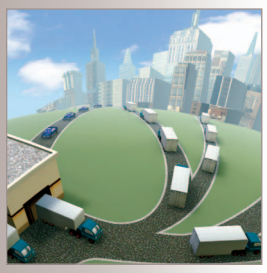

Asynchronous Messaging between Web Services Using SSDL

The SOAP Service Description Language (SSDL) is designed for describing asynchronous, message-oriented, and multimessage interactions between Web services. SSDL provides the basis for a range of protocol description frameworks. At one end of the spectrum, such frameworks can be simple, SOAP-centric replacements for the Web Services Description Language. At the other end, they're a more expressive contract-definition language enabling formal verification of asynchronous application protocol properties. This is possible because SSDL focuses on the "message" abstraction as the building block for service-oriented applications.

Savas Parastatidis and Simon Woodman
University of Newcastle

Jim Webber
ThoughtWorks

Dean Kuo and Paul Greenfield
CSIRO Information and Communication Technology Centre

Designing and developing asynchronous, message-oriented, distributed applications can be difficult. To support these tasks, middleware toolkits attempt to automate distributed applications' design, development, monitoring, maintenance, and evolution using metadata about process models, messaging behavior (interactions), policies, quality-of-service requirements, and so on. As more metadata becomes available to middleware toolkits and runtime environments, the level of automation in an application's life cycle can generally increase.

Today, the standard protocol for transferring messages between Web services is SOAP,[1] which defines an extensible processing model that's suitable for building asynchronous, message-oriented distributed applications. However, the default contract language for Web services — the Web Services Description Language (WSDL)[2] — doesn't explicitly target SOAP. Instead, it provides a generic framework for describing network-exposed software artifacts. WSDL's transfer-protocol independence makes describing SOAP message transfers more complex than simply adopting SOAP from the outset. Although the motivation to define a language that Web services application architects can use with other underlying transfer and transport technologies is understandable, the cost is high in terms of solution complexity for the dominant SOAP-based Web services.

WSDL's focus on an "interface" abstraction for describing services

 Published by the IEEE Computer Society 1089-7801/06/$20.00 © 2006 IEEE

SOAP Service Description Language Highlights

- SSDL assumes SOAP as the means of transferring messages between Web services over arbitrary transport (and transfer) protocols. Consequently, defining bindings for all possible transport protocols is unnecessary.
- SSDL assumes WS-Addressing as the standard means for embedding addressing information within SOAP envelopes and for binding those addresses onto underlying transport protocols. We assume that a SOAP binding for a transport protocol exists.
- SSDL focuses on messages and protocols. Consequently, articles such as "interface," "inheritance," and "operation" are unnecessary.
- SSDL assumes XML Infoset as its underlying component model. We don't need (or want) to create a new component model simply for contract description.
- SSDL handles contract modularization using XInclude. It also provides a shortcut mechanism that's defined in terms of XInclude elements and simplifies componentization as much as possible.
- SSDL promotes protocol framework extensibility. It lets us plug different protocol description models into the base SSDL framework, which helps promote protocol-based integration and expose Web services' tools messaging behavior. Tools such as model checkers can verify the correctness of protocols defined in an SSDL contract or automate the reasoning about Web services' compatibility. Hosting environments can even use the SSDL contract to validate message exchanges between Web services.

makes it difficult to change the object-oriented or Remote Procedure Call (RPC) mindset and focus on message-orientation and asynchrony as the means for achieving Web services integration. Furthermore, WSDL doesn't attempt to model any interaction patterns that involve more than two message exchanges between communicating Web services — the upper boundary of WSDL's capability is the classic request-response pattern. Furthermore, it's difficult to use WSDL to describe infrastructure protocols (such as transactions or reliable messaging) that use SOAP headers. Finally, technologies such as the Web Services Business Process Execution Language (WS-BPEL) — which uses WSDL as the basis for business process orchestration and other techniques — are more verbose and complex than they would be if they had more fundamental underlying messaging abstractions.

Motivated by architectural constraints for Internet-scale integration, we developed the SOAP Service Description Language (SSDL) to define a language for capturing Web services' messaging behavior. SSDL is an XML-based vocabulary for writing message-oriented contracts for Web services. It focuses specifically on SOAP messages and protocol frameworks to capture, in a contract document, Web services' messaging behaviors. (Given that we intend SSDL to fit naturally with the SOAP processing model, we assume that every Web service supports SOAP. We also developed the SSDL.exe tool to process SSDL contracts and generate .NET code with Microsoft's Web Services Enhancements framework (http://msdn.microsoft.com/webservices/building/wse).

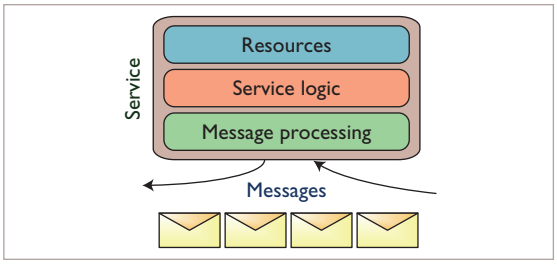

Figure 1. A typical service structure. The structure comprises resources, service logic, and a message-processing layer.

Message-Oriented, Asynchronous Services

Before we start exploring SSDL, we should briefly describe service-orientation, which is the contemporary architectural paradigm for building distributed applications.

Service-Orientation

Although service-orientation isn't a new architectural paradigm, emerging Web services technologies have reinvigorated interest in the approach. It's a misconception that Web services are a form of software "magic," automatically corralling the architect toward a loosely coupled solution that's scalable, robust, and dependable. We can certainly build service-oriented applications using Web services protocols and toolkits, but with the same technologies, we can also build applications according to the principles of other architectural paradigms (such as object-orientation).

As researchers and developers have rebranded their work to be in vogue with the latest buzz-

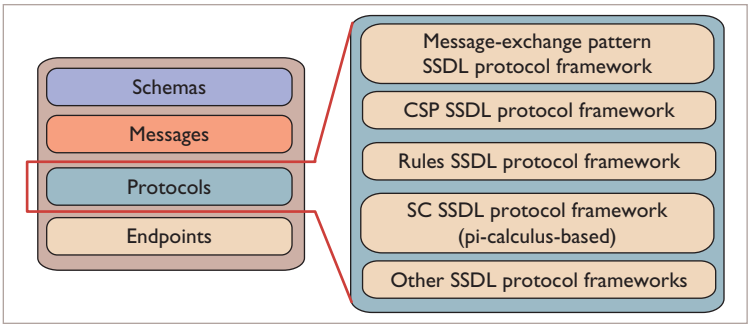

Figure 2. An SSDL contract. The contract has four major sections: schemas, messages, protocols, and endpoints. We can define a protocol using any available or future protocol frameworks.

```
<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>
```

Figure 3. A message and a fault message. The described message has two header and two body elements, and the fault message identifies a SOAP fault with value `Sender` *for* `soap:Code`.

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

Figure 4. Examples of `ssdl:msgref` *elements. The two message references point to the same message – defined elsewhere in the SSDL contract – with the first reference declaring the message as incoming, whereas the second is outgoing.*

words, the term *service-oriented architecture* (SOA) has become overloaded. In the absence of a widely accepted definition, we'll define a service as the logical manifestation of some physical or logical resources (databases, programs, devices, humans, and so on) or some application logic that is exposed to the network. Services interact through the exchange of messages.

Figure 1 shows a typical service consisting of resources, service logic, and a message-processing layer that manages message exchanges. Using a service's resources as necessary, its service logic acts on messages arriving at any point in time. Such services can be of any scale, from an operating system process to a multi-enterprise business process.

Devices hosting services can be of arbitrary size (for example, workstations, servers, phones, and personal digital assistants) and provide network applications with different types of functionality. This promotes a connected world in which no single device or service is isolated. Subsequently, we can build interesting applications by composing services and coordinating message exchanges between them.

One-Way Messages

Messages are units of communication between services. Service-oriented systems don't expose abstractions such as classes, objects, methods, or remote procedures; instead, services bind to the messages transferred between them. Service architects can logically group several message transfers to form message-exchange patterns (MEPs) – an incoming and a related outgoing message can form a *request–response* MEP, for example. Furthermore, an architect might group multimessage interactions to form protocols that are associated with some well-defined behavior for the participating services.

An important property of messages is their direction, which the source and destination addresses define. Such information is necessary for building asynchronous, message-oriented applications using one-way messages. By allowing a service to replay, cache, or store messages for delayed retrieval or processing, one-way messages enable – though they don't guarantee – loose coupling between services.

One-way messages might also have a certain order within message-interaction patterns (for instance, "reply to a previously sent message," "fault message as a result of a previously sent message," or "message *m* in an interaction pattern of *n* messages"). Although a message need not convey such information explicitly, metadata can help implicitly correlate messages into interaction patterns that capture service-oriented applications' behaviors. Metadata can also capture the semantics of infor-

mation exchanged between services. Mechanisms for declaratively capturing metadata about services are an important part of modern SOAs.

Protocols, Policies, and Contracts

In the traditional object-oriented world, behavioral semantics are associated with types, exposed through methods, and coupled with particular endpoints. However, we can also specify a service's messaging behavior in a distributed application via a set of messages and the order in which it sends and receives them (that is, the protocol that the service supports).

Service architects describe protocols and other metadata in contracts to which services adhere. Such contracts describe policy (security requirements or encryption capabilities, for example), quality-of-service characteristics (such as support for reliable messaging), and the semantics of the exchanged information a service supports or requires. In addition, contracts set forth the set of messages and MEPs that support the conveyance of functional information to and from the service. Contracts are critical to any realistic distributed application because they enable heterogeneity of implementation, ease of maintenance, portability, automation, and more when designing, building, and deploying services.

The SSDL Contract

An SSDL contract primarily aims to provide the mechanisms for service architects to describe the structure of the SOAP messages that a Web service supports. Once architects have described these messages, they can use available (or future) protocol frameworks to combine those messages into protocols that capture the service's messaging behavior. SSDL defines an extensible mechanism for using various protocol frameworks.

Using the SSDL-defined protocols, architects and developers can create systems that can meaningfully participate in conversations of arbitrary duration and complexity. Service designers or automated tools can discover SSDL contracts and compare the protocol descriptions against a given application's requirements to determine whether interactions will sensibly occur — reach an agreed termination state, for example, or not race or starve. (A race occurs when multiple possibilities about an interaction's forward progress exist, but, due to timing issues, participants observe different paths forward. Starvation indicates a situation whereby contracts are incompatible with each other due to certain messages never being sent. This could occur, for example, because one service expects a particular mes-

```
<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" />
      <ssdl:msgref ref="msgs:Fault1"
         direction="out" />
      <ssdl:msgref ref="msgs:Fault2"
         direction="out" />
    </mep:in-out>
  </ssdl:protocol>
</ssdl:protocols>
```

Figure 5. An in-out message-exchange pattern. `Msg2` *is sent as a reply to the* `Msg1` *request message. It's also possible that fault messages will be sent.*

```
<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>
```

Figure 6. Messaging behavior defined using the communicating sequential processes SSDL protocol framework. This very simple behavior shows the relative relation of three messages and a fault in an interaction between two services captured as a contract.

```
<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>
```

Figure 7. Race and starvation. The described interaction could lead to a race after `Msg1` *is received, given that the outgoing* `Msg2` *or the incoming* `Msg3` *might be in transit at the same time. If the sender service never sends* `Msg4`*, then the receiving service will starve.*

sage in order to progress, but the communicating service never sends this message.)

Figure 2 illustrates how we define an SSDL contract in a namespace that uniquely identifies the contract, which has four major sections: schemas, messages, protocols, and endpoints.

Schemas

In the schemas section, we define the structure of all the elements used to describe the SOAP messages. We can use any schema language to define global schema elements, although XML Schema[3] is the default choice.

Messages

The messages section is where the messages that a service understands are described. We can define many groups of messages in different namespaces. Irrespective of the namespace in which they're defined, however, messages included in the SSDL document are all part of the contract. Architects describe SOAP messages in terms of header and body elements and name them so that protocol frameworks can reference them.

In Figure 3, a message `"msg"` has two header elements (children of `soap:Header`) and two body elements (children of `soap:Body`), which refer to schema-defined global elements in the `elements` namespace. Although the SOAP processing model permits the existence of multiple body elements, the Web Services-Interoperability (WS-I) Basic Profile 1.0a (www.ws-i.org) mandates that `soap:Body` have a single child element. However, SSDL doesn't impose this restriction. Figure 3 also demonstrates how we can declare a SOAP fault message.

The header element provides the `mustUnderstand`, `role`, and `relay` attributes, which correspond to the SOAP processing model's equivalent attributes, thus making describing the Web services infrastructure (WS-*) protocols straightforward. Similarly, the fault construct's structure captures the information we usually find in SOAP fault messages.

Protocols

Protocol designers can describe how the messages declared in a contract might relate to each other. SSDL provides an extensible mechanism, based on the *protocol frameworks* concept.

A protocol framework is an XML-based vocabulary that can capture the relationship between messages in interactions across Web services and use the messages declared in a contract to describe everything from simple MEPs to multimessage interactions. SSDL is silent on interaction scope; a Web service can support one or more of a given protocol's instantiations concurrently. If it supports more than one, the service will need a contextualization mechanism to associate messages with particular instantiations of the protocol (for example, WS-Context,[4] WS-Security,[5] WS-Addressing,[6] or service-specific information). This applies to all protocol frameworks.

SSDL defines the semantics of the `msgref` element, which protocol frameworks must use when referencing messages. The `msgref` element provides the mandatory `ref` (an XML QName) and `direction` attributes, as well as optional `action` attributes. The `ref` attribute points to a declared message, whereas the `direction` attribute defines whether that particular message is incoming or outgoing. The `action` attributes define the uniform resource identifier (URI) that senders must use and receivers must expect for the SOAP message's WS-Addressing[6] `action` addressing property. A protocol description can refer to a message multiple times, and the same message can even be used for different parts of the same protocol, as Figure 4 shows. The current draft version of the WS-Addressing specification makes the `wsa:Action` header information element mandatory; thus, when it is omitted, we assume its value is `urn:ssdl:ProcessMessage`.

Protocol designers can define the same protocol

```
<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" />
            <rls:condition>
               <ssdl:msgref ref="msgs:Msg1" direction="in" />
               <rls:not>
                  <ssdl:msgref ref="msgs:Msg3" direction="in"/>
               </rls:not>
            </rules:condition>
         </rls:rule>

        <rls:rule>                                                     <!-- (3) -->
        <ssdl:msgref ref="msgs:Msg3" direction="in" />
            <rls:condition>
               <ssdl:msgref ref="msgs:Msg1" direction="in" />
               <rls:not>
                  <ssdl:msgref ref="msgs:Msg2" direction="out"/>
               </rls:not>
            </rules:condition>
         </rls:rule>

         <rls:rule>                                                    <!-- (4) -->
            <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:or>
            </rls:condition>
         </rls:rule>
      </rls:rules>
   </ssdl:protocol>
</ssdl:protocols>
```

Figure 8. The same interaction as that in Figure 7, captured using preconditions.

multiple ways using the same or different protocol frameworks, as best meets their needs. Furthermore, depending on the source and target frameworks, it's possible to translate the description of a service's messaging behavior from one protocol framework to another without losing any semantics.

We can associate some frameworks with the semantics of a formal model. Consequently, we can use model checkers, such as Spin[7] and FDR (www.fsel.com/documentation/fdr2/), to verify the defined protocols' safety (for example, agreed termination and absence of starvation) and liveness (such as an eventual termination guarantee) properties. Given that various protocol frameworks might meet different requirements when describing protocols, Web services architects must choose

```
<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>
```

Figure 9. A simple multiparty protocol specified using the sequencing constraints protocol framework. Participants `serviceX` *and* `serviceY` *interact through message exchange.*

those that are most suitable for the protocols they're developing.

SSDL's initial release comes with four protocol frameworks:

- MEP,
- communicating sequential processes (CSP)
- rules, and
- sequencing constraints (SC).

Rather than critically comparing the CSP, rules, and SC protocol frameworks, we merely introduce their capabilities in this article. (More information on supporting Web services with formal methods is available elsewhere.[8])

Message-exchange pattern. The MEP SSDL protocol framework[9] defines the semantics and structure of XML elements to match the MEPs found in the WSDL 2.0 specification.[10]

The MEP SSDL protocol framework's approach to defining MEPs enables the easy validation of messages' structures through appropriate tooling. Furthermore, we define some of the MEP elements' semantics in terms of WS-Addressing, which makes it easier to develop tools to produce code stubs. Figure 5 shows the semantics of the `<in-out>` element, for example, which requires that the incoming message's `"Message ID"` WS-Addressing header value become the outgoing message's `"Relates To"` header value.

Unlike the other protocol frameworks we describe in this article, the MEP protocol isn't suitable for describing multimessage interaction patterns. Instead, it is intended to be a simple SOAP-native replacement for WSDL 2.0 and thus supports a much more limited and specific set of messaging patterns.

Communicating sequential processes. The CSP SSDL protocol framework[11] is based on the semantics of Hoare's Communicating Sequential Processes.[12] The protocol framework models interactions between services as sequential processes that communicate with each other. Sent or received messages represent events in CSP processes.

Figure 6 describes a very simple behavior for a Web service captured as a series of message exchanges. The behavior suggests that after the incoming `Msg1` message, the recipient service will send either `Msg2` or `Fault1`. If it sends `Msg2`, then `Msg3` will be expected by the partner service. Any deviation from this message sequence breaks the contract. The key difference between the MEP and CSP protocol frameworks is that MEPs can specify only very simple relationships between in and out messages, whereas CSP can capture more intricate interaction patterns.

Figure 7 shows a race condition — that is, what happens when a given Web service violates the contract when interacting with another, by sending `Msg3` before receiving `Msg2`, for example. Additionally, if the interacting service never sends

Msg4, the receiving service can starve. Although such a situation might not be fatal for a Web service, architects and application developers must ensure that protocols are free from race conditions and starvation if successful termination of protocol-based interactions is required. However, protocol frameworks supported by formal models benefit from tools such as model checkers that can detect if a protocol is prone to such problems.

Rules. The Rules SSDL protocol framework uses preconditions on send and receive events to describe (and constrain) messaging behavior. As with the CSP SSDL framework, we can use model checkers to verify that a protocol is free from starvation and race conditions. The protocol in Figure 8 is an example of this framework.

We see that a service adhering to the described contract can receive Msg1 at any time (rule 1); send Msg2 or receive Msg3 after receiving Msg1 (rule 2); and receive Msg4 after sending Msg2 or receiving Msg3 (rule 3). Msg4 is also flagged as the final message in the protocol. The Rules framework lets us use and, or, xor, and not logical operators. The protocol description in Figure 8 suffers from the same race condition as the one in Figure 7; model checkers would bring this fact to the protocol designer's attention.

Sequencing constraints. The SC SSDL protocol framework describes multiparty (generally two or more) interactions and enforces an agreed-on contract for a single conversation among all parties involved. The framework's semantics are based on pi-calculus[13] and use an SSDL contract's declared messages to define the allowed interactions between other services with specific participant roles in the multiservice interaction.[14] Protocols are structured in terms of pi-calculus processes, which execute in parallel and can communicate with each other.

We define multiparty interactions by defining multiple participants and annotating the ssdl:msgref element with a participant that's interacting with the service. We assume that one participant is the service itself and another is implicitly bound by sending the initial message that starts the protocol. The protocol can identify the introduction of new participants through the participant-binding-name attribute by using some other out-of-band method.

Figure 9 shows a simple, multiparty protocol that defines two participants, serviceX and serviceY. We define the protocol as a sequence of messages, beginning with the receipt of msg1 from serviceX. The endpoint references (EPR) in the message binds serviceY and then sends msg2 to serviceX. Following this message, serviceX will send either msg3 or msg4 to serviceY (as bound by the message

```
<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>
```

Figure 10. Two endpoints in an SSDL contract. Web service endpoints are defined as WS-Addressing references.

```
<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/ser-
   vice/messages.ssdl" />
  </ssdl:messages>

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

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

Figure 11. SSDL contract document composed using XInclude. XInclude supports modularization and obviates the need to define new semantics.

Related Work in Protocol Description Languages

In addition to Web Services Description Language, the Web Services Business Process Execution Language (WS-BPEL)[1] and WS-Choreography[2] have gained some prominence within the Web services community as candidates for describing complex Web service contracts. Both (abstract) WS-BPEL and WS-Choreography layer on top of WSDL contracts and augment those contracts with additional information pertaining to the choreography of the message-exchange patterns contained therein.

Although both approaches have merit, they also have drawbacks. In particular, given that both rely on WSDL, a high level of complexity exists. The building blocks for the process or choreography descriptions isn't the "message" abstraction, as you might expect, but rather the "operation." SOAP Services Description Language, on the other hand, enables protocol description directly through message correlation. Consequently, we should be able to define both WS-BPEL and WS-Choreography as readily as SSDL protocol frameworks.

References

1. *Web Services Business Process Execution Language*, specification by the Organization for the Advancement of Structured Information Standards (Oasis), May 2003; www.oasis-open.org/committees/download.php/2046/BPEL%20V1-1%20May%205%202003%20Final.pdf.
2. *WS-Choreography*, W3C working draft, Dec 2004; www.w3.org/TR/2004/WD-ws-cdl-10-20041217.


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

Figure 12. Example of the `include` *element. This element is used as a shortcut to XInclude elements for modularization.*

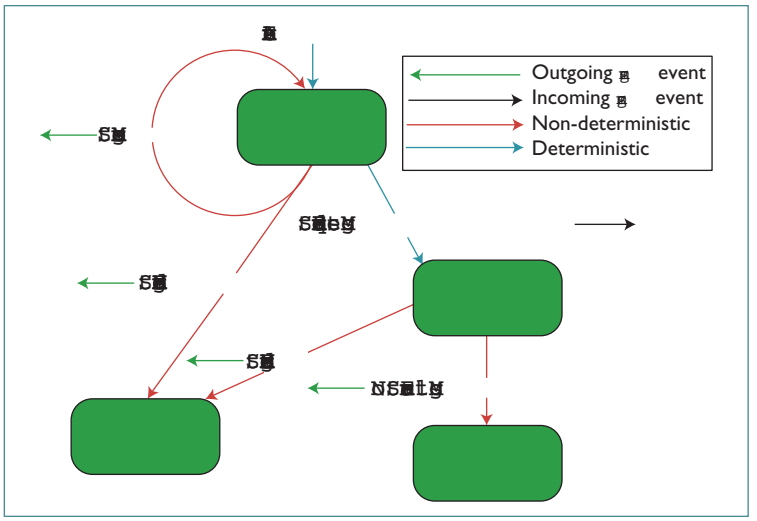

Figure 13. The WS-Streaming protocol. The nodes represent states, and arcs represent events.

received from `serviceX`). Finally, `serviceY` sends `msg5`. In addition to defining choice and sequence structures, the SC protocol framework defines multiple and parallel constructs, enabling the description of more complex interaction patterns.

Appropriate tooling can automatically translate protocols defined in SC to a pi-calculus form or to an input language for a model checker. Analyzing such definitions lets us reason about Web services' composability and compatibility characteristics as well as employ model checkers to prove properties such as liveness and consistency.[14]

Endpoints

An SSDL contract can also define endpoints — as WS-Addressing EPRs — of Web services that support the defined contract (as in Figure 10). Although a contract's schemas, messages, and protocols are constant (as per its namespace), endpoints supporting that contract might vary over time.

Modularization

We've defined an SSDL contract's structure in a way that supports modularization without the need to define new semantics for document composition. Instead, SSDL's document structure supports modularization through XInclude elements.[15] SSDL contract authors should use XInclude when a contract is composed from smaller XML documents. The resulting document must be a valid SSDL document, such as that in Figure 11.

We can assume that XInclude elements are used for all a contract author's modularization needs. Additionally, SSDL also provides the `include` element, which is defined in terms of specific XInclude elements, so it doesn't introduce new semantics into XML document composition. If an SSDL contract contains an `include` element, the SSDL processor expands it to the corresponding

```
<?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:Stream
            EndRequest"
               mustUnderstand="true"
```

continued on p. 36

Figure 14. The CSP-based SSDL contract for the WS-Streaming protocol. Unlike WSDL, the SSDL contract can capture a multimessage interaction.

XInclude elements before validating the document.

WS-Streaming Example

The WS-Streaming example demonstrates how we can use SSDL to describe a Web services infrastructure protocol — a fairly straightforward process, given SSDL's focus on SOAP and its explicit support for describing header and body elements. Figure 13 illustrates the protocol as a state machine, in which nodes represent states and arcs represent events.

We use the CSP SSDP protocol framework to describe the WS-Streaming protocol (see Figure 14) from the stream originator's viewpoint. However, we could also use the Rules or SC protocol frameworks.[16]

As in the example in Figure 6, a race condition exists in this protocol. A `StreamEndRequestMsg` might be headed toward the Web service generating the `StreamMsg` messages, while a `StreamEndMsg` is being transferred the other way. In the WS-Streaming example, the streaming service can deal easily with the condition by ignoring the `StreamEndRequestMsg` message. However, some applications might require their protocols to be free of such races. SSDL's strength is that model checkers can highlight the issue so that a service architect can make an informed decision on whether a redesign of the protocol is necessary.

Implementation

We created the SSDL.EXE tool using the .NET 2.0 platform (www.microsoft.com/net/) and Web Ser-

continued from p. 35

```
                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:Stream
                  EndRequestMsg" direction=
                  "in" />
               <csp:non-d-choice>
                  <msgref ref="msgs:Stream
                     EndMsg" direction="out" />
                  <msgref ref="msgs:NoStream
                     FaultMsg" direction="out" />
               </csp:non-d-choice>
            </csp:sequence>
         </csp:sub-process>

      </protocol>
   </protocols>
</contract>
```

Fig. 14, continued.

vices Enhancements 2.0 (http://msdn.microsoft.com/webservices/building/wse/) to consume SSDL contracts. SSDL.EXE can generate C# and VB.NET code, which we can use in turn for sending and receiving messages when implementing Web services. Such implementations would be extensible through plug-ins that could further process validated SSDL documents. Figure 15 shows the SSDL.EXE architecture.

Validation

We validate an SSDL contract document's structure and semantics against the language and protocol framework schemas in the following stages:

1. a .NET XML reader loads the unvalidated SSDL document.
2. The tool expands any `ssdl:include` element information items and adds their equivalent `xi:xinclude` element information items to the document's XML infoset representation.
3. Using a .NET 2.0 XML validating reader, SSDL.EXE validates the document's XML infoset representation against the XML schemas describing the contract's structure.
4. The tool checks the validated XML infoset for semantic correctness for those properties the schema languages can't describe (for example, it checks whether qualified names in the `ssdl:msgref` elements' `ref` attributes refer to declared messages and confirms that the `ssdl:body` and `ssdl:header` refer to declared elements).

Once the tool has validated the input SSDL contract document, it passes it to the set of plug-ins available

for processing the document. Currently, the code generator is the only plug-in available, and the SSDL framework plug-ins have yet to be implemented.

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 message declared in an SSDL contract. The plug-in builds on Microsoft's Web Services Enhancements (WSE) 2.0 library to provide a message-oriented programming interface for sending and receiving messages defined in a contract. This approach contrasts with popular Java and .NET tooling, which both attempt to provide object-oriented abstractions for implementing Web services. Instead, SSDL.EXE generates the necessary code for representing to developers the interactions between Web services through asynchronous, one-way messages and event programming abstractions. For example, SSDL.EXE converts the `StreamMsg` message defined in Figure 14 into the pseudo class of Figure 16.

Figure 17 shows how an application might send a `StreamMsg` message. Each message is a subclass of the `Ssdl.Wse.SsdlMessage`, which encapsulates a `Microsoft.Web.Services2.Soap Envelope` instance that in turn represents the contents of a SOAP message. Before sending a message, the implementation constructs an instance of the appropriate class and passes it to the `Send(SsdlMessage)` method of an `SsdlSender` instance. The `SsdlSender` class provides functionality equivalent to that of WSE's SoapSender (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wseref/html/T_Microsoft_Web_Services2_Messaging_SoapSender.asp).

Whereas `SsdlSender` is used to send messages to Web services, the `SsdlReceiver` class is used when implementing the receiving part of an interaction. Figure 18 shows the code generated to represent Figure 14's `StreamMsg` message as an event.

Service implementers use the events for the messages to implement the service logic, which is executed once a specific message arrives, as Figure 19 shows. An implementer might choose to call more than one handler to deal with a message's arrival.

Although it might seem odd to dismiss the method-call abstraction in favor of an event-driven mechanism, exposing a truthful picture of a service's messaging behavior to its implementation offers a more robust approach. Ignoring the difference between local and remote components leads to brittle systems.[17] By exposing such behavior to a service implementation, we let that implementation be robust against the underlying network's real behavior rather than against some idealized RPC-centric view. Given access to the full set of protocols that a service supports, we can develop implementations to anticipate and more gracefully tolerate failures.

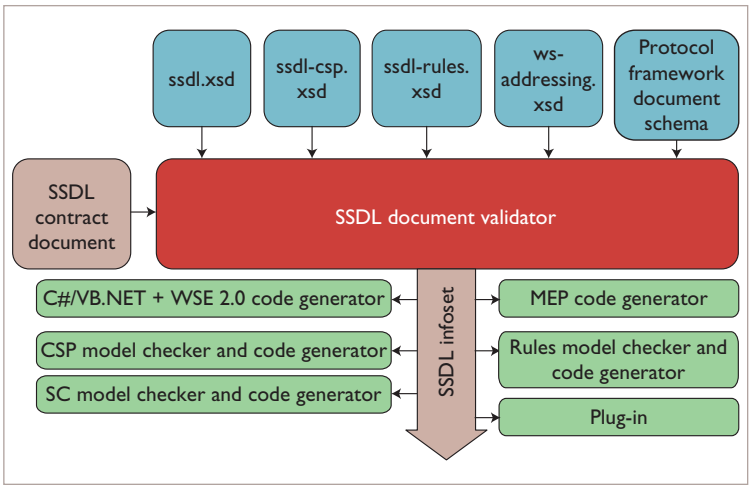

Figure 15. The SSDL .NET processing tool architecture. The SSDL.EXE tool reads an SSDL document and validates its structure against the appropriate SSDL XML Schema documents. Then, it makes use of plug-ins to generate code or pass the protocol through model checkers.

```
public class StreamContext
{
   public URI StreamID
   public int Sequence
}

public class StreamMsg : Ssdl.Wse.SsdlMessage
{

[Ssdl.Wse.SsdlHeaderAttribute(ElementName="Stream
   Context",
        Namespace="urn:ssdl:example:ws-
           streaming:contract:schema")]
   public StreamContext_StreamContext = new Stream
      Context();
}
```

Figure 16. C# class representing Figure 14's `StreamMsg` message. A message abstraction is generated to represent the message with the necessary attributes to indicate how the SOAP message is to be serialized or deserialized.

```
class Program
{
   static void Main(string[] args)
   {
      EndpointReference epr =
         new EndpointReference(new
Uri("soap.tcp://localhost:10001/service"));
      Ssdl1Sender sender = new Ssdl1Sender(epr);
      StreamMsg msg = new StreamMsg();
      sender.Send(msg);
   }
}
```

Figure 17. Sending a `StreamMsg` *message. Messages, rather than method calls, are the abstraction for sending messages to Web services.*

```
public sealed class ServiceReceiver :
Ssdl.Wse.SsdlService
{
   [Ssdl.Wse.SsdlEventAttribute(MsgType =
      typeof(StreamMsg))]
   public event StreamMsgEventHandler
      StreamMsgReceived;

   private void OnStreamMsgReceived(StreamMsg msg)
   {
       StreamMsgEventHandler evnt = this.Stream
          MsgReceived;
      if ((evnt != null)) {
         evnt(msg);
      }
   }
   public delegate void
StreamMsgEventHandler(StreamMsg msg);
}
```

Figure 18. A specialization of an `SsdlReceiver` *class for an SSDL contract. Events are made available for incoming messages in an SSDL contract.*

Clearly, SSDL is a departure from the orthodoxy of Web services technologies. However SSDL has emerged at a point where deep misgivings about the Web services orthodoxy have begun to surface, and indeed at a point at which there is open revolt in some circles about the RPC nature of existing approaches (a criticism leveled at both .NET and Java). Although it might be wishful thinking for us to hope that SSDL could replace the incumbent WSDL as a more suitable contract language for Web services, it does provide a useful vehicle for experimenting with alternative message-oriented approaches. We note with particular enthusiasm that SSDL is a natural bedfellow of workflow systems and hope to pursue further investigation in that area.

Acknowledgments

We thank Alan Fekete and Jon Burton for their significant contributions to the SSDL suite of specifications and documents. We also thank the *IEEE Internet Computing* reviewers for their constructive feedback.

References

1. M. Gudgin et al., *SOAP v. 1.2, Part 1: Messaging Framework*, W3C recommendation, 24 June 2003; www.w3.org/TR/2003/REC-soap12-part1-20030624/.
2. R. Chinnici et al., *Web Services Description Language (WSDL) v. 2.0, Part 1: Core Language*, W3C recommendation, 3 Aug. 2004; www.w3.org/TR/2004/WD-wsdl20-20040803/.
3. *XML Schema*, W3C recommendation, 2 May 2001; www.w3.org/XML/Schema/.
6. *Web Services Addressing (WS-Addressing)*, W3C recommendation, 17 Aug. 2005; www.w3.org/2002/ws/addr/.
7. G. Holzmann, *SPIN Model Checker, The Primer and Reference Manual*, Addison-Wesley Professional, 2004.
8. D. Kuo et al., *Maintaining Consistency for Service-Oriented Systems*, tech. report 05/017, CSIRO ICT Centre, Australia, 2005.
9. S. Parastatidis and J. Webber, *MEP SSDL Protocol Framework*, tech. report CS-TR-900, School of Computing Science, Univ. of Newcastle, 2005.
10. *Web Services Description Language (WSDL) v. 2.0, Part 2: Predefined Extensions*, W3C recommendation, 3 Aug. 2004; www.w3.org/TR/2004/WD-wsdl20-extensions-20040803/.
11. S. Parastatidis and J. Webber, *CSP SSDL Protocol Framework*, tech. report CS-TR-901, School of Computing Science, Univ. of Newcastle, 2005.
12. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
4. "Web Services Composite Application Framework (WS-CAF)," specification by the Organization for Structured Information Systems (Oasis), in progress; www.oasis-open.org/committees/ws-caf/.
5. "Web Services Security (WS-Security)," specification by the Organization for Structured Information Systems (Oasis), Jan. 2004; www.oasis-open.org/committees/wss/.
13. S. Woodman, S. Parastatidis, and J. Webber, *Sequencing Constraints SSDL Protocol Framework*, tech. report CS-TR-903, School of Computing Science, Univ. of Newcastle, 2005.
14. S. Woodman et al., "Notations for the Specification and Verification of Composite Web Services," *Proc. 8th IEEE*

```
class Program
{
   static void Main(string[] args)
   {
      EndpointReference epr =
         new EndpointReference(new Uri("soap.tcp://localhost:10001/service1"));
      ServiceReceiver service = new ServiceReceiver();
      service.StreamMsgReceived +=
         new ServiceReceiver.StreamMsgEventHandler(StreamMsgReceived);
      // This event is fired for all messages that arrive (it is defined by
      // the base class Ssdl1Service
      service.MessageReceived +=
         new Ssdl1Service.MessageReceivedDelegate(MessageReceived);
      SoapReceivers.Add(new EndpointReference(epr), service);
   }
   static void StreamMsgReceived(StreamMsg msg)
   {
      // Do something with the StreamMsg message
   }
   static void MessageReceived(SoapEnvelope msg)
   {
      // Do something with the message
   }
}
```

Figure 19. A service implementation as events.

Int'l Enterprise Distributed Object Computing Conf. (EDOC), IEEE CS Press, 2004, pp. 35–46.

15. *XML Inclusions (XInclude) v. 1.0*, W3C recommendation, 20 Dec. 2004; www.w3.org/TR/xinclude/.
16. S. Parastatidis et al., *An Introduction to the SOAP Service Description Language*, tech. report CS-TR-898, School of Computing Science, Univ. of Newcastle, 2005.
17. J. Waldo et al., *A Note on Distributed Computing*, tech. report SMLI TR-94-29, Sun Microsystems, 1994.

Savas Parastatidis is a program manager at Microsoft Corporation. Previously, he was a principal research associate at the School of Computing Science, University of Newcastle upon Tyne. His research interests include service-oriented architecture and Web services technologies. Savas has a PhD in computing science from the University of Newcastle. He is a member of the IEEE and the ACM. Contact him through his blog at http://savas.parastatidis.name.

Jim Webber is a senior consultant with ThoughtWorks, where he leads the service-oriented architecture practice. Previous research includes developing strategies for aligning grid computing with Web services practices and architectural patterns for dependable service-oriented computing. Webber has a PhD in parallel computing from the University of Newcastle upon Tyne. He is coauthor of the book *Developing Enterprise Web Services — An Architect's Guide* (Prentice Hall, 2003). Contact him through his blog at http://jim.webber.name.

Simon Woodman is a PhD student in the School of Computing Science, University of Newcastle upon Tyne. His research interests include Web services, workflow enactment technologies, business process modeling, and process algebra. Woodman has a BSc in computing science from the University of Newcastle. Contact him at s.j.woodman@ncl.ac.uk.

Dean Kuo is the grid architect at the eScience North West Centre (ESNW) Manchester UK. His research interests include service-oriented systems and grid computing. Kuo has a PhD in computer science from the University of Sydney. Contact him at dean.kuo@manchester.ac.uk.

Paul Greenfield is a researcher and consultant based in Australia. His current research focuses on the impacts that global connectivity will have on enterprise computing technologies, and includes Web services, application correctness, security, and trusted computing. Greenfield has a BSc and an MSc in computer science from the University of Sydney. He is a member of the IEEE and the ACM. Contact him at p.greenfield@computer.org.