An Introduction to the SOAP Service Description Language

Savas Parastatidis[1], Jim Webber[2], Simon Woodman[1], Dean Kuo[3], Paul Greenfield[3]
Savas@Parastatidis.name, Jim@Webber.name, S.J.Woodman@newcastle.ac.uk, Dean.Kuo@csiro.au, Paul.Greenfield@csiro.au

Abstract

The SOAP Service Description Language (SSDL) is a SOAP-centric contract definition language for Web Services. SSDL provides the base framework for a range of protocol description frameworks which at one end of the spectrum can be a simpler, SOAP-focussed, direct replacement for WSDL while at the other end of the spectrum a more expressive contract definition language that can enable formal validation and reasoning about the protocols that a Web Service supports. Four protocol description frameworks are provided with the base SSDL specification though third-parties are free to implement others to fit their needs.

SSDL documents

An Introduction to the SOAP Service Description Language [1] (this document)

SOAP Service Description Language (SSDL) v1.3 [2]

MEP SSDL Protocol Framework v1.3 [3]

CSP SSDL Protocol Framework v1.3 [4]

Rules SSDL Protocol Framework v1.3 [5]

Sequencing Constraints SSDL Protocol Framework v1.3 [6]

Status of this document

Version: 1.3

Date: April 2005

http://ssdl.org

Disclaimer

[1] School of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK
[2] ThoughtWorks Australia Pty. Ltd.
[3] CSIRO Information and Communication Technology (ICT) Centre, CSIRO, Australia

Table of Contents

1. Introduction

SOAP [7] is the standard message transfer protocol for Web Services. However, the default description language for Web Services (WSDL [8]) does not explicitly target SOAP but instead provides a generic framework for the description of network-exposed software artefacts. WSDL's transfer protocol independence makes describing SOAP message transfers more complex than if SOAP had been assumed from the outset. While the motivation to define a language that can be used with other underlying transfer and transport technologies is understandable, the cost in terms of complexity of the solution for SOAP-based Web Services is high. Furthermore, WSDL's focus on the interface abstraction for describing services makes it difficult for the community to move away from the object-oriented or remote procedure call mindset and focus on message-orientation as the means through which integration is achieved. Finally, although this is a known issue, it is difficult to use WSDL to describe infrastructure protocols (e.g. transactions, reliable messaging) that make use of SOAP headers.

The SOAP Service Description Language (SSDL) is an XML-based vocabulary for writing message-oriented contracts for Web Services. SSDL focuses on the use of messages and protocol frameworks to describe a SOAP-based Web Service and is intended to provide a natural fit with the SOAP processing model.[4]

This whitepaper discusses the relationships between Web Services, messages, and contracts. The goals of SSDL are presented and its structure is discussed and exemplified using the four protocol frameworks that comprise the base suite of SSDL specifications: MEP [3], CSP [4], Rules [5], and SC [6].

1.1. Key Points

- SSDL assumes SOAP as the means of transferring messages between Web Services over arbitrary transport (and transfer) protocols. As a result, there is no need to define bindings for all possible transport protocols;
- SSDL assumes WS-Addressing as the standard means for embedding addressing information within SOAP envelopes and for binding those addresses onto underlying transport protocols;
- SSDL focuses on messages and protocols. As a result, there is no need for articles like 'interface', 'inheritance', and 'operation';
- XML Infoset is assumed as the underlying SSDL component model. There is no need (nor desire) to create a new component model simply for contract description;
- Modularisation of contracts is handled using XInclude. A shortcut mechanism is provided which is defined in terms of XInclude elements to simplify componentisation as far as is possible;
- SSDL promotes protocol framework extensibility. It allows different protocol description models to be plugged into the base SSDL framework which helps promote protocol-based integration and exposure of the messaging behaviour of a Web Service. Tools, such as model checkers, can verify the correctness of protocols defined in an SSDL contract, or automate the reasoning about the compatibility of Web Services. Hosting environments can even use the SSDL contract to validate the message exchanges between Web Services.

1.2. Namespaces

These namespaces and their prefixes are used throughout this document.

[4] From now on we will assume that a "Web Service" supports SOAP.

Prefix	Namespace	Notes
ssdl	urn:ssdl:v1	Where elements are not qualified with a namespace prefix, urn:ssdl:v1 is assumed
mep	urn:ssdl:mep:v1	
rls	urn:ssdl:rules:v1	
csp	urn:ssdl:csp:v1	
sc	urn:ssdl:sc:v1	
xs	http://www.w3.org/2001/XMLSchema	
wsa	http://www.w3.org/2004/12/addressing	
xi	http://www.w3.org/2001/XInclude	
soap	http://www.w3.org/2003/05/soap-envelope	

2. Web Services

2.1. Service-orientation

While service-orientation is not a new architectural paradigm, the advent of Web Services has reinvigorated interest in the approach. It is a common misconception that Web Services are a form of software magic which automatically corrals the architect towards a loosely coupled solution which is scalable, robust, and dependable. Certainly it is possible to build service-oriented applications using Web Services protocols and toolkits but it is equally possible to build applications according to the principles of other architectural paradigms using the same set of technologies (e.g. object-orientation).

As researchers and developers have re-branded their work to be in vogue with the latest buzzwords, the term Service Oriented Architecture (SOA) has become overloaded. Due to lack of a widely-accepted definition of a service, we provide the following:[5]

> A service is the logical manifestation of some physical or logical resources (like databases, programs, devices, humans, etc.) and/or some application logic that is exposed to the network;

and

> Service interaction is facilitated by message exchanges.

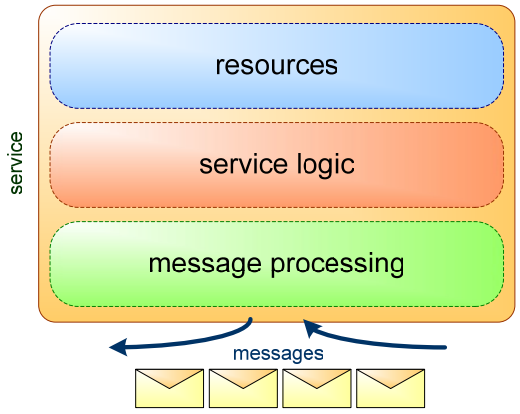

Figure 1: The typical structure of a service

A service such as that in Figure 1 consists of some resources (e.g. data, programs, or devices), service logic, and a message processing layer which deals with message exchanges. Messages

[5] Please note that although we are using the term “service”, any other term could have been used with this definition. In fact, the terms “business entity”, “business agent”, “application unit”, etc. have already been proposed as alternatives in an attempt to find the next differentiating marketing buzzword for product offerings. The fact remains, however, that we are talking about entities that can exchange messages.

arrive at the service and are acted on by the service logic, utilising the service's resources (if any) as required. Services may be of any scale: from a single operating system process to enterprise-wide business processes.

Services may be hosted on devices of arbitrary size (e.g. workstations, databases, printers, phones, personal digital assistants, etc.) providing different types of functionality to a network application. This promotes the concept of a connected world in which no single device and/or service is isolated. Interesting applications are built through the composition of services and the exchange of messages (Figure 2).

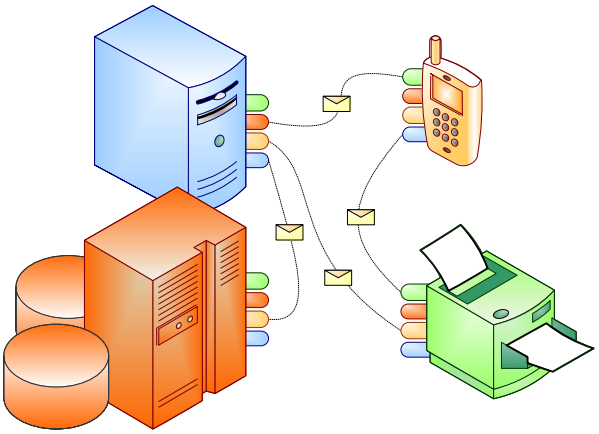

Figure 2: Networked applications are built through the exchange of messages between services hosted in devices. In this example, an application running on a mobile device makes use of the distributed resources through services running on a workstation (e.g. job execution), a database, and a printer.

2.2. Messages

A message is the unit of communication between services. Service-oriented systems do not expose abstractions like classes, objects, methods, remote procedures; the abstraction to which services bind is the message and communication is achieved through the transfer of such messages. A number of message transfers can be logically grouped to form message exchange patterns (MEPs) (e.g. an incoming and an outgoing message that are related can form a "request-response" MEP). Multi-message interactions can be grouped to form a protocol which is associated with some well-defined behaviour for the participating services.

2.3. Protocols, policies, and contracts

The messaging behaviour of a service in a distributed application is specified by a set of messages and the order in which they are sent and received (i.e. the supported protocols). This is a departure from the traditional object-oriented world where behavioural semantics are associated with types, exposed through methods, and coupled with particular endpoints.

Protocols and other metadata are usually described in contracts to which services have to adhere. A contract is a description of the policy (e.g. security requirements or encryption capabilities), quality of service characteristics (e.g. support for reliable messaging), and semantic description which a service supports and/or requires, in addition to the set of messages and MEPs which convey functional information to and from the service.

2.4. The Web Services stack

The composable Web Services specifications are layered as shown in Figure 3 to form the Web Services stack. Here we make the assumption that in order to achieve interoperability and ease integration, all Web Services must support SOAP. Services interact through the transfer of SOAP messages and no other communication means (e.g. RMI) are logically supported.

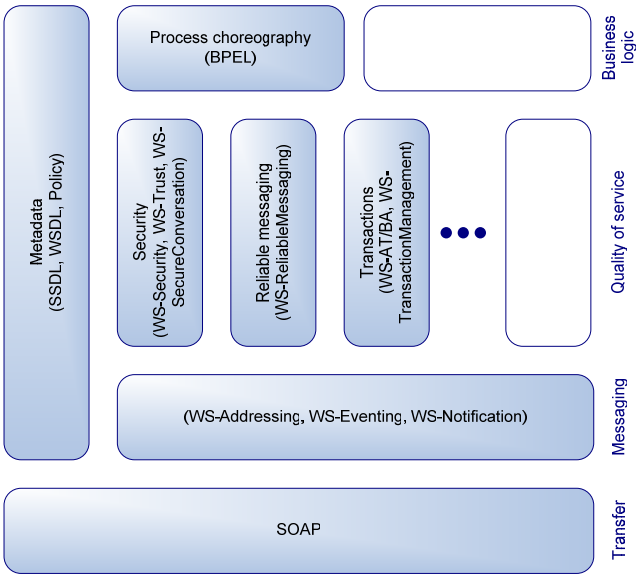

Figure 3: The Web Services stack (adapted from [9])

3. SSDL Contract

The primary goal of an SSDL contract is to provide the mechanisms for service architects to describe the structure of the SOAP messages a Web Service supports. Once the messages of a Web Service have been described, one of the currently available (or future) protocol frameworks can be used to combine the messages into protocols that expose the messaging behaviour of that Web Service. SSDL defines an extensible mechanism for various protocol frameworks to be used.

SSDL contracts communicate the supported messaging behaviour of a Web Service, in terms of messages and protocols, so that architects and developers can create systems that can meaningfully participate in conversations. SSDL contracts may be dynamically discovered from registries and its protocol descriptions compared against with an application's requirements in order to determine whether an interaction can sensibly take place.

An SSDL contract is defined in a namespace that uniquely identifies it and consists of four major sections, as shown in Figure 4.

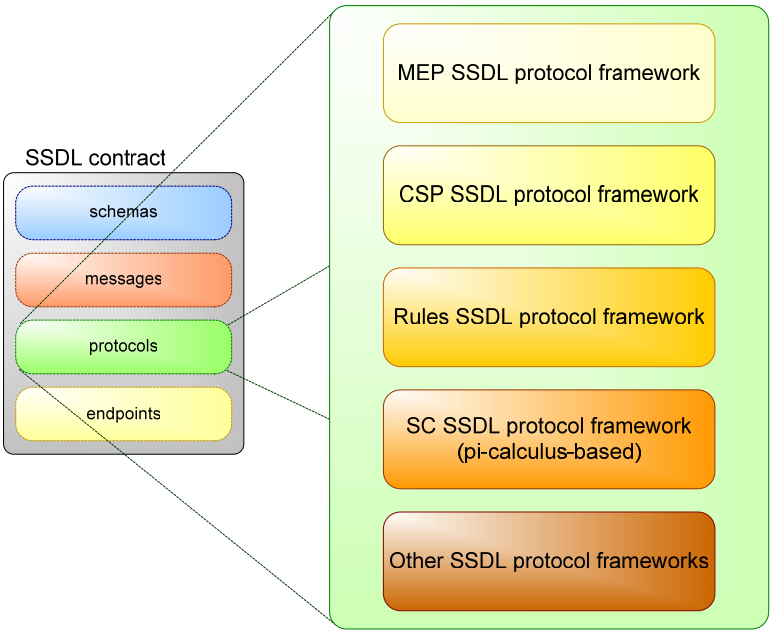

Figure 4: The structure of an SSDL contract

3.1. Schemas

The “schemas” section is used to define the structure of all the elements that will be used for the description of the SOAP messages. Any schema language may be used to define global schema elements, though XML Schema is the default choice.

3.2. Messages

The ‘messages’ section is where the messages that a Web Service supports are declared. There can be many groups of messages defined in different namespaces. However, irrespective of the namespace in which they are defined, the messages included in the SSDL document are all part of the contract. SOAP messages are described in terms of header and body elements and are named so that protocol frameworks can reference them.

```
<ssdl:messages targetNamespace="uri">
  <ssdl:message name="msg">
    <ssdl:header ref="elements:header1" mustUnderstand="true" />
    <ssdl:header ref="elements:header2"
                 role="urn:ssdl:example:role"/>
    <ssdl:body ref="elements:body1" />
    <ssdl:body ref="elements:body2" />
  </ssdl:message>

  <ssdl:fault name="fault">
    <ssdl:code role="http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver">
      <ssdl:value>Sender</ssdl:value>
    </ssdl:code>
  </ssdl:fault>
</ssdl:messages>
```

Listing 1: An example of a message and a fault message

In Listing 1 a message “msg” is defined to have two header elements (children of `soap:Header`) and two body (children of `soap:Body`) elements. Note that while the SOAP processing model permits it, the WS-I Basic Profile 1.0a [10] mandates a single element as a child of `soap:Body`. However, SSDL does not impose that restriction. Listing 1 also demonstrates how a SOAP fault message could be declared.

The `header` element provides the `mustUnderstand`, `role`, and `relay` attributes which correspond to the equivalent attributes defined by the SOAP processing model. This makes it possible and straightforward to describe Web Services infrastructure protocols.

3.3. Protocols

Once the messages in a contract have been declared, we can move on to describe how they may relate to each other. SSDL provides an extensible mechanism, based on the concept of protocol frameworks.

A protocol framework uses messages already declared in a contract to describe the simple message exchange patterns or multi-message interactions which are observed by other services. A protocol framework is an XML-based model for capturing relationships between message exchanges which may or may not be supported by an underlying formal model.

SSDL defines the semantics of the `msgref` element which must be used by protocol frameworks when referencing messages. The `msgref` element provides the mandatory `ref` and `direction` and the optional `action` attributes. The value of the `ref` attribute (an XML QName) points to a message, while the value of the `direction` attribute defines whether that particular message is incoming or outgoing. The value of the `action` attribute defines the URI that must be used for the WS-Addressing [11] `Action` addressing property of the SOAP message. A declared message can be referred multiple times in a protocol description and also be declared as having multiple directions or multiple actions as per Listing 2. Since the current draft version of the WS-Addressing specification defines that the `wsa:Action` header information element is mandatory,

when the `action` attribute is omitted an `ssdl:msgref` element `urn:ssdl:ProcessMessage` is assumed.

```
<ssdl:msgref ref="msgs:Msg" direction="in" action="urn:service:actions:MsgRequest" />
<ssdl:msgref ref="msgs:Msg" direction="out" action="urn:service:actions:MsgRequestResponse" />
```

Listing 2: Examples of `ssdl:msgref` elements

It may be possible that the same protocol is defined in multiple ways using the same or different protocol frameworks. It is up to the designers to choose which protocol framework is best for their needs. Also, it may be possible to translate the description of a service's messaging behaviour from one protocol framework to another without losing any semantics depending on the source and target frameworks.

Some protocol frameworks may be associated with the semantics of a formal model (e.g. CSP, Rules, SC). As a result, it is possible to use model checkers, such as SPIN [12] and FDR [13], to verify the safety (e.g. absence of deadlocks and agreed termination) and liveness (e.g. eventual termination guarantee) properties of the defined protocols. Different protocol frameworks may be suitable for meeting different requirements when describing protocols. It is up to the Web Service architect to choose the one that is most suitable for the protocol(s) under development.

The initial release of SSDL comes with four protocol frameworks: MEP, CSP, Rules, and SC. It is not in the scope of this whitepaper to compare the CSP, Rules and SC protocol frameworks (for an initial discussion the reader should read [14]).

3.3.1. MEP

The MEP (Message Exchange Pattern) is defined to be a superset of the MEPs defined by the WSDL specification [15]. The MEP specification defines the semantics and structure of XML elements representing several message exchange patterns.

```
<ssdl:protocols>
  <ssdl:protocol targetNamespace="urn:service:mep">
    <!-- A request-response -->
    <mep:in-out>
      <msgref ref="msgs:Msg1" direction="in" />
      <msgref ref="msgs:Msg2" direction="out" />
    </mep:in-out>

    <!-- A request-response with the possible fault messages -->
    <mep:in-out>
      <ssdl:msgref ref="msgs:Msg3" direction="in"
                   action="urn:service:actions:Msg3Request" />
      <ssdl:msgref ref="msgs:Msg3" direction="out"
                   action="urn:service:actions:Msg3Response" />
      <ssdl:msgref ref="msgs:Fault1" direction="out" />
      <ssdl:msgref ref="msgs:Fault2" direction="out" />
    </mep:in-out>
  </ssdl:protocol>
</ssdl:protocols>
```

Listing 3: Two examples of message exchange patterns defined using two of the MEP protocol framework elements

The approach taken by the MEP SSDL protocol framework for defining message exchange patterns allows tooling to easily validate the structure of the description. Furthermore, the semantics of some of the MEP elements are defined in terms of WS-Addressing, hence making it easier for tooling to produce interoperable code stubs. For example, the semantics of the `<in-out>` element require that the value of the "Message ID" WS-Addressing header of the incoming message becomes the value of the "Relates To" header for the outgoing message.

The patterns currently supported by the MEP are: in-only, robust-in-only, in-out, in-optional-out, in-out-with-faults, out-only, robust-out-only, out-in, and out-optional-in.

3.3.2. CSP

The CSP SSDL protocol framework is based on the Communicating Sequential Processes [16] semantics. A protocol is defined in terms of one or more sequential processes which may

communicate with each other. Messages which are sent or received represent the events in CSP processes.

```
<ssdl:protocols>
   <ssdl:protocol targetNamespace=" urn:service:csp:1"
                  xmlns:msgs="http://example.org/service/messages"
                  xmlns:csp="urn:ssdl:csp:v1">
      <csp:process>
         <csp:sequence>
            <ssdl:msgref ref="msgs:Msg1" direction="in"/>
            <csp:d-choice>
               <csp:sequence>
                  <ssdl:msgref ref="msgs:Msg2" direction="out" />
                  <ssdl:msgref ref="msgs:Msg3" direction="in" />
               </csp:sequence>
               <ssdl:msgref ref="msgs:Fault1" direction="out" />
            </csp:d-choice>
         </csp:sequence>
      </csp:process>
   </ssdl:protocol>
</ssdl:protocols>
```

Listing 4: An example of a very simple messaging behaviour defined using the CSP SSDL protocol framework

In Listing 4 a very simple behaviour for a Web Service is described in terms of a series of message exchanges. The behaviour suggests that after the incoming `Msg1` message, either `Msg2` or `Fault1` will be sent. If `Msg2` is sent, then `Msg3` will be expected. Any deviation from this series of messages breaks the contract. The key difference between MEP and CSP protocol framework is MEP can only specify very simple relationship between *in* and *out* messages while CSP can specify complex interrelationships between messages.

A race condition can occur if a Web Service interacting with the Web Service which adheres to the contract of Listing 5 sends `Msg3` before it receives `Msg2`. Also, a deadlock can occur if the interacting service never sends `Msg4`. It is the responsibility of the architect and application developer to ensure that the protocols are free from race conditions and deadlocks. While such a race condition may not be fatal for a Web Service. However, tools such as model checkers can detect if a protocol can deadlock or a race condition can occur, if that is important.

```
<ssdl:protocols>
   <ssdl:protocol targetNamespace="urn:service:csp:2"
      <csp:sequence>
         <ssdl:msgref ref="msgs:Msg1" direction="in" />
         <csp:d-choice>
            <ssdl:msgref ref="msgs:Msg2" direction="out" />
            <ssdl:msgref ref="msgs:Msg3" direction="in" />
         </csp:d-choice>
         <ssdl:msgref ref="msgs:Msg4" direction="in" />
      </csp:sequence>
   </ssdl:protocol>
</ssdl:protocols>
```

Listing 5: A race and a deadlock

Note that SSDL says nothing about the scope of an interaction. A Web Service may support one or more instantiations of this protocol at the same time. If more instantiations are supported, a contextualisation mechanism is necessary for messages to be associated with a particular instantiation of the protocol (e.g. WS-Context [17], WS-Security [18], WS-Addressing [11] Reference Parameters, service-specific information, etc.).

3.3.3. Rules

The Rules SSDL protocol framework uses preconditions on ‘send’ and ‘receive’ events as the means to describe messaging behaviour. As with the CSP SSDL Protocol Framework, it is possible to use model checkers to verify that a protocol is free from deadlock and race conditions.

```
<ssdl:protocols>
   <ssdl:protocol targetNamespace="urn:service:rls:1"
      <rls:rules>
         <rls:rule>                                                    <!-- (1) -->
            <ssdl:msgref ref="msgs:Msg1" direction="in" />
```

```
            <rls:condition />
         </rls:rule>

         <rls:rule>                                                          <!-- (2) -->
            <ssdl:msgref ref="msgs:Msg2" direction="out" />
            <ssdl:msgref ref="msgs:Msg3" direction="in" />
            <rls:condition>
               <ssdl:msgref ref="msgs:Msg1" direction="in" />
            </rules:condition>
         </rls:rule>

         <rls:rule>                                                          <!-- (3) -->
            <ssdl:msgref ref="msgs:Msg4" direction="out" rls:final="true"/>
            <rls:condition>
               <ssdl:or>
                  <ssdl:msgref ref="msgs:Msg2" direction="out" />
                  <ssdl:msgref ref="msgs:Msg3" direction="in" />
               </rls:and>
            </rls:condition>
         </rls:rule>
      </rls:rules>
   </ssdl:protocol>
</ssdl:protocols>
```

Listing 6: The protocol of Listing 5 described using the Rules SSDL protocol framework

Listing 6 is described using the Rules-based SSDL protocol framework. `Msg1` can be received at any time (Rule 1). `Msg2` can be sent or `Msg3` can be received after `Msg1` has been received (Rule 2). `Msg4` can be received after `Msg2` has been sent or `Msg3` has been received (3). `Msg4` is also flagged as the final message in the protocol. The SSDL Rules protocol framework permits the use of the 'and', 'or', 'xor' and 'not' logical operators. As with the description of the protocol in Listing 5 the description of Listing 6 suffers from the same race condition.

Note that SSDL says nothing about the scope of an interaction. A Web Service may support one or more instantiations of this protocol at the same time. If more instantiations are supported, a contextualisation mechanism is necessary for messages to be associated with a particular instantiation of the protocol (e.g. WS-Context [17], WS-Security [18], WS-Addressing [11] Reference Parameters, service-specific information, etc.).

3.3.4. SC (Sequencing Constraints)

The SC SSDL protocol framework is used to describe multi-service interactions and its semantics are based on the pi-calculus [6]. SC uses the declared messages in an SSDL contract to define the allowed interactions between other services with specific participant roles in the multi-service interaction [19]. Protocols are structured in terms of pi-calculus processes which execute in parallel and may communicate with each other. The SC framework can be used to define protocols consisting of just the service being described and its consumer, or multi-party protocols where more than two services are involved.

Defining multi-party interactions is achieved by defining multiple participants and annotating the `ssdl:msgref` element with the participant that is interacting with the service. It is assumed that one participant is the service itself and another participant is implicitly bound by sending the initial message which starts the protocol. Other participants may be bound during the protocol by declaring the `participant-binding-name` attribute, or may be bound using some other out-of-band method. The SC makes no assumptions about the number of instantiations of a protocol that a Web Service supports. If more that one instantiation is supported, a contextualisation mechanism is necessary for messages to be associated with a particular instantiation of the protocol (e.g. WS-Context [17], WS-Security [18], WS-Addressing [11] Reference Parameters, service-specific information, etc.).

```
<ssdl:protocols>
  <ssdl:protocol targetNamespace="urn:service:sc">
    <sc:sc>
      <sc:participant name="serviceX"/>
      <sc:participant name="serviceY"/>
      <sc:protocol name="example">
        <sc:sequence>
```

```
          <ssdl:msgref ref="msgs:msg1" direction="in"
                       sc:participant="serviceX" participant-binding-name="serviceY"
                       participant-binding-content="types:epr"/>
          <ssdl:msgref ref="msgs:msg2" direction="out" sc:participant="serviceX"/>
          <sc:choice>
            <ssdl:msgref ref="msgs:msg3" direction="out" sc:participant="serviceY"/>
            <ssdl:msgref ref="msgs:msg4" direction="out" sc:participant="serviceY"/>
          </sc:choice>
          <ssdl:msgref ref="msgs:msg5" direction="in" sc:participant="serviceY"/>
        </sc:sequence>
      </sc:protocol>
    </sc:sc>
  </ssdl:protocol>
</ssdl:protocols>
```

Listing 7: A simple multi-party protocol specified using the SC protocol framework

Listing 7 shows a simple, multi-party protocol which defines two participants, `serviceX` and `serviceY`. The protocol is defined as a sequence of messages, beginning with the receipt of `msg1` from the `serviceX` paricipant. Part of the contents of this message bind the `serviceY` participant. Note that the protocol does not specify which part of the message binds the participant, only that it is this message that contains the binding. The protocol then proceeds by `msg2` being sent to `serviceX`. Following that message there is a non-deterministic choice of either `msg3` or `msg4` being sent to `serviceY` (as bound by the message received from `serviceX`), and finally `msg5` is received from `serviceY`. In addition to `choice` and `sequence` structures the SC protocol framework also defines `multiple` and `parallel` constructs.

It is possible to automatically translate protocols defined in SC to a pi-calculus form or other formal languages such as Promella. Analysis of such definitions allows us to reason about the composability and/or compatibility characteristics of Web Services and also employ model checkers to prove properties like liveness and consistency [19].

Note that SSDL says nothing about the scope of an interaction. A Web Service may support one or more instantiations of this protocol at the same time. If more instantiations are supported, a contextualisation mechanism is necessary for messages to be associated with a particular instantiation of the protocol (e.g. WS-Context [17], WS-Security [18], WS-Addressing [11] Reference Parameters, service-specific information, etc.).

3.4. Endpoints

An SSDL contract may also define endpoints, as WS-Addressing Endpoint References (EPRs), of Web Services that are known to support the defined contract. While the schemas, messages, and protocols of a contract (the contract is identified by its namespace) remain constant, the endpoints may change. Also, additional endpoints not defined in the contract may also exist.

```
<ssdl:endpoints>
  <ssdl:endpoint>
    <wsa:Address>http://www.example.org/service</wsa:Address>
  </ssdl:endpoint>

  <ssdl:endpoint>
    <wsa:Address>urn:service:1</wsa:Address>
    <wsa:ReferenceParameters>
      <example:element>10</example:element>
    </wsa:ReferenceParamters>
  </ssdl:endpoint>
</ssdl:endpoints>
```

Listing 8: An example of two endpoints in an SSDL contract

3.5. Modularisation

The structure of an SSDL contract has been defined in such a way as to support modularisation without the need to define new semantics for document composition. Instead, modularisation is supported through the inclusion of XML documentation as defined by the XInclude specification

[20]. It is recommended that XInclude is used when a contract is composed from smaller XML documents. The resulting document has to be a valid SSDL document as shown in Listing 9.

```
<ssdl:contract targetNamespace="urn:service:1:contract">
  <xi:include href="http://www.example.org/service/schemas.ssdl " />

  <ssdl:messages targetNamespace="urn:service:messages:group1" />
    <xi:include href="http://www.example.org/service/messages.ssdl" />
  </ssdl:messages>

  <xs:include href="http://www.example.org/service/contract2.ssdl"
    xpointer="
      xmlns(ssdl='urn:ssdl:v1')
      xpointer(/ssdl:contract/ssdl:messages[targetNamespace='urn:service:messages:group2')" />

  <xi:include href="http://www.example.org/service/protocols.ssdl" />
  <xs:include href="http://www.example.org/service/endpoints.ssdl" />
</ssdl:contract>
```

Listing 9: Using XInclude to make an SSDL contract document

In addition to the assumption that XInclude elements are used for all the modularisation needs of a contract author, SSDL also provides the `include` element as a shortcut. The `include` element does not introduce new semantics into XML document composition. Instead, it is defined in terms of specific XInclude elements. If an `include` element is found in an SSDL contract, the SSDL processor is responsible for expanding it into its corresponding XInclude elements before validating the document.

```
<ssdl:contract targetNamespace="urn:service:1:contract">
  <ssdl:include location="http://www.example.org/service/contract.ssdl">
  <!-- rest of SSDL contract -->
</ssdl:contract>
```

Listing 10: An example of `ssdl:include`

4. Message-orientation

The SSDL contract allows Web Service designers to focus on the structure of messages and the message exchange patterns. SSDL enables the contract-first approach to designing, building, and deploying Web Services. In SSDL, there are no abstractions like interfaces, inheritance, operations, etc. Architects define protocols by correlating messages together.

The move away from WSDL's interface and operation abstractions and the requirement for a unique Global Element Definition (GED) brings up the issue of how SOAP processors are to distinguish messages that have the same structure. SSDL does not provide a solution for this since it delegates such decisions to the Web Service architect through the implicit support of a number of alternatives:

- The message may be distinguished from other messages depending on the current state of a protocol execution. For example, messages Msg1 and Msg2 may have the exact same structure (and even contents) but the protocol may define that Msg2 can only be received after Msg1 has been received in the same context of execution. The Web Service can very easily distinguish between the two.
- A Web Service implementation may use content-based rules to distinguish between incoming messages. There may not be a need for the SOAP processor to distinguish messages from one another. Instead, the service logic of a Web Service may wish to receive an event signifying the arrival of any message and, hence, there is no requirement for the "operation" abstraction. Furthermore, the Web Service logic may infer the semantics of the messages based on its contents (e.g. the order identifier found inside a business document that is included within the message) or the overall protocol-based interaction. It may be possible that SOAP processors provide mechanisms for the declarative description of how messages are to be distinguished from one another, if that is required, and then appropriate events fired.

- If messages with the same structure need to be distinguished between one another, the `ssdl:action` attribute can be used which results in the WS-Addressing `wsa:Action` addressing header to be set.

The messaging behaviour of a service in a distributed application is captured through the set of protocols which it supports. The notion of protocol is a departure from the traditional object-oriented world where behavioural semantics are associated with types, exposed through methods, and coupled with particular endpoints. Instead a protocol describes the externally visible messaging behaviour of a service only in terms of the messages, message exchange patterns, and ordering of those MEPs which are supported by the service.

5. Examples

Here we present a set of examples and their contracts using different protocol frameworks.

5.1. StockQuote

The StockQuote example uses a simple request-response message exchange pattern. A graphical illustration of the two messages involved is presented in Figure 5 while the listing for the contract using the MEP SSDP protocol framework is given in Listing 11.

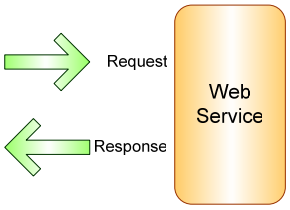

Figure 5: A graphical representation of a request-response MEP

```
<?xml version="1.0" encoding="UTF-8" ?>
<contract xmlns="urn:ssdl:v1"
          targetNamespace="urn:ssdl:example:StockQuote:contract">
   <documentation>
     This is an example of an SSDL contract for a StockQuote Web Service
   </documentation>

   <schemas xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:schema targetNamespace="urn:ssdl:example:StockQuote:contract:schema">
         <xs:element name="company" type="xs:string" />
         <xs:element name="value" type="xs:positiveInteger" />
      </xs:schema>
   </schemas>

   <messages targetNamespace="urn:ssdl:example:StockQuote:contract:messages"
             xmlns:s="urn:ssdl:example:StockQuote:contract:schema">
      <message name="StockQuoteRequestMsg">
         <body ref="s:company" />
      </message>

      <message name="StockQuoteResponseMsg"
               xmlns:s="urn:ssdl:example:StockQuote:contract:schema">
         <body ref="s:value"/>
      </message>
   </messages>

   <protocols>
      <protocol targetNamespace="urn:ssdl:example:StockQuote:contract:protocol"
                xmlns:mep="urn:ssdl:mep:v1"
                xmlns:msgs="urn:ssdl:example:StockQuote:contract:messages">
         <mep:in-out>
            <msgref ref="msgs:StockQuoteRequestMsg" direction="in" />
            <msgref ref="msgs:StockQuoteResponseMsg" direction="out" />
         </mep:in-out>
      </protocol>
   </protocols>

   <endpoints xmlns:wsa="http://www.w3.org/2004/12/addressing">
      <endpoint>
```

```
         <wsa:Address>http://www.example.org/StockQuoteWebService</wsa:Address>
      </endpoint>
      <endpoint>
         <wsa:Address>smtp:StockQuoteWebService@example.org</wsa:Address>
      </endpoint>
   </endpoints>
</contract>
```

Listing 11: The SSDL contract for the StockQuote example

5.2. WS-Streaming

The WS-Streaming example demonstrates how SSDL could be used to describe a Web Services infrastructure protocol. Given SSDL's focus on SOAP and the explicit support for describing header and body elements, protocol description becomes easy. Figure 6 illustrates the protocol as a state machine with the nodes representing the states and the arcs the events. The protocol assumes that a Web Service will send a stream of SOAP messages. The stream is initiated by an event which is not defined by this protocol. Any number of messages may be sent as part of the stream and the stream may be cancelled at any time.

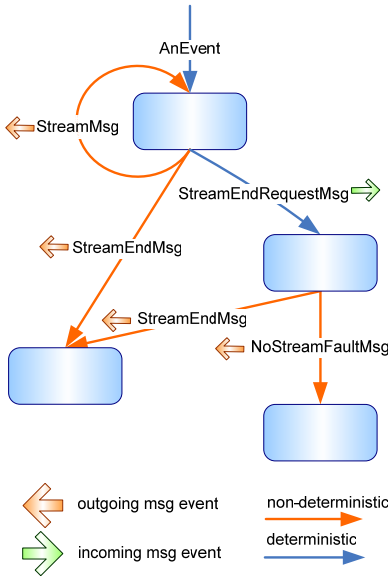

Figure 6: Graphical representation of the WS-Streaming protocol

We use the CSP SSDP protocol framework to describe the WS-Streaming protocol (Listing 12).

```
<?xml version="1.0" encoding="utf-8" ?>
<contract xmlns="urn:ssdl:v1"
          targetNamespace="urn:ssdl:example:ws-streaming:contract">
   <documentation>
     This is an example of an SSDL contract for a WS-Streaming protocol
     using the CSP SSDL Protocol Framework
   </documentation>

   <schemas xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:schema targetNamespace="urn:ssdl:example:ws-streaming:contract:schema">
         <xs:element name="StreamContext">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="StreamId" type="xs:anyURI" />
                  <xs:element name="Sequence" type="xs:positiveInteger" />
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="StreamEndRequest">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="StreamId" type="xs:anyURI" />
                  <xs:element name="Time" type="xs:duration" />
               </xs:sequence>
            </xs:complexType>
         </xs:element>
```

```
         <xs:element name="StreamEnd">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="StreamId" type="xs:anyURI" />
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </schemas>

   <messages targetNamespace="urn:ssdl:example:ws-streaming:contract:messages"
             xmlns:elements="urn:ssdl:example:ws-streaming:contract:schema">
      <message name="StreamMsg">
         <header ref="elements:StreamContext"
                 mustUnderstand="true"
                 role="urn:ssdl:example:ws-streaming:soap-role" />
      </message>

      <message name="StreamEndRequestMsg">
         <header ref="elements:StreamEndRequest"
                 mustUnderstand="true"
                 role="urn:ssdl:example:ws-streaming:soap-role"/>
      </message>

      <message name="StreamEndMsg">
         <header ref="elements:StreamEnd"
                 mustUnderstand="true"
                 role="urn:ssdl:example:ws-streaming:soap-role" />
      </message>

      <fault name="NoStreamFaultMsg">
         <code value="Sender" />
         <reason xml:lang="en">
            <text>No such stream!</text>
         </reason>
         <role>urn:ssdl:example:ws-streaming:soap-role</role>
      </fault>
   </messages>

   <protocols>
      <protocol targetNamespace="urn:ssdl:example:ws-streaming:contract:protocol:csp"
                xmlns:prtcl="urn:ssdl:example:ws-streaming:contract:protocol:csp"
                xmlns:csp="urn:ssdl:csp:v1"
                xmlns:msgs="urn:ssdl:example:ws-streaming:contract:messages">
         <csp:process>
            <csp:non-d-choice>
               <msgref ref="msgs:StreamEndMsg" direction="out" />
               <csp:d-choice>
                  <csp:sub-process-ref ref="prtcl:subprocess" />
                  <msgref ref="msgs:StreamMsg" direction="out" />
               </csp:d-choice>
            </csp:non-d-choice>
         </csp:process>

         <csp:sub-process name="subprocess">
            <csp:sequence>
               <msgref ref="msgs:StreamEndRequestMsg" direction="in" />
               <csp:non-d-choice>
                  <msgref ref="msgs:StreamEndMsg" direction="out" />
                  <msgref ref="msgs:NoStreamFaultMsg" direction="out" />
               </csp:non-d-choice>
            </csp:sequence>
         </csp:sub-process>

      </protocol>
   </protocols>

</contract>
```

Listing 12: The CSP-based SSDL contract for the WS-Streaming protocol

5.3. Rules

The WS-Streaming example of Figure 6 can also be described using the Rules SSDL Protocol Framework without the need to change any of the message definitions (Listing 13).

```
<contract xmlns="urn:ssdl:v1"
          targetNamespace="urn:ssdl:example:ws-streaming:contract">
   <documentation>
     This is an example of an SSDL contract for a WS-Streaming protocol
     using the Rules SSDL Protocol Framework
   </documentation>

   <!-- Messages are the same as in Listing 12 -->

   <protocols>
      <!-- The CSP-based protocol description of Listing 12 can also appear here -->

      <protocol targetNamespace="urn:ssdl:example:ws-streaming:contract:protocol:rules"
                xmlns:rls="urn:ssdl:rules:v1"
                xmlns:msgs="urn:ssdl:example:ws-streaming:contract:messages">
         <rls:rules>
            <rls:rule>
               <msgref ref="StreamMsg" direction="out" />
               <rls:condition>
                  <rls:not>
                     <msgref ref="StreamEndMsg" direction="out"/>
                     <msgref ref="StreamEndRequestMsg" direction="in" />
                  </rls:not>
               </rls:condition>
            </rls:rule>

            <rls:rule>
               <msgref ref="StreamEndMsg" direction="out" rls:final="true" />
               <rls:condition>
                  <rls:not>
                     <ssdl:msgref ref="NoStreamFaultMsg" direction="out" />
                  </rls:not>
               </rls:condition>
            </rls:rule>

            <rls:rule>
               <msgref ref="StreamEndRequestMsg" direction="in" />
               <rls:condition>
                  <rls:not>
                     <msgref ref="StreamEndMsg" direction="out" />
                  </rls:not>
               </rls:condition>
            </rls:rule>

            <rls:rule>
               <msgref ref="NoStreamFaultMsg" direction="out" rls:final="true" />
               <rls:condition>
                  <msgref ref="StreamEndRequestMsg" direction="in" />
               </rls:condition>
            </rls:rule>
         </rls:rules>
      </protocol>
   </protocols>
</contract>
```

Listing 13: The Rules-based SSDL contract for the WS-Streaming protocol

For a more complex example of the Rules SSDL Protocol Framework refer to [5].

5.4. SC

The WS-Streaming example of Figure 6 can also be described using the SC SSDL Protocol Framework without the need to change any of the message definitions (Listing 13).

```
<contract xmlns="urn:ssdl:v1"
          targetNamespace="urn:ssdl:example:ws-streaming:contract">
   <documentation>
     This is an example of an SSDL contract for a WS-Streaming protocol
     using the SC SSDL Protocol Framework
   </documentation>

   <!-- Everything is the same as in Listing 12 -->

   <protocols>
      <!-- The CSP-based protocol description of Listing 12 and
           the Rules-based protocol description of Listing 13 can also appear here -->
```

```
      <protocol targetNamespace="urn:ssdl:example:sc:ws-streaming"
xmlns:msgs="http://example.org/service/messages" xmlns:sc="urn:ssdl:protocols:sc:v1">

         <sc:sc>
            <sc:participant name="stream-receiver"/>

            <sc:protocol name="send-stream">
               <sc:choice>
                  <msgref ref="StreamMsg" direction="out"
                          sc:participant="stream-receiver"/>
                  <msgref ref="StreamEndMsg" direction="out"
                          sc:participant="stream-receiver"/>
                  <sc:sequence>
                     <msgref ref="StreamEndRequestMsg" direction="in"
                             sc:participant="stream-receiver"/>
                     <sc:choice>
                        <msgref ref="StreamEndMsg" direction="out"
                                sc:participant="stream-receiver"/>
                        <msgref ref="NoStreamFaultMsg" direction="out"
                                sc:participant="stream-receiver"/>
                     </sc:choice>
                  </sc:sequence>
               </sc:choice>
               <sc:protocolref ref="send-stream"/>
            </sc:protocol>
         </sc:sc>
      </protocol>
   </protocols>
</protocol>
```

Listing 14: The Rules-based SSDL contract for the WS-Streaming protocol

For a more complex example of the Rules SSDL Protocol Framework refer to [6].

6. SSDL.EXE

A tool, called SSDL.EXE has been created using the .Net 2.0 platform [21] and Web Services Enhancements 2.0 [22] to consume SSDL contracts. SSDL.EXE is able to generate C# and VB.NET code which in turn can be used in implementation of Web Services for sending and receiving messages. The implementation is extensible through plug-ins that can further process a validated SSDL document.

The architecture of SSDL.EXE is shown in Figure 7 and its current functionality is described in the following sections.

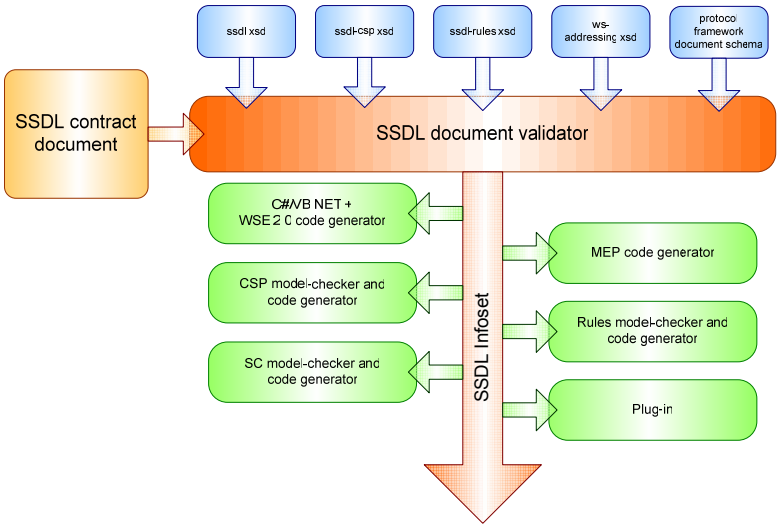

Figure 7: The architecture of the SSDL .NET processing tool

6.1. Validation

The structure and semantics of a SSDL contract document is validated against the language and protocol framework schemas. This happens through the following stages:

- The SSDL document is loaded by a .NET XML reader without validation.
- Any `ssdl:include` element information items are expanded and their equivalent `xi:xinclude` element information items are added into the XML Infoset representation of the loaded SSDL document.
- The XML Infoset representation of the SSDL document is validated, using a .NET 2.0 XML validating reader, against the XML Schemas describing the structure of a contract.
- The validated XML Infoset is checked for semantic correctness for those properties that cannot be described by schema languages (e.g. check whether the qualified names in the `ref` attribute of all the `ssdl:msgref` elements refer to declared messages or that the `ssdl:body` and `ssdl:header` refer to declared elements).

Once the input SSDL contract document has been validated, it is passed to a set of plug-ins that each processes the document.

6.2. Source Code Generation for Message-Oriented Programming Plug-in

The Ssdl.Wse plug-in uses .NET's CodeDOM API to generate C# or VB.NET classes for each of the messages declared in an SSDL contract. The plug-in builds on Microsoft's Web Services Enhancements 2.0 library to provide a message-oriented programming interface for sending and receiving messages defined in a contract. For example, the `PurchaserOrder` message defined in Listing 15 is converted to the classes of 16.

```
<?xml version="1.0" encoding="utf-8" ?>
<ssdl:contract targetNamespace="urn:e-procurement:example:contract"
               xmlns:ssdl="urn:ssdl:v1"
               xmlns:elemnts="urn:e-procurement:example:elements">
   <ssdl:schemas>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 elementFormDefault="qualified"
                 targetNamespace="urn:e-procurement:example:elements">
         <xs:element name="PurchaseOrder">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="Product" type="xs:string" />
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </ssdl:schemas>

   <ssdl:messages targetNamespace="urn:e-procurement:example:messages">
      <ssdl:message name="PurchaseOrder">
         <ssdl:body ref="elemnts:PurchaseOrder"/>
      </ssdl:message>
   </ssdl:messages>
</ssdl:contract>
```

Listing 15: An example of a message that is going to be represented as a C# class

```
namespace example.types.urn.e.procurement.example.elements
{
   using System;
   using System.Xml;
   using System.Xml.Serialization;

   [SerializableAttribute()]
   [XmlTypeAttribute(Namespace="urn:e-procurement:example:elements")]
   [XmlRootAttribute(Namespace="urn:e-procurement:example:elements", IsNullable=false)]
   public class PurchaseOrder
   {
      private string productField;
      public string Product
```

```
      {
        get { return this.productField; }
        set { this.productField = value; }
      }
   }
}

namespace example.messages
{
   using System;
   using System.Xml;
   using example.types.urn.e.procurement.example.elements;

   public class PurchaseOrderMsg : Ssdl.Wse.SsdlMessage
   {
      private PurchaseOrder  PurchaseOrder = new PurchaseOrder();
      [Ssdl.Wse.SsdlBodyAttribute(ElementName="PurchaseOrder",
                  Namespace="urn:e-procurement:example:elements")]
      public PurchaseOrder PurchaseOrder
      {
         get { return this._PurchaseOrder; }
         set { this. PurchaseOrder = value; }
      }
   }
}
```

Listing 16: C# class representing the message of Listing 15

Each message is a subclass of the `Ssdl.Wse.SsdlMessage` which encapsulates a `Microsoft.Web.Services2.SoapEnvelope` instance which in turn represents the contents of a SOAP message. When a message is to be sent, an instance of the appropriate class is constructed and passed to the *Send(SsdlMessage)* method of an `SsdlSender` instance. The `SsdlSender` class provides functionality equivalent to that of WSE's `SoapSender`. Listing 17 shows an example of how a `PurchaseOrderMsg` message can be send.

```
class Program
{
   static void Main(string[] args)
   {
      EndpointReference epr =
         new EndpointReference(new Uri("soap.tcp://localhost:10001/service"));

      SsdlSender sender = new SsdlSender(epr);

      PurchaseOrderMsg msg = new PurchaseOrderMsg();
      msg.PurchaseOrder.Product = "Product 1";

      sender.Send(msg);
   }
}
```

Listing 17: Example of sending a `PurchaseOrderMsg` message

A `ServiceReceiver` class, which inherits from `SsdlReceiver`, is created for the input SSDL contract as illustrated in Listing 18. For each message in the contract, an event is created.

```
namespace example
{
   using example.messages;

   public sealed class ServiceReceiver : Ssdl.Wse.SsdlService
   {
      [Ssdl.Wse.SsdlEventAttribute(MsgType = typeof(example.messages.PurchaseOrderMsg))]
      public event PurchaseOrderMsgEventHandler PurchaseOrderMsgReceived;

      private void OnPurchaseOrderMsgReceived(PurchaseOrderMsg msg)
      {
         PurchaseOrderMsgEventHandler evnt = this.PurchaseOrderMsgReceived;
         if ((evnt != null))
         {
            evnt(msg);
         }
      }

      public delegate void PurchaseOrderMsgEventHandler(PurchaseOrderMsg msg);
```

```
    }
}
```

Listing 18: A specialisation of an SsdlReceiver class for an SSDL contract

Service implementers use the events for the messages to implement the service logic that should be executed on arrival of a specific message as illustrated in Listing 19. More than one handler may be called to deal with the arrival of one message.

```
class Program
{
   static void Main(string[] args)
   {
      EndpointReference epr =
         new EndpointReference(new Uri("soap.tcp://localhost:10001/service1"));

      ServiceReceiver service = new example.ServiceReceiver();

      service.PurchaseOrderMsgReceived +=
         new ServiceReceiver.PurchaseOrderMsgEventHandler(PurchaseOrderMsgReceived);
      // This event is fired for all messages that arrive (it is defined by
      // the base class SsdlService
      service.MessageReceived +=
         new SsdlService.MessageReceivedDelegate(MessageReceived);

      SoapReceivers.Add(new EndpointReference(epr), service);
   }

   static void PurchaseOrderMsgReceived(PurchaseOrderMsg msg)
   {
      // Do something with the PurchaseOrderMsg message
   }

   static void MessageReceived(SoapEnvelope msg)
   {
      // Do something with the message
   }
}
```

Listing 19: An example of a service implementation as events

6.3. MEP Protocol Framework Plug-in

This plugin is currently under development. It will generate code that presents a familiar method abstraction for message exchange patterns defined using the MEP protocol framework.

6.4. CSP Protocol Framework Plug-in

This plugin is currently under development. Initially it will generate code that can be validated using the SPIN model checker. At a later stage it will generate code that can validate incoming/outgoing messages for conformance to the described protocol. Outgoing messages that are not part of the protocol will never reach the network while incoming messages will not reach the service logic.

6.5. Rules Protocol Framework Plug-in

This plugin is currently under development. Initially it will generate code that can be validated using the SPIN model checker. At a later stage it will generate code that can validate incoming/outgoing messages for conformance to the described protocol. Outgoing messages that are not part of the protocol will never reach the network while incoming messages will not reach the service logic.

6.6. SC Protocol Framework Plug-in

This plugin is currently under development.

6.7. SSDL to/from WSDL 2.0 Plug-in

An investigation is under way in order to identify the feasibility of converting SSDL contracts to/from WSDL 2.0 documents when the MEP SSDL Protocol Framework is used.

6.8. .NET class to SSDL document

There are plans for a .NET to SSDL document generator for message descriptions using attributes. High-level tools for describing protocols will be considered.

6.9. Other Plug-ins

The ssdl.exe tool is designed to be extended by third parties. This extensibility means that anyone choosing to create a new protocol framework within SSDL may also choose to create and publish a plugin for ssdl.exe which supports that protocol. Plugins that are not related to protocol frameworks can also be written.

Acknowledgements

Alan Fekete (fekete@it.usyd.edu.au, University of Sydney, Australia) and Jon Burton (J.I.Burton@newcastle.ac.uk, School of Computing Science, University of Newcastle upon Tyne, UK) for their significant contributions to the SSDL suite of specifications and documents.

References

[1] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield, "An Introduction to the SOAP Service Description Language," School of Computing Science, University of Newcastle, Newcastle upon Tyne CS-TR-898, 2005.

[2] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield, "SOAP Service Description Language (SSDL)," School of Computing Science, University of Newcastle, Newcastle upon Tyne CS-TR-899, 2005.

[3] S. Parastatidis and J. Webber, "MEP SSDL Protocol Framework," School of Computing Science, University of Newcastle, Newcastle upon Tyne CS-TR-900, 2005.

[4] S. Parastatidis and J. Webber, "CSP SSDL Protocol Framework," School of Computing Science, University of Newcastle, Newcastle upon Tyne CS-TR-901, 2005.

[5] D. Kuo, S. Parastatidis, and J. Webber, "Rules SSDL Protocol Framework," School of Computing Science, University of Newcastle, Newcastle upon Tyne CS-TR-902, 2005.

[6] S. Woodman, S. Parastatidis, and J. Webber, "Sequencing Constraints SSDL Protocol Framework," School of Computing Science, University of Newcastle, Newcastle upon Tyne CS-TR-903, 2005.

[7] W3C, "SOAP 1.2 Part 1: Messaging Framework," M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, and H. F. Nielsen, Eds. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/: W3C, 2003.

[8] W3C, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana, Eds. http://www.w3.org/TR/2004/WD-wsdl20-20040803/, 2004.

[9] Microsoft, "Web Services Specifications Index." http://msdn.microsoft.com/webservices/understanding/specs/.

[10] WS-I, "Web Services Interoperability (WS-I) Interoperability Profile 1.0a." http://www.ws-i.org.

[11] W3C, "Web Services Addressing (WS-Addressing)." http://www.w3.org/2002/ws/addr/.

[12] G. Holzmann, SPIN Model Checker, The Primer and Reference Manual: Addison Wesley Professional, 2004.

[13] "FDR2 User Manual." http://www.fsel.com/documentation/fdr2/, 1994.

[14] D. Kuo, A. Fekete, P. Greenfield, and S. Nepal, "Maintaining Consistency for Service Oriented Systems," CSIRO ICT Centre, Australia, Technical Report 05/017, 2005.

[15] W3C, "Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions." http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803/, 2004.

[16] C. A. R. Hoare, Communicating Sequential Processes: Prentice Hall International, 1985.

[17] OASIS, "Web Services Composite Application Framework (WS-CAF)." http://www.oasis-open.org/committees/ws-caf.

[18] OASIS, "Web Services Security (WS-Security)." http://www.oasis-open.org/committees/wss.

[19] S. Woodman, D. Palmer, S. Shrivastava, and S. Wheater, "Notations for the Specification and Verification of Composite Web Services," presented at 8th IEEE International Enterprise Distributed Object Computing (EDOC) Conference, Monterey, California, 2004.

[20] W3C, "XML Inclusions (XInclude) Version 1.0." http://www.w3.org/TR/xinclude/, 2004.

[21] Microsoft. "NET." http://www.microsoft.com/net/.

[22] Microsoft, "Web Services Enhancements (WSE)." http://msdn.microsoft.com/webservices/building/wse.