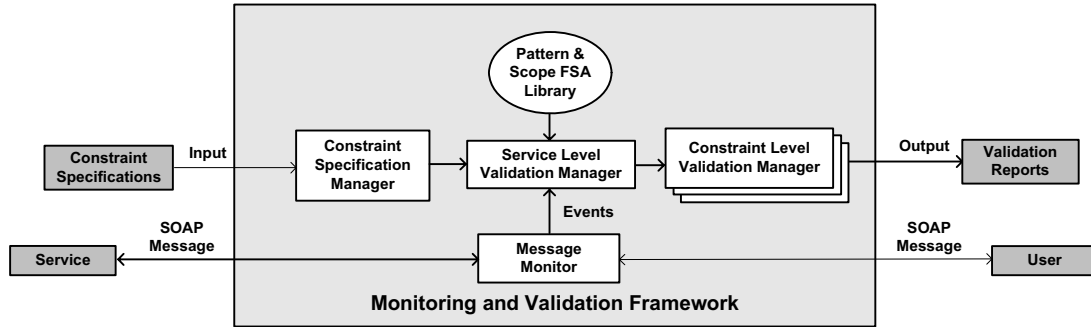


Figure 4. Architecture of Monitoring and Validation Framework

management and validation can be independent of the IC specification language. The internal representation identifies the following five key elements of an interaction constraint:

- Pattern type
- Pattern related messages
- Scope type
- Scope related messages
- Constraint conditions

As an example, Figure 5 depicts the internal syntactic representation for the constraint (2) given in section 2.

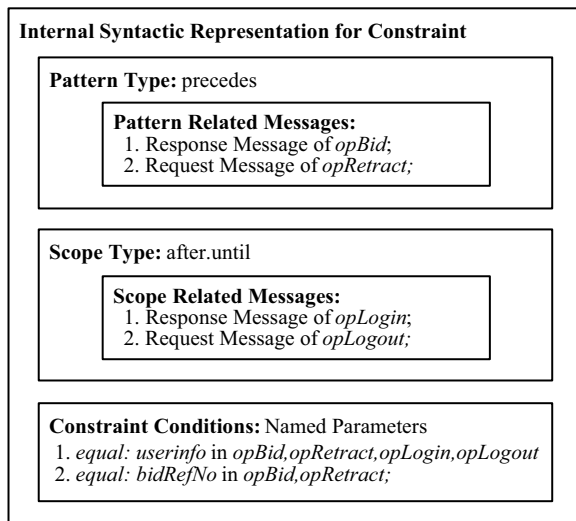


Figure 5: Internal Syntactic Representation for an Example Constraint

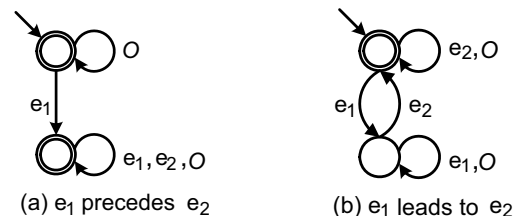
It is worth noting that, for constraint conditions, currently we only consider equality relationship among named parameters. However, the syntactic representation can be extended to incorporate other types of relationships.

### 3.3 Pattern FSAs and Pattern Validation

In section 2, we described the semantics of pattern and scope operators informally. To enable automated tool support for validation, we need to formally define their semantics. In our approach, we choose FSAs as the underlying representation. Instead of using one single FSA to define a combination of pattern and scope, we define FSAs for pattern operators and scope operators separately to make the approach more scalable as the number of pattern and scope operators increases. In this section, we introduce the FSAs for pattern operators and outline their validation method. In the next section, we further describe the FSAs for scope operators, their validation, and their coordination with pattern validation.

**FSAs for Pattern Operators.** In general, each pattern operator has a corresponding FSA representation where arc labels are events or event sets, and circles denote states. Figure 6 lists the FSA representations for the pattern operators shown in section 2.

In these figures, a double-circle denotes a final state of the FSA, where the FSA can be safely terminated or destroyed. A single-circle denotes the otherwise. Further,  $e_1$  and  $e_2$  denote two distinctive events. In figure 6(d-f), the value of natural number  $n$  is equal or greater 1 ( $n \geq 1$ ). And  $O$  is the set of events not explicitly mentioned.



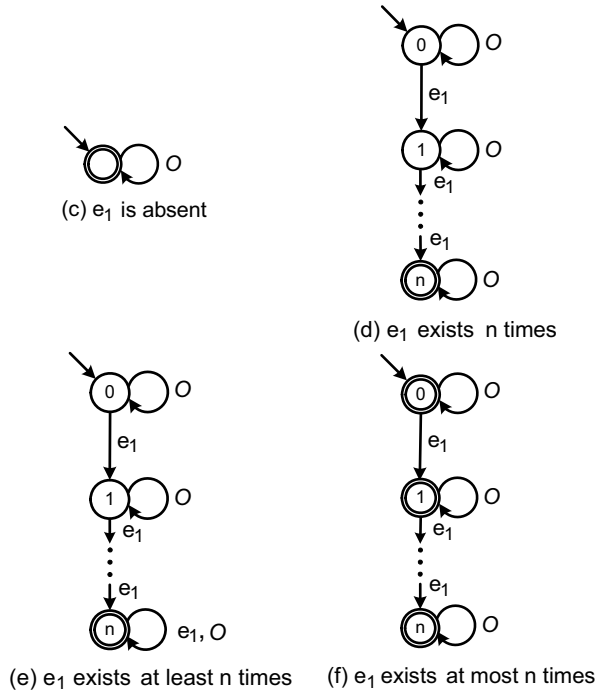


Figure 6: FSA Representation for Patterns

**Validation of Pattern FSAs.** In the validation process, a FSA runs in parallel with the monitored services. It transits from one state to another via an arc by taking an event in a labelling set of the arc. A finite sequence of events is considered valid if taking the events sequentially leads the FSA to a final state. A violation will occur if the FSA receives an event that is unspecified at the current state. For example, in Figure 6(a),  $e_2$  is not allowed to occur at the start state since the pattern requires that at least one  $e_1$  event must occur before any occurrence of  $e_2$ . A violation may also occur if one attempts to terminate a FSA when it is at a single-circled state. The termination of a pattern FSA may be caused by an event indicating the end of its validation scope (explained below) or the termination of the Web Service. For instance, In Figure 6(b), after receiving  $e_1$  in the start state, FSA transits to the bottom single-circle state where termination is not acceptable. It means, before terminating the FSA, an  $e_2$  event is expected so that it can move to the upper final state. In the pattern validation, a violation to the FSA will cause the pattern validator to issue a “pv” message to the managing scope validator (detailed later).

### 3.4 Scope FSAs and Scope Validation

Scope operators are used to specify portions of the interaction history of a service, where the pattern constraint must hold [8]. Since a pattern FSA is always

associated with a scope, in this work, we develop FSAs for scope operators to manage the lifecycle of pattern FSAs. Validation of a pattern FSA is enabled when the stated portions are entered. Correspondingly, when the portions are exited, validation of the pattern FSA is disabled, resulting in the pattern FSA to be destroyed.

**FSAs for Scope Operators.** Besides the events that delimit the scope, scope FSAs also concern a special type of events: violations to its associated pattern FSAs. Only these two types of events will cause a scope FSA to transit states. It is worth noting that a violation to the pattern FSA does not necessarily mean a violation to an interaction constraint. The state of the scope FSA must also be taken into consideration when judging a violation on the constraint level.

The FSAs for scope operators are shown in Figure 7, where  $P$  is the associated pattern and  $pv$  denotes a violation event to  $P$ . Letter  $s$  and  $t$  are two distinctive events that delimit the scope. The circles in shadow with a letter  $C$  denote the states where a constraint violation has been detected. Circles with different letters indicate states with different semantics, which are detailed below.

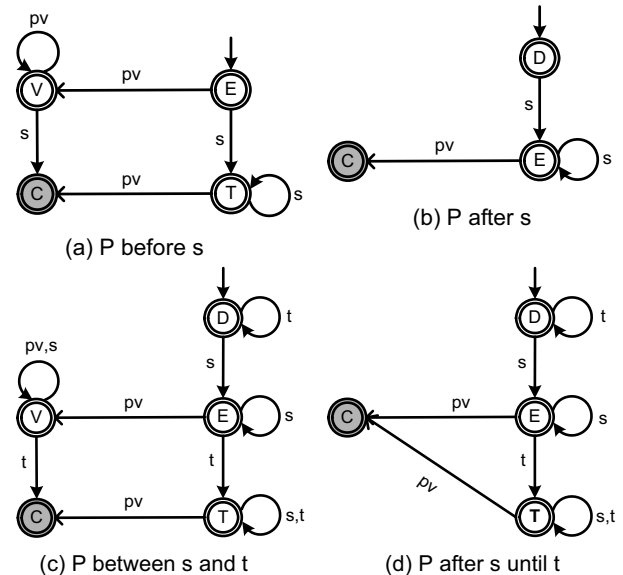


Figure 7: FSA Representation for Scopes

**State E:** The scope where pattern must hold is entered. The validation process of the pattern FSA is enabled. Messages related to the pattern FSA are used to advance its states for validation.

**State D:** Program execution is not in the scope where the pattern must hold. The validation of the pattern

FSA is disabled. Messages related to the pattern FSA will be discarded.

**State V:** A violation on the pattern FSA level is detected.

**State C:** A violation on the constraint level is detected.

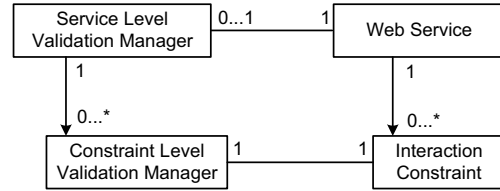
**State T:** A single round of validation of the interaction constraint has ended.

**Validation of Scope FSA.** At each state of a scope FSA, certain actions are carried out to validate the constraint. At state D, validation for the pattern FSA is disabled. It becomes enabled when state E is entered. At state T, a Terminate/Destroy message will be sent to the pattern FSA. If it causes a violation to the pattern FSA (i.e. a pattern violation message (pv) is observed), state C is entered, resulting in a constraint violation to be reported. If no pattern violation is observed, both the pattern FSA and the scope FSA will be terminated/destroyed.

In figure 7, (a) denotes the scope FSA for “*P before s*”. At its initial state, validation of the FSA for *P* is enabled. The scope validator will not report any violations of *P* until it receives the first *s* event. If event *s* is received before any violation of *P* is detected, the pattern FSA will be destroyed since the scope has been exited. Figure 7(b) is for scope “*P after s*”. The validation of *P* is disabled until the first event *s* occurs. After that, the validation is enabled and any violation to *P* will lead to a constraint-level violation. Figure 7(c) depicts “*between...and*” scope, which looks like the combination *after* and *before*. The validation of *P* will not be enabled until the first event *s* occurs and will be terminated when *t* occurs. Any violations of *P* will not be reported until the first *t* occurs. Figure 7(d) is for “*after...until*”. The validation of *P* is enabled after the first *s* occurs and any violation that happens afterwards will lead to a constraint level violation. An occurrence of event *t* will result in terminating the pattern FSA for *P*.

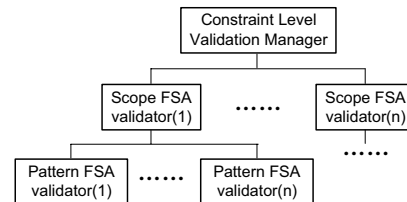
### 3.5 Validation Management

The validation of service run-time interaction is carried out by CLVMs. Each CLVM represents an interaction constraint corresponding to an IC specification and is instantiated by a value combination of its named parameters. CLVMs carry out the actual validation function, detecting violations and issuing violation reports. They are created, destroyed, and managed by the SLVM. Figure 8 depicts the relationship between SLVM, CLVMs, Web services, and interaction constraints.



**Figure 8: Relationship between Validation Managers and Web services with constraints**

To manage the validation process, the framework creates one SLVM instance for each monitored Web service and one or multiple CLVMs for its ICs. One CLVM instance administers one or multiple scope FSA validators identified by the values of named parameters. Each scope FSA validator may associate one or several pattern FSA validators, which are also identified by the values of their named parameters. This structure is shown hierarchically in Figure 9:



**Figure 9: Hierarchical Structure for Constraint Validation**

The scope and pattern FSA validators are identified by the values of the named parameters in their related events. Scope FSA validators and pattern FSA validators are correlated by their named parameters. Recall the example IC (2) in sections 2, one scope FSA validator is associated the pattern FSA validators whose named parameters *userinfo* have the same value. The validation process example given in next section shows how CLVM creates scope FSA validators and their associated pattern FSA validators.

### 3.6 Validation Process

In general, the validation process is carried out in the following steps:

1. In the initialization phrase, FSAs in PSFL are initialized. CSM reads constraint specifications in OWL-S files and translates them into the internal format. A SLVM is created for each monitored service. MM establishes connections to the server hosting the Web services and is ready to receive intercepted messages.

2. When a message is intercepted and observed, MM deposits an event into the queue of the SLVM of concern.
3. SLVM fetches one event at a time from its event queue. If the event interests no ICs, the event will be discarded. Otherwise SLVM will try to find corresponding CLVMs representing the IC(s) and notify them. Upon being notified, a CLVM will use the event to advance the state of its scope or pattern FSAs, depending on which one is related. A constraint level violation is detected if the event leads a scope FSA to state *C*.
4. If an event interests an IC, but CLVM found no corresponding scope and pattern FSA instance, CLVM will create new scope or pattern FSA instances from PSFL using the values of named parameters in the event. It then asks the FSA instance to advance its states using the event.
5. When the processing of an event is completed and no violation is detected, SLVM fetches the next event in the queue for further validation.

Please note that MM and SLVM run in parallel. Message interception process and validation process are asynchronous.

As an example, consider the auctioneer service given in section 2 and its example constraint (2), assuming the service operations are invoked in following sequence:

```
opRegister(user1)
opLogin(user1)
opRetract(user1,0001)
opBid(user1, item01, 100)
opLogout(user1)
```

and that the initialization process is completed, a SLVM instance is created for auctioneer service and is waiting for events.

**a.** When *opRegister(user1)* invocation occurs: MM observes SOAP request and response messages for operation *opRegister* with parameter value *user1*. Then it generates an event to the SLVM for auctioneer service. SLVM finds out from CSM that no IC concerns the event. Thus this event is discarded.

**b.** When *opLogin(user1)* invocation occurs: MM sends the corresponding event to SLVM. SLVM learnt from CSM that this event is related to the scope operator in example IC (2). However, the existing scope FSA list in the corresponding CLVM is empty. Then CLVM creates one “*after...until*” scope FSA validator instance (Cf. Figure 7(d)) using value *user1*. The scope FSA is moved from its initial state *D* to state *E*, where

the validation of pattern FSA is enabled. Please note, at this stage, the scope FSA does not contain a pattern FSA validator because its related events do not occur yet.

**c.** When *opRetract(user1,0001)* invocation occurs: CLVM regards this as a pattern FSA related message. Then it uses the values of two named parameters, *user1* and *0001*, as the ID to create one pattern FSA validator and associates it to the scope FSA validator identified by the value of parameter *userinfo (user1)*. Then the pattern FSA validator uses the message to advance its state. However this message is not acceptable at the current state (initial state) of the pattern FSA, so a violation of pattern occurs. A pattern FSA violation event (*pv*) is sent to the scope FSA validator, which causes state transition from State *E* to State *V*. Now, a constraint violation is detected and is reported to the user.

## 4. Implementation

We choose Java-based open source platforms and tools to implement the monitoring and validation framework. For the web server, we use Apache Tomcat 5 which is a high efficient servlet container with a lightweight HTTP support. To run Web services, we employ Apache Axis1.2, which is an implementation of W3C SOAP standard, as the SOAP engine to process SOAP messages. Combined these two, we establish a reliable and stable platform to run Java Web Services.

To monitor the behaviours of Web service, we need to intercept the SOAP request messages which are used to invoke operations of services along with their corresponding response messages. The Apache Axis1.2 toolkit contains a SOAP Monitor utility which is able to intercept the SOAP messages going in and out of the service at run-time without requiring any special configuration or modification on the Web service itself. As SOAP requests and responses are received, the messages go through a chain of handlers for processing and are delivered to the right target service. In the Axis1.2 tool kit, a new message handler has been written and added to the global message handler chain so that the messages can be intercepted. To allow client to retrieve the intercepted messages, Axis engine provides a SOAP Monitor Service (SMS) which is implemented as a servlet. When the Axis engine starts, SMS opens a server socket on port 5001 and waiting for connection requests from clients. Once the handler intercepts a message, it will forward the message to SMS and then SMS forwards the message

to all connected clients. In our framework, MM is the component we developed to connect to SMS in Axis to retrieve messages for validation.

In implementing other validation related components in the framework, we have reused and redesigned the architecture of a runtime validation tool developed in [13] for CORBA-based systems. However, these modules have been enhanced to better deal with the full range of interaction property patterns we developed for Web services in this paper. We have also modified the Constraint Specification Manager module for processing the XML-based specifications of service interaction constraints.

Our framework can be deployed and executed relatively independently from the monitored services and client processes. The monitoring is non-intrusive to the service execution and the validation is conducted asynchronously. It is not required to deploy the framework in the same server node as the services. The only inevitable overhead is message interception forwarding conducted by the service execution platform like the Axis engine. This nevertheless will only incur a marginal performance penalty to the execution of monitored services.

## 5. Related work and discussion

Currently major research effort has been put into the area of Web service composition. Orchestration and choreography based standards, such as BPEL4WS [3] and WS-CDL [14], are in the spotlight of many active researchers. However, they are composition languages in nature and specify service behaviours from the service composition or business process point of view [5]. What they specify is the required behaviours for services rather than the behaviours services can provide. The later issue is what this paper tries to address: specifying behaviours properties for individual services and validating their behaviours at run-time. Some research efforts also identify the needs to specify behaviours properties and propose FSM based or ontological approach to represent interaction protocols for a individual Web service. A more detailed discussion of their approaches is provided in [18]. [2] proposes a SOAP-centric contract description language for Web Services. The approach they employ to specify the messaging sequencing constraints is quite similar to the process model in OWL-S, which also suffers from the limitation as a procedural approach.

A body of work on Web service monitoring has been reported. [15] proposes an approach to specifying and monitoring Service Level Agreements. It focuses

on Quality of Service, and monitors properties such as performance and costs, rather than interaction behaviours. [17] develops a service request language to specify goals of service composition and provide a framework to monitor the execution of the composite process. Their approach focuses on service composition rather than a single service. A log-based architecture is described in [6] to track utilization of a Web service. Their work only concerns message interception and logging. [4, 19] aim at monitoring service compositions at run-time to see whether services satisfy the assertions or requirements from the service composition specified by BPEL. [4] uses predicate logic to express assertions and embed them into the BPEL process specification. [19] extracts assumptions and behavioural properties from a BPEL assumptions. The assertions in [4] and behavioural properties in [19] are the required properties from the service consumer, rather than services' properties. And both of them do not cater for behavioural properties for one single service. In contrast, our work not only offers a practitioner friendly approach to specifying ICs for one individual service, but also includes a framework that can monitor service behaviours and validate them against those ICs. In addition, our monitor attaches to the service itself rather than to a service consumer such as the BPEL process. [20] is targeting the same problem as we try to address: validate the occurrence sequence of SOAP message against the protocol specification. But their approach to constraint specification and validation is different from ours. They utilize XQuery to specify protocol while we use more intuitive pattern-based operators to define constraints. Furthermore, our constraints specifications are compatible with OWL-S framework and can be published as part of the Web Service interface description.

When putting our approach into practical use, the service designer needs to ensure the consistency of all the interaction constraints of a service. Inconsistency among constraints will leads to a situation where calls to an operation will always violate some rules. This issue is addressed in [12] where consistency checking is done by testing the non-emptiness of the language intersection of the interaction constraints and proving that each operation has its role in the intersection.

It is worth noting that some of the example constraints mentioned in this paper can be implemented internally using programming logic. However, the purpose of our work is to spell out those constraints explicitly in the interface description of Web Services so that the clients can have an accurate understanding of service's behaviour before actual

interactions happen and thus reduce the possibilities of exceptions in the service's run-time execution.

## 6. Conclusion

Precise specification and run-time validation of behavioural properties for Web service is critical to ensure the proper use of services and build a loose-coupled, service-based software system.

In this paper, we have presented the design and implementation of a framework to monitor and validate run-time behaviour of Web service against pre-defined interaction constraints. We gave a thorough treatment to the framework architecture, the functionalities and collaboration of its major components, and a detailed description of the overall validation process. Compared to [18], the approach we employ in this paper to representing the semantic of IC is more scalable as the number of scope and pattern operators increases. Instead of using one FSA to represent the combination of pattern and scope, we separate the scope FSAs from pattern FSAs.

Currently we are seeking solutions for incorporating the required operations of services into our framework, and plan to investigate the issue of static compatibility checking between constraints of a service and service composition specification. We also plan to investigate the issue of incorporating violation handling mechanisms in our validation framework.

## References

- [1] "OWL-S", <http://www.daml.org/services/owl-s/1.0/>. Last accessed on January 27<sup>th</sup>, 2006
- [2] "SSDL - the SOAP Service Description Language", <http://ssdl.org>. Last accessed on January 27<sup>th</sup>, 2006
- [3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business Process Execution Language for Web Services version 1.1", <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>, 2003. Last accessed on January 27<sup>th</sup>, 2006
- [4] L. Baresi, C. Ghezzi, and S. Guinea, "Smart Monitors for Composed Services", in proc. 2<sup>nd</sup> International Conference on Service-Oriented Computing (ICSOC), 2004.
- [5] B. Benatallah, F. Casati, H. Skogsrud, and F. Toumani, "Abstracting and Enforcing Web Service Protocols", *International Journal of Cooperative Information Systems*, vol. 13, pp. 413-440, 2004.
- [6] S. M. S. d. Cruz, M. L. M. Campos, P. F. Pires, and L. M. Campos, "Monitoring E-Business Web Services Usage through a Log Based Architecture", in proc. *IEEE International Conference on Web Services (ICWS)*, 2004.
- [7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-state Verification", in proc. *International Conference on Software Engineering (ICSE)*, 1999.
- [8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-State Verification", *Workshop on Formal Methods in Software Practice*, 1998.
- [9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Specification Patterns", <http://www.cis.ksu.edu/santos/spec-patterns>, Last accessed on January 27<sup>th</sup>, 2006.
- [10] E. Gottesdiener, "Procedural versus declarative", in *Application Development Trends*, 1997.
- [11] D. Guillaume and R. Plante, "Declarative Metadata Processing with XML and Java", *Journal of Astronomical Society of the Pacific*, 2001.
- [12] Y. Jin and J. Han, "Consistency and Interoperability Checking for Component Interaction Rules", in proc. 12<sup>th</sup> Asia-Pacific Software Engineering Conference (APSEC), 2005.
- [13] Y. Jin and J. Han, "Runtime Validation of Behavioural Contracts for Component Software", in proc. 5<sup>th</sup> International Conference On Quality Software (QSIC), 2005.
- [14] N. Kavantzaz, D. Burdett, and G. Ritzinger, "Web Services Choreography Description Language Version 1.0", <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>, 2004. Last accessed on January 27<sup>th</sup>, 2006
- [15] A. Keller and H. Ludwig, "Defining and Monitoring Service-Level Agreements for Dynamic e-Business", in proc. 16<sup>th</sup> Conference on Systems Administration (LISA), 2002.
- [16] R. Lara, H. Lausen, S. Arroyo, J. d. Bruijn, and D. Fensel, "Semantic web services: description requirements and current technologies", in proc. *International Workshop on Electronic Commerce, Agents, and Semantic Web Services*, In conjunction with the 5<sup>th</sup> International Conference on Electronic Commerce (ICEC), 2003.
- [17] A. Lazovik, M. Aiello, and M. Papazoglou, "Associating Assertions with Business Processes and Monitoring their Execution", in proc. 2<sup>nd</sup> International Conference on Service Oriented Computing (ICSOC), 2004.
- [18] Z. Li, J. Han, and Y. Jin, "Pattern-Based Specification and Validation of Web Services Interaction Properties", in proc. 3<sup>rd</sup> International Conference on Service Oriented Computing (ICSOC), Amsterdam, Netherlands, 2005.
- [19] K. Mahbub and G. Spanoudakis, "A Framework for Requirements Monitoring of Service Based Systems", presented at 2<sup>nd</sup> International Conference on Service Oriented Computing (ICSOC), 2004.
- [20] M. Venzke, "Specifications using XQuery Expressions on Traces", in proc. 1<sup>st</sup> International Workshop on Web Services and Formal Methods, Pisa, Italy, 2004.