Soya: a Programming Model and Runtime Environment for Component Composition using SSDL

Patric Fornasier[1], Jim Webber[2], and Ian Gorton[3]

[1] Empirical Software Engineering, National ICT Australia and
School of Computer Science and Engineering, University of New South Wales
`patricf@cse.unsw.edu.au`
[2] ThoughtWorks
United Kingdom
`jim@webber.name`
[3] Pacific Northwest National Lab
WA 99352, USA
`ian.gorton@pnl.gov`

Abstract. The SOAP Service Description Language (SSDL) is a SOAP-centric language for describing Web Service contracts. SSDL focuses on message abstraction as the building block for creating service-oriented applications and provides an extensible range of protocol frameworks that can be used to describe and formally model component composition based on Web Service interactions. Given its novel approach, implementing support for SSDL contracts presents interesting challenges to middleware developers. At one end of the spectrum, programming abstractions that support message-oriented designs need to be created. At the other end, new functionality and semantics must be added to existing SOAP engines. In this paper we explain how component developers can create message-oriented Web Service interfaces with contemporary tool support (specifically the Windows Communication Foundation) using SSDL. We show how SSDL can be used as an alternative and powerful metadata language natively alongside existing tooling without imposing additional burdens on application developers. Moreover, we describe the design and architecture of the Soya middleware which supports SSDL-based development of Web Services on the WCF platform.

1 Introduction

Complex software systems can be constructed by composing many independently developed components using rules from an architectural framework. Service-oriented Architecture (SOA) [12] is the most recent design paradigm which guides software architects during the composition of component-based distributed software systems. In an SOA, independent components are called services. Services use messages to communicate and exchange structured information among each other while descriptions capture the form and patterns of these interactions.

Together, services, messages and descriptions form the three main components of a basic SOA [5].

Web Services technology offers a suitable platform for building component-based service-oriented systems. However, simply using Web Services technologies will not automatically lead to a service-oriented system [25]. In particular, WSDL, one of the oldest Web Services specifications, is procedure call-centric and constrains Web Services practitioners from adopting a more message-oriented mindset.

The SOAP Service Description Language (SSDL) [19] is an XML-based language for describing message-oriented Web Services and can, in its crudest form, be a direct replacement for WSDL. SSDL provides an extensible mechanism, known as protocol frameworks, for capturing a service's messaging behavior into interaction protocols. These protocol frameworks are typically derived from formal modeling techniques, which allow model checkers to verify the correctness of a service (component) composition. Most importantly, the message-centric concepts underlying SSDL provide a natural fit with service-oriented design principles and promise to hold solutions for some of the problems which limit WSDL.

This paper introduces a programming model that uses metadata embedded in source code to describe SSDL contracts in a familiar, declarative manner. An implementation of this programming model called Soya is presented and further used to demonstrate how a runtime environment for SSDL-based Web Services can be implemented.

In section 2 we discuss the background and motivation of our research. Section 3 describes the notions and concepts of the SSDL language. We follow with a detailed explanation of Soya in section 4. Finally, we conclude the paper by providing an assessment of our current work and indicating future research directions.

2 Background and Motivation

2.1 Why SSDL?

Web Services have matured into a commoditized platform for building service-oriented systems and are having an enormous impact on interoperable distributed computing [23]. Using Web Services technology for creating distributed applications, however, does not mean that a component-based architecture will magically become service or message-oriented [25]. Specifically, WSDL's focus on operations as the primary abstraction for communication, for example, can encourage developers to use it as a traditional Interface Description Language (IDL) [14] and build systems that are architecturally similar to RPC-based systems. These solutions hence suffer from tight coupling at component and distribution boundaries.

Vendor products often encourage developers to use WSDL to generate service proxies for existing components (e.g. [22,10]) in order to shield the details involved in accessing remote services. While this seems reasonable at first, it eventually leads to brittle systems, because users of the service will not be aware of the inherently fundamental differences between local and remote invocations in terms of latency, memory space, concurrency and partial failure scenarios [31]. This is why calls across a network must be addressed by the programmer in ways fundamentally different to invocations on local components.

In its current (draft) version, the WSDL 2.0 core specification [29] is over a 100 pages long and comments on it have not been favorable, mainly complaining of its unnecessary weight and complexity [7]. Still, WSDL does not provide any support for describing service protocols, apart from the eight simple message exchange patterns that are defined in the WSDL adjuncts specification [30]. This means that given a WSDL contract, it is — except for the most trivial cases — generally not possible to determine if a certain sequence of service invocations will succeed or fail. To describe more complex interactions, additional specifications such as WS-BPEL [13] or WS-Choreography [26] have to be used in addition to WSDL. Unfortunately, this further increases the complexity of the Web Service description [21].

Even though the W3C's *Web Services Architecture* note [25] defines that a Web Service has "an interface described in a *machine-processable* format (specifically WSDL)"[4], it acknowledges that there might be some other semantics apart from the Web Services description (WSD) that are crucial for components to successfully interact with each other. The note further states that the information may not necessarily be "explicit, written or machine processable, but implicit, oral or human oriented"[5]. Clearly, it is desirable to have Web Service descriptions that are expressive enough to capture every aspect of the contract, therefore enabling full automation of the agreement on semantics and the subsequent component interactions.

SSDL offers solutions to some of the problems that WSDL currently exhibits. It presents a more lightweight approach to describing Web Services. By focusing on messages, SSDL encourages the creation of loosely-coupled service-oriented applications. Furthermore, SSDL supports developers working directly with messages as their fundamental abstraction and discourages them from thinking about exposing component interfaces directly as Web Services. Finally, providing mechanisms for capturing a service's messaging behavior can be leveraged in a number of ways by middleware. As a result, it can have positive effects on service development, binding and execution and thus considerably simplify the service lifecycle [3].

2.2 SSDL Tool Support

Unfortunately, almost no data exists that reports on experiences using SSDL as part of Web Services-based SOAs. There is only one set of published results from a project known to have used SSDL to model its services [4]. The lack of rich

[4] W3C, Web Services Architecture, 2004, section 1.4.
[5] W3C, Web Services Architecture, 2004, section 1.4.4

empirical data makes it hard to assess the capabilities and potential of SSDL as a service description language in a general sense.

One reason why SSDL has not been used more widely is the lack of tool support that aids developers in creating and consuming SSDL contracts. More importantly, no runtime environment exists for executing SSDL-based Web Services, thus preventing SSDL from being more than a specification on paper. Therefore, we have developed Soya, which is a programming model and runtime environment for creating and executing SSDL-based Web Services. Soya is intended to serve as a research vehicle and its use in future case studies will provide us with the empirical data we need to determine if the message-centric and formally verifiable SSDL approach has significant benefits compared to the incumbent approaches.

The implementation of Soya has presented a number of interesting challenges. At one end of the spectrum, we wanted to create straightforward programming abstractions that foster SSDL's underlying message-oriented practices. At the other end, we wanted to reuse contemporary SOAP-processing middleware and equip it with new functionality and semantics related to SSDL. The solutions we adopted to these issues, as well as the fundamentals of SSDL, are described in the following sections.

3 SSDL Language Features

The SOAP Service Description Language (SSDL) is an XML-based language for describing Web Services. It describes Web Services in a purely message-oriented way, focusing on messages as the building blocks for creating service-oriented applications. Hence fundamentally, SSDL provides the necessary mechanisms for describing the structure of SOAP messages. It further offers an extensible range of protocol frameworks that can be used to combine and relate messages into protocols. These protocols describe the messaging behavior of a Web Service and define how other services can interact with it. Additionally, some protocol frameworks may be formally verified using model checkers to ensure the absence of deadlocks or race conditions.

An SSDL contract can be separated into the following four major sections:

- **Schemas**: Defines the structure of SOAP message elements used by the service, normally using XML Schema [28];
- **Messages**: Declares the SOAP messages that a service supports, including body elements and header elements not inferred by an associated policy document;
- **Protocols**: Defines how messages relate to each other and the valid sequences in which they can be exchanged. Different protocol frameworks can be used, depending on the required level of formal verification and the number of parties involved in a protocol;
- **Endpoints**: Uses WS-Addressing [24] to define endpoints of Web Services that are known to support the given contract.

SSDL assumes SOAP (over arbitrary transport protocols) together with WS-Addressing as the only means of transferring messages between services. Consequently, defining bindings for different transport protocols is unnecessary and messages can be described in a more lightweight way compared to WSDL, which does not explicitly target SOAP. Likewise, adopting SOAP from the outset gives developers greater control over message structures, because it makes it possible to define SOAP header elements as part of the contract. Figure 1 illustrates how a message is defined in an SSDL contract.

```
<ssdl:messages targetNamespace="urn:my:messages" xmlns:s="urn:my:schema">
 <ssdl:message name="MsgA">
  <ssdl:header ref="s:MyHeaderX"
    mustUnderstand="true" />
  <ssdl:header ref="s:MyHeaderY"
    role=".../ultimateReceiver"/>
  <ssdl:body ref="s:MyBody" />
 </ssdl:message>
</ssdl:messages>
```

Fig. 1. A message defined as part of an SSDL contract. The `header` and `body` refer to XML schema elements.

Messages defined in this way can be combined and related into protocols that capture a service's messaging behavior. Making this information available to consumers promotes protocol-based integration [21] rather than interface-centric solutions. Currently, four protocol frameworks — MEP (Message Exchange Pattern) [18], CSP (Communicating Sequential Processes) [17], Rules [8] and SC (Sequencing Constraints) [32] — have been specified, but additional protocol frameworks can be created and plugged into SSDL, if needed. Figure 2 exemplifies how a service's messaging behavior can be captured using the SSDL Rules [8] framework, which constrains incoming and outgoing messages using preconditions. If desired, the same behavior could also be expressed using a different protocol framework, for example if multiparty choreography is required.

4 Soya

Soya [6] is an open-source implementation of the SSDL specification [20]. It provides a programming model and runtime environment for creating and enacting SSDL contracts. Soya supports developers in building message-centric applications and offers mechanisms to define message structures and messaging behavior in a straightforward manner using metadata in order to express Web Service contracts within the host language environment for a component. Soya uses this component metadata to infer SSDL contracts that can then be exposed to other services. Most importantly, Soya enables users to execute SSDL-based Web Services. It ensures that incoming and outgoing service messages adhere to

```
<ssdl:protocol xmlns:rls="urn:ssdl:rules:v1">
 <rls:rule>
  <ssdl:msgref ref="m:MsgB" direction="out"/>
  <rls:condition>
   <ssdl:msgref ref="m:MsgA" direction="in"/>
   <rls:not>
    <ssdl:msgref ref="m:MsgC" direction="out"/>
   </rls:not>
  </rls:condition>
 </rls:rule>
</ssdl:protocol>
```

Fig. 2. Messaging behavior specified using the SSDL rules protocol framework. The protocol defines that `MsgB` can only be sent after `MsgA` has been received and before `MsgC` has been sent.

the messaging behavior defined in a deployed SSDL contract and dispatches the incoming messages to the the underlying component implementation.

The current prototype implementation of Soya is built on top of the Windows Communication Foundation (WCF) [11]. WCF is an extensible framework which can, amongst other styles, be used to build message-centric distributed applications.[6] This and the comprehensive set of XML APIs included in the .NET framework [9] were the main reasons why we chose WCF as the underlying communication system for Soya. Figure 3 highlights the relationship between Soya and WCF.

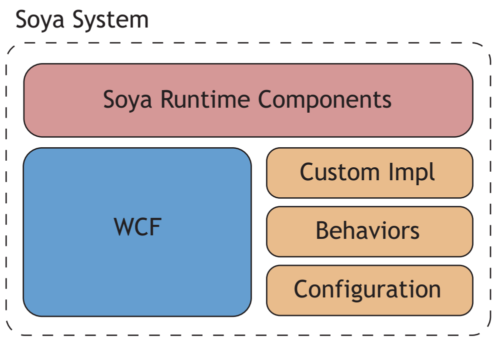

Fig. 3. Soya. Configuration files, injected behaviors and custom classes modify WCF's runtime behavior. The Soya runtime provides the SSDL specific functionality.

[6] Using WCF allows us to concentrate on implementing SSDL protocol support and delegate issues like session management, failure recovery, efficient processing of SOAP messages and so on to the underlying framework.

4.1 Defining SSDL Contracts Using C# Attributes

C# attributes are a mechanism for declaratively embedding metadata in C# component source code. This metadata adds additional information to the code that can be retrieved, processed and interpreted by other programs. In WCF's programming model, service and message contracts are typically defined in this declarative manner [15]. Soya reuses this programming model and provides additional SSDL-specific attributes and functionality. On one side this allows developers to define the structure of messages supported by an SSDL contract. On the other side it describes how these messages relate to each other through the use of different protocol frameworks. This attribute-oriented approach makes it possible to specify contract data with very little code yet provides extensive control over the contract when warranted.

In Soya, we have adopted the attribute-oriented programming model for the following reasons:

- less code and hence less scope for error introduction;
- more easily maintainable due to single source location;
- seamless integration with WCF's programming model and provision of familiar idioms to existing C# programmers.

Defining Messages Where possible, we reused existing WCF attributes to make the transition from WCF to Soya as smooth as possible. For concepts unique to SSDL, however, we had to introduce additional attributes (e.g. message names, message name-spaces, protocols . . .). The following code shows how messages are defined in Soya using C# attributes:

```
[SsdlMessageContract] // Soya attribute
public class MsgA {
    [MessageHeader]      public string MyHeader;
    [MessageBodyMember]  public MyData MyBody;
}

[DataContract]  // WCF attribute
public class MyData {
    [DataMember] public int id;
    [DataMember] public string code;
}
```

Attributes can take additional property parameters that can be used to override default values and give developers more control over the message data. For example, to explicitly specify the qualified name of the SSDL message element in the code above, one would simply define values for the `Name` and `Namespace` properties as shown in the following code fragment:

```
[SsdlMessageContract(Name="...", Namespace="...")]
```

From the above examples, Soya infers the following XML Schema code, which is part of the SSDL contract:

```
<xs:element name="MyHeader" type="xs:string"/>
<xs:element name="MyBody" type="s:MyData"/>
<xs:complexType name="MyData">
  <xs:sequence>
    <xs:element  name="id" type="xs:int"/>
    <xs:element  name="code" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Additionally, Soya generates the following SSDL message element, which is likewise included in the SSDL contract description:

```
<ssdl:message name="MsgA">
  <ssdl:header ref="s:MyHeader"/>
  <ssdl:body ref="s:MyBody"/>
</ssdl:message>
```

As illustrated in the above examples, Soya reuses attributes defined by WCF wherever possible (e.g. `MessageHeader`, `MessageBodyMember`). Instead of generating WSDL, however, it uses the attribute metadata to create SSDL contracts.

Defining Messaging Behavior Apart from defining messages supported by an SSDL contract, Soya's programming model may also be used to describe how these messages relate to each other. Soya has been designed to accommodate SSDL's extensible model and provides the necessary hooks to plug in new protocol frameworks. Of the four initial SSDL protocol frameworks, the MEP framework [18] is the simplest and least sophisticated. It does not demonstrate SSDL's full strength and has primarily been designed for capturing the Message Exchange Patterns defined by WSDL [30] so it can be used as a simple SOAP-centric language replacement for WSDL. The following lines show how simple MEP protocol interactions can be modeled using Soya's `MEP` attributes.

```
[Mep(Style=MepStyle.InOnly)]
public void Process(MsgA msg);

[Mep(Style=MepStyle.InOptionalOut, Out=typeof(MsgC),
 Fault=typeof(FaultX))]
public void Process(MsgB msg);
```

The attribute on the first method declaration defines an *in-only* MEP in which `MsgA` represents the incoming message. The second method declaration defines an *in-optional-out* MEP with `MsgB` representing the incoming message, `MsgC` being the outgoing message and `FaultX` standing for the optional fault message. From this code, Soya can generate the following SSDL protocol information which captures the messaging behavior in the SSDL contract:

```
<ssdl:protocol xmlns:mep="urn:ssdl:mep:v1">
  <mep:in-only>
    <ssdl:msgref ref="m:MsgA" direction="in"/>
  </mep:in-only>
  <mep:in-optional-out>
    <ssdl:msgref ref="m:MsgB" direction="in"/>
    <ssdl:msgref ref="m:MsgC" direction="out"/>
    <ssdl:msgref ref="m:FaultX" direction="out"/>
  </mep:in-optional-out>
</ssdl:protocol>
```

These examples show how Soya can use class information and attribute metadata to infer SSDL contracts. The examples also show how little additional code is necessary to create an entire SSDL contract including XML Schema definitions, protocol descriptions and method and fault declarations.

Exposing SSDL Contracts The most fundamental purpose of a Web Service description is to capture the semantics that describe how two or more services can interact meaningfully, in a machine-processable format. It is thus crucial that this description is exposed, so interested parties can retrieve it and reason about the described service. This reasoning might range from simply checking a Web Service's compatibility to performing protocol-based integration of services [21]. The previous sections have suggested that Soya can infer SSDL from the service classes and attribute metadata. Soya builds an internal service model from this data. Using this model, Soya can generate an SSDL contract represented as XML information set [27]. The infoset can then be serialized into XML and published using, for example, HTTP or WS-Metadata Exchange [2].

4.2 Architecture and System Design

To better understand Soya, we distinguish between service deployment and service execution. First, we describe how a service implementation is turned into an executable instance and exposed to the network. Then, we illustrate what happens inside Soya when other services interact with a deployed service and how the Soya runtime enforces a service's SSDL contract.

Service Deployment In Soya a service implementation typically consists of code representing the core application logic, metadata attributes describing the service's SSDL contract and configuration files specifying service endpoints, security settings and so on. A developer can deploy a service implementation by opening a `SoyaServiceHost` instance, which is a custom host implementation of WCF's `ServiceHostBase`. The following two lines show how a service (in this case `MyService`) is deployed:

```
host = new SoyaServiceHost(typeof(MyService));
host.Open();
```

Opening a `SoyaServiceHost` triggers the three following major activities, which are also graphically illustrated in Figure 4:

1. Reflect over service classes (i.e. service code and attribute metadata) and build an internal model representing the SSDL service contract from it;
2. Process application configuration files and add further artifacts, such as service endpoints or custom behaviors, to the internal model;
3. Create and configure WCF and Soya runtimes based on internal model.

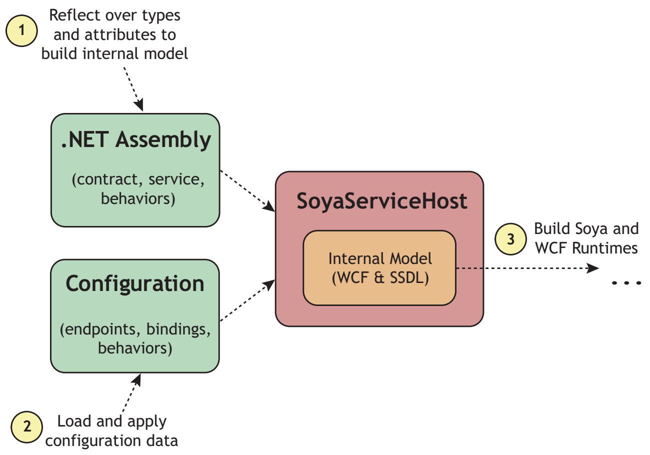
1
Reflect over types and attributes to build internal model
.NET Assembly
(contract, service, behaviors)
Configuration
(endpoints, bindings, behaviors)
2
Load and apply configuration data
SoyaServiceHost
Internal Model (WCF & SSDL)
3
Build Soya and WCF Runtimes
...

Fig. 4. Deployment of a service. The model is created from service type and attribute information as well as configuration data. Then, it is used to initialize the WCF and Soya runtimes.

Building the internal model constitutes most of the work the `SoyaServiceHost` performs after the `Open()` method is called. The internal model can be seen as an intermediary language between the service implementation and the SSDL contract description. It is used by the runtime as a blueprint for creating new stateful protocol instances and also to generate and expose SSDL metadata.

The `SoyaServiceHost` builds the internal model by reflecting over the service classes and applying configuration settings. It first of all identifies the SSDL protocol framework that has been used to model the service's messaging behavior and then uses protocol specific classes to process the class and attribute metadata. This includes inferring XML schemas, interaction protocols, and message descriptions and adding them to the model. Next, it processes the application's configuration files and adds further artifacts, such as service endpoints or custom behaviors, to the internal model. Finally, both the WCF and the Soya runtimes are created and configured. This includes injecting a message inspector into the WCF runtime that will later be used to intercept inbound and outbound messages. The message inspector, shown in Figure 5, bridges the two runtimes by passing intercepted messages from WCF to the Soya runtime. This terminates the service deployment and enables other services to start interacting with the deployed service through the specified endpoints.

Service Execution When an incoming message is received from the network, it is first of all pushed through WCF's channel stack. The channel stack consists of different elements that deserialize, decode and decrypt the incoming bits into an untyped `Message` instance. Immediately after the message exits the channel stack, it is intercepted by a custom message inspector and passed to the Soya runtime for further processing.

Once a message is passed to the Soya runtime, the runtime firstly uses an `XsdValidator` to validate the structure of the message's elements. It compares the header and body elements with the SSDL contract that is represented by the service's internal model and tries to locate the message in the current contract. If the message validation or location fails, the message is rejected. Otherwise, the message is further processed by an `IProtocolValidator`. This validator checks if the incoming message is valid in terms of the messaging behavior defined in the service's contract (i.e. the protocol definition).

As opposed to the `XsdValidator`, which is stateless, the `IProtocolValidator` needs to maintain state between interactions, as validation is based on the state of a conversation in which the interacting services are at a given point in time. Internally, this validation is performed with a state machine. It is built from the protocol definition and the incoming and outgoing messages represent the state transitions. If the message causes the state machine to transit to an invalid state, the message is rejected. Otherwise, it is returned to the WCF runtime, where the untyped `Message` instance is mapped into a user-defined message instance. Finally, this user-defined message instance is dispatched to a local method of the service implementation. Figure 5 illustrates this mechanism and the same process (in reverse) applies to outgoing messages.

4.3 Intelligent Message Dispatching

In SSDL the concept of *operations* or *service invocations* does not exist. Interactions between services are modeled purely as messages that are exchanged among services. Messages represent self-contained units of information and do not convey details of underlying APIs. SSDL expects that applications reason about the sequence of messages and derive appropriate actions from this. This concept has been described as the MEST (MESsage Transfer) architectural style [16].

Just like SOA, SSDL and MEST do not have operation abstractions, Soya does not have them either. Of course, since Soya is built using an object-oriented programming language, a local API method is ultimately invoked. This method, however, is not part of the service contract, but belongs to the service's internal implementation, thus enforces loose coupling. Soya inspects incoming messages and decides to which internal method the message should be dispatched. This

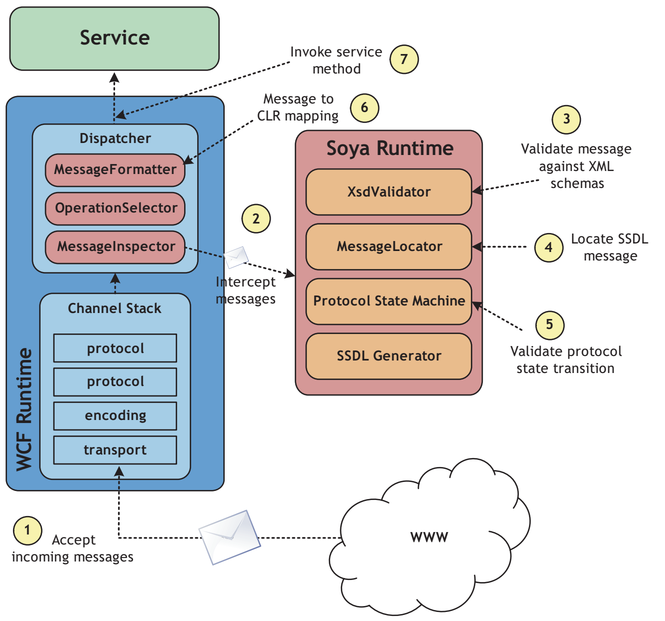

Fig. 5. Soya runtime architecture.

decision is exclusively based on the messaging behavior defined in the service's contract and the state of the current conversation. Method names play no role in this decision-making process, meaning that given a different protocol state multiple arrivals of the same message type can result in a different dispatching behavior. This is illustrated in the following C# pseudo-code.

```
[Rule(Condition="!(MsgB == In)")]                (1)
public void ProcessX(MsgA msg) {}

[Rule(Condition="MsgB == In")]                   (2)
public void ProcessY(MsgA msg) {}

[Rule(Condition="MsgA == In && !(MsgB == In)")]  (3)
public void ProcessZ(MsgB msg) {}
```

The above pseudo-code defines three different methods for processing incoming messages. Messages of type `MsgA` are dispatched to the first method as long as no message of type `MsgB` has been received. `MsgB` can be received exactly once (after one or more messages of type `MsgA` have been received) and is dispatched to the third method. After that, incoming messages of type `MsgA` are dispatched to the second method. The state machine that we can infer from this is shown in Figure 6.

Each method contains service logic that does something based on the type of message and the current conversation state. If we had no protocol metadata, the first and second methods would be ambiguous. A service developer would need to write application code to determine the conversation state of the application, correlate messages and finally dispatch them to the correct logic. We understand that this imposes a significant burden on the developer. Therefore, Soya takes advantage of the protocol metadata and infers a state machine that defines the correct order of the exchanged messages. The state machine is used to decide to which methods messages need to be dispatched. Presenting the developer with this abstraction eliminates the confusion as what needs to be implemented.

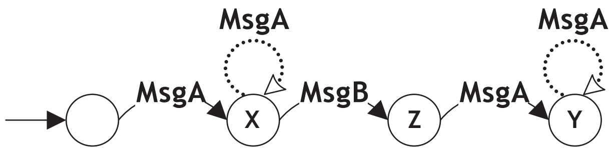

Fig. 6. State machine inferred from protocol metadata. `X`, `Y` and `Z` stand for the methods to which the message relating to the incoming transition will be dispatched.

5 Conclusion and Future Work

Web Service descriptions are machine-processable documents that capture the semantics that define how two services can interact meaningfully. Normally, Web Service descriptions are written in WSDL. In this paper we accept that there are significant drawbacks with WSDL for building service-oriented applications (e.g. focus on operations rather than messages, insufficient control over SOAP messages, high complexity, not expressive enough to capture sophisticated messaging behavior). The target for our empirical work instead uses an alternative Web Services description language called SSDL. Given the lack of empirical data on using SSDL as part of Web Services-based SOAs, we identified the need to further investigate and assess the capabilities of SSDL through empirical studies.

To that end we have developed Soya, a programming model and runtime environment for creating and executing SSDL-based Web Services. We have presented programming abstractions that not only allow developers to build SSDL services in a straightforward way but also encourage the creation of truly service-oriented applications without imposing unrealistic development burdens. Further, we have provided a detailed explanation of how we leveraged a contemporary SOAP engine by adding functionality and semantics related to SSDL, thus providing an advanced runtime environment for executing SSDL-based Web Services to the community.

The development of Soya has provided an extremely valuable insight into the creation, deployment and runtime enactment of SSDL contracts. In future investigations and case studies, we will use Soya as a research vehicle through which we can express our needs and experiences related to SSDL. Specifically, we will use Soya and SSDL to create a service-oriented system in the context of the Australian lending industry [1]. One one side, this will help us to validate the usability of Soya's programming model and the proper functioning of its runtime environment. On the other side, these experiments will provide us with the empirical data we need to determine, whether describing Web Services in SSDL has significant benefits compared to the incumbent approaches.

6 Acknowledgments

We would like to thank Liming Zhu, Savas Parastatidis and Pawel Kowalski for their constructive feedback and valuable support in creating this document.

National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

1. Lending Industry XML Initiative (LIXI). http://www.lixi.org.au.
2. K. Ballinger et al. Web services metadata exchange, version 1.1, 2006.
3. B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols.
4. E. R. de Mello, S. Parastatidis, P. Reinecke, C. Smith, A. van Moorsel, and J. Webber. Secure and provable service support for human-intensive real-estate processes. Technical report, University of Newcastle upon Tyne: Computing Science, 2006.
5. T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005.
6. P. Fornasier. Soya - SSDL engine. http://soya.sourceforge.net.
7. D. Hinchcliffe. Web service description languages: When there is nothing left to take away. http://hinchcliffe.org/archive/2005/05/10/215.aspx, 2005.
8. D. Kuo, S. Parastatidis, and J. Webber. Rules SSDL protocol framework. Technical Report CS-TR-902, School of Computing Science, University of Newcastle upon Tyne, 2005.
9. Microsoft Corporation. .NET framework 3.0. http://www.netfx3.com/.
10. Microsoft Corporation. Web services description language tool (wsdl.exe). http://msdn2.microsoft.com/en-us/library/7h3ystb6.aspx.
11. Microsoft Corporation. Windows communication foundation (WCF). http://wcf.netfx3.com/.
12. OASIS. Reference model for service oriented architecture v 1.0. http://www.oasis-open.org/committees/soa-rm/, 2006.
13. OASIS. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/, 2006.
14. Object Management Group. IDL syntax and semantics chapter. http://www.omg.org/cgi-bin/doc?formal/02-06-39.
15. D. Pallmann. *Programming INDIGO*. Microsoft Press, 2005.
16. S. Parastatidis. The MEST architectural style. http://savas.parastatidis.name/2004/11/09/92ede84c-ca1f-41ab-8feb-8ba50d07e86f.aspx, 2004.

17. S. Parastatidis and J. Webber. CSP SSDL protocol framework. Technical Report CS-TR-901, School of Computing Science, University of Newcastle upon Tyne, 2005.
18. S. Parastatidis and J. Webber. MEP SSDL protocol framework. Technical Report CS-TR-900, School of Computing Science, University of Newcastle upon Tyne, 2005.
19. S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. An introduction to the SOAP service description language. Technical Report CS-TR-898, School of Computing Science, University of Newcastle upon Tyne, 2005.
20. S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. SOAP service description language (SSDL). Technical Report CS-TR-899, School of Computing Science, University of Newcastle upon Tyne, 2005.
21. S. Parastatidis, S. Woodman, J. Webber, D. Kuo, and P. Greenfield. Asynchronous messaging between web services using SSDL. *Internet Computing, IEEE*, 10(1):26–39, 2006.
22. The Apache Software Foundation. Axis2. http://ws.apache.org/axis2/.
23. W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.
24. W3C. Web services addressing. http://www.w3.org/Submission/ws-addressing/, 2004.
25. W3C. Web services architecture. http://www.w3.org/TR/ws-arch/, 2004.
26. W3C. WS choreography model overview. http://www.w3.org/TR/ws-chor-model/, 2004.
27. W3C. XML information set (second edition). http://www.w3.org/TR/xml-infoset/, 2004.
28. W3C. XML schema. http://www.w3.org/XML/Schema, 2004.
29. W3C. Web services description language (WSDL) version 2.0 part 1: Core language. http://www.w3.org/TR/wsdl20/, 2006.
30. W3C. Web services description language (WSDL) version 2.0 part 2: Adjuncts. http://www.w3.org/TR/wsdl20-adjuncts/, 2006.
31. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories, Mountain View, CA, 1994.
32. S. Woodman, S. Parastatidis, and J. Webber. Sequencing constraints SSDL protocol framework. Technical Report CS-TR-903, School of Computing Science, University of Newcastle upon Tyne, 2005.