

# **Providing Context in WS-BPEL Processes**

by

Allen Ajit George

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Master of Applied Science  
in  
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

©Allen Ajit George 2008

## AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## ABSTRACT

Business processes are increasingly used by organizations to automate their activities. Written in languages like Web Services Business Process Execution Language (WS-BPEL), they allow a company or an institution to describe precisely its internal operations and identify the actors and processing steps involved in completing them. As the pace of change increases, however, both organizations and their internal processes are required to be more flexible; they have to account for an increasing amount of externally-driven environment state, or *context*, and modify their behavior accordingly. This puts a significant burden on business-process programmers, who now have to source, track, and update context from multiple entities, in addition to implementing and maintaining core business logic. Implementing this state-maintenance logic in a WS-BPEL business process is involved. This is because WS-BPEL business processes are modeled as if they were the only thing operating in, and making changes to, the business environment. In this mental model all environment state is changed by the business process – either as a result of its activities, or a query it makes to external entities. This mental model does not reflect the real world, where organizations and entities depend on state that is outside their control – state that is modified independent of, and concurrent with, the organization’s activities. The WS-BPEL language is, however, implicitly designed around this mental model, and the idea that all environment state and the variables they are stored in are updated and maintained by the process itself. This makes it hard for business-process programmers to write context-dependent processes in a concise manner.

This thesis presents a solution to this problem based on the notion of a context variable for WS-BPEL business processes. It describes how the WS-BPEL language-extension mechanism was used to design context variables, and how these variables can be used in business processes. This thesis outlines an architecture for offering context in the web services environment that uses constructs from the WS-Resource Framework specification. It shows how changes in context can be propagated from these context sources to WS-BPEL context variables using WS-Notification-based publish/subscribe. The proposed design also includes a standards-compliant method for extending web-service response messages with pointers to context sources. Finally, a prototype validat-

ing these extensions and the overall system is described, and enhancements for increasing the utility of context variables proposed.

The solution outlined in this thesis offers significant advantages: it builds on established practices and well-understood message-exchange patterns, leverages widely used languages, frameworks and specifications, is standards compliant, and has a low barrier-to-entry for business-process programmers. Moreover, when compared to either polling or onEvent handlers, this solution requires significantly less process logic and fewer interface changes to maintain constantly changing environment state. This makes it much simpler for business-process programmers to write flexible WS-BPEL business processes.

## ACKNOWLEDGEMENTS

First, I would like to thank both Kamran Jamshaid and Cecil Reid for their friendship, support, and for the times they simply listened. They have enriched my time at Waterloo.

Next, I would like to thank both Professors Jay P. Black and Krzysztof Czarnecki for agreeing to read my thesis. I enjoyed the classes I took with them, and have learned a lot from the way they looked at research problems.

I would be remiss if I did not acknowledge the many contributions of my supervisor, Professor Paul A.S. Ward. His support and encouragement during the course of my research was invaluable. I thank him for the many times I walked away from our one-on-one meetings with a head full of ideas and possibilities, and with a new appreciation for the joy in research. He has taught me a lot about the research process and how to develop ideas, and he gave me considerable flexibility in how I worked towards the solutions. I greatly appreciate his many insights during my years here.

Finally, I would like to thank my parents, for their love and understanding, and for their unending support. They have done so much for me.

## DEDICATION

To conversations, photography, and weights.

## TABLE OF CONTENTS

List of Figures .....	x
Chapter 1 Introduction.....	1
1.1 Contributions .....	2
1.2 Thesis Organization.....	3
Chapter 2 Background and Related Work .....	4
2.1 Context.....	4
2.2 Previous Work .....	5
2.3 WSDL .....	9
2.4 WS-BPEL.....	11
2.5 WS-Addressing .....	14
2.6 WS-Notification .....	16
2.7 WS-Resource Framework .....	18
Chapter 3 Context in WS-BPEL .....	20
3.1 Motivating Scenario .....	20
3.2 Polling.....	22
3.3 Message Handling .....	22
3.4 Proposed Approach.....	24
Chapter 4 Solution Outline .....	27
4.1 Modeling Context Sources .....	28
4.2 Context-Source References.....	33
4.3 Extending WS-BPEL.....	40
4.3.1 Language Support .....	40
4.3.2 WS-BPEL Engine Enhancements .....	44
4.4 Overall System .....	46
Chapter 5 Prototype .....	48
5.1 WSRF-Based Context Source Framework .....	48
5.1.1 Apache Muse .....	49
5.1.2 Managed Resources.....	52
5.1.2.1 Context-Source Factory .....	54

5.1.2.2 Context Source.....	54
5.1.2.3 ServiceGroup and ServiceGroupEntry .....	57
5.1.2.4 Subscription Manager .....	59
5.1.3 Development Impressions .....	60
5.2 Intermediate Web Service .....	61
5.2.1 Programming Model .....	62
5.2.2 Interface .....	65
5.2.3 Implementation .....	70
5.2.3.1 Interacting with the Invoking Service.....	71
5.2.3.2 Interacting with the Context-Source System.....	73
5.2.4 Development Impressions .....	75
5.3 Modified WS-BPEL Process Engine .....	75
5.3.1 ActiveBPEL 4.1 .....	76
5.3.2 Extension Validation .....	78
5.3.3 Generating Runtime Structures.....	80
5.3.4 Processing Extended Responses .....	83
5.3.5 Subscribing to Context Sources.....	84
5.3.5.1 Creating WSN NotificationConsumers.....	85
5.3.5.2 Subscribing.....	93
5.3.6 Processing Notifications .....	96
5.3.6.1 Propagating Notifications to a Process Instance.....	97
5.3.6.2 Updating the Context Variable.....	99
5.3.7 Development Impressions .....	99
5.4 Test Setup .....	100
5.5 Prototype Limitations .....	101
Chapter 6 Evaluation .....	102
Chapter 7 Enhancements .....	106
7.1 The <conditionWithTimeout> Extension Activity.....	106
7.2 Context Handlers .....	110
7.3 Context Handoff.....	112



7.4 Context Sources with Semantics .....	113
7.5 Other Enhancements .....	115
Chapter 8 Conclusions.....	116
Bibliography .....	118

## LIST OF FIGURES

Figure 1 WSDL <message> definition .....	9
Figure 2 WSDL operation .....	10
Figure 3 WSDL Port Type .....	10
Figure 4 Relationship between WS-BPEL Partner Links and Port Types .....	11
Figure 5 Sample WS-BPEL Partner Link and Partner Link Type .....	11
Figure 6 Sample WS-BPEL business process .....	13
Figure 7 XML representation of WSA EPR .....	14
Figure 8 WSA EPR with single Reference Parameter .....	15
Figure 9 WSA SOAP-Binding message-information header blocks .....	15
Figure 10 Translation of EPR reference parameters to SOAP-header elements .....	16
Figure 11 WSN NotificationProducer offering a set of topics in a TopicNamespace .....	17
Figure 12 Interaction between NotificationProducer, NotificationConsumer and Subscriber .....	17
Figure 13 WS-Resource with a set of WS Resource Properties in a WS Resource Properties Document .....	18
Figure 14 Sample Scenario .....	20
Figure 15 Example of WS-BPEL onEvent handlers and polling .....	21
Figure 16 High-level conceptual model of context variables .....	25
Figure 17 High-level interaction between scenario actors .....	27
Figure 18 Design of the <code>Shipment</code> context source .....	29
Figure 19 Context variable system using WSRF constructs .....	30
Figure 20 Two EPRs differentiated using reference parameters .....	31
Figure 21 Two different mappings between EPRs are WS-Resource instances .....	31
Figure 22 Sample message flow for context source creation .....	32
Figure 23 Elements in a context variable linked to different context sources .....	35
Figure 24 Context parameter schema .....	37
Figure 25 Use of <StatefulParameterName> .....	39
Figure 26 Functionality a WS-BPEL engine needs to support context variables .....	44
Figure 27 High-level message flow for context variable setup .....	45
Figure 28 Routing notifications from a context source to a WS-BPEL process .....	46

Figure 29	ShipmentCondition context type.....	48
Figure 30	Apache Muse high-level design.....	49
Figure 31	Port Type with both a specification-defined and custom operation.....	51
Figure 32	Mapping between a WSA Action URI and the Muse receiving class.....	51
Figure 33	Interaction between Muse resources .....	52
Figure 34	Business interaction to create Shipment context sources.....	53
Figure 35	Capabilities for the MainInterface Muse resource (context-source factory).....	53
Figure 36	WS Resource Properties Document for ShipmentTracker.....	55
Figure 37	Capabilities for ShipmentTracker Muse resource (context source).....	56
Figure 38	Capabilities for the ShipmentResourceGroup Muse resource.....	58
Figure 39	Capabilities for the ShipmentResourceGroupEntry Muse resource.....	58
Figure 40	Relationship between NotificationConsumer, context source (NotificationProducer), SubscriptionManager and Subscription WS-Resource.....	59
Figure 41	Capabilities for SubscriptionManager Muse resource .....	59
Figure 42	Interaction between the supplier web service, the WS-BPEL engine and the context- source system .....	61
Figure 43	Relationship between an SEI, SIB, Port Component and the WSDL .....	62
Figure 44	Request-message structure for purchaseItem operation .....	65
Figure 45	Relationship between message, part, element and parameters.....	66
Figure 46	Type hierarchy for the intermediate web service's response message.....	69
Figure 47	Relationship between ActiveBPEL *Def, *Impl objects and process instances .....	77
Figure 48	Simple WSN NotificationConsumer setup.....	86
Figure 49	Alternative WSN NotificationConsumer setup .....	87
Figure 50	Relationship between Axis services and Muse resources.....	89
Figure 51	The BPELConsumer Muse resource in the service/resource heirarchy .....	89
Figure 52	PurchaseItemResponse response message.....	93
Figure 53	Format of a WSN notification message .....	96
Figure 54	Example of NotificationMessage routing to Muse resources that implement the NotificationMessageListener interface.....	97
Figure 55	Scenario message flow required to create a Shipment context source .....	102

Figure 56 Java-EE web service response message .....	103
Figure 57 Response-message context parameter .....	103
Figure 58 Pseudocode for "blocking condition with timeout" scenario process logic .....	106
Figure 59 Scenario process logic using polling .....	107
Figure 60 Scenario process logic using context variables and standard activities only.....	108
Figure 61 Design of <conditionWithTimeout> extension activity.....	109
Figure 62 Scenario process logic using context variables and <conditionWithTimeout>.....	110
Figure 63 Design of the <contextHandler> activity .....	110
Figure 64 High-level view of context variable handoff .....	112
Figure 65 Use of modelReference attribute with ShipmentCondition context type .....	114

# Chapter 1

## INTRODUCTION

Business processes are integration programs that tie together various entities, both within and between organizations, to achieve business goals. They are ideally suited to implementation in a Service-Oriented Architecture (SOA), where functionality is provided by services with well-defined interfaces. The problems they target often require extensive I/O, data mediation, and knowledge of relevant environment state. As organizations demand greater efficiency from their operations, adaptability in the face of competitive forces, and nuanced responses to clients, the code required to perform these tasks only increases. Business processes are then no longer a rote repetition of steps: they have to find and account for changes in the business environment, choose a concrete entity from several alternatives and route task requests to them, *etc.* This poses challenges when it comes to evolving current business process languages like Web Services Business Process Execution Language (WS-BPEL) 2.0 [39].

WS-BPEL 2.0 (hereafter, WS-BPEL) is a business process language tailored to integrating web-services. It is designed to run in a business-process engine, a virtual machine that abstracts away much of the implementation minutiae of addressing, format conversion, persistence, etc. WS-BPEL allows programmers to think of a business process's constituent services only in terms of XML messages and WSDL [67] interfaces, not implementation technology or details. As WS-BPEL business processes are tasked with being more flexible, they will need knowledge of their environment state to customize their actions. However, WS-BPEL business processes are modeled as if they were the only thing operating in, and making changes to, the business environment. This mental model does not reflect the real world, where organizations and entities depend on state that is outside their control – state that is modified independent of, and concurrent with, the organization's activities. The WS-BPEL language itself is built around this assumption – that all relevant environment state is sourced and updated by the process itself, and at its discretion. Thus, if a WS-BPEL business process is interested in a piece of environment state, its designer has to pick the state source at design time, decide the state-update interval, define state update and fault compensation activities within the process, etc. The amount of work the programmer must do to achieve this may be tolerable when the process is small, the relevant state is static, or when only

one or two pieces of state have to be sourced and maintained, but it becomes a significant burden when multiple pieces of changing environment state are sourced from many entities, each with different interfaces, characteristics and Message-Exchange Patterns (MEP). Forcing the programmer to implement and maintain all this state-maintenance logic distracts from their core task: implementing the business goal.

Simplifying state maintenance in WS-BPEL is challenging, but it delivers a number of benefits: it becomes easier and faster to write adaptable business processes, since programmers no longer need to define state-maintenance activities for each piece of relevant state; it allows these processes to be written with greater clarity since state-maintenance activities will no longer be scattered throughout the process logic; and it raises the abstraction level in thinking about environment state – programmers can think of state and its effects on their business process instead of the minutiae of locating, updating and propagating changes in this state. There is a range of problems that would benefit from the simplification of state maintenance, including applications in health care, transit, and logistics.

State maintenance in WS-BPEL is hard because while WS-BPEL has excellent support for the pull-style interactions, where environment state is collected by the process on its timeline at its behest, it has extremely poor support for push-style interactions. This omission is glaring when one considers that most changes in a process' business environment occur independent of its execution, on an often unpredictable schedule. Thus, if business processes are to be more flexible, then WS-BPEL needs substantially better support for externally driven environment-state changes.

## 1.1 CONTRIBUTIONS

This thesis presents a solution for dealing with a large amount of changing environment state, or *context*. It centers on the idea of a *context variable* for WS-BPEL. In our approach we:

1. Describe context as it applies to business processes;
2. Model context sources for web services using WS-Resource-Framework (WSRF) [43] WS-Resources;

3. Use publish/subscribe (pub/sub), as provided by WS-Notification (WSN) [38], to propagate context changes from context sources to WS-BPEL context variables;
4. Demonstrate a standards-compliant way of extending web-service messages with references to context sources.

The solution outlined in this thesis offers significant advantages: it builds on established practices and well-understood message-exchange patterns, leverages widely used languages and frameworks, is standards compliant, and has a low barrier-to-entry for business-process programmers. It allows these programmers to write processes that rely on external state using substantially fewer activities and interface changes than either polling or onEvent message handlers. We also demonstrate the feasibility of the proposed solutions by constructing a prototype using a modified standards-compliant WS-BPEL engine, a Java Platform Enterprise Edition 5 (Java EE) web service, and a WSRF toolkit.

## 1.2 THESIS ORGANIZATION

This thesis begins by describing context and reviewing previous research into integrating context with WS-BPEL. It then covers Web Services Description Language (WSDL), WS-BPEL and other relevant specifications. Chapter 3 presents the motivating scenario used throughout the thesis, describes how the environment state in this scenario can be updated using constructs in WS-BPEL, the limitations of these approaches, and ends with a proposal based on context variables. In the following chapter the context-variable approach and its supporting architecture is described in detail. Chapter 5 then describes how the proposed solution was implemented in the prototype. Finally, the thesis closes with an evaluation of the work in Chapter 6, and outlines some enhancements to the core concept in Chapter 7.

## Chapter 2

### BACKGROUND AND RELATED WORK

Our context-variable solution draws on developments from both academia and industry. On the academic side, it builds on research into context, and context-aware and adaptable WS-BPEL processes. The solution architecture was designed to be standards compliant and, as a result, builds on various web-services standards, including WS-BPEL, WSRF, WSN, *etc.* In the following sections we first cover research that informed the design of context variables. We then outline the specifications used to design the solution architecture and relevant constructs within them.

#### 2.1 CONTEXT

As individuals, before we act we consider various ‘relevant factors’ and modify our behavior accordingly; this allows us to be flexible, and adapt to changing situations. These relevant factors form the context for an action and ensure that our responses are appropriate to the situation. In the computing community context awareness has been interpreted as applications “adapting their behavior based on information sensed from the physical or computational environment” [2]. With context awareness, systems would be able to incorporate user preferences, knowledge of device capabilities, and environmental factors in their decision-making, all in a manner transparent to users, thus enabling a new generation of highly responsive, yet minimally intrusive applications. Initial research into context often focused on the user’s location [1, 70]. Schilit expanded this to include other aspects as well, including available resources, the people around the user, and even the activity or social situation in which the user was involved [53]. Since then, extensive work has been done to incorporate context in application areas from ad-hoc communication [71] and telecom [26], to web services [24, 72] and mobile devices [16]. Despite this, there is yet to be a universally accepted definition of context since it varies by domain and application. The most cited one is Dey and Abowd’s [19], in which they state:

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.



This definition is very broad and provides developers little guidance as to what qualifies as context in their application. Recognizing this, many researchers have tried to define relevant context and ways of representing context in specific application areas [28, 30, 35]. However, none of these have been used widely.

There is also research on how to use context to adapt application behavior. This can be separated into two parts: acquiring and processing context from multiple sources, and using it in context-aware software. Research into the former has led to context middleware frameworks, the most well-known of which are Gaia [52] and the Context-Broker Architecture (CoBrA) [17]. These frameworks abstract away the variances in how context is sourced. There are many projects in which context is used to adapt application behavior. For example, Ranganathan and McFaddin present a system in which an individual's context is used to guide service selection and populate a Business Process Execution Language for Web Services (BPEL4WS) business process [51]. In their Semantic Web Services paper McIlraith et al. describe how context can be used to influence goal-based automated service composition [37]. While in the mobile devices space, Lemlouma and Layaida describe their Negotiation and Adaptation Core (NAC) prototype in which the capabilities of the target platforms influence service composition [32]. These are but a few examples of how researchers have leveraged context to improve the flexibility of their solutions.

## 2.2 PREVIOUS WORK

The work in this thesis builds on concepts in context-aware computing, event-driven systems, and distributed shared data in general. It is closest to research into context-aware and adaptable WS-BPEL processes.

Researchers have recognized that using context effectively is key to building more flexible, interoperable web services [36, 54]. Kiedl and Kemper present a context framework for web services in which context parameters like location and client device type are carried in Simple Object Access Protocol (SOAP) header blocks [28]. Separate context services then modify a web service's request and response messages based on the contents of those blocks. However, since their approach only deals with messages to/from a service, their project cannot use context to modify its internal operation. This is a major difference from our proposal.

Subsequently, modifying a service's internal operation has been explored. Many projects have used context to modify BPEL<sub>4</sub>WS [4] or WS-BPEL business processes, workflows and other service-composition languages. Context is often used to:

- Determine what tasks (and thus the services performing them) are used in the composition
- Choose between multiple service implementations for a given task
- Determine whether a service should participate in future compositions

In their research Vukovic and Robinson use context information – which they define as a user's location, network capabilities, device constraints, *etc.* – to determine which constituent tasks should be performed in a composition to meet a given goal [61]. This context information is used, along with other parameters, as input into a Simple Hierarchical Ordered Planner 2 (SHOP<sub>2</sub>) planner in order to generate a BPEL<sub>4</sub>WS business process. Using context up-front to generate an executable process is also pursued by Ranganathan et al., who describe an architecture for coordinating web services in a pervasive computing environment [51]. There, BPEL<sub>4</sub>WS workflow templates are specified in advance, and context is automatically gathered from the environment to determine which service implementations should be chosen for the placeholders in the template. In both these projects context is only used at the composition stage; once the executable business process is created, changes in context no longer affect process execution. Other solutions have gone further, using Quality of Service (QoS) or other monitored information to modify a process information at runtime [9, 27]. In their system, Karastoyanova et al. use a BPEL<sub>4</sub>WS language extension and architecture support to create a “find-and-bind” mechanism through which service instances can be substituted at runtime in response to infrastructure failures, a process' execution domain, *etc.* [27]. Baresi and Guinea use context for a different purpose. In their system, independent monitoring rules are combined with a proxy-based service invocation approach to modify process instances [9]. These solutions address flexibility by treating it as a service selection and composition problem. While promising, they do not use context within control structures or structured activities to change the process' execution path, only the specific services invoked in that path. In fact, a common theme running through these projects is that to modify the process you modify the services invoked – by changing constituents the overall composition can be made to run faster, slower, account for user preferences, *etc.* Again, this approach differs from ours. We

do not use context to drive service selection, but instead make it available to WS-BPEL constructs, thus allowing it to influence process control flow.

More complex workflow adaptation has been explored in projects like eFlow [13]. There, workflows are viewed as a graph consisting of service, event and decision nodes, with user preferences and other context information used to drive composition both at the preliminary stage and at runtime. What differentiates eFlow from the above projects is its notion of event nodes – points at which a composition may send and receive events. These nodes are subsequently explored by Casati and Shan [14]. In that work they acknowledge that workflows depend on data that changes independent of its own execution. They then:

1. Propose an event model, describing the different kinds of events the eFlow architecture generates and responds to
2. Decompose event nodes into publish and request process nodes – points at which a composition halts, and then generates or waits for an event

In eFlow, the developer has to explicitly define points in the composition (request nodes) where execution halts and the composition waits for events. Variables can only be updated at these nodes, not whenever an event occurs. Moreover, programmers must define a capturing rule within the request node to copy an event's value into a composition's variable. Finally, eFlow is based on a custom language and engine, not on WS-\* standards, communication protocols and composition languages like WS-BPEL. Our context-variable proposal, while sharing many of the same concepts and concerns as eFlow, is a much lighter-weight approach that builds on existing standards, frameworks and toolkits.

The idea of using pub/sub during workflow execution has also been explored extensively. Projects like LEAD [50] explore how event subscription and notification can be used to guide the execution of grid-based workflows. It uses WS-Eventing [11] and the idea of a centralized event channel to which a workflow instance's invoked services can publish status and other metadata. This workflow engine then uses this information to guide an instance's execution. In [29], the authors present the idea of workflow engine having a separate notification channel to its invoked services. This channel would be used by the services to inform the engine of their processing state. In both of these projects events and notifications are used to communicate a service's ex-

ecution state, not transfer data between executing services or workflows. Recent work has taken the use of pub/sub even further: in NINOS [33], a BPEL4WS process is executed in a distributed fashion on the PADRES [21] distributed content-based pub/sub routing infrastructure. There, a process is devolved into its constituent activities, each of which is executed by a light-weight activity agent. These activity agents are all publish/subscribe clients that use the PADRES infrastructure. NINOS is primarily concerned with showing how distributed control flow can be achieved using pub/sub primitives – subscriptions and notifications. It also briefly describes how these primitives can be used for variables as well: activities use PADRES primitives to subscribe to, and notify others of, changes to variables in which they are interested. Though NINOS uses pub/sub for process variables, the semantics governing these variables remain unchanged: a variable’s value is only modified by in-process activities or explicit request/response interactions with external web services. This is in contrast to our work, in which we recognize the limitations of standard WS-BPEL variables in reflecting externally-driven state, and show how our pub/sub-based context-variable design can simplify this task.

This idea of shared data or resources has a long history. In fact, context can be thought of as shared data that a WS-BPEL process can access, but not modify. There are a number of research projects and standards that explore this concept in the web services arena. One of these is WS-Context [41], an OASIS specification in which the concept of an *activity* is used to group interactions between a set of web services. An activity is analogous to the idea of a shared session. The WS-Context specification identifies each activity instance using a context. Note that this notion of context is not the same as ours: it is best thought of as a shared session. Since all the services participating in the activity use a context identifier in their messages’ header block, they can access shared resources implicitly. They can also use a *Context Manager* to retrieve and set data associated with the activity instance. Unlike our approach, WS-Context’s view of shared data is still a request/reply-oriented one. Shared data must be explicitly set by the service generating changes, and explicitly retrieved by services using them. As a result, this specification does not address the concerns highlighted in this thesis. Researchers have also explored how web services can leverage a shared session, or space, in their execution. In the WSSecSpace project [34], services in a composition execute not as a result of direct service invocations, but by watching a shared space for data on which they are dependent. This project extends work in Linda [22] and adapts it to the

```

<types>
  <schema targetNamespace="http://example1.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="GetPrice">
      <complexType>
        <all>
          <element name="ItemCode" type="string"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>

<message name="GetPriceRequest">
  <part name="wrapper" element="pr:GetPrice"/>
</message>

```

**Figure 1 WSDL <message> definition**

web services arena with additional support for security. In the Active Shared Space project [49], a shared space is used not just to facilitate execution, but monitoring as well. Both of these projects focus on the data that is generated, or changed, as a result of service execution, and use it to initiate behavior in other, dependent services. Unlike our approach they do not consider the question of how to change a service implementation – e.g., a WS-BPEL process – to transparently pick up these changes.

Finally, external variables as defined by Chandy et al. [15] come close to our notion of context variables. They are, however, limited when compared to context variables since each one has to be linked to a different real-world source. Further, it is not described how these variables are populated at runtime. External variables have been added to WS-BPEL by Baresi et al. [8], who use them in monitoring rules outside a WS-BPEL process. In their design the variables are populated using ad-hoc, RPC-style WSDL operations, and their values are not used within the process. This differs from our approach in that context variables are used within the business logic, we describe the design of the standards-based entities that provide context, and we use WSN pub/sub to deliver their values.

## 2.3 WSDL

Web Services Description Language (WSDL) is the de facto XML format for defining web service interfaces. A WSDL interface includes a set of data types, and abstract definitions of messages and operations. It also defines how to bind these abstract messages and operations to specific message

```
<operation name="GetPrice">
  <input message="tns:GetPriceRequest"/>
  <output message="tns:GetPriceResponse"/>
</operation>
```

**Figure 2 WSDL operation**

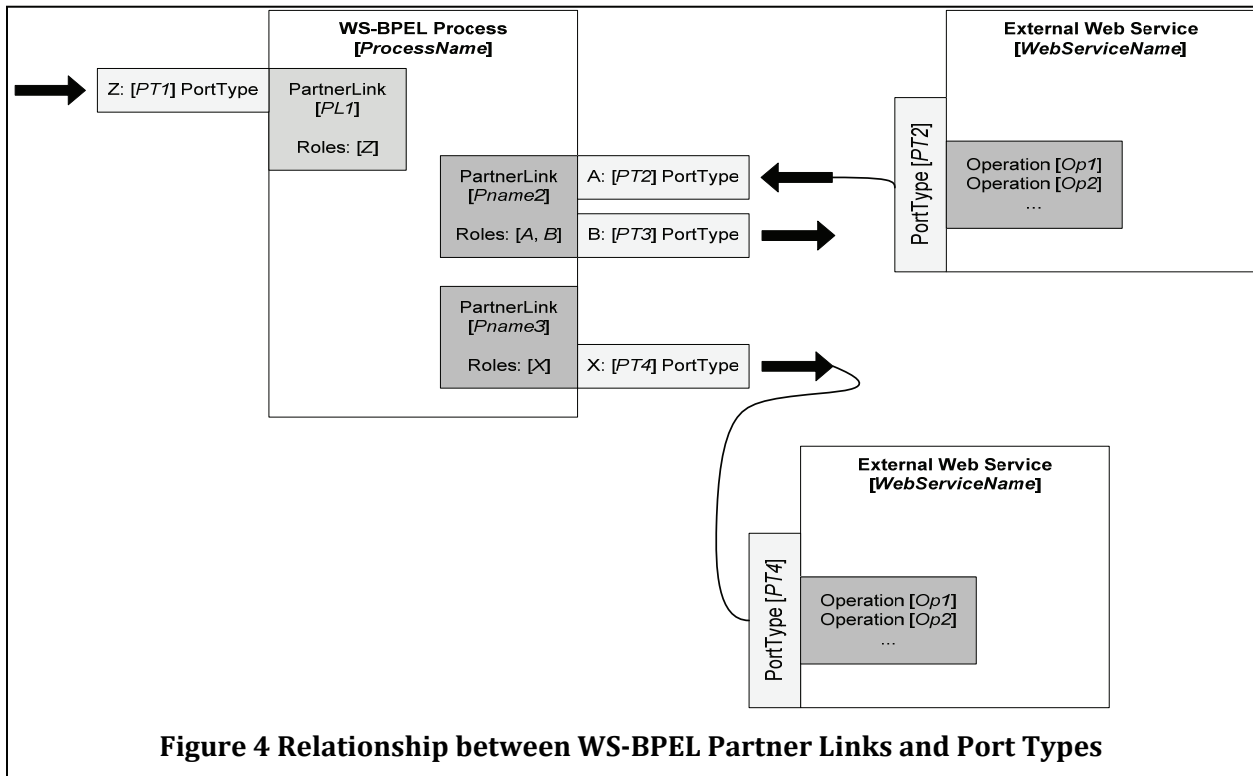
```
<portType name="RetailerPortType">
  <operation name="GetPrice">
    ...
  </operation>
  <operation name="GeQuantity">
    ...
  </operation>
</portType>
```

**Figure 3 WSDL Port Type**

formats and protocols. For the purposes of this thesis we shall cover only a few key features of WSDL. For more details, the WSDL specification from the W3C should be consulted [67]. The WSDL `<types>` element contains a list of XML-Schema type definitions that are used within WSDL messages. Other, pre-existing schemas can also be referenced within the `<types>` element. A WSDL `<message>` is an abstract definition of the data that is transferred in a WSDL operation. Each `<message>` consists of one or more parts, each referencing an XML Schema type or element. In this thesis messages with only a single part are considered. An example of a `<message>` definition is shown in Figure 1.

A WSDL interface can contain an arbitrary number of messages. Messages are transmitted to, and from, a web service by WSDL operations. The WSDL specification defines four operation styles, of which one-way and request-response are the most widely used and supported. Each operation defines an input and/or output message, as well as an error message that may be returned as a result of the operation. The input message is defined using the `<input>` element, the output message using the `<output>` element, and the error message using the `<fault>` element. A sample of a WSDL operation is shown in Figure 2. Since WSDL operations are defined as a set of message exchanges, they are also referred to as Message-Exchange Patterns or MEPs.

A set of WSDL operations and messages are bundled into a WSDL Port Type. Port types define an abstract endpoint, the operations offered by this endpoint, and the messages used by these operations. Each port type can be thought of as a named interface to the web service. An example of a port type is shown in Figure 3.



## 2.4 WS-BPEL

WS-BPEL is an XML-based programming language for business processes that describes how these processes are specified and the model governing their operation. A WS-BPEL business process orchestrates interactions with web service partners to achieve a business goal. Each partner has a WSDL interface with at least one port type; a WS-BPEL process also exposes a WSDL interface with at least one port type, thus allowing itself to be included in even larger composi-

```

<partnerLinks>
  <partnerLink name="Store"
    partnerLinkType="emp:eCommerceStoreType"
    partnerRole="eCommerceStore" />

  <partnerLink name="Notification"
    partnerLinkType="not:NotificationServiceType"
    partnerRole="NotificationServiceProvider" />
</partnerLinks>

In referenced WSDL:
<plnk:partnerLinkType name="eCommerceStoreType">
  <plnk:role name="eCommerceStore" portType="stor:RetailerPortType" />
</plnk:partnerLinkType>

```

**Figure 5 Sample WS-BPEL Partner Link and Partner Link Type**

tions. The relationship between a partner service and a WS-BPEL business process is codified in a mandatory Partner Link; each partner link has up to two roles, and declares which port type each role requires and that the communicating services should satisfy, for the interaction to occur. The relationship between a WS-BPEL process, external web services, partner links and port types is shown in Figure 4. A sample of a partner link is shown in Figure 5. Neither the port type nor the partner link specify message formats or transport-specific information about the web-service partners. This concrete binding information is set at design-time in an implementation-specific manner, or at runtime through the use of endpoint references.

The WS-BPEL specification fully outlines the grammar of the WS-BPEL language. All business processes are written in process-definition files, or process definitions. Each process definition is loaded and run on a WS-BPEL engine, a virtual machine that abstracts away much of the minutiae in communicating with web services and executing long-running business processes. Process logic is performed through the use of *activities*, which come in two forms: *basic* and *structured*. Activities are represented in process definitions as XML elements, with their properties described as attributes or child elements. Certain activities may contain child activities, and all activities are themselves children of `<process>` element, which is the root of a WS-BPEL process.

Structured activities in WS-BPEL describe control-flow logic and have analogs in common programming languages. They include constructs like `<if>`, `<while>`, and `<repeatUntil>` (similar to `do . . . while`). On the other hand, basic activities describe the “elemental steps of the process behavior” and include `<invoke>`, the `<receive>/<reply>` pair, `<assign>` and `<wait>`. The `<invoke>` activity is used to call an operation in an external web-service; `<receive>` allows an operation offered by a WS-BPEL business process to be invoked by an external entity, and `<reply>` allows the process to respond with the results of an operation execution. The `<wait>` activity enables programmers to pause process execution either for a certain *duration*, or until a *deadline* is reached. Finally, `<assign>` is used to copy values to WS-BPEL constructs. A sample WS-BPEL process with a few activities is shown in Figure 6.

In addition to the activities above, the language also includes scoping constructs (`<scope>`), and supports parallel execution (`<flow>`). WS-BPEL differs from popular programming platforms like Java and .NET in that it does not include a suite of libraries and APIs. In fact, it does not even



```

<bpel:process . . . name="purchase" ns4:isStateful="yes"
  suppressJoinFailure="yes"
  targetNamespace="http://ece.uwaterloo.ca/aag/internalpurchase/purchase">
  <bpel:extensions>
    <bpel:extension mustUnderstand="yes"
      namespace="http://ece.uwaterloo.ca/aag/statefulbpel"/>
  </bpel:extensions>

  <bpel:partnerLinks>
    <bpel:partnerLink name="eCommerceStoreLink"
      partnerLinkType="ns1:eCommerceStoreLT" partnerRole="eCommerceStore"/>
  </bpel:partnerLinks>

  <bpel:flow>
    <bpel:links>
      <bpel:link name="L4"/>
      . . .
    </bpel:links>

    <bpel:invoke inputVariable="purchaseItemRequest"
      operation="PurchaseItem" outputVariable="purchaseItemResponse"
      partnerLink="eCommerceStoreLink" portType="ns2:Purchase">
      <bpel:targets>
        <bpel:target linkName="L3"/></bpel:targets>
        <bpel:sources>
          <bpel:source linkName="L4"/>
        </bpel:sources>
      </bpel:invoke>

      . . .

    <bpel:receive createInstance="yes" operation="issuePurchaseOrder"
      partnerLink="purchaseProcessLink" portType="ns3:InternalPurchase"
      variable="issuePurchaseOrderRequest">
      <bpel:sources>
        <bpel:source linkName="L2"/>
      </bpel:sources>
    </bpel:receive>

    . . .

  </bpel:flow>
</bpel:process>

```

**Figure 6 Sample WS-BPEL business process**

have the concept of libraries; any functionality not offered by the language must be provided by external web-services, expression languages, or extensions to WS-BPEL itself.

WS-BPEL variables represent the state of a business process. They come in three types: WSDL message type, XML-Schema type, and XML-Schema element. WSDL message-type variables reference type definitions in either the process' or its partners' WSDL interfaces. It is often used to store the result of a web service invocation or send an invocation message. The other variable type

```

<wsa:EndpointReference>
  <wsa:Address>xs:anyURI</wsa:Address>
  <wsa:ReferenceProperties>... </wsa:ReferenceProperties> ?
  <wsa:ReferenceParameters>... </wsa:ReferenceParameters> ?
  <wsa:PortType>xs:QName</wsa:PortType> ?
  <wsa:ServiceName PortName="xs:NCName"?>xs:QName</wsa:ServiceName> ?
  <wsp:Policy> ... </wsp:Policy>*
</wsa:EndpointReference>

```

**Figure 7 XML representation of WSA EPR**

that needs explanation is the XML-Schema-element variable type. If one imagines an XML-Schema type definition as a structure then variables of type XML-Schema element refer to a component in that structure.

The WS-BPEL language was designed to be extensible. The standard allows namespace-qualified attributes to be added to any WS-BPEL element and also allows elements from other namespaces to appear within existing WS-BPEL elements. The extensions a process uses are listed within its `<extensions>` element, with each extension designated as optional or mandatory; this list of extensions allows the WS-BPEL processor to decide whether to load the process definition at all, process extended elements, or ignore them altogether.

WS-BPEL is central to the research and prototype described in this thesis. For more details on the language, the WS-BPEL 2.0 specification [39] and the WS-BPEL 2.0 Primer [40] should be consulted.

## 2.5 WS-ADDRESSING

The WS-Addressing (WSA) specification [12] defines a transport-neutral way of addressing web services and web-service messages. This thesis is primarily concerned with the WSA Endpoint Reference (EPR) construct and how an EPR is translated into SOAP [64] header elements. A WSA EPR has the XML representation shown in Figure 7. The `wsa:EndpointReference/wsa:Address` element is the only mandatory element, and is the transport or logical Universal Resource Identifier (URI) that identifies the endpoint. Also of interest is the `wsa:EndpointReference/wsa:ReferenceParameters` element. Reference parameters are optional and each EPR can contain as many of them as necessary. They are issued by the entity that generates EPRs and are associated with the endpoint represented by an EPR. Reference parameters are meant to be opaque to the consuming entity – there is no standardized way to determine

```

<wsa:EndpointReference xmlns:wsa="..." xmlns:retailer="...">
  <wsa:Address>http://www.example.com/retailer</wsa:Address>
  <wsa:ReferenceParameters>
    <retailer:CustomerOrderCart>Cart-7668</retailer:CustomerOrderCart>
  </wsa:ReferenceParameters>
</wsa:EndpointReference>

```

**Figure 8 WSA EPR with single Reference Parameter**

```

<wsa:MessageID>xs:anyURI</wsa:MessageID>
<wsa:RelatesTo RelationshipType="..."?>xs:anyURI</wsa:RelatesTo>
<wsa:To>xs:anyURI</wsa:To>
<wsa:Action>xs:anyURI</wsa:Action>
<wsa:From>endpoint-reference</wsa:From>
<wsa:ReplyTo>endpoint-reference</wsa:ReplyTo>
<wsa:FaultTo>endpoint-reference</wsa:FaultTo>

```

**Figure 9 WSA SOAP-Binding message-information header blocks**

the organization of the underlying system using these reference parameters. Moreover, the specification does not codify the relationship between reference parameters and endpoints. For example, an endpoint may be addressable through multiple EPRs each with different sets of reference parameters, *etc.* This means that the association between reference parameters and endpoint is implementation specific. An example of an EPR with a single reference parameter called `<retailer:CustomerOrderCart>` is shown in Figure 8.

When a message needs to be sent to an EPR-addressed endpoint, the properties of the EPR must be bound to the message format. In web services, SOAP is the de facto message format, and the WS-Addressing Technical Committee (TC) has defined a SOAP Binding for WSA [66]. This binding defines the message-information header blocks in Figure 9. These header blocks appear as SOAP header elements. Of these header blocks only `wsa:To` and `wsa:Action` are mandatory. The value of `wsa:To` is the same as the value of the `wsa:EndpointReference/wsa:Address` element in the XML representation of an EPR. The `wsa:Action` element contains a URI identifying the semantics associated with the message. Section 3.3 of the WSA specification outlines how WSA-Action URIs are associated with WSDL operations [12]. The most common approach is to associate a WSA Action URI with the input and output message for each WSDL operation as shown below:

```

<portType name="RetailerPortType">
  <operation name="GetPrice">
    <input message="tns:GetPriceRequest" wsa:Action="http://retailer/GetPrice"/>
    <output message="tns:GetPriceResponse" wsa:Action="http://retailer/Price"/>
  </operation>
</portType>

```

```

<!-- EPR -->
<wsa:EndpointReference xmlns:wsa="..." xmlns:retailer="...">
  <wsa:Address>http://www.example.com/retailer</wsa:Address>

  <!-- Reference parameter -->
  <wsa:ReferenceParameters>
    <retailer:CustomerOrderCart>Cart-7668</retailer:CustomerOrderCart>
  </wsa:ReferenceParameters>

</wsa:EndpointReference>

<!-- SOAP Header Equivalent -->
<wsa:MessageID>UUID:gg</wsa:MessageID>
<wsa:To> http://www.example.com/retailer</wsa:To>
<wsa:Action> http://retailer/GetPrice </wsa:Action>

<!-- SOAP translation of reference parameter -->
<retailer:CustomerOrderCart wsa:IsReferenceParameter="true">
  Cart-7668
</retailer:CustomerOrderCart>

```

**Figure 10 Translation of EPR reference parameters to SOAP-header elements**

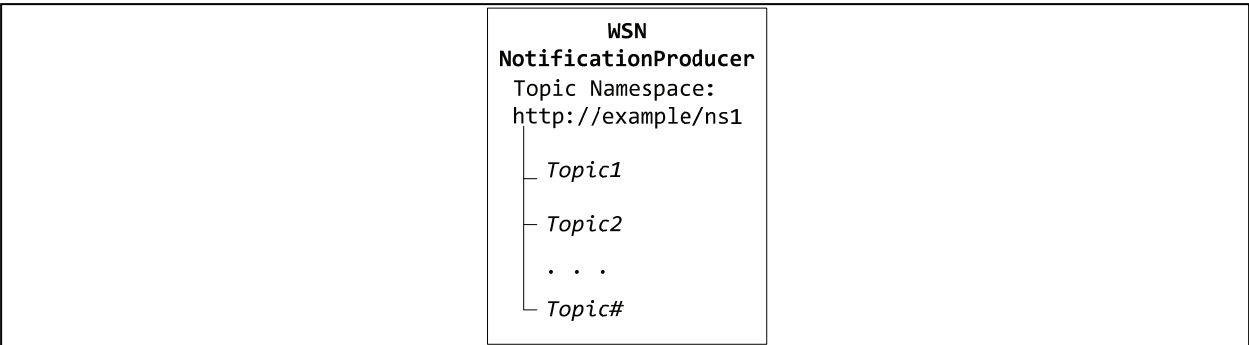
When it comes to reference parameters, each parameter element in the EPR is translated into a separate SOAP header element. An example of this translation is shown in Figure 10.

## 2.6 WS-NOTIFICATION

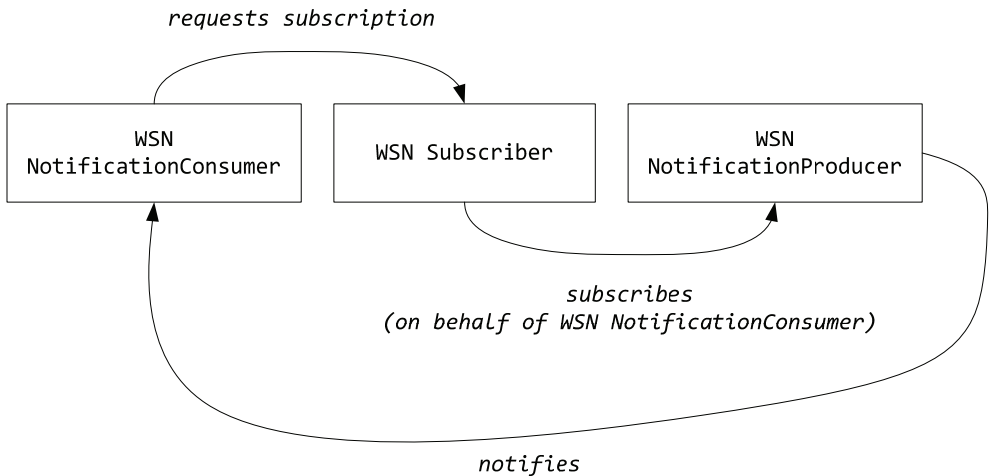
WS-Notification (WSN) is a set of standards and white papers describing a web-services approach to topic-based publish/subscribe (pub/sub). It details the entities participating in these pub/sub interactions, a set of standard MEPs for providing and consuming notifications, WSDL interfaces, the XML topic model, etc. For more information on WS-Notification, the Publish-Subscribe Notification for Web services white paper [23], the WS-BaseNotification specification [38] and the WS-Topics specification [48] should be consulted.

This thesis uses only the basic architectural constructs and MEPs of WS-N. The key entities that we are concerned with are the *NotificationProducer* (notification producer, or producer), *NotificationConsumer* (notification consumer, or consumer) and *Subscriber* (subscriber).

A NotificationProducer is a web service that implements the standardized NotificationProducer WSDL interface and provides a set of WS-Topic *Topics*. A Topic is the basic unit for which a NotificationMessage is generated. An external entity subscribes to a Topic offered by a NotificationProducer to receive a NotificationMessage whenever a situation occurs, and message is



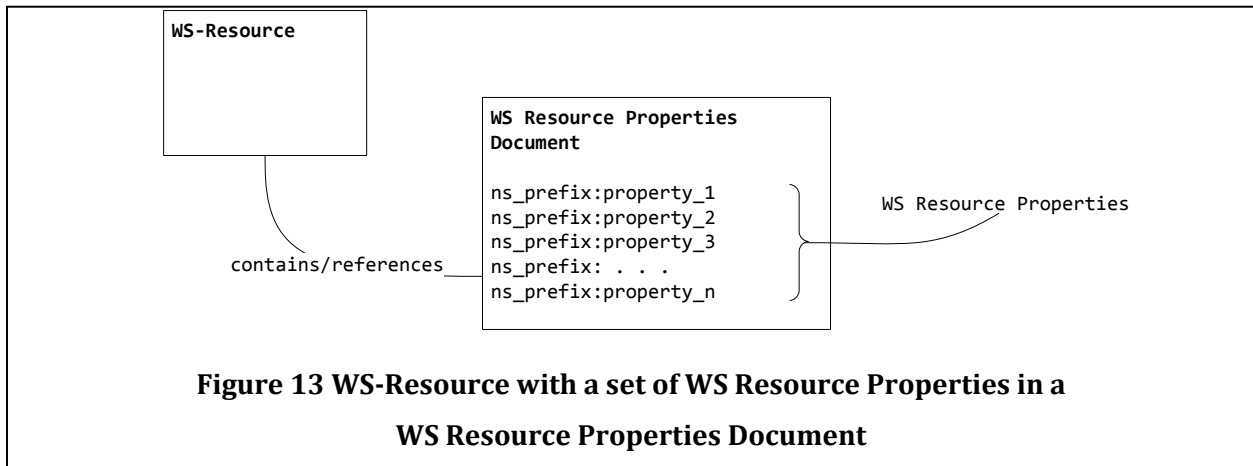
**Figure 11 WSN NotificationProducer offering a set of topics in a TopicNamespace**



**Figure 12 Interaction between NotificationProducer, NotificationConsumer and Subscriber**

generated. A diagram demonstrating the abstract design of a WSN NotificationProducer is shown in Figure 11.

The eventual destination of a `NotificationMessage` is a `NotificationConsumer` web service. Every `NotificationConsumer` in this thesis implements the standard `NotificationConsumer` WSDL interface, which exposes a single `Notify` operation. A `NotificationConsumer` uses a `Subscriber` to subscribe to a `Topic` offered by a `NotificationProducer`. The interaction between `NotificationProducer`, `NotificationConsumer`, and `Subscriber` is shown in Figure 12. The WSN specification states that each subscription (the result of a `Subscribe` operation a `Subscriber` performs on behalf of a `NotificationConsumer` on a `NotificationProducer`) results in a subscription resource. This resource represents the relationship between the producer, consumer and topic, as well as other information. The `NotificationProducer` maintains a list of subscription resources.



## 2.7 WS-RESOURCE FRAMEWORK

WS-Resource Framework (WSRF) is a set of specifications standardizing web services' interactions with state. It is based around the idea of a *stateful resource* as a collection of *state components*. Stateful resources are how WSRF models state; each resource has a set of state components, of which a subset is exposed through web services. In WSRF, when a web service is associated with a stateful resource the resulting composition is called a *WS-Resource*. In this setup the resource's state components are exposed as *resource properties* in a *WS-Resource properties document*, a set of XML types and values corresponding to individual properties. A reference to this WS-Resource properties document is embedded as an attribute in the WS-Resource's port type. Figure 13 shows how a WS-Resource is composed. Programmers can place additional constraints on these resource properties by referencing a WS-Resource Metadata Descriptor (WSRMD) [45] in the same port type as the WS Resource Properties Document.

WS-Resource instances are identified using WSA EPRs. This allows messages to those instances to be differentiated using WSA-message-information header blocks in the message header, as opposed to explicit instance identifiers in a WSDL operation's input-message parameters. By using EPRs requesters can access a specific WS-Resource instance and through it, its underlying stateful resource.

WSRF defines a set of standard WSDL MEPs like `GetResourceProperty` and `SetResourceProperty` for interacting with WS-Resources. Each WS-Resource must provide the mandatory operations defined in the spec; it can, however, also offer its own, custom operations. By defining

a set of standardized operations for WS-Resources, the WSRF specification makes it easier for developers to interact with different types of WS-Resources, and for tooling to automate these interactions.

WS-Resources also do not have to exist in advance: they can be created by a WSRF *WS-Resource Factory* – a web service that creates WS-Resources. The interface and implementation of a WS-Resource Factory is not standardized. WSRF also defines a number of other constructs for organizing WS-Resources, of which this thesis only uses *ServiceGroup* and *ServiceGroupEntry*. A *ServiceGroup* allows administrators to automatically group WS-Resources using custom, pre-defined criteria, while a *ServiceGroupEntry* represents a WS-Resource's inclusion in a *ServiceGroup*.

The WSRF specification also describes how WSN can be integrated with a WS-Resource to provide notifications on changes in the values of its resource properties. In a WS-Resource/WSN configuration each WS-Resource acts as a notification producer. The WS-Resource exposes each resource property in its WS Resource Properties Document as a WS-Topic subscription topic of the same name. A WSN *NotificationConsumer* can then use a WSN *Subscriber* to subscribe to the WS-Resource instance for change notifications from one, or many, resource properties.

For more details on the WSRF documents used in this thesis the Web services Resource Framework Primer v1.2 whitepaper [44] and the WSRF WS-Resource v1.2 [42], WSRF WS-Resource Properties v1.2 [46], WSRF WS-Service Group v1.2 [47] and WSRF WS-Resource Metadata Descriptor v 1.0 [45] specifications should be consulted.

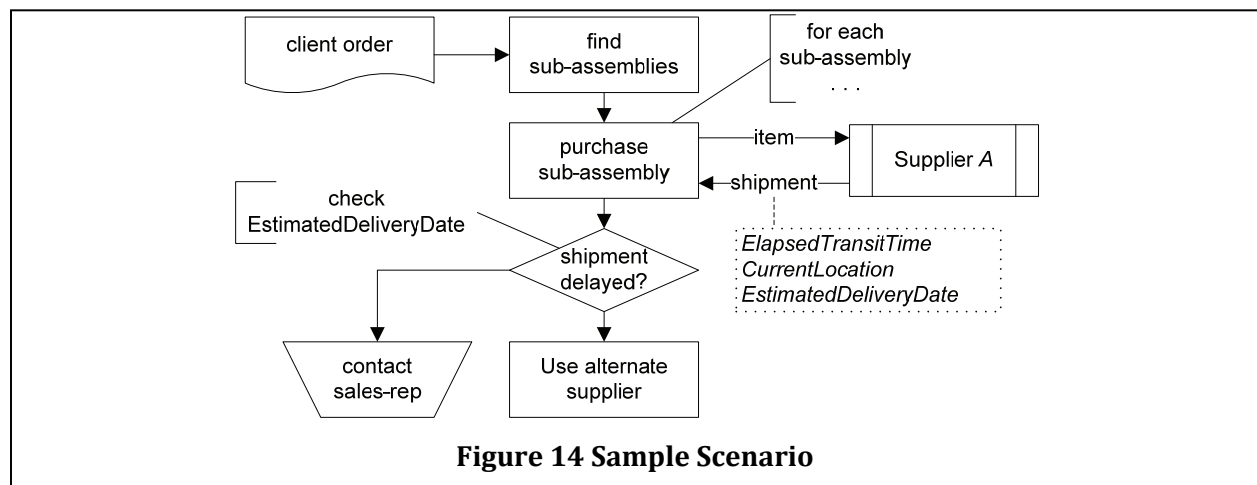
## Chapter 3

### CONTEXT IN WS-BPEL

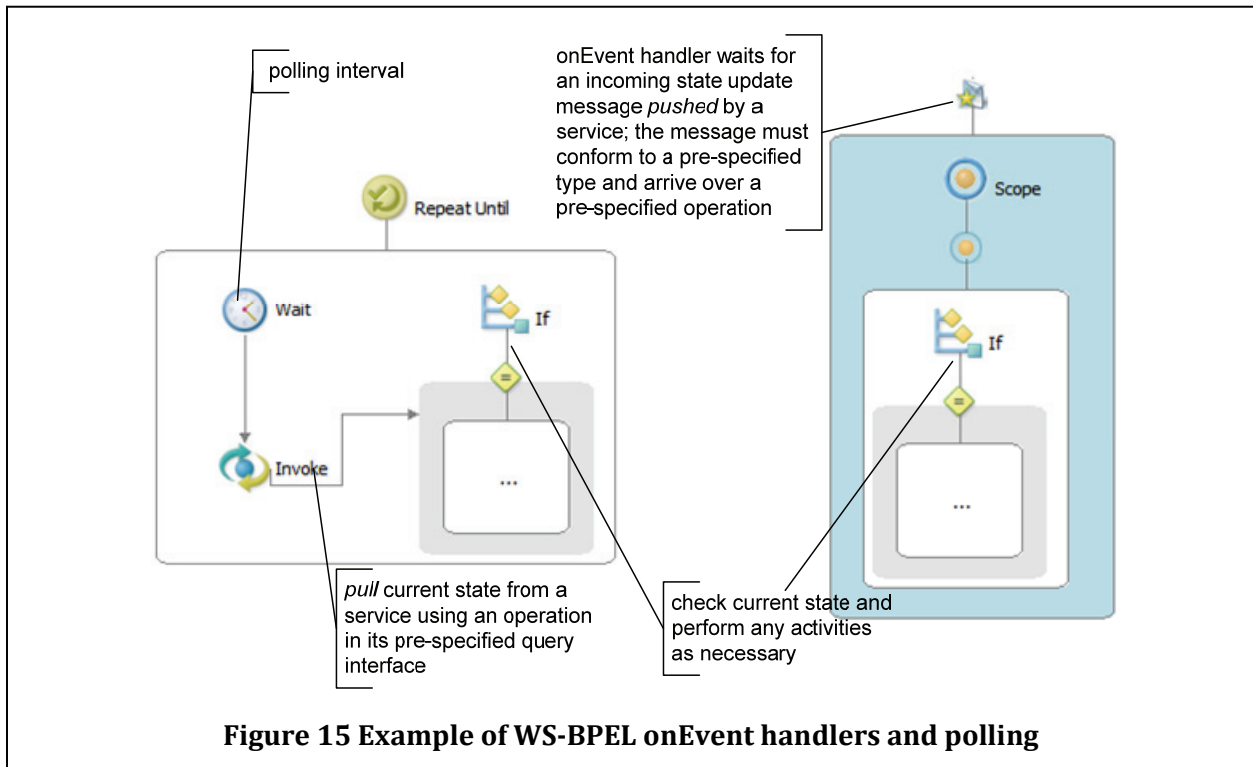
WS-BPEL can be used to create business processes in application areas from logistics to administration, healthcare, and transit. In all these areas the flexibility and responsiveness of business processes can be increased if context awareness is incorporated. In this chapter we present a motivating scenario and use it to illustrate what context is in a business process. We then examine two methods for accessing and manipulating context in a WS-BPEL process – polling and onEvent message handlers – and describe their design and limitations. Finally, we outline our context-variable solution and show how it incorporates the advantages of the existing WS-BPEL approaches to interacting with context, while discarding the need for extensive in-process logic to manage these interactions.

#### 3.1 MOTIVATING SCENARIO

Our motivating scenario is based on a common business application: a shipping situation in which three entities interact to deliver goods to a customer. Consider a business that offers its customers strict product-delivery guarantees. Each product uses multiple sub-assemblies for which there are a number of approved suppliers. As the company receives new purchase requests it orders the constituent sub-assemblies from its suppliers, which are then shipped to it by a third-party delivery service.







Not every shipment will arrive on time. Instead of finding this out when the sub-assembly fails to arrive at the loading dock on the expected delivery date, the company would like advance warning so alternate arrangements can be made. To do this they track the shipment's estimated delivery date and, if it exceeds a certain threshold, perform a compensating activity (reschedule assembly, purchase from an alternate supplier, *etc.*) A high-level diagram of this process is shown in Figure 14. Note that this process relies on `EstimatedDeliveryDate`, a piece of external state whose value it cannot affect, and which changes unpredictably, but which still influences its execution.

We now demonstrate how the value of `EstimatedDeliveryDate` is transferred to a WS-BPEL process using two existing methods: polling and onEvent message handlers. First, we describe the limitations of these approaches. We then outline a solution architecture that enables programmers to access this externally driven environment state within business processes without having to write extensive in-process logic.

## 3.2 POLLING

Polling is a conceptually simple and frequently used way of maintaining environment state. Based on the pull paradigm, it assumes that state sources provide an interface with a state query operation that can be invoked in a control loop to retrieve a snapshot of the current state. Polling is attractive for a number of reasons: it presents a simple mental model because it occurs synchronously with the main process logic, it is easy to code, and most state sources implicitly support this paradigm by offering a state query interface.

Despite its widespread use, polling is not ideal. First, the choice of polling interval is not obvious: it changes based on state, and its rate of change may vary over its lifetime; what may be an appropriate polling interval at one point may be too fast or too slow at another. This presents a major issue for WS-BPEL programmers: the simpler their approach to dealing with a state's dynamic behavior, the less responsive their system is, but the more factors they consider, the more complex the logic for dealing with it – logic that has nothing to do with the business problem they are trying to solve. Also, as state updates occur within the main process, loop and invoke activities (i.e., state-maintenance code) are scattered throughout the code, making the business logic harder to identify and the process harder to maintain. The situation worsens when the process uses multiple pieces of state sourced from disparate entities requiring different polling intervals – what was manageable before becomes untenable now. Tight coupling between process and state source is a second problem. Since the process calls a specific operation in a specific port type at a specific endpoint, any interface changes will break it. Finally, polling is expensive for the WS-BPEL engine and queried service. Engines have to continually wake processes to perform queries, even though most will report no actionable change. Queried services can also be heavily loaded if multiple process instances request updates. While hardware addresses this, it comes at a cost – for the equipment, its use, and its administration. A high-level view of polling using WS-BPEL activities is shown in Figure 15.

## 3.3 MESSAGE HANDLING

WS-BPEL offers a push-based alternative that enables an event-driven approach to writing business processes: onEvent message handlers. Every scope can have its own set of message handlers – one per message type – each defined separately from the main process activities. These handlers

can be triggered by a message at any point in their associated scope's lifetime, and run in parallel with their associated scope's activities. Although an event-driven design offers many advantages, its support in the form of WS-BPEL onEvent handlers has many issues, with the first being coupling. Ideally, message handlers would be loosely coupled, i.e. defined only in terms of the incoming message type. This is not the case in WS-BPEL, where programmers have to specify the operation and the partner link for each onEvent message handler. This requires modifying the process' WSDL and adding a new operation, its input and/or output messages, and the XML Schema types for the message contents. Furthermore, this operation is the "operation that is invoked by the partner in order to cause the event" (Section 12.7.1, WS-BPEL specification [39]). The resulting coupling is extremely problematic: state sources now need advance knowledge of the operations and port types of the business processes they will be delivering state-update messages to before they can send notifications. As a result, if a state-update source has to notify a variety of WS-BPEL processes – each with a different onEvent message handler – its programmers have to implement custom notification code for each one. This also means that any change to the process' interface without a corresponding change to the state source breaks state delivery.

Moreover, process programmers are still forced to define extra non-business logic when using onEvent handlers. They need partner links with roles for every state-update port type used – yet another coupling point. In addition, though the *wait*, *invoke*, and *repeatUntil* activities required for polling are no longer necessary, *assign* activities are still required to copy state updates from the handler to the main process. Maintainability is yet another concern. While onEvent handlers allow process programmers to separate concerns, this separation, and their concurrent execution, makes it harder to understand the overall process flow. Furthermore, the rules governing variable access in WS-BPEL onEvent handlers are insufficiently specified and can vary by implementation. In Section 12.7.1 [39] the WS-BPEL specification outlines the rules on variable *referencing*, but not modification; and even if modification of variables outside an onEvent handler's immediate scope were allowed, it is unclear what the semantics of concurrent access and update are. All these conceptual and practical issues reduce the attractiveness of onEvent handlers. A high-level view of a WS-BPEL onEvent message handler is shown in Figure 15.

### 3.4 PROPOSED APPROACH

Existing methods clearly have significant drawbacks when dealing with externally-driven environment state: polling is poorly suited to dynamic-state behavior, intersperses state-maintenance activities with the business logic, and scales poorly; onEvent handlers introduce excessive coupling, are non-trivial to understand, and have insufficiently specified semantics. Both complicate process design by making programmers concentrate on the state-maintenance mechanisms instead of the business logic. These limitations spring from assumptions around which the WS-BPEL language itself.

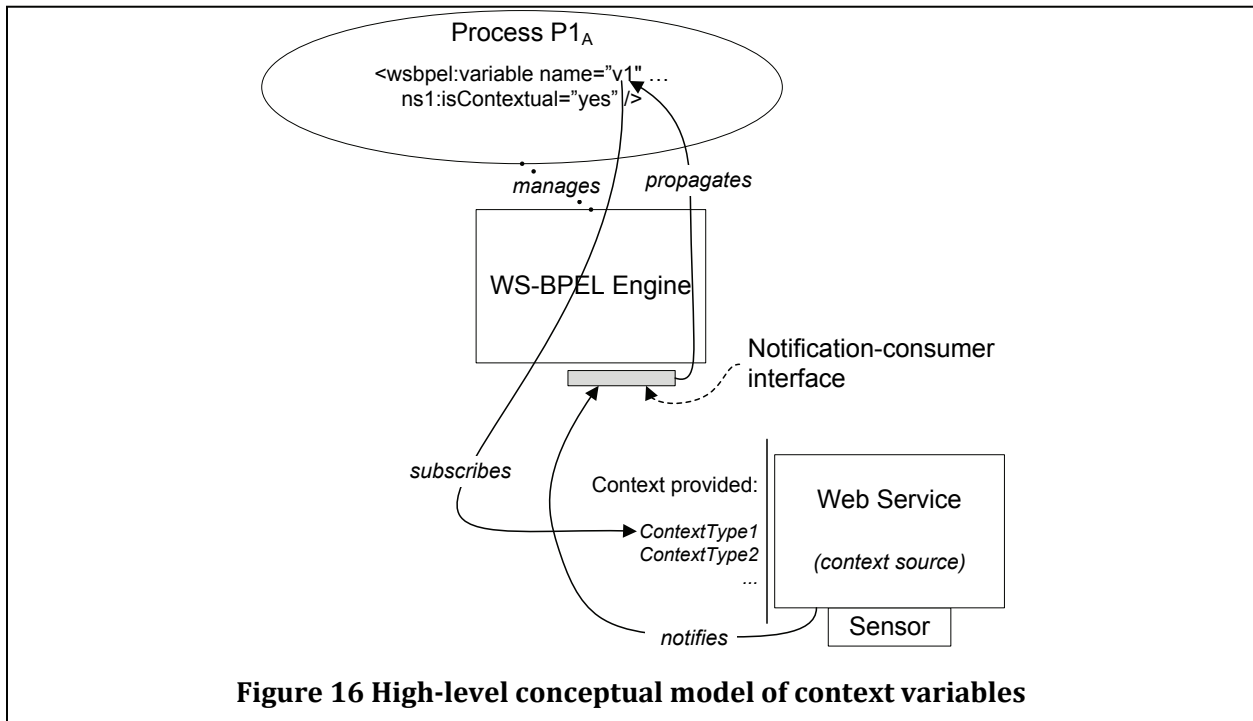
WS-BPEL processes are modeled as if they were the only thing operating in, and making changes to, the business environment. In this mental model all environment state is changed by the business process – either as a result of its activities, or a query it makes to external entities. This mental model does not reflect the real world, where organizations and entities depend on state that is outside their control – state that is modified independent of, and concurrent with, the organization’s activities. This is evident in our scenario, where the environment state the process is interested in (`EstimatedDeliveryDate`) changes independent of the process’ own actions.

`EstimatedDeliveryDate` is an example of *context*, and demonstrates characteristics that are common to all kinds of context over a range of business processes:

1. It is state that is driven and maintained outside the business process
2. Its value cannot be directly affected by a business process; i.e., the process cannot use activities like `<assign>` to change its real value
3. Its value changes unpredictably, and independent of the business process’ lifecycle
4. It influences process execution

Note that the relationship between a WS-BPEL process and context is similar to the relationship between events and event-based systems.

Unlike previous research, this thesis does not categorize context into a hierarchy or define what constitutes relevant context for various applications. We believe that a more feasible, and more widely applicable approach, is to allow WS-BPEL programmers to declare within their business process what qualifies as context. This provides visibility into what environment state is influencing the overall business logic; it also allows a programmer to show how context is represented



using XML data types, and where it influences process execution. Once the programmer has identified what they consider context in their business process, we can offload the minutiae of maintaining this context to the WS-BPEL engine. The tasks involved in this are:

- Sourcing context and storing it into the process' syntactic representation
- Monitoring it for changes and updating affected references when it occurs
- Facilitating comparisons between compatible context

What would be ideal is a solution for using context in a WS-BPEL process that:

1. Combines the conceptual simplicity and synchronicity of polling with the loose coupling and reduction in non-business logic of event handlers
2. Makes it simple for the WS-BPEL engine to achieve the above tasks in an automated fashion

This thesis proposes *context variables* as a way of representing context in a WS-BPEL business process. Context variables are not populated by the process, but by an external pub/sub context source. A context source contains a set of context types, and each variable subscribes to one or more of these types. Whenever the value of that context type changes its source publishes an update, which is then propagated to the variable. Figure 16 shows the conceptual model of a context variable.

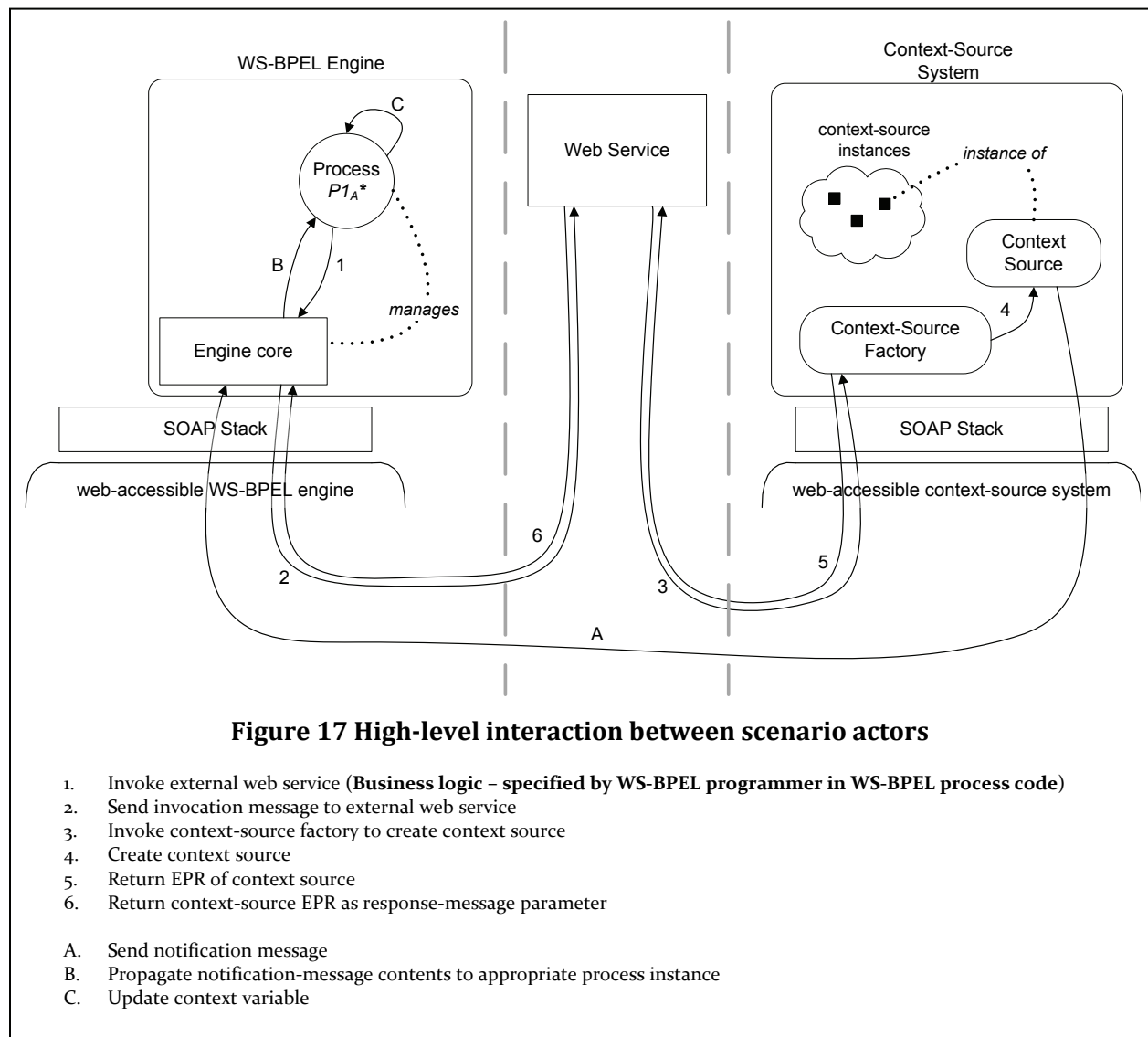
Context variables combine the best aspects of polling and onEvent message handlers. They can be accessed inline like standard WS-BPEL variables, so a process can be viewed as a single flow of control. WS-BPEL programmers no longer have to scatter state maintenance code throughout their process, and they can introduce additional context variables without having to write extra non-business logic. Coupling is also reduced since context variables depend only on the context type, not on the source providing the update, the operation generating it, or the partner link it arrives over. Finally, by building on pub/sub, context variables are ideally suited to dynamic state behavior. This proposal also allows context be handled consistently across WS-BPEL processes, obviating the need for ad-hoc solutions. Moreover, MEPs are associated with sourcing context, so context-source interactions can be automated and managed by a WS-BPEL engine. All these characteristics allow programmers to concentrate on implementing their business logic, making context variables ideal for incorporating externally-driven state in WS-BPEL processes.

## Chapter 4

### SOLUTION OUTLINE

The motivating scenario was used to guide the high-level design. In the scenario a manufacturer's WS-BPEL process invokes a supplier's web service and receives a reference to the shipment containing its order. The question is, then, how can the manufacturer's process be modified to use context variables without changing the process' activities? To do so requires the following:

- In the WS-BPEL process the output variable for the invoke activity used to call the supplier's



web service would be defined as a context variable

- Shipments would be modeled as context sources and their properties (`EstimatedDeliveryDate`, etc.) would be exposed as context types
- On receiving an order the supplier's web service would contact the shipping company to create a Shipment. It would then return the current values of the Shipment's context types, as well as parameters the invoker could use to subscribe to the Shipment for updates to these types
- The WS-BPEL engine would use the subscription parameters in the response message to subscribe to the Shipment for updates to its context types

The scenario requires three entities to interact for a WS-BPEL process to use context variables: the context source, the invoked web service, and the WS-BPEL engine. Figure 17 shows these entities, and the high-level message flows between them that are required to setup and populate context variables. An even simpler setup is possible – one in which a WS-BPEL process communicates directly with context sources – but this cannot be expected of all WS-BPEL processes. In many cases it is more likely, and in line with current practice, for processes to call intermediate web services that return links to managed context sources. This thesis considers the more complex setup shown in Figure 17, which leads to the following design challenges:

1. How are context sources modeled in a web-services environment?
2. Both WS-BPEL processes with context variables and those without can invoke the same web service. How can this web service return links to context types in its responses without making major interface changes, *while* allowing both types of WS-BPEL processes to function correctly?
3. What changes have to be made to the WS-BPEL language to support context variables?

The sections below describe these problems and the solutions designed to solve them.

#### 4.1 MODELING CONTEXT SOURCES

Context sources are entities that expose a collection of environment state parameters, or context. They organize and present different types of context at a single endpoint, thus simplifying interactions for context users. Our design challenge is to translate this conceptual model of context sources to the web-services space in a natural manner while satisfying the following criteria:

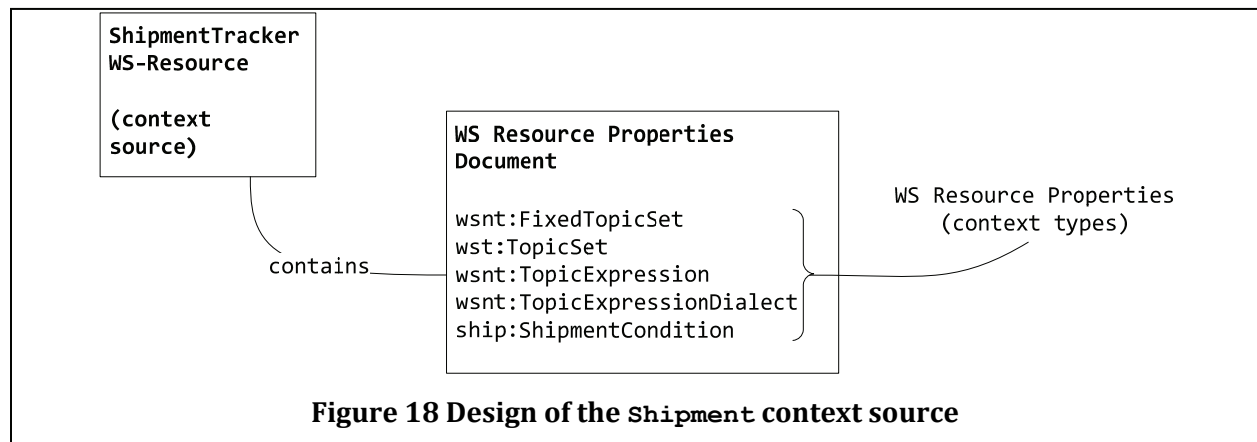


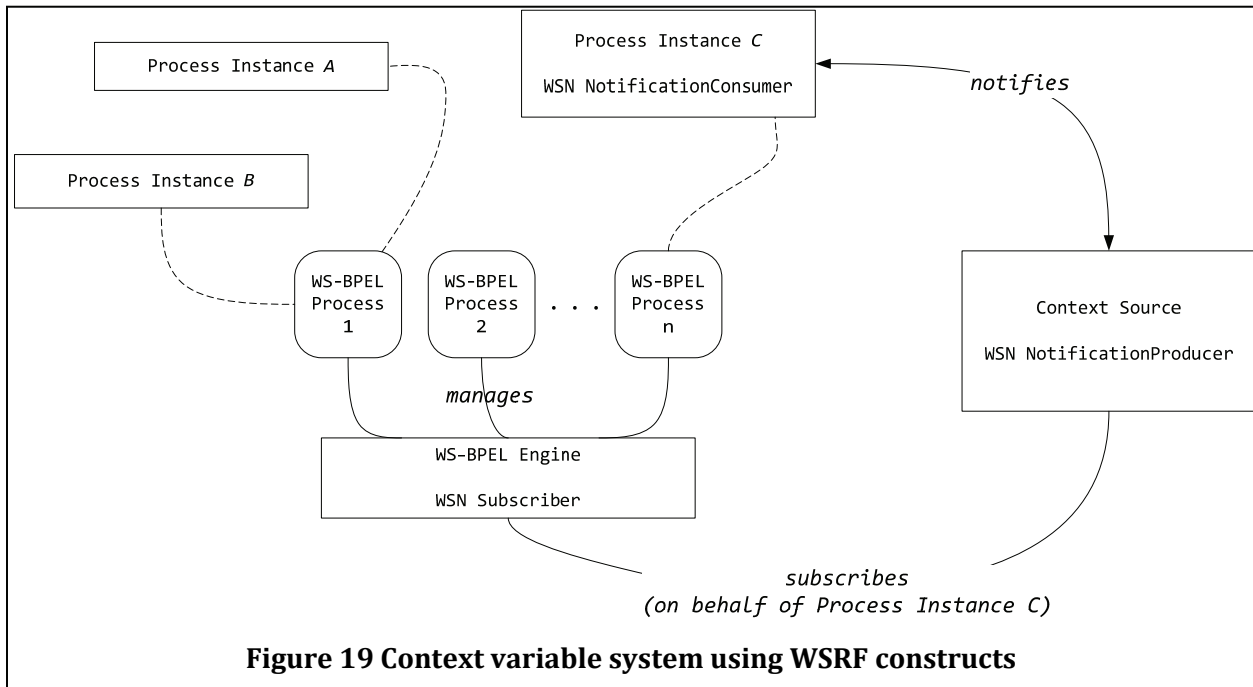
- Context sources have to be uniquely addressable
- There needs to be a way to create and return new context sources, or references to existing context sources, on demand

That these characteristics are needed is evident in our scenario: `Shipments` are not interchangeable – they are instances of the same kind of entity, but differ in one or more ways (e.g. their contents, current location, or estimated delivery date); and they do not exist *a priori*: they result from business activities. Given these requirements, this thesis proposes a context-source system based on constructs in WSRF.

To build context sources we return to the WSRF model of a stateful resource and its state components. Note that this is very similar to the idea of a context source offering a collection of context. For example, a `Shipment` can be thought of as a stateful resource, with its state components including ‘estimated delivery date’, ‘current location’, *etc.* Now, WSRF exposes stateful resources as web services by using the concept of a WS-Resource; it also dictates that the state components of this resource become XML Schema types in the WS-Resource’s Resource Properties Document. Since context sources can be viewed as a type of stateful resource, and context types as the state components of this resource, they can also be exposed in the web-services environment using the WS-Resource and WS Resource Properties Document patterns. Figure 18 shows how the `Shipment` context source can be represented using a WS-Resource and WS-Resource Properties.

WSRF places no constraints on the nature of a WS-Resource – whether it should represent a single real-world resource or a composition of real-world resources. This flexibility is essential when designing context sources, for while certain context sources – a `Shipment`, for example –





may represent a single resource, others may not.

The design of context variables envisions that each variable be subscribed to one or more context types at a context source, and receive updates whenever the values of those context types change. This design goal can be achieved using WSN. In a combined WSRF/WSN configuration a WS-Resource exposes each of its resource properties as a WS-Topic with the same name; external entities can then invoke the `Subscribe` operation on the WS-Resource to receive notifications for changes to its resource properties. This setup can be translated to the model of context variables, context sources, and context types. In this translation it is envisioned that:

1. Each context source acts as a WSN provider
2. Each context type in a context source is exposed as a WS-Topic in the source's XML namespace
3. Each WS-BPEL process that contains the context variables to be subscribed to the context types acts as a WSN consumer
4. The WS-BPEL engine act as a WSN subscriber, and subscribes to context types at a context source on behalf of a WS-BPEL process

This setup is shown in Figure 19.

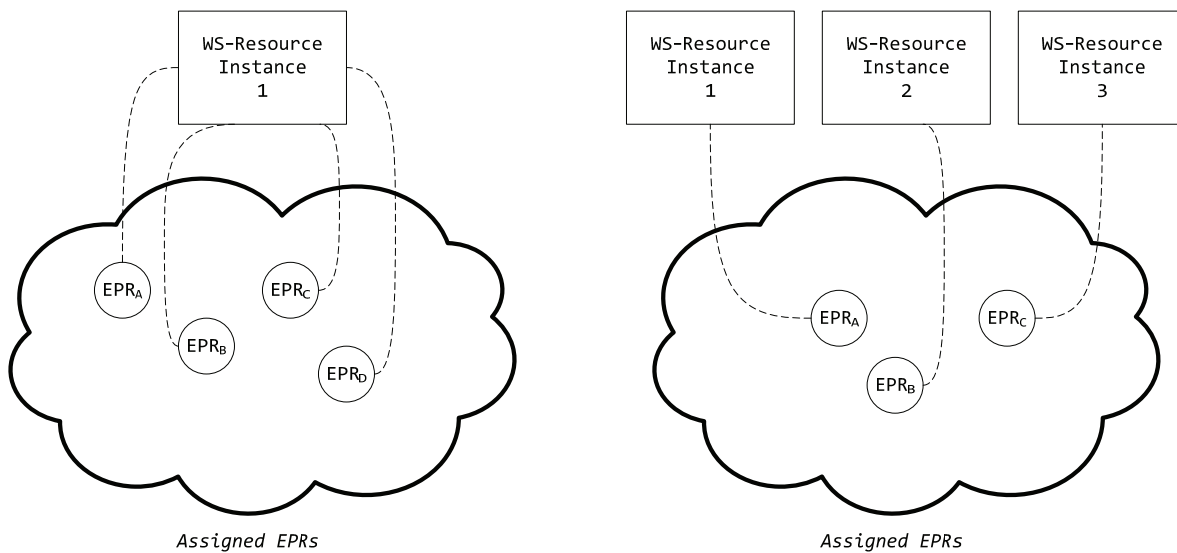
```

<wsa:EndpointReference xmlns:wsa="..." xmlns:retailer="...">
  <wsa:Address>http://www.example.com/retailer</wsa:Address>
  <wsa:ReferenceParameters>
    <retailer:CustomerOrderCart>Cart-7668</retailer:CustomerOrderCart>
  </wsa:ReferenceParameters>
</wsa:EndpointReference>

<wsa:EndpointReference xmlns:wsa="..." xmlns:retailer="...">
  <wsa:Address>http://www.example.com/retailer</wsa:Address>
  <wsa:ReferenceParameters>
    <retailer:CustomerOrderCart>Cart-7879</retailer:CustomerOrderCart>
  </wsa:ReferenceParameters>
</wsa:EndpointReference>

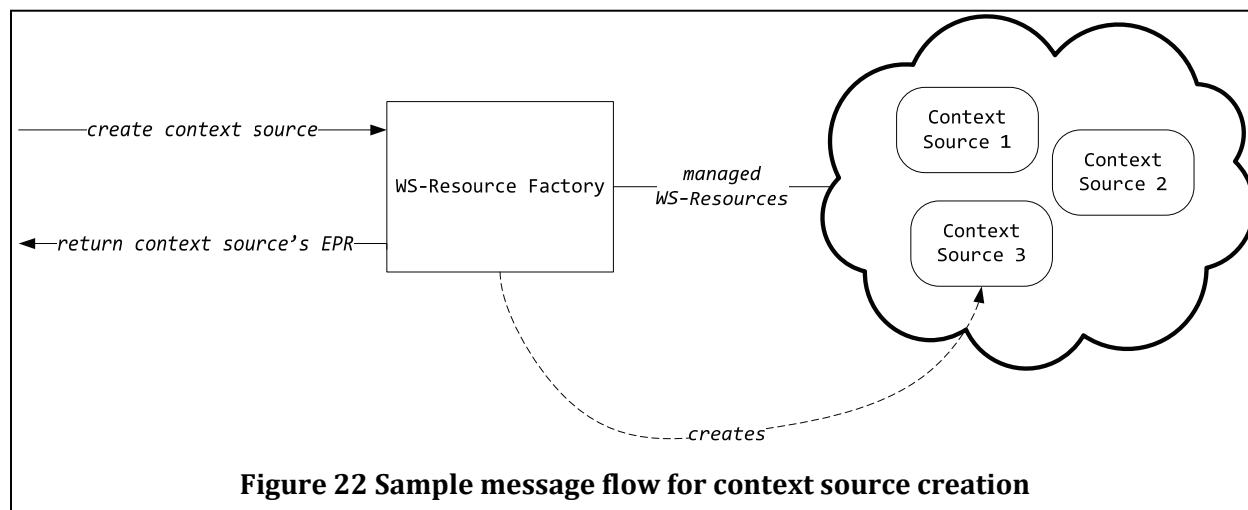
```

**Figure 20 Two EPRs differentiated using reference parameters**



**Figure 21 Two different mappings between EPRs are WS-Resource instances**

Modeling context sources as WS-Resources also allows the first criteria to be satisfied, since, as noted in Section 2.7, each WS-Resource is uniquely addressable using WSA EPRs. In this way a WS-BPEL process can access a specific context-source instance and the specific context-type values for that instance. This approach can be used to differentiate *Shipment* context sources in the motivating scenario. It is obvious that one *Shipment* is not the same as another, so EPRs can be used to differentiate them. This allows each WS-BPEL process instance to query, manipulate and share the *Shipment* context-source that its business activities created. The exact way in which these EPRs will be differentiated is implementation specific, but it is likely that most WSRF frameworks will do so using reference parameters, as shown in the example in Figure 20. The WSRF specification does not dictate the relationship between EPRs and WS-Resource instances.



While certain situations may call for a 1:1 correspondence between an EPR and a WS-Resource instance, there may be others where multiple EPRs are aliased to the same WS-Resource instance. Figure 21 shows these two possible relationships. This flexibility allows implementers of context sources to structure and address their system as they see fit in order to achieve administrative or business objectives. From the perspective of a developer using a context source, the relationship between an EPR and the context source is a non-issue; they simply request a context source, receive a reference to it, and query or manipulate it as necessary. Whether other users have received a reference to the same context source is not the developer's concern – the system's implementers are responsible for mediating access to context-source instances.

WS-Resources also do not have to exist in advance – they can be created by a WSRF WS-Resource Factory. This means that a developer can:

1. Invoke an operation on a WS-Resource Factory to create the WS-Resource corresponding to a context source
2. Receive an EPR identifying it
3. Use this EPR to interact with that context source

A sample message flow for this interaction is shown in Figure 22.

There is no standardized WSDL for a WS-Resource Factory, so its `create` operation is implementation specific. Also, the semantics of that operation will vary by context-type. In some situations a new context source will be instantiated on every invocation of the WS-Resource Factory; this would occur in a factory that manages `Shipment` context sources. In other situations invoc-

ing the WS-Resource Factory multiple times would return a reference to a singleton context source; *e.g.*, a factory that returns references to an `Environment` context source – one that represents the temperature, weather conditions, *etc.*, in a region.

There are many advantages to using WSRF constructs to represent context sources. First, the design builds on existing practices in the web services community. Many web services offer a view of a managed item's state in addition to other functionality; a WS-Resource represents the logical next step: it represents *only* the managed item. Also, by using WS-Resources designers no longer have to worry about how to expose context sources as web services. They are also freed from defining the MEPs and mechanics for context-source interactions, since WSRF standardizes the WSDL operations for manipulating resource properties in a WS-Resource. This obviates the need for ad-hoc solutions, and allows context-type query, retrieval, *etc.* to be automated. This allows developers to focus on the core issues in providing context through web services: the context types each source exposes, their XML representation, and their lifecycle.

From the perspective of a developer using context sources, the approach outlined above is extremely advantageous. First, the mental model of a context source as WS-Resource and context types as its resource properties is simple to understand. Instead of having to learn a custom specification for context awareness, they can leverage their knowledge of, and extensive resources available for, established specifications like WSRF. Moreover, the standardized MEPs for WS-Resources also simplify interactions with different types of context sources. While each source may represent a different real-world construct, and offer different context types, the broad outlines of its web-service representation, and the WSDL operations for manipulating its context types, remain the same. This allows developers to use the same programming logic in interacting with a range of context sources and makes it easier to implement basic context awareness in WS-BPEL.

## 4.2 CONTEXT-SOURCE REFERENCES

While some WS-BPEL processes will contact context sources directly, others will receive references to them from invoked web services. These web services have to return messages that can be used and validated by WS-BPEL engines supporting context variables and those that do not. A

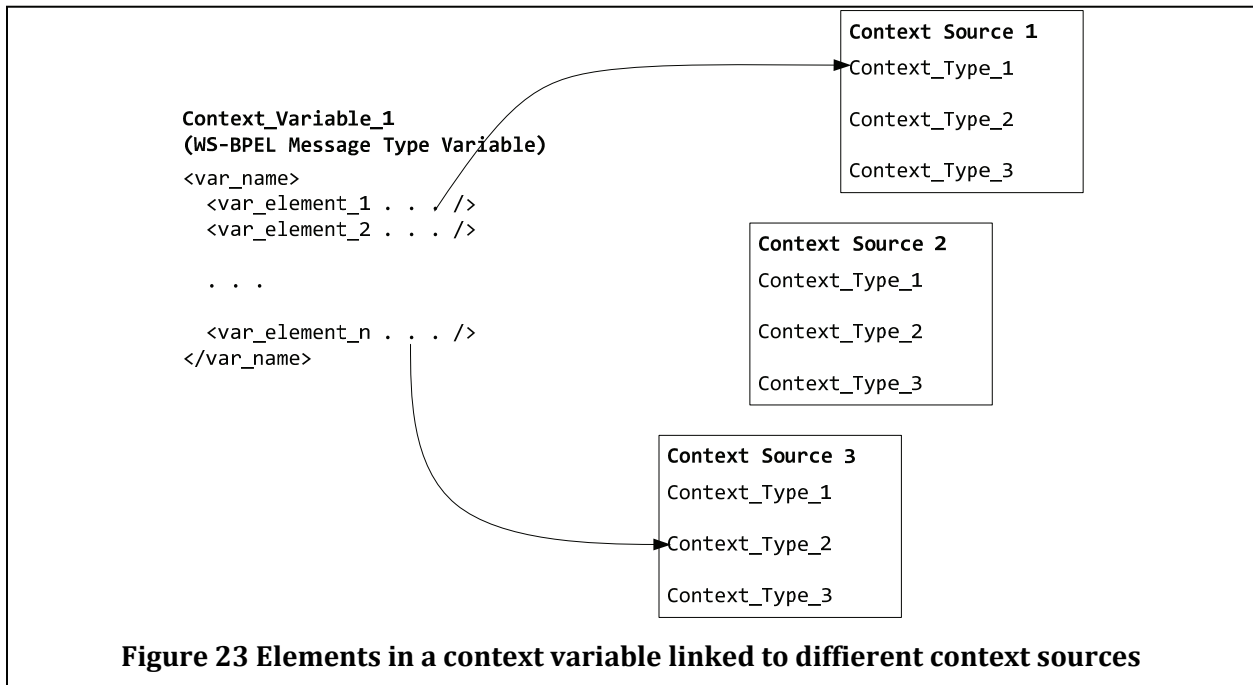
standard engine would only expect and use a snapshot of context types; those supporting context variables would expect both this snapshot and additional context-source subscription parameters.

There are multiple ways to support both types of WS-BPEL engines. The first, and simplest approach, is to expose two different WSDL operations – a basic one that returns only the context types' current values, and a context-enabled one that returns those values along with references to context sources. This is, however, a maintenance burden. The time and effort involved in synchronizing the message schemas for each operation pair, and for programming their logic, increases drastically with the number of operations in a WSDL. Simply put, this solution does not scale. Another option is to define a two-step MEP, where the snapshot of the context types is sent in the first message, followed by references to those context types in a second message. In this approach, when the WS-BPEL engine receives the first message it would automatically invoke a second operation on the sending web service that requested references to the context types. Again, this approach is inconvenient: implementers of the web service have to add significant logic to ensure that the reference-generating operation is invoked in the correct order, and that it returns the correct context-source/context-type references. Yet another option is the use of out-of-band signaling, where a separate message containing links to the context sources is sent in parallel with the operation response to a special interface on the WS-BPEL engine. This however, does not conform to web service best practices, and requires that all WS-BPEL engines implement a proprietary interface to support this signaling.

There is a lower overhead, standards-compliant alternative. This is to have the web service use a single WSDL operation and XML-Schema defined response that both engine types could validate. This response would include parameters that would allow a WS-BPEL process to subscribe to the relevant context types of a context source.

In designing this solution, the first challenge is deciding what parameters are required. Remember that:

1. Each context source is implemented as a WS-Resource and also acts as a WSN producer
2. The context types that this source offers are exposed as WS-Topics
3. The WS-BPEL engine acts as a WSN subscriber on behalf of a WS-BPEL process instance



In this configuration, if a WS-BPEL engine wishes to receive notifications of context type changes, it has to invoke the `Subscribe` operation on the context source using the message below:

```

<wsnt:Subscribe>
  <wsnt:ConsumerReference>wsa:EndpointReferenceType</wsnt:ConsumerReference>
  <wsnt:Filter>
    <wsnt:TopicExpression Dialect="xsd:anyURI">
      ns:anyTopicName
    </wsnt:TopicExpression>
    . . .
  </wsnt:Filter> ?
  . . .
</wsnt:Subscribe>

```

This `Subscribe` message requires the engine to have:

1. The topic corresponding to the context type
2. The context-type topic's namespace
3. The EPR of the context source
4. The EPR of the business process

Since the process' EPR is already known to the engine, 1, 2 and 3 have to be supplied by the invoked web service in its responses. Web-service response messages will vary by application. This thesis considers messages that conform to the guidelines outlined in the WS-Interoperability

(WS-I) Basic Profile. The WS-I Basic Profile is a best-practices paper that clarifies the inconsistencies resulting from multiple interpretations of the WSDL and SOAP specifications. By following these recommendations web services can be created and consumed regardless of toolkit used. WS-I dictates that Basic-Profile-compliant WSDL interfaces use the document/literal-wrapped binding style. One of the requirements of this style is that SOAP message parameters be grouped under a single ‘wrapper’ element that is the only direct child of the SOAP body. An example of a response message using such a style is show below:

```
<soap:envelope>
  <soap:body>
    <ns2:purchaseItemResponse . . .">
      <ns2:return xsi:type="ns2:ExtendedReturn">
        <ns2:itemCode>Stapler-101</ns2:itemCode>
        <ns2:purchaseStatus>SUCCEEEED</ns2:purchaseStatus>
      </ns2:return>
      . . .
    </soap:body>
  </soap:envelope>
```

In this example `<purchaseItemResponse>` is the wrapper element. Contained within `<purchaseItemResponse>` are a number of child elements that represent the return values of the invoked operation. As shown in Figure 23, only some of the values of one or more of these elements may be snapshots of context types at context sources.

If a WS-BPEL process wants to use this response to populate context variables, then the response message needs to include the three parameters listed earlier – the WS-Topic corresponding to the context type, its namespace, and the EPR of the context source – for each element that may be linked to a context type. Thus, using the example in Figure 23, `<var_element_1>` and `<var_element_n>` need these extra parameters.

The first approach considered was adding these parameters as attributes to each affected element. Closer investigation revealed that XML Schema does not allow complex types as attributes (Section 3.2.1, XML Schema Part 1: Structures specification [69]). A second option was to add the context-type subscription parameters as SOAP header elements in the response message. This is similar to the approach taken by WS-Addressing to bind an EPR’s properties to a SOAP message. Although this appeared to be a promising approach two issues arose:



## Context

↳ *ContextParameter* (\*)

## ContextParameter

↳ *StatefulParameterName*: *string*  
↳ *NotificationTopicNamespace*: *XSD anyURI*  
↳ *NotificationTopic*: *WS-T TopicType*  
↳ *ServiceEPR*: *WSA EndpointReferenceType*

**Figure 24 Context parameter schema**

1. There did not seem to be a clean way to reference specific elements in the response message from within SOAP-header elements
2. Many toolkits return only the response message's SOAP body to the WS-BPEL process. This makes it difficult to access the context-type subscription parameters and complicates the task of a programmer who wants to take their own approach to dealing with context sources

Given that neither of the approaches is suitable, we are left with one option: adding the context-type subscription parameters as elements in the response. Unfortunately it is not possible to simply add these parameters as extra elements to a response (without defining them in the schema) because strict parsers will reject them. So this thesis defines a solution that uses XML Schema type-derivation. Using the `<extension>` element, interface-designers can append extra attributes or elements to the base type used for the response-message element; when the service sends its response to invokers it indicates the element's concrete type with an `xsi:type` attribute. Endpoints using only the base type will ignore the additional elements, preserving backwards-compatibility.

Consider the schema for the following complex type:

```
<xsd:complexType name="Student">
  <xsd:sequence>
    <xsd:element name="Faculty" type="xsd:String" />
  </xsd:sequence>
</xsd:complexType>
```

Let this be the base type that a schema designer wishes to extend. Base types can be built-in types, simple types, or complex types. By using the `<extension>` element a designer can derive a new

type from the base type by adding new elements *after those* already defined in the base type's content model. This is a key point: new elements in the extended type cannot be interleaved with those already defined in the base type – they must be added at the end. We would have preferred an approach where the required information could be added to, or immediately after, each response element in the base type – but this is not possible. The following schema fragment shows how `<extension>` is used to derive a new type from `<Student>`:

```
<xsd:complexType name="EngineeringStudent">
  <xsd:complexContent>
    <xsd:extension base="tns:Student">
      <xsd:sequence>
        <xsd:element name="EngmailID" type="xsd:String"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

As noted earlier concrete types are identified in an instance document using the `xsi:type` attribute. Instances of both the base type and extended type from the example are shown below:

```
<!-- Instance of base type -->
<ns2:CurrentStudent xsi:type="ns2:Student" xmlns:xsi=" . . .">
  <ns2:Faculty>Math</Faculty>
</ns2:CurrentStudent>

<!-- Instance of extended type -->
<ns2:CurrentStudent xsi:type="ns2:EngineeringStudent" xmlns:xsi=" . . .">
  <ns2:Faculty>Engineering</ns2:Faculty>
  <ns2:EngmailID>noidperp</ns2:EngmailID>
</ns2:CurrentStudent>
```

A more complete description of the `<extension>` element and its use is given in Sections 4.2 and 4.3 of the XML Schema Part 0: Primer Second Edition [68].

Our solution isolates the subscription parameters to the schema fragment shown in Figure 24. Interface designers can simply append this fragment to the bottom of the content model of their base type. An example of the fragment's use is shown below:

```

<xsd:complexType name="ExtendedReturn">
  <xsd:complexContent>
    <xsd:extension base="tns:BaseReturn">
      <xsd:sequence>
        <xsd:element name="Context" maxOccurs="unbounded" minOccurs="1"
          type="tns:ContextParameter" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

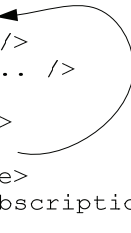
Since a derived type's elements can only be appended after those already defined in the base type, we use `StatefulParameterName` to link a set of subscription parameters in the extension with its associated element in the base type. This is best demonstrated in Figure 25, where the base type in the response message has three elements, but only `CurrentLocation` is exposed as a context type:

**WSDL Message:**

```

<CurrentLocation ... />
<ElapsedTransitTime ... />
<EstimatedDeliveryDate ... />
<Context>
  <StatefulParameterName>
    CurrentLocation
  </StatefulParameterName>
  <!-- Context source subscription elements -->
</Context>

```



**Figure 25 Use of <StatefulParameterName>**

The value of `StatefulParameterName` in the `Context` element indicates that these subscription parameters are associated with `CurrentLocation`. By using the `<extension>` element a web service can define two messages:

1. A base type containing only a snapshot of the context types
2. A derived type that contains references to context sources

This allows a web service to send a response message containing the derived type to any WS-BPEL engine – whether it supported context variables or not – and have it validate properly. As a result, backwards compatibility is maintained. For its part, the WS-BPEL engine does not have to send any extra information when it invokes an operation on a web service using this approach. The solution outlined above also minimizes interface changes and is standards-compliant.

## 4.3 EXTENDING WS-BPEL

Implementing context variables in WS-BPEL consists of two parts:

1. Using the language extensibility features of WS-BPEL, defined in the WS-BPEL standard, to support context variables
2. Defining the interactions WS-BPEL engines need to implement to setup and update these variables

### 4.3.1 LANGUAGE SUPPORT

In WS-BPEL a variable's value cannot be constantly updated by an external entity. It can be populated indirectly by an external entity through an `<invoke>`, a message handler, *etc.*, but once populated no link is maintained between the variable and the entity. Thus, to implement context variables we have to extend the WS-BPEL language. This is common practice. As stated in Section 2.4, the WS-BPEL language is built to be extensible, and its specification clearly dictates (Sections 5.3, 8.4, 10.9, 14, WS-BPEL specification [39]) how language extensions are to be designed and used. This allows the language to adapt to concerns that may not have been foreseen at the time of its creation. In fact, the web-services community has already taken advantage of WS-BPEL's extension support to create, among others, the BPEL4People extension [3] and the WS-BPEL 2.0 Extensions for Sub-Processes [25].

Before defining our extension syntax tokens and deciding where they were to be added, the following principles were created and used to drive the design of the context-variable extension:

1. Programmers should be able to specify within a process definition if a WS-BPEL variable is a context variable or not
2. After initialization a context variable's value cannot be changed by any in-process activities

The other characteristics of WS-BPEL variables were retained (scoping rules, *etc.*) so that programmers could think of context variables as another type of variable, not as a different construct altogether.

The first step in creating a WS-BPEL extension is to declare its namespace. As defined in the WS-BPEL specification, the presence of an extension in a process is indicated by a child `<exten-`

sion> element under the <extensions> element in a process definition. This child <extension> element contains:

1. The namespace of the extension
2. An attribute indicating whether the WS-BPEL processor must understand the semantics of the extension element

The <extension> element chosen for the context variable extension is shown below:

```
<bpel:extensions>  
  <bpel:extension mustUnderstand="yes"  
    namespace="http://ece.uwaterloo.ca/aag/statefulbpel"/>  
</bpel:extensions>
```

Simply adding an <extension> declaration does not change the behavior of a WS-BPEL process or its constituent activities – it only determines whether the WS-BPEL engine can load the process definition for execution or not. For behavior to change:

. . . an extension syntax token, in the form of an element or attribute qualified by the URI value of a namespace attribute in an <extension> element that is used outside of an <extension> element MUST appear in the WS-BPEL process definition . . . It is this extension syntax token, rather than the extension declaration, that indicates that the new semantics apply. [39]

In short, for an extension to change process behavior, extension-namespace qualified attributes or elements have to be added to one or more predefined WS-BPEL elements.

It was decided that the context variable extension syntax token would be a namespace-qualified attribute called `isStateful`. The presence of this token on a predefined WS-BPEL element should cause the WS-BPEL engine to change the behavior of that element and all elements in its subtree. Having defined the extension syntax token, we met the first design principle by adding the `isStateful="yes"` attribute to <variable> elements to denote context variables. An example of a context variable is shown below:

```
<bpel:variable messageType="ns2:PurchaseItemResponse" name="purchaseItemResponse"  
  ns4:isStateful="yes"/>
```

A WS-BPEL variable with this attribute is subscribed by the WS-BPEL engine to a context type at a context source. Its value can only be updated by that context source – not by any other activities

within the process definition. This link between a context source and a context variable can only be terminated when the WS-BPEL process itself terminates, either normally or through a fault.

This thesis defines one way of initializing context variables and receiving the parameters required for its subscription: through an extended response message from an external web service. In practice, this requires that the context variable must be used as the output variable of the `<invoke>` activity used to interact with this web service. An example of an `<invoke>` activity using the `purchaseItemResponse` context variable as its output variable is shown below:

```
<bpel:invoke inputVariable="purchaseItemRequest" operation="PurchaseItem"
  outputVariable="purchaseItemResponse" partnerLink="eCommerceStoreLink"
  portType="ns2:Purchase">
  <bpel:targets>
    <bpel:target linkName="L3"/>
  </bpel:targets>
  <bpel:sources>
    <bpel:source linkName="L4"/>
  </bpel:sources>
</bpel:invoke>
```

In the future other ways of initializing context variables may be defined.

Prior to this `<invoke>` activity being executed by the WS-BPEL engine, the variable's elements will not be subscribed to any context types and will not receive any notifications. Once the engine receives a response message containing context-type subscription parameters, it matches each context-type subscription parameter to the appropriate element in the output variable and performs the necessary subscriptions. A child element in a context variable can only be subscribed to one context type at a context source. If however, a context variable has multiple elements, not all of them have to be linked to a context type. Moreover, if a context variable has multiple elements, each element may be subscribed to context types at different context sources – there is no requirement that all the elements in a context variable should only be subscribed to context types at a single context source. An example of this is seen in Figure 23.

Section 4.2 describes how a response message can contain context-type subscription parameters for any or all of its elements. If the `<invoke>` activity's output variable is a WSDL message type, then the WS-BPEL engine must subscribe all elements with a corresponding context parameter to the appropriate context-type/context-source. This may result in a variable in which certain elements will be updated by a context-source, while others will be static. This can be seen in Figure

23. The WS-BPEL engine must ensure that once a variable has any element that is subscribed to a context source, none of its elements (whether linked to a context source or not) must be modified by other WS-BPEL activities.

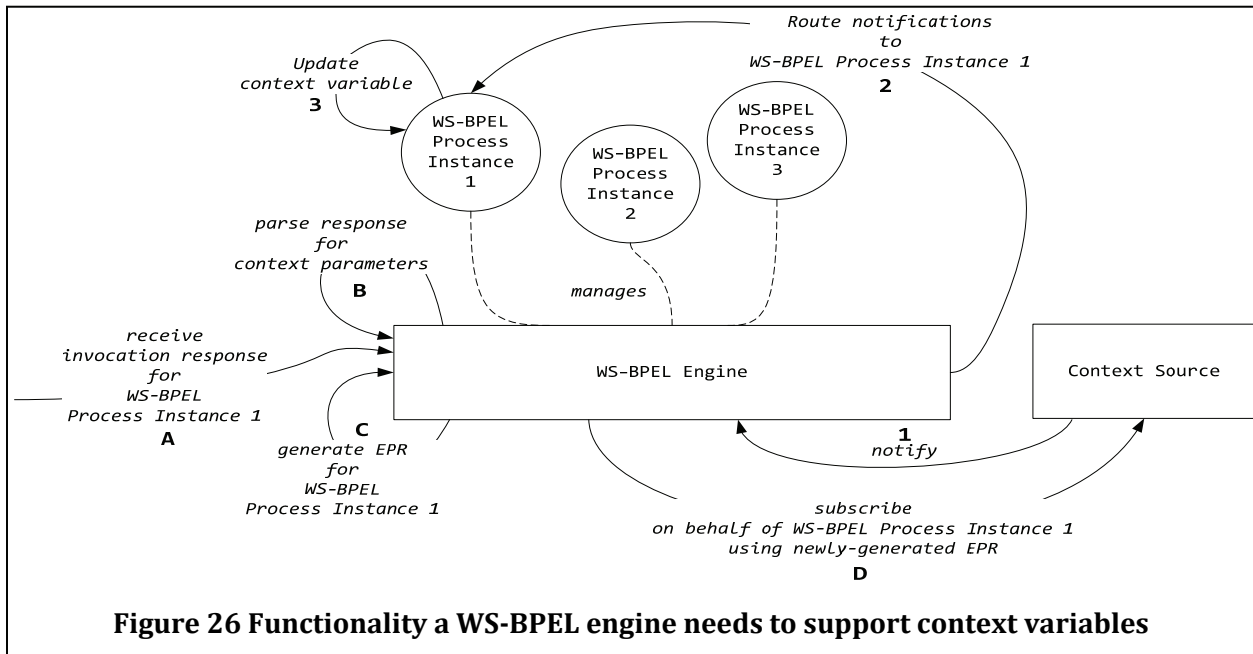
WS-BPEL also allows two other variable types: XML Schema type (simple or complex) or XML Schema element, both of which can be used to create the output variable for an `<invoke>` activity. In either case, once the WS-BPEL engine receives a response message that contains context-type subscription parameters, it must:

1. Determine if the response message elements/types they reference are present in the output variable
2. Perform subscriptions for any element in the output variable with a context-type subscription parameter in the response

It may be the case that none of context-type subscription parameters apply to the elements in the output variable. When this occurs the WS-BPEL engine should perform no subscriptions, but should still prevent the variable from being updated by other in-process activities.

Satisfying the principle that no in-process activity be able to modify the value of a context variable required that the same extension attribute be added to `<process>` as to `<variable>`. This is because Section 14 of the WS-BPEL specification states: “An extension syntax token can only affect WS-BPEL constructs within the syntax sub-tree of the parent element of the token.” As our second design principle governs other activities’ behavior, their containing element, `<process>`, has to be changed. Note that adding `isStateful="yes"` to `<process>` only affects those process activities using context variables. If an activity uses regular variables it is unaffected, and if a process has no context variables it behaves like a process without the extension attribute.

The WS-BPEL language extension this thesis proposes is minimally intrusive: by retaining most characteristics of `<variable>` and `<process>`, WS-BPEL programmers do not have to learn all-new semantics for context variables; it minimizes the changes needed in WS-BPEL engines to support context variables; and it conforms to the language extension rules in Section 5.3 of the WS-BPEL specification. Moreover, the addition of `isStateful="yes"` to `<variable>` and `<process>` elements is a minimal change and allows graceful downgrading; in unmodified engines a `<variable>` with the extension attribute will function as normal, but will not be linked to



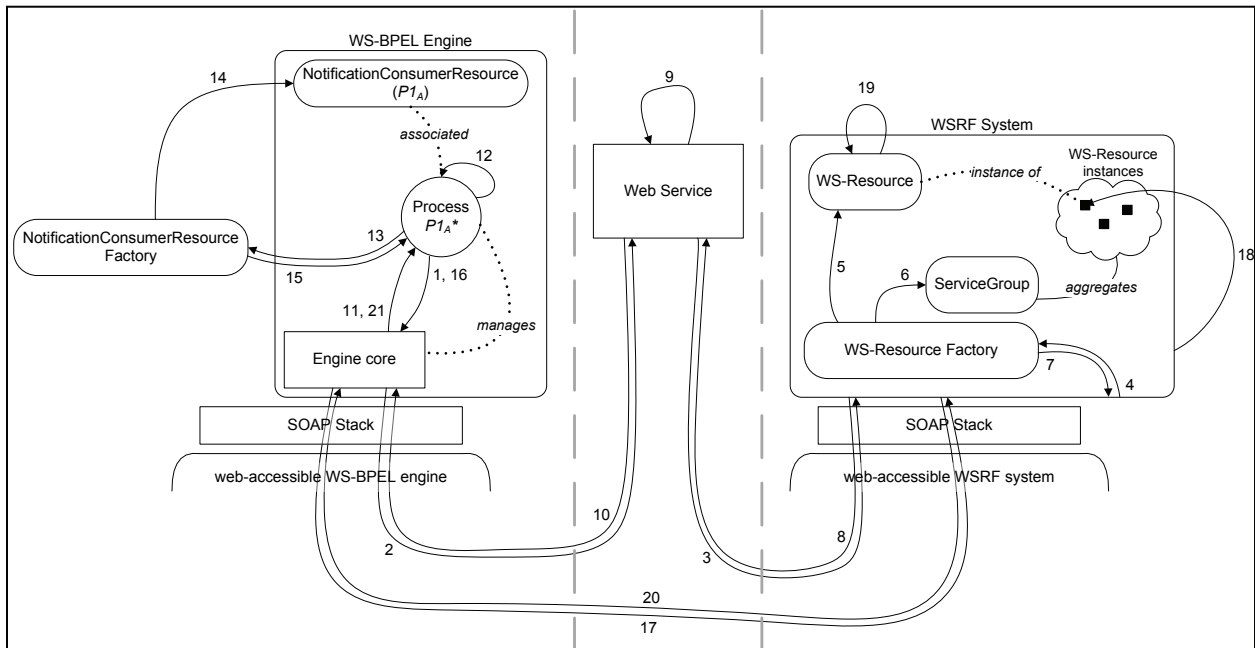
context sources; in unmodified engines a `<process>` with the extension attribute executes in exactly the same way as a WS-BPEL process without context variables. Finally, since the only change this extension requires in the process definition is the addition of the `isStateful="yes"` attribute to the `<variable>` and `<process>` elements, programmers can continue to use their existing WS-BPEL development environments. This is a major advantage, since developers do not have to abandon their existing methodologies and development tools to use context variables.

#### 4.3.2 WS-BPEL ENGINE ENHANCEMENTS

Existing WS-BPEL engines must be modified to support context variables. First, the WS-BPEL engine must be changed so that it can parse and load process definition files containing the language extension defined in Section 4.3.1. Then it needs to implement the MEPs and logic necessary to link a context variable to a context-source, and update the value of this variable on receiving notifications from the context source. For the latter, a WS-BPEL engine must, at minimum:

- Intercept and parse invocation responses for context parameters
- Subscribe context variables to context sources using the WSN `Subscribe` operation and the information in these parameters
- Route WSN `Notify` messages containing context changes to the appropriate context variable in a process instance



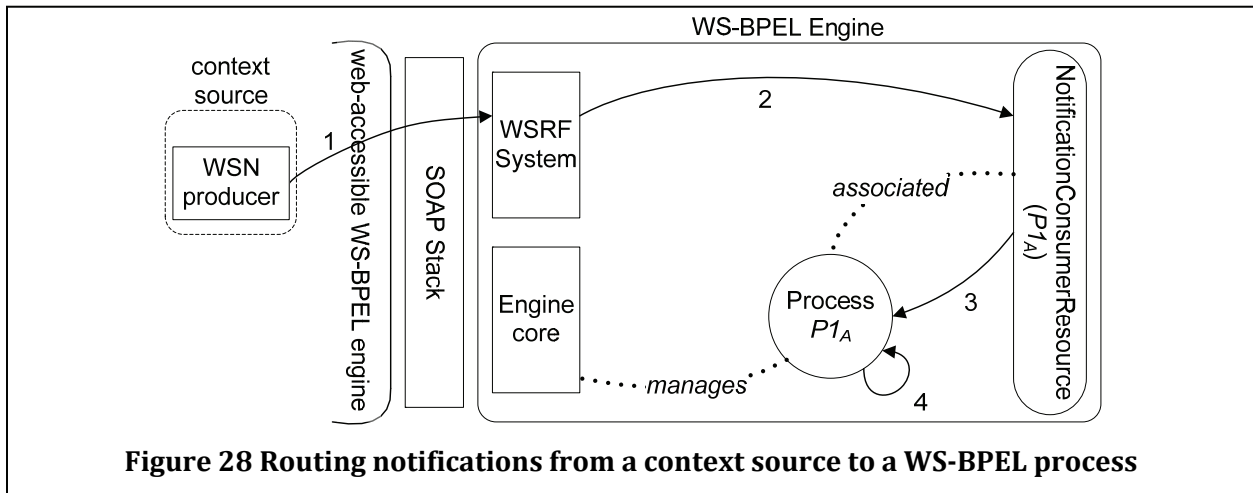


**Figure 27 High-level message flow for context variable setup**

1. Invoke external web service (**Business logic – specified by WS-BPEL programmer in WS-BPEL process code**)
2. Send invocation message to web service
3. Request address for stateful resource instance of a given type
4. Forward request to WS-Resource Factory that handles resources of that type
5. Create/get a WS-Resource
6. Add WS-Resource reference to the appropriate ServiceGroup
7. Return WS-Resource EPR and list of its notification topics (i.e. WS-Resource Properties)
8. Respond to web service with WS-Resource EPR and notification topics to web service
9. Create mapping between web-service response-message elements and the WS-Resource notification topics and publisher address
10. Return invocation response
11. Forward invocation response to the invoking process instance
12. Inspect returned message for subscription information elements
13. Request a NotificationConsumer WS-Resource instance associated with this process instance
14. Create a NotificationConsumer WS-Resource
15. Return the EPR of the NotificationConsumer WS-Resource associated with the process instance
16. Create subscription request using the NotificationConsumer WS-Resource EPR as the listener and returned WS-Resource EPR as publisher
17. Send subscription message to the WS-Resource EPR
18. Route subscription request to correct WS-Resource instance
19. Record the subscription (consumer, watched topics, etc.)
20. Return subscription response
21. Forward subscription response to process

- Update the context variable with the new values in these notifications

While the exact implementation of this functionality will vary depending on the WS-BPEL engine being changed, all modified engines will have to support the high-level message flow outlined in Figure 26. In Figure 26, lines A – D outline the steps required to link a context variable to a context source, while lines 1 – 3 denote those necessary to update the context variables in a process instance.



**Figure 28 Routing notifications from a context source to a WS-BPEL process**

#### 4.4 OVERALL SYSTEM

Using these solutions we now create the integrated system shown in Figure 27. Consider the message flow that occurs when a WS-BPEL process uses context variables and subscribes to a context type. The WS-BPEL process definition contains an invoke activity that calls a WSDL operation in an external web-service. When this activity is executed the invoked web service creates a context source using the WSRF system. It then returns subscription parameters to the relevant context types as extensions in its response message. On receiving this response the WS-BPEL engine creates a `NotificationConsumerResource` WS-Resource as a facet of the destination process. This WS-Resource is a WSN consumer, and the engine subscribes to the context sources in the response message using its EPR as the consumer endpoint. Note that the *only* step the process programmer has to specify is the first: invoking the web service and defining its output to be a context variable; the rest are executed by our solution infrastructure. Although the resulting system flow is involved, this is a reasonable tradeoff: by burying a feature's complexity in the infrastructure, the productivity of many target developers is improved. Figure 28 shows the message flow for context-type updates. Notifications are addressed to the `NotificationConsumerResource` WS-Resource of the process with context variables. When the resource receives these notifications it extracts the new values of the context types and sends them to its associated process, which then propagates them to the relevant variables.

This design does not rely on any vendor-specific features and can be constructed using any WS-BPEL engine, application server hosting a web service, and WSRF-capable system. It is based on a

simple conceptual model, and process programmers do not have to add non-business process logic to use context variables. It uses accepted standards and MEPs, so context sources can be created using existing toolkits, and the difficulty of modifying a WS-BPEL engine is reduced. Moreover, the WS-BPEL engine, web service, and context-source system do not have to support any non-standard communication patterns. Finally, being push-based, it imposes low overhead on the WS-BPEL engine.

## Chapter 5

### PROTOTYPE

A prototype for our WS-BPEL context architecture was developed using free, open-source, well-maintained toolkits. It demonstrates that existing development frameworks can be leveraged to build the proposed system, and that such a system achieves its design purpose. The prototype was implemented using:

- The ActiveBPEL 4.1 WS-BPEL engine, which we modified
- A Java EE 5 web service
- An Apache Muse 2.2.0 WSRF/WSN system for context sources

The initial implementation functions as a proof of concept. It features only basic functionality, and highlights the challenges developers face in developing architectures across multiple frameworks. In the following sections we describe how the high-level requirements described in Chapter 3 are put into practice. We highlight interesting implementation details and discuss the impact each toolkit's design had on the architectural entity that was implemented on it. Finally, note that nothing in the high-level architecture is toolkit-specific – an architecture for supporting context variables can be built using any WS-BPEL engine, web-services toolkit and WSRF/WSN system. This is simply one example.

#### 5.1 WSRF-BASED CONTEXT SOURCE FRAMEWORK

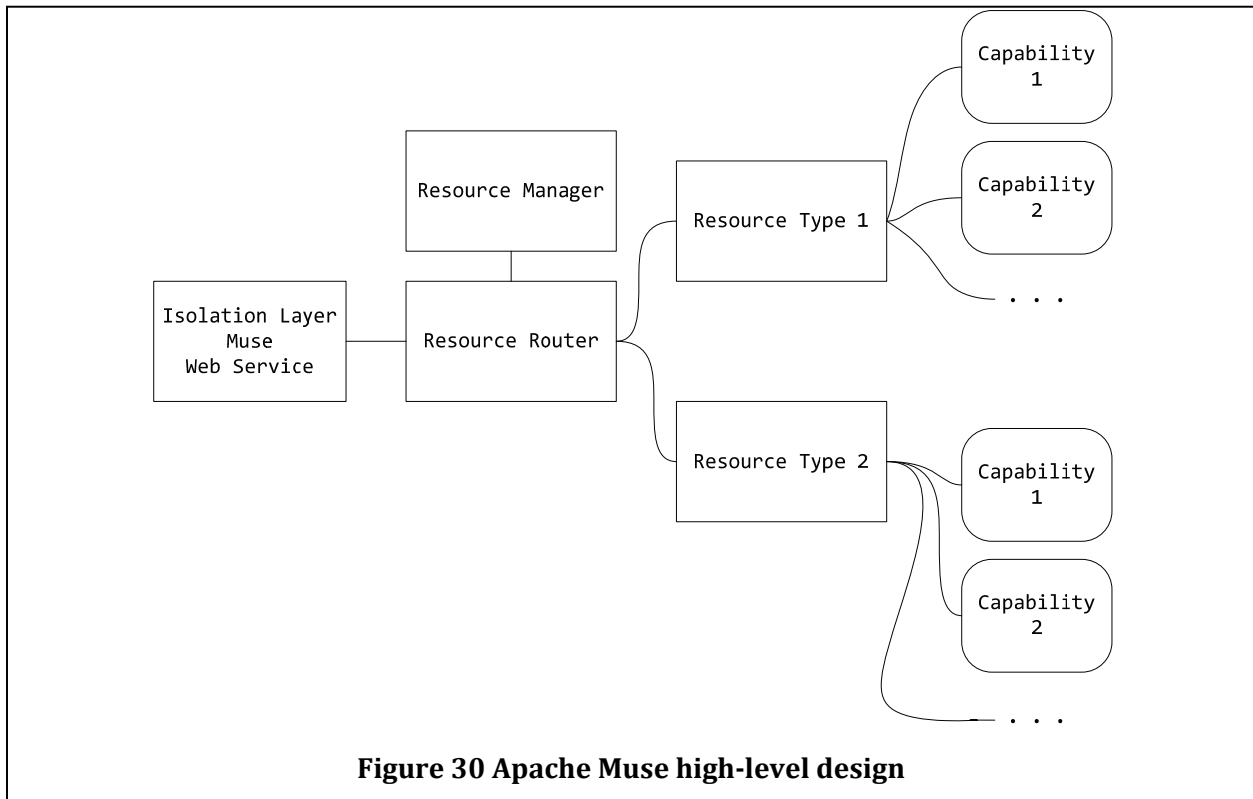
The proposed system uses WSRF WS-Resources as context sources. Context sources expose con-

```
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://delivery.company/shipping/Shipment">

  <xsd:element name="ShipmentCondition" type="xsd:string"/>

  <xsd:element name="ShippingResourceProperties">
    <xsd:complexType>
      <xsd:sequence>
        . . .
        <xsd:element ref="ship:ShipmentCondition" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

**Figure 29** ShipmentCondition context type



text types as resource properties in their WS Resource Properties document. These context sources may exist *a priori*, or result from business activities.

It is impossible to build an abstract WS-Resource. Instead, one has to build a WS-Resource that exhibits the behavior and characteristics of interest in the system being exercised. For this reason the context source in the prototype is the one described in the motivating scenario: the `Ship-ment`. It exposes only a single context type that is described using the XML schema in Figure 29. This single context source is the simplest one possible that highlights:

1. The WSRF entities are required to create a working WSRF context-source system
2. The design considerations that have to be accounted for when creating real-world context sources

### 5.1.1 APACHE MUSE

We used Apache Muse 2.2.0, a WSRF/WSN/Web Services Distributed Management (WSDM)-compliant framework, to create our context-source system. Apache Muse (hereafter, Muse) is structured as a resource manager that sits on top of a SOAP stack, and runs as a web application

(webapp) in common servlet containers like Apache Tomcat. Each Muse resource type has its own WSDL interface, and is constructed by aggregating a number of Java capabilities. Together, these capabilities implement the operations and store the properties described in the Muse resource-type's interface. Instances of these resource types are created at runtime (one can think of resource types as templates and resource-type instances as concrete instantiations) and are addressable through WSA EPRs.

Muse affords programmers a lot of flexibility over the design of managed resources. Resources can be singletons or have many instances; they can be persisted across application restarts; they can represent both real-world or virtual entities. This flexibility allows Muse a wide range of application requirements to be supported.

At a high level Muse is a router for managed resources, and it has the structure shown in Figure 30. A brief description of the major components follows:

- **Capability:** A Java class composing a set of properties and operations that are exposed via web services. Capabilities are aggregated to create a resource type, and provide the functionality for that resource type. They are either written by the programmer or are default implementations of the operations and behavior specified by WSRF, WSN or WSDM.
- **Resource Type:** An aggregation of capabilities that is exposed as a web service. Each resource type is described by a WSDL interface. A resource type can have many runtime instances which are addressable using WSA EPRs.
- **Resource Manager:** Maintains a list of known resource types and their constituent capabilities. Also functions as the component through which resources can be located, set up, and shut down, etc. from within Muse.
- **Resource Router:** Uses the EPR and WSA `Action` headers to route incoming web service requests to the appropriate resource-type instance.
- **Isolation Layer:** An abstraction over the communication and deployment artifacts of a given deployment environment. Components above this layer are insulated from implementation-specific constructs, allowing Muse to be ported between environments without changes to the code in the managed resource types.

A full description of Muse's design is given in *Apache Muse – Programming Model* [7].

```

<wsdl:portType name="ShipmentTrackerPortType"
wsrf-rp:ResourceProperties="ship:ShippingResourceProperties"
wsrmd:Descriptor="ShipmentResourceMetadata"
wsrmd:DescriptorLocation="ShipmentResource.rmd">

  <!-- Specification defined operation -->
  <wsdl:operation name="GetMetadata">
    <wsdl:input wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata"
      name="GetMetadataMsg" message="tns:GetMetadataMsg"/>
    <wsdl:output
      wsa:Action="http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadataResponse"
      name="GetMetadataResponseMsg" message="tns:GetMetadataResponseMsg"/>
  </wsdl:operation>

  <!-- Custom operation -->
  <wsdl:operation name="GetShipmentContents">
    <wsdl:input wsa:Action="http://delivery.company/GetShipmentContents"
      message="tns:GetShipmentContentsRequest"/>
    <wsdl:output wsa:Action="http://delivery.company/ShipmentContents"
      name="GetShipmentContentsResponse" message="tns:GetShipmentContentsResponse"/>
  </wsdl:operation>
  .
  .
  .
</wsdl:portType>

```

**Figure 31 Port Type with both a specification-defined and custom operation**

Any resource can offer a mixture of WSRF, WSDM, WSN and custom functionality using operations in its WSDL interface. This interface always consists of a single port type with a set of specification-defined and custom operations. Specification-defined operations are simply copied from the WSDL interfaces defined in the corresponding standards. A simple example of a port type having both a specification-defined and custom operation is shown in Figure 31.

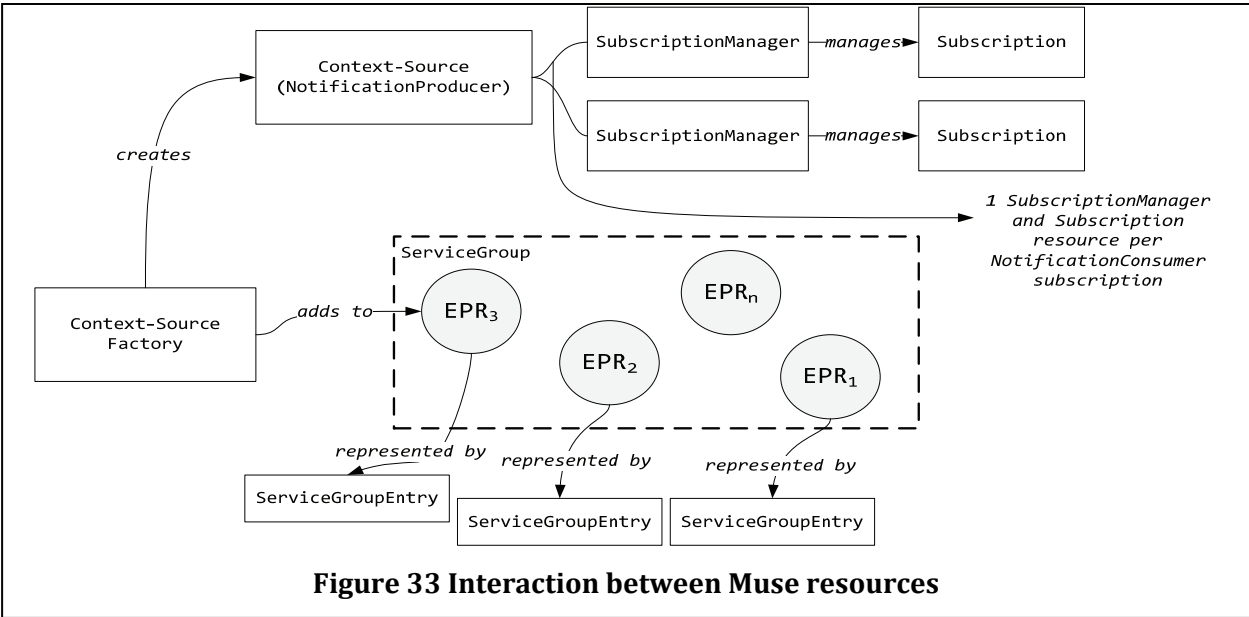
When a SOAP message is sent to a Muse-managed resource, two levels of routing take place. The first is performed by the SOAP stack. From its perspective Muse is one of many web services

```

<serviceGroup>
  <service name="MainInterface">
    <parameter locked="false" name="ServiceClass">
      org.apache.muse.core.platform.axis2.AxisIsolationLayer
    </parameter>
    <parameter name="useOriginalwsdl">true</parameter>
    <operation name="handleRequest">
      <messageReceiver
        class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
      <actionMapping>
        http://delivery.company/MainInterface/ShipmentTrackingRequest
      </actionMapping>
    </operation>
  </service>
</serviceGroup>

```

**Figure 32 Mapping between a WSA Action URI and the Muse receiving class**



**Figure 33 Interaction between Muse resources**

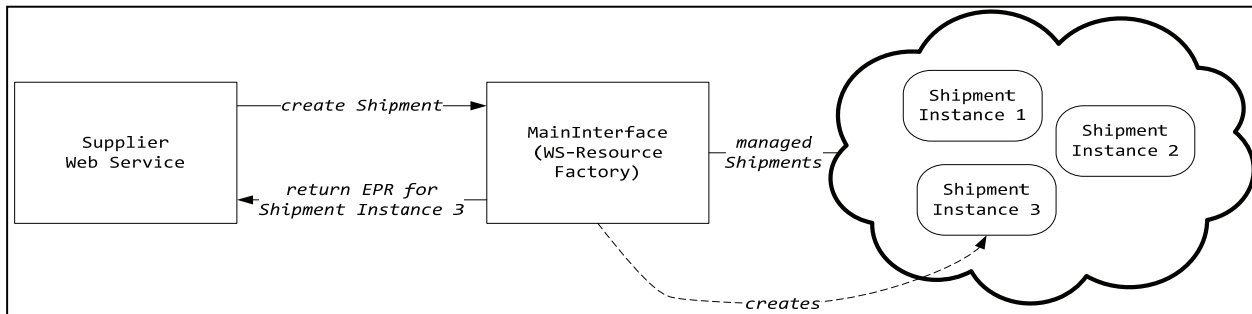
utilizing the stack. As a result, the programmer has to define a mapping between any incoming WSA Action URIs and the Java web-service class handling the SOAP messages with which they are associated. As the prototype WSRF system uses Apache Axis2 for its SOAP stack, this mapping is defined in Axis2's services.xml file. An example mapping from `http://delivery.company/MainInterface/ShipmentTrackingRequest` (the URI) to `org.apache.axis2.receivers.RawXMLINOutMessageReceiver` (the Java web-service class) is shown in Figure 32. The second level of routing occurs within Muse itself. Once it receives the web service request from the SOAP stack it has to determine which resource instance, and capability within that instance, should service the request. The capabilities within a resource type are specified in the Muse deployment descriptor `muse.xml` [6].

Muse uses code generators to simplify the development process. As usual, however, these are best used with simple resource designs, and are poorly suited to iterative development.

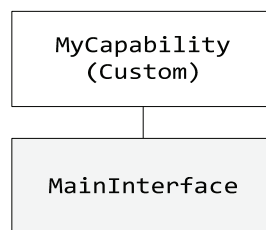
5.1.2 MANAGED RESOURCES

In the prototype context-source system Muse runs as an Axis2 web service on Apache Tomcat 5.5.x. The system uses five resource types: a context source, a context-source factory, a ServiceGroup, a ServiceGroupEntry, and a SubscriptionManager. The context-source factory is invoked by an external web service to create a context-source instance. This context-source is in-





**Figure 34 Business interaction to create shipment context sources**



**Figure 35 Capabilities for the MainInterface Muse resource (context-source factory)**

serted into a `ServiceGroup` and its presence in the group denoted by a `ServiceGroupEntry`. When an external entity subscribes to a context-source instance to receive notifications when the source’s context types change, a subscription WS-Resource is generated; this resource is managed by the `SubscriptionManager`. The relationship between these resources is shown in Figure 33.

The simplest way to create Muse resources is by using the “start from WSDL” approach. In this approach the programmer first designs the WSDL interface. They compose a port type from a mix of specification-defined and custom operations, then add the necessary XML-Schema types, WSDL messages and faults, and finally, specify any resource properties and resource metadata, as required. This WSDL interface is then used fed into the Muse code generator, which:

1. Creates the files required for an AXIS2 web-service deployment
2. Uses the operation names and namespaces to generate the appropriate capabilities
3. Creates the `muse.xml` deployment descriptor

The operation of this code generator is detailed in Apache Muse – WSDL2Java Tool [8]. This process is repeated for each resource type that needs to be hosted on Muse. Note that a new `Axis2 services.xml` and `muse.xml` deployment descriptor is created each time a new resource type is created, and must be manually combined to list all the resource types hosted in a single Muse in-

stance. If the WSDL interface is later changed, the best course of action is not to rerun the code generator, but to create the required capabilities manually and add the appropriate entries to the existing `services.xml` and `muse.xml` files. This approach was used to create all the resource types in the prototype context source system. We describe these resource types in the following sections.

#### 5.1.2.1 CONTEXT-SOURCE FACTORY

The context source factory is represented by a Muse resource type named `MainInterface`, which functions as an entry point into the context-source system. In the motivating scenario `MainInterface` represents the shipping company's web service. It is invoked by the supplier's web service to create a `Shipment` (also a context source) from the supplier to the manufacturer. The business interaction between these three entities is detailed in Figure 34.

`MainInterface` is a WS-Resource factory. There is only one instance of a `MainInterface`, and that is created when Muse starts up. Since it is a singleton, it was decided that its WSA EPR would not include any reference parameters. As a result, all SOAP messages directed to it only have the mandatory WSA `Action` and `Address` header elements. `MainInterface` exposes a custom WSDL with a single operation: `TrackShipment`. When this operation is invoked the `MainInterface` instance creates a new `ShipmentTracker` resource instance to represent the `Shipment`. It then returns the EPR of this newly created resource instance and the notification topics representing its context types to the invoking web service. As shown in Figure 35, this resource type is implemented using a single custom capability. It uses a custom serializer to generate the XML for its web service response messages.

#### 5.1.2.2 CONTEXT SOURCE

A Muse resource type named `ShipmentTracker` acts as the context source. `ShipmentTracker` is a WS-Resource: it exposes a set of resource properties in a WS Resource Properties Document, and each instance is addressed using a unique WSA EPR. Each `ShipmentTracker` represents a real-world context source – a shipment – in the scenario and is created on each non-faulting invocation of the `TrackShipment` operation on `MainInterface`. There is only one `ShipmentTracker` per shipment, and each `ShipmentTracker` represents only one shipment.

```

<wsdl:types>
. . .
<!-- Resource Properties -->
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://delivery.company/shipping/Shipment">

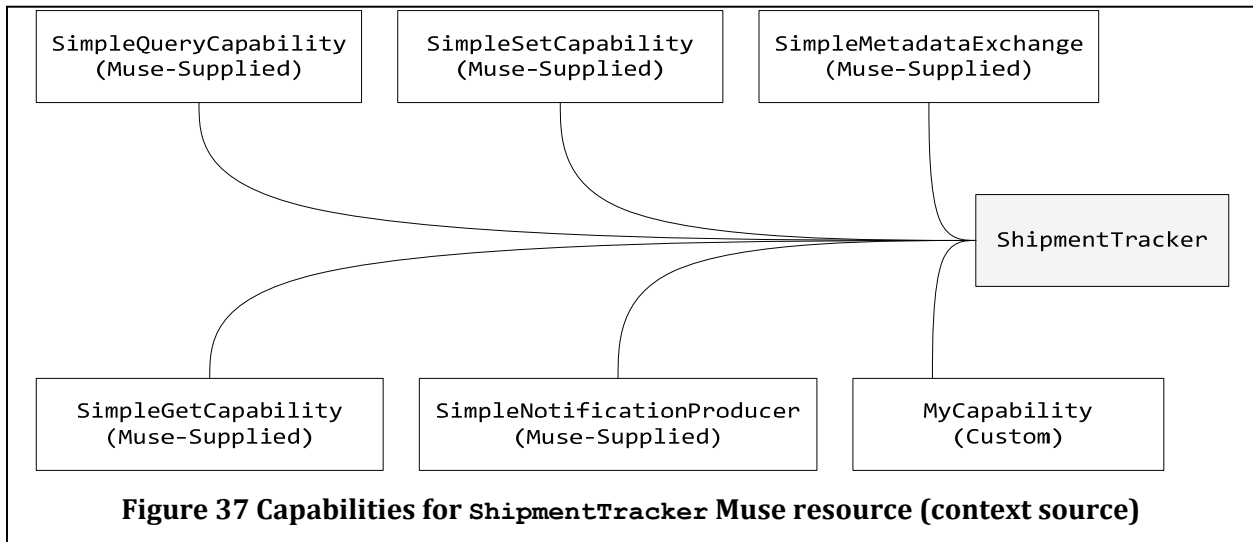
  <xsd:element name="ShipmentCondition" type="xsd:string"/>

  <xsd:element name="ShippingResourceProperties">
    <xsd:complexType>
      <xsd:sequence>
        <!-- Required for NotificationProducers by WS-Notification spec -->
        <xsd:element ref="wsnt:FixedTopicSet"/>
        <xsd:element ref="wst:TopicSet" minOccurs="0"/>
        <xsd:element ref="wsnt:TopicExpression" minOccurs="0"
          maxOccurs="unbounded"/>
        <xsd:element ref="wsnt:TopicExpressionDialect" minOccurs="0"
          maxOccurs="unbounded"/>
        <xsd:element ref="ship:ShipmentCondition" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</wsdl:types>

```

**Figure 36 WS Resource Properties Document for ShipmentTracker**

The resource properties contained by the `ShipmentTracker` WS-Resource represent the context types it offers. We have defined only a single resource property, a `ShipmentCondition`, of type `xsd:string`. A `ShipmentTracker` also includes additional metadata about this resource property. This is done by adding links to a WS-Resource Metadata Descriptor to its port type using the `wsrmd:Descriptor` and `wsrmd:DescriptorLocation` attributes. This descriptor allows programmers to provide extra information about each resource property in a WS Resource Property Document. It can include mutability constraints, valid values for each property, its initialization value, etc. Using a WSRMD allows the programmers of the invoking web service to understand which resource properties can be changed, what changes are valid, what the WS-Resource's default state is, etc. In the prototype, the WS-Resource Metadata Descriptor is only used to set the initial value for `ShipmentCondition`. A `ShipmentTracker` also includes four other resource properties: `TopicSet`, `FixedTopicSet`, `TopicExpression`, and `TopicDialect`. These properties are included because `ShipmentTracker` acts as a WSN notification producer, and the WSN specification states that notification producers must provide the `TopicExpression`, `FixedTopicSet`, and `TopicDialect` properties. `TopicSet` is included because the resource could not be generated



**Figure 37 Capabilities for `ShipmentTracker` Muse resource (context source)**

without it. The full WS Resource Properties Document for `ShipmentTracker` is shown in Figure 36.

The `ShipmentTracker` WS-Resource exposes a custom WSDL interface with eight operations. One, `GetResourceProperty`, must be provided by every WS-Resource (Section 5, WS Resource Properties specification [46]). Two, `Subscribe` and `GetCurrentMessage` must be provided by every WSN notification producer (Section 4, WS-BaseNotification specification [38]). The other five are optional WS-Resource-Properties and WSRMD operations: `GetResourcePropertyDocument`, `GetMultipleResourceProperties`, `QueryResourceProperties`, `SetResourceProperties` and `GetMetadata`. We have included these because Muse supplies capabilities implementing these operations, so there is no developer overhead incurred in including them. In addition, `SetResourceProperties` is required by the test framework to change the value of the `ShipmentCondition` context type. Now, a WS-Resource is not limited to these operations – developers can add custom operations for application-specific behavior. A `ShipmentTracker` could, for example, include a `CompleteDelivery` operation that automatically changes the value of `ShipmentCondition` from `Undelivered` to `Delivered`.

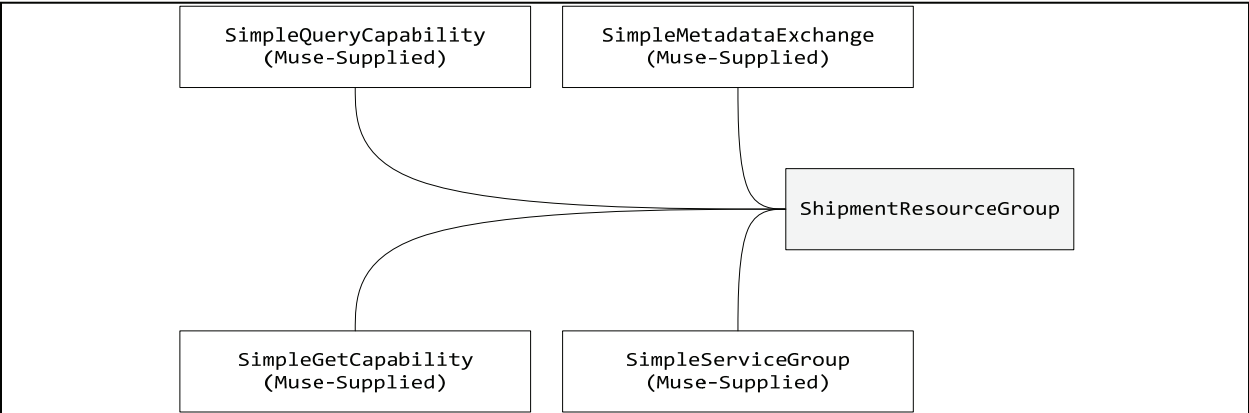
As shown in Figure 37, the `ShipmentTracker` resource type is implemented using five Muse-supplied capabilities that implement the WSRF, WSN and WSRMD operations, and a single custom capability. The custom capability manages the state and lifecycle of the `ShipmentCondition` resource property.

Each `ShipmentTracker` instance acts as a WSN notification producer. It generates a notification when the value of `ShipmentCondition` changes. This notification message is automatically created and sent by the Muse-supplied `SimpleNotificationProducer` capability, which listens for changes to the properties of the resource type it is included in. When an external entity wants to subscribe for notifications from a `ShipmentTracker` it invokes the `Subscribe` operation on a specific `ShipmentTracker` instance using an EPR (received from `MainInterface` or an intermediate web service). `ShipmentTracker` is the only resource type that is persisted when the context source system restarts.

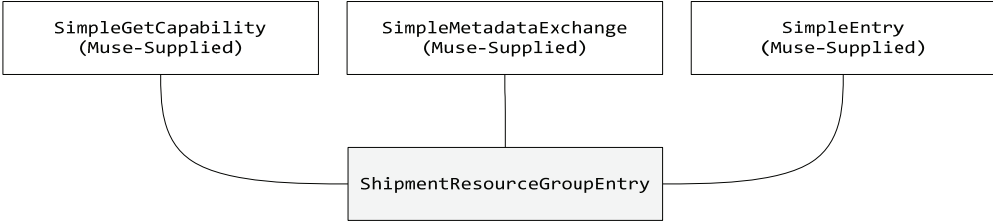
### 5.1.2.3 SERVICEGROUP AND SERVICEGROUPENTRY

A designer can get away with minimal organization in a small context-source system with a single context-source type. In systems with multiple context sources or context-source types it is useful to have flexible organizational structures that can model a variety of grouping requirements. Administrators in context-source systems may want to group instances by type, by the value of their resource properties, or by invoking web service. This compartmentalization can simplify administrative tasks or facilitate external interactions. For example, the supplier web service in the motivating scenario may want to query the shipping company for all outstanding shipments and their delivery dates. In an unorganized context-source system, such a query could be expensive, but in a system where grouping is supported the query would be substantially faster and less resource intensive.

WSRF provides the `ServiceGroup` and `ServiceGroupEntry` WS-Resources for organization, and the prototype uses both. The Muse resource types corresponding to these two WS-Resources are `ShipmentResourceGroup` and `ShipmentResourceGroupEntry`. A `ServiceGroup` represents a collection of member WS-Resources whose membership in the group is constrained by the value of its `MembershipContentRule` resource property. `ShipmentResourceGroup` uses a WSRMD file to specify its constraints. In this file a `MembershipContentRule/@ContentElement` indicates that members of the `ShipmentResourceGroup` must have a namespace-qualified resource property named `ShipmentCondition`. As before, this WSRMD file is linked to the port type of `ShipmentResourceGroup` using the `wsrmd:descriptor` and `wsrmd:descriptorLocation` attributes. `ShipmentResourceGroup` exposes a WSRF-defined `ServiceGroup` WSDL interface with five op-



**Figure 38 Capabilities for the ShipmentResourceGroup Muse resource**



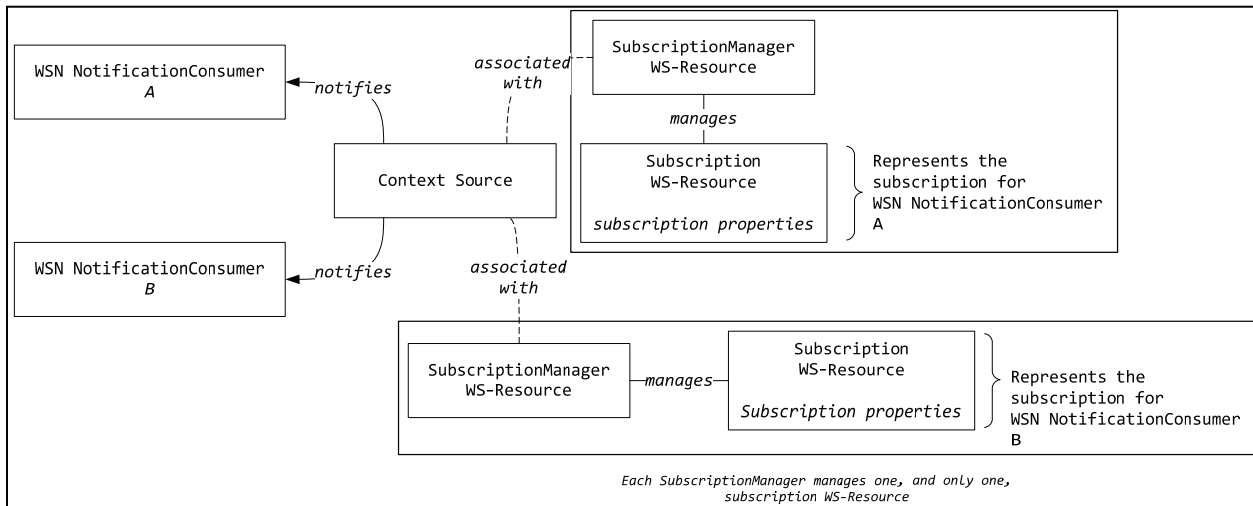
**Figure 39 Capabilities for the ShipmentResourceGroupEntry Muse resource**

erations; the only changes made to this interface were a change in its name, and adding a link to the WSRMD file in its port type.

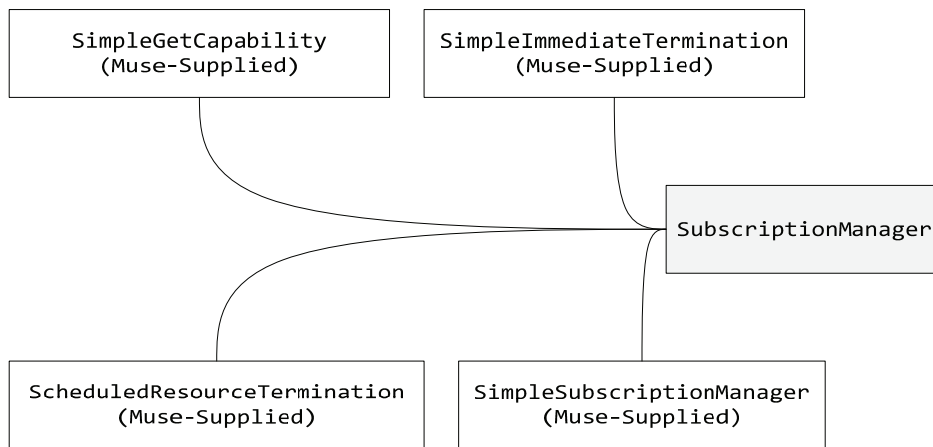
As shown in Figure 38, `ShipmentResourceGroup` is implemented as a Muse resource type using four Muse-supplied capabilities. No custom capabilities were used because no custom operations were defined. An example of a custom operation that could have been included is `CheckOutstandingShipments`, which would return each outstanding shipments and its associated status to the supplier.

A WS-Resource’s membership in a `ServiceGroup` is represented by a `ServiceGroupEntry`. Like `ServiceGroup`, it too is a WS-Resource. In the prototype `ShipmentResourceGroupEntry` exposes a WSRF-defined `ServiceGroupEntry` WSDL interface with five operations; only its name was changed. `ServiceGroupEntry` (Figure 39) is implemented as a Muse resource type that uses three Muse-supplied capabilities.

In the prototype the intermediate web service and WS-BPEL engine interact only with the context source and context-source factory, not with the `ServiceGroup` or `ServiceGroupEntry`. This reflects the simplified nature of the scenario and the prototype. Nevertheless, these resource types



**Figure 40 Relationship between NotificationConsumer, context source (NotificationProducer), SubscriptionManager and Subscription WS-Resource**



**Figure 41 Capabilities for SubscriptionManager Muse resource**

are included to show how they fit into a context-source system, and we expect them to be used extensively in more full-featured scenarios.

#### 5.1.2.4 SUBSCRIPTION MANAGER

When a subscriber invokes the `Subscribe` operation on a WSN notification producer a subscription resource is created. The EPR of this resource is returned to the subscriber along with the EPR of a `SubscriptionManager`, an entity for managing subscription resources. A `SubscriptionManager` allows external entities to manipulate their subscriptions – for example, terminating or pausing them. When a programmer creates a WSN notification producer WS-Resource in Muse

the code-generator automatically creates an associated `SubscriptionManager` resource type; no work is required on the developer's part.

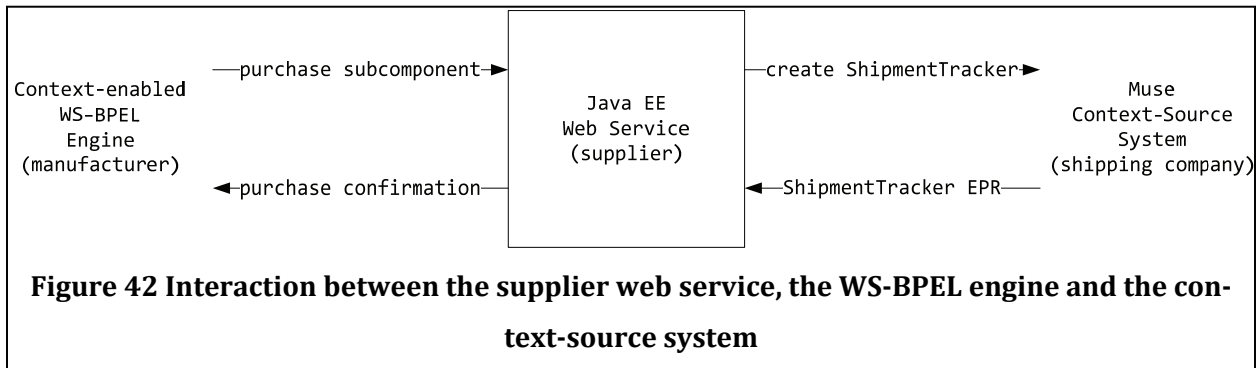
The relationship between `SubscriptionManager` instances and subscription resources is implementation specific. For example, the framework could use a single `SubscriptionManager` instance to manage all subscription resources, regardless of which notification-producer instance or notification-producer resource type with which they are associated. Alternatively, it could use a single `SubscriptionManager` per notification-producer resource type. Muse uses a single `SubscriptionManager` instance per subscription resource. To clarify: when a subscriber invokes the `Subscribe` operation on a notification-producer instance, a subscription resource is created. Simultaneously, a new `SubscriptionManager` instance is created to manage this subscription resource. The EPRs of both the subscription resource and the `SubscriptionManager` instance are then returned to the subscriber. Invoking `Subscribe` again on another notification-producer instance (or even the same instance) results in the creation of a new subscription resource and a new `SubscriptionManager` instance. This is shown in Figure 40. This means that in Muse, each `SubscriptionManager` instance manages only a single subscription resource.

Muse's implementation of the `SubscriptionManager` resource type is unusual. Since it represents subscription resources as WS-Resources, the `SubscriptionManager` resource-type's WSDL interface does not implement the operations defined in either the WSN-defined `SubscriptionManager` or `PausableSubscriptionManager` port types. Instead, a `SubscriptionManager` instance exposes the standard WSRF resource-properties operations (`GetResourceProperty`, etc.) so that external entities can retrieve the subscription WS-Resource it manages, and manipulate or terminate the subscription directly. The `SubscriptionManager` resource type is implemented using four Muse-supplied capabilities, as shown in Figure 41.

### 5.1.3 DEVELOPMENT IMPRESSIONS

The WSRF-based context-source system above is not the most minimal, but it contains all the resources needed to support a range of requirements. In building the system the goal was to understand how much developer effort was required to get a single context source, along with its supporting entities, up and running. Notable challenges included:





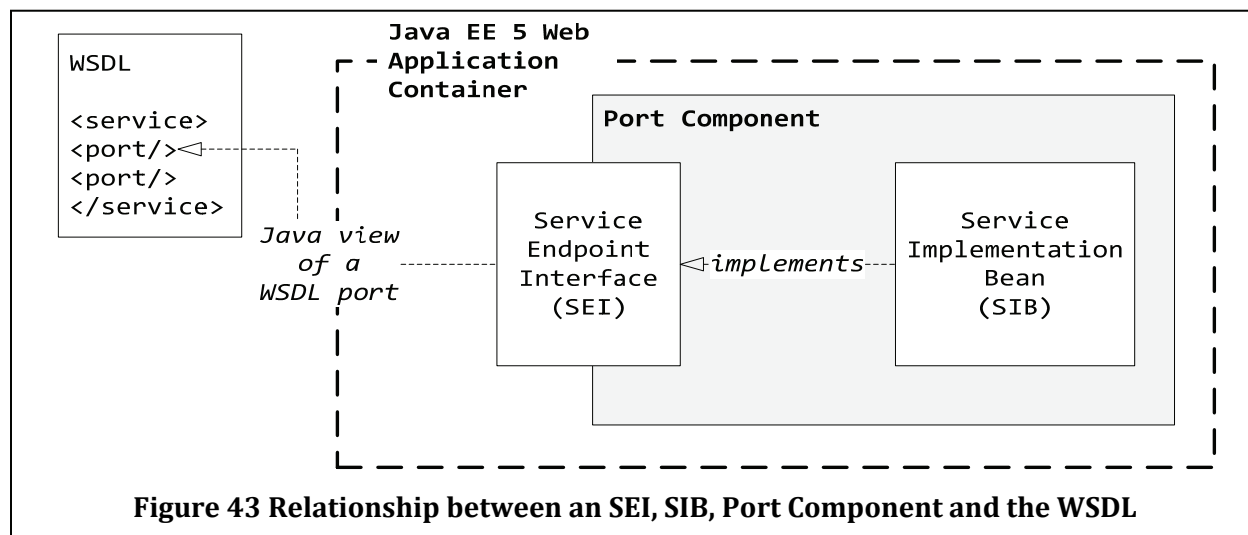
- Understanding how WSA Action headers influenced routing to a capability's methods, and changing the code and deployment artifacts to reflect this
- Composing a WSDL interface for a Muse resource-type. Including the correct XML schemas, schema references, *etc.*, was far more involved than expected
- Maintaining the mapping between the `muse.xml` deployment descriptor, the WSDL interface, and the Java capabilities

The Muse programming model is relatively straightforward and did not pose many difficulties in implementing the relatively simple system above. It is possible that in a configuration with a more constraints – for example, a single WS-Resource that represented multiple real-world resources, or a WS-Resource that composed multiple pre-defined schema and WSDL operations – that this would no longer be the case.

## 5.2 INTERMEDIATE WEB SERVICE

While some business processes will interact with context sources directly, most will receive references to context sources through intermediate web services. In fact, it is this configuration – the manufacturer receiving a reference to a shipment from its supplier's web service – that is used in the motivating scenario. There, an intermediate web service plays the role of the supplier. It receives an order from the manufacturer, contacts the shipping company to create a shipment and its associated context source, and returns a reference to this context source to the manufacturer. The interaction between these high-level components is shown in Figure 42.

In Section 4.2 we describe how `xsd:extension` can be used to add a mapping between the response-message parameters, the context-source EPR, and context-type WS-Topic in such a way that any WS-BPEL engine could process it. Of course, many designs function well in theory but



fail in practice. To test the feasibility of this method, we implemented the supplier web service using Java EE 5, JAXB 2.x and JAX-WS 2.x.

Java EE 5 (henceforth Java EE) is the latest in a line of Java enterprise-centric middleware frameworks. It is one of the broadest and oldest enterprise frameworks, and its predecessors (J2EE 1.4, J2EE 1.2, etc.) have been used in many organizations for a wide range of applications. As a result, there is a considerable developer experience and legacy code tied up in it. Java EE is also supported by most major software vendors and has a healthy open-source community surrounding it. These factors make it likely that Java EE (or some subset of it) will continue to be heavily used in the near future. The following sections detail the basics of the JAX-WS 2.x and JAXB 2.x programming model, and describe the various implementation components of the prototype web-service.

### 5.2.1 PROGRAMMING MODEL

Java EE is a large framework comprising a number of specifications [10, 55-60]. It includes solutions for a number of application areas such as persistence, transactions, web-application development, web-service development, etc. The prototype only uses a subset of the web services technologies provided by the framework. It would be impossible for this thesis to detail all the inner workings of Java EE's web services support, so only a broad outline is provided. To implement the prototype an understanding of the following specifications is necessary:

- JSR 224: The Java API for XML-Based Web Services (JAX-WS)

- JSR 222: The Java Architecture for XML Binding (JAXB)
- JSR 181: Web Services Metadata for the Java Platform (WS-Metadata)

In addition, some knowledge of JSR 67: SOAP with Attachments API for Java (SAAJ) is also required. It is not crucial to have in-depth knowledge of the all the above JSRs – in fact it is unlikely that any developer will require all the capabilities they provide. At minimum, however, they need to know:

- The development process for implementing web services in Java EE
- What artifacts (code, WSDL interfaces, XML schemas, etc.) are required
- How SOAP messages are mapped to Java methods
- The request/response framework
- The basics of how XML<->Java data binding works

In its simplest, most automated configuration, a JAX-WS web service consists of a Plain Old Java Object (POJO) annotated with `@WebService`, with one or more of its public methods annotated with `@WebMethod`. An example of such a service follows:

```
package example;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class StudentFunctions {

    @WebMethod
    public String getGreeting(String name) {
        return "Hello" + name;
    }
}
```

It is also possible to implement a web service as a stateless EJB; this approach was tried in the prototype but could not be made to work, as a result of a bug in the implementation or a subtlety of which we are unaware.

When the web service is packaged appropriately – for example, in a web archive (`.war`) – and loaded into a Java-EE-compliant application server like Glassfish 2.0, the Web-Services-for-Java-EE container (web-service container) generates additional code artifacts, a WSDL interface for the service, and then waits for incoming requests.

A Java EE web service exists as a port component in a container. There is a different port component for each port in a WSDL interface. This port component consists of a Service Endpoint Interface (SEI) and an `@WebService`-annotated Service Implementation Bean (SIB). The SEI is a Java mapping of a WSDL port type that is generated using the JAX-WS rules for WSDL<->Java mapping. It defines the methods implemented by an underlying SIB. This SIB contains the business logic of the web service. It has the same methods as those in the SEI but does not have to implement (the Java `implement` keyword) the SEI. The relationship between WSDL, port component, SEI and SIB is shown in Figure 43. When developing and packaging a Java EE web service, only the SIB is required. Including the WSDL and SEI is optional, since the container will generate the missing artifacts using annotations in the code and the default binding and mapping rules. Section 5 of JSR 109 has a complete overview of the server-side view of Java EE web services [60].

The mapping between WSDL operations and Java methods, and XML Schema types and Java objects/types, is governed by the JAX-WS WSDL<->Java mapping and JAXB XML<->Java binding rules, respectively. These rules are detailed in JSR-224 [55], JSR-222 [56, 57], and JSR-181 [10]. A programmer has some flexibility in how these rules are applied through the use of annotations, custom binding files and extension elements in a WSDL. When a web-service operation is invoked, or a web-service response needs to be sent, the container's JAX-WS and JAXB runtimes use the annotations and default binding rules to choose which method in a port component to call, how to serialize/deserialize XML/Java, etc.

The Java EE toolchain makes extensive use of code generation. There are three major approaches to web-service development:

- Start from WSDL: The developer has a pre-defined WSDL and needs to write Java code to implement its operations
- Start from Java: This is the approach shown in the example above, and is the one used in most tutorials. The developer starts with the implementation objects and annotates them as necessary. The WSDL interface is generated automatically
- Meet in the middle: The developer has existing Java code, but needs to conform to a pre-defined WSDL. This is, by far, the hardest approach and involves a lot of iterative experimenting, tweaking annotations and code generation

```

<wsdl:definitions . . .>
  <wsdl:types>
    . . .
    <xsd:schema>
      . . .
      <xsd:element name="PurchaseItem" type="tns:PurchaseItem" />

      <xsd:complexType name="PurchaseItem">
        <xsd:sequence>
          <xsd:element name="ItemCode" type="xsd:string"
            minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  . . .
  <wsdl:message name="PurchaseItemRequest">
    <wsdl:part element="tns:PurchaseItem" name="wrapper" />
  </wsdl:message>
  . . .
</wsdl:definitions>

```

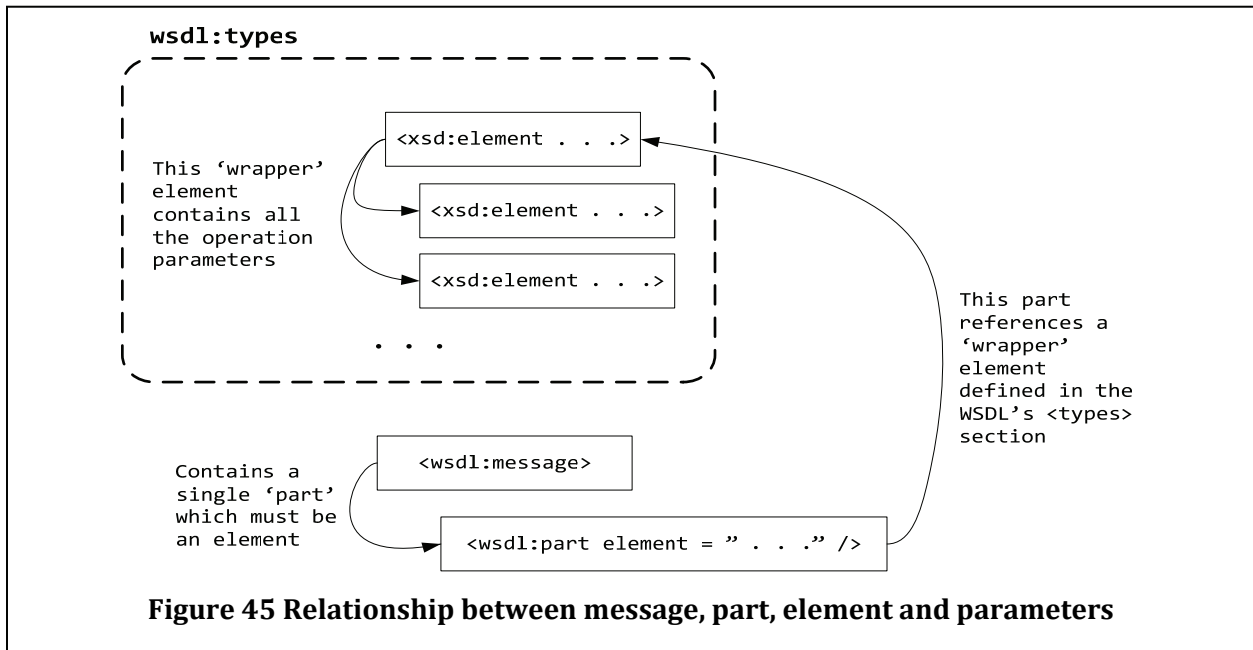
**Figure 44 Request-message structure for purchaseItem operation**

The prototype used the “Start from WSDL” approach with some constraints on the generated SEI method signatures.

## 5.2.2 INTERFACE

Java-EE web services are advertised using WSDL interfaces. Although it is possible to start from Java and have the container generate a WSDL automatically, this approach was not ideal because:

1. We were not fluent in the JAX-WS/JAXB mapping and binding rules, so were unsure what the generated WSDL would look like
2. Changing the Java method signatures and data-type structures so that the generated WSDL would meet our requirements would be a tedious, iterative and error-prone task
3. We had very specific requirements for the structure of the return messages (the use of `xsd:extension`, for example)
4. Our context parameters made use of XML-Schema-defined types, like the `WSN TopicType`, which had no Java equivalent. It would have been hard to quickly write mapping classes for these pre-defined XML types
5. In the scenario our supplier (the Java-EE web service) interacts with an ordering WS-BPEL business process. Writing this process would be simpler if we had a known WSDL with which to work



These factors made us choose the “Start from WSDL” approach. With this approach, a pre-existing WSDL is fed into a tool like `wsimport`, which generates the corresponding SEI and Java data structures. As noted above, the design of the WSDL interface has a major impact on how the generated code looks – from the names of operations, to the data types and method signatures. In creating a WSDL interface the developer must first decide what WSDL style they will use. The choice of style determines how a web service maps SOAP messages to operations. There are three alternatives: `rpc/literal`, `document/literal` and `document/literal wrapped`, and their details can be found in Section 3 of the WSDL specification [67]. As mentioned in Section 4.2, for interoperability reasons this thesis only considers those web services that conform to the WS-I Basic Profile. This necessitates the use of the `document/literal-wrapped` WSDL style for the supplier web service. Briefly, this binding style requires that:

1. Messages carry data that conforms to the XML-Schema types defined in a WSDL's `<types>` section
2. The parameters in a SOAP message are grouped under a single 'wrapper' element that is the only direct child of the SOAP body

The use of `document/literal wrapped` also brings into play other conventions that must not be forgotten since they affect the Java artifacts generated during the WSDL->Java mapping phase.

Using document/literal wrapped has its advantages and disadvantages. While it allows schema-type validation, it also makes both the WSDL and the SOAP messages very verbose. For a novice developer it can be hard, when designing the schema and WSDL, to keep in mind which element is the wrapper, the contained type, etc.

The supplier web service has a single port type named `Purchase` with only one operation: `PurchaseItem`. This operation's request message has the structure show in Figure 44. Some of the document/literal wrapped conventions come into play here. The message can contain only a single part which must be an element, not a type (Section 2.3.1.2, JAX-WS specification [55]). This element is the wrapper element and must have a local name that corresponds to the name of the operation it is used in (Section 2.3.1.2 (ii), JAX-WS specification [55]) – so, `PurchaseItem`. This element is defined using XML Schema in the WSDL interface's `<types>` section, and it contains the actual parameters for the operation. These parameters are, in turn, mapped to Java method parameters in the SEI. The relationship between message, part, element, and parameters is shown in Figure 45.

Creating the WSDL entries for the operation's return types was more involved. `PurchaseItem` had to return two types of messages: one with a snapshot of relevant properties from the `ShipmentTracker` context source (`BaseReturn`), and another that included the context parameters as well (`ExtendedReturn`). `BaseReturn` was defined as a complex type with the following schema:

```
<xsd:complexType name="BaseReturn">
  <xsd:sequence>
    <xsd:element name="ItemCode" type="xsd:string" />
    <xsd:element name="PurchaseStatus" type="xsd:string" />
    <xsd:element name="ShipmentStatus" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
```

Extended return included these elements as well as the context parameters shown in Figure 24. It used the `xsd:extension` element and had the following schema:

```

<xsd:complexType name="ExtendedReturn">
  <xsd:complexContent>
    <xsd:extension base="tns:BaseReturn">
      <xsd:sequence>
        <xsd:element name="Context" maxOccurs="unbounded" minOccurs="1"
          type="tns:ContextParameter" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

Remember that the document/literal-wrapped WSDL style stipulates that a message part refer to a wrapper element, not an XML-Schema type. This requirement made it impossible to simply reference the `BaseReturn` type above. Instead, a three-layer hierarchy was used. First, a wrapper element named `PurchaseItemResponse` was created. It contained a single element of type `BaseReturn`. Using `xsd:extension` a new `ExtendedReturn` type was derived from `BaseReturn`, allowing the response message to contain either type. The schema for `PurchaseItemResponse` is:

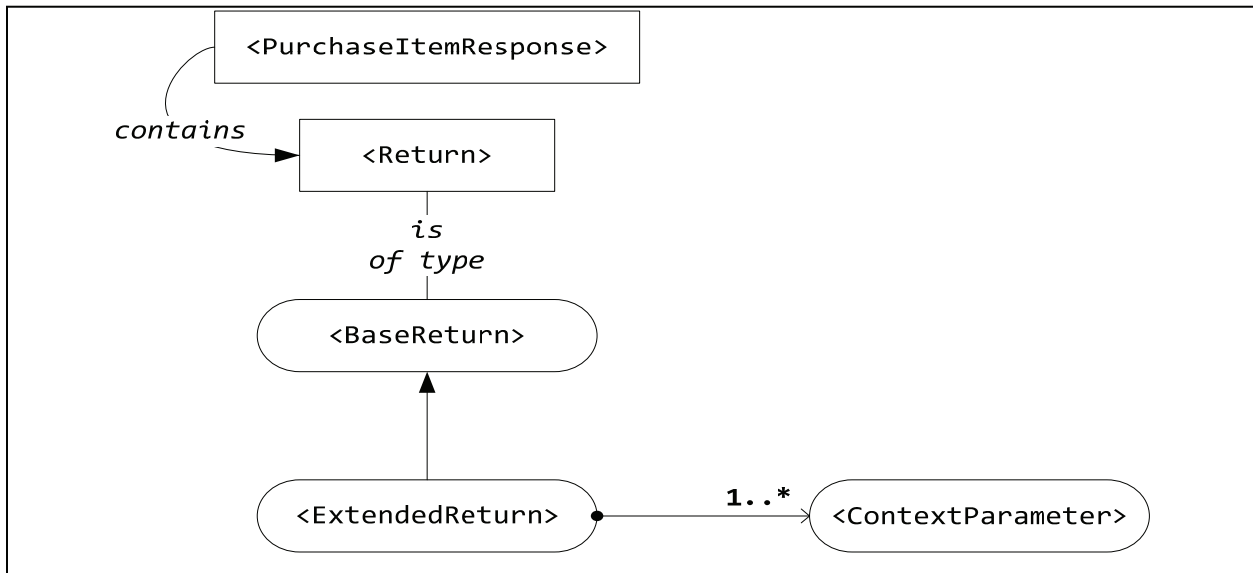
```

<xsd:element name="PurchaseItemResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Return" type="tns:BaseReturn" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

The hierarchy formed by the wrapper element and the two return types is shown in Figure 46.





**Figure 46** Type hierarchy for the intermediate web service's response message

One final consideration had to be made in defining the schema for `PurchaseItemResponse`. The element of type `BaseReturn` was originally named `ResponseContents`. When the WSDL interface incorporating this schema was fed into `wsimport` it generated an SEI with Java method signatures with the following form:

```
public void purchaseItem(Holder<BaseReturn> ResponseContents, String itemCode);
```

Note that `PurchaseItem` has a `void` return type and uses `Holder<T>` parameters, where `<T>` is the class hierarchy created by `BaseReturn` and `ExtendedReturn`. Using a `void` return is not standard Java practice if a method has only one return type. After investigation it was discovered that Section 2.3.2 of the JAX-WS specification states:

If there is a single out wrapper child then it forms the method return type, if there is an out wrapper child with a local name of “return” then it forms the method return type, otherwise the return type is void. [55]

As a result, we changed the name of the contained element from `ResponseContents` to `Return`, as shown in the following schema fragment:

```

<!-- Before -->
<xsd:element name="PurchaseItemResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="ResponseContents" type="tns:BaseReturn" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- After -->
<xsd:element name="PurchaseItemResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Return" type="tns:BaseReturn" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

This resulted in the following Java method signature:

```
public BaseReturn purchaseItem(String itemCode);
```

Once the supplier's schema and WSDL interface was created, and acceptable SEI and data structures for the web service created, all that was left was its implementation.

### 5.2.3 IMPLEMENTATION

This web service was implemented as a single SIB that referenced an SEI with a single operation. Since the web service needed to support WS-Addressing and the XML-Schema `xsd:extension` element, it needed to use the JAX-WS 2.1.x Reference Implementation (RI) and the JAXB 2.1.x RI. The 2.0.x versions of these RIs did not support either WS-Addressing or XML-Schema type extension.

Although using an SEI is optional, including it has many advantages. It allows a developer to:

1. Follow standard practice by separating interface from implementation
2. Separate most of the annotations concerning method behavior, JAXB data binding behavior, etc. from the web service's implementation code
3. Better control the public interface of the web service

Using an SEI however, has its own idiosyncrasies. First, the SIB's `@WebService/endpointInterface` annotation must reference the SEI's fully-qualified name even though the interface and class are linked through the use of the Java `implements` keyword.

Second, although the SEI may have been generated from a pre-existing WSDL interface, it does not reference this WSDL interface by default. This means that when the web service's WSDL is requested, it will not return the hand-created one used to generate the SEI. Instead, it will return a congruent WSDL that is auto-generated by the web service runtime. Not linking the SEI to the hand-created WSDL caused unexpected behavior with the prototype web service – a situation in which the web service .war could not be loaded into the container. It was fixed by annotating the SEI with `@WebService/wsdlLocation` and setting it to the location of the pre-existing WSDL:

```
package ca.uwaterloo.ece.aag.ecommercestore.purchase;

import . . .

@WebService(name = "Purchase",
    targetNamespace = "http://ece.uwaterloo.ca/aag/ecommercestore/purchase",
    wsdlLocation = "WEB-INF/wsdl/Purchase.wsdl"
)

@Addressing()
@XmlSeeAlso({
    ca.uwaterloo.ece.aag.ecommercestore.purchase.ObjectFactory.class,
    org.oasis_open.docs.wsn.t_1.ObjectFactory.class
})
public interface Purchase {
    . . .
}
```

We also added the `@Addressing` annotation to the SEI. This is because Muse uses WSA headers to route SOAP messages, not the the `SOAPAction` HTTP header field.

As shown in Figure 42 the supplier web service in the prototype has to interact with both the context source system and the invoking WS-BPEL process. Its implementation can be split along those lines, with a different approach used for both.

### 5.2.3.1 INTERACTING WITH THE INVOKING SERVICE

The WS-BPEL process invokes the supplier by calling its `PurchaseItem` operation. The invocation request is sent in a SOAP message. The SIB implementing the supplier web service leverages the JAXB data-binding system in interactions with this WS-BPEL process. From a Java developer's perspective, using JAX-WS/JAXB is extremely convenient since it allows them to ignore the underlying use of XML, any issues associated with its parsing and processing, etc. In fact, they deal

only with Java data types. For example, the `BaseReturn` complex type is translated into the following Java class:

```
package ca.uwaterloo.ece.aag.ecommercestore.purchase;

import . . .

public class BaseReturn {
    protected String itemCode;
    protected String purchaseStatus;
    protected String shipmentStatus;

    public String getItemCode() {
        return itemCode;
    }

    public void setItemCode(String value) {
        this.itemCode = value;
    }

    public String getPurchaseStatus() {
        return purchaseStatus;
    }

    public void setPurchaseStatus(String value) {
        this.purchaseStatus = value;
    }

    public String getShipmentStatus() {
        return shipmentStatus;
    }

    public void setShipmentStatus(String value) {
        this.shipmentStatus = value;
    }
}
```

Note that the code snippet above does not include all the annotations added by JAXB during code generation. These annotations are required, and classes used in the prototype contain them.

The invoked `PurchaseItem` operation returns a snapshot of the `Shipment` parameters. Since JAX-WS/JAXB is used, web service responses are simple: the developer simply returns an instance of the `BaseReturn` type in the standard way:

```
public class PurchaseService implements Purchase {
    public BaseReturn purchaseItem(String itemCode) {
        . . .
        return new BaseReturn;
    }
}
```

Correspondingly, to return the `ExtendedReturn` XML type, the developer simply creates an instance of the `ExtendedReturn` Java class (which corresponds to its namesake in the WSDL) and returns that instead:

```
public class PurchaseService implements Purchase {
    public BaseReturn purchaseItem(String itemCode) {
        . . .
        return new ExtendedReturn;
    }
}
```

XML serialization, matching the response up to the appropriate operation, and other logistical tasks are handled by the JAXB and JAX-WS runtimes. This is the closest-to-ideal Java web-service development experience, since the developer does not have to care about the underlying XML mechanics.

### 5.2.3.2 INTERACTING WITH THE CONTEXT-SOURCE SYSTEM

The supplier SIB interacts with the context source system through the `MainInterface` `Muse` resource type. It invokes the `MainInterface` resource's `TrackShipment` operation, receiving a reference to the newly-created `ShipmentTracker` context source and a list of its notification topics in return. The structure of the response message is as follows:

```
<xsd:element name="TrackingResponse">
  <xsd:element name="ShipmentDetails">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ShipmentUpdateService"
          type="wsa:EndpointReferenceType"/>
        <xsd:element name="ShipmentNotificationTopics" type="tns:TopicList"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:element>
```

When the SIB requests a `ShipmentTracker` from `MainInterface` it acts as a web-service client. A Java-EE web-service client can use the `Service` abstraction, detailed in Section 4 of the JAX-WS specification, to make requests, and receive responses from, web-service providers [55]. Using this approach involves leveraging JAX-WS/JAXB to create Java interfaces and data-types from the WSDL interface of the invoked web service. This, however, poses a problem in dealing with `Muse` resources. Note that the `Muse` code generator's default behavior is to merge the all the

referenced schemas into a single WSDL interface. Feeding this merged WSDL interface into the JAX-WS/JAXB code generator would result in hundreds of Java data types being created – most unnecessary. Moreover, it was unclear if the JAXB data binding rules would work for all the schema types or, even if they did, whether sensible Java versions of the XML schema types would be generated. Given the simplicity of the `TrackShipment` message schema and the potential for error when using JAXB, working directly with the request/response XML seemed the safer option.

JAX-WS allows web-service clients to use the low-level `Dispatch` API to interact with clients. This API allows programmers to work at the XML level and deal with protocol-specific message structures. With this flexibility, however, comes the requirement that programmers fully understand what the request/response message and payload structure look like. The SIB uses the `SOAPMessage` version of the `Dispatch` API – `Dispatch<SOAPMessage>` – with WS-Addressing support. `Dispatch<SOAPMessage>` allows a SOAP message – both body and headers – to be constructed from scratch. The prototype implementation uses SAAJ to construct the request message. Although there was some time lost in learning another API, the effort was worth it: the resulting implementation was lightweight in comparison to the alternative. Note that this was, at least in part, because of the simple message schema for the `TrackShipment` operation. Using the full capabilities of JAXB and JAX-WS would be a better move if the service provider had a large WSDL interface with multiple operations and a complex XML schema.

The response message from `TrackShipment` was parsed using XML Document Object Model (DOM) and XML Path Language (XPath). The content of the SOAP body was extracted as a DOM tree, and the notification topics and context-source EPR retrieved using XPath expressions:

```
<!-- Prefix assignment -->
xmlns:mainInt="http://delivery.company/MainInterface"

<!-- Expression to retrieve the EPR -->
//mainInt:ShipmentUpdateService

<!-- Expression to retrieve the notification topics -->
//mainInt:Topic
```

Note that using XPath required creating a `NamespaceContext` object that mapped namespace prefixes to URIs, since without it expression matching failed. After these notification topics and

context-source EPRs were retrieved, they were inserted into JAXB data types and returned to the invoking WS-BPEL service.

#### 5.2.4 DEVELOPMENT IMPRESSIONS

Developing web services using Java EE was a mixed experience. While the amount of Java code that to be written was fairly low, getting to the point where one could write all that code was quite involved. In developing the prototype service there were two major sources of frustration:

1. The effort required to create Java data types and method signatures with the structure we wanted. This often required close reading of the JAX-WS and JAXB specifications to understand which rules were being applied for a given circumstance and why. Note that it is entirely possible to have valid XML schema and WSDL interfaces, but be unable to generate Java implementations of the same
2. Trying to figure out which APIs (for XML processing, etc.) to use in the SIB implementation. While in many cases it was possible to use more than one API for a task, it was often unclear what the tradeoffs in choosing one API over the other were

Finally, the framework's heavy reliance on code generation complicated iterative development. Changes to the WSDL interface meant re-running the JAXB/JAX-WS mapping/binding tool, figuring out what classes had been added or modified, and updating the project piecemeal. Although one got used to the process, it was error prone and far from fluid.

#### 5.3 MODIFIED WS-BPEL PROCESS ENGINE

Neither Muse nor the Java EE framework had to be modified to support context sources and extended web-service responses. This is because the architecture outlined in Chapter 4 leverages existing constructs and functionality within WSRF and the WS-\* standards. The same cannot be said for a WS-BPEL engine that has to support context variables. Introducing context variables to WS-BPEL requires extending the WS-BPEL language. It also requires the engine to support additional messaging, implicit interaction with external entities, and other non-standard behavior. These requirements necessitate significant changes to an existing standards-compliant WS-BPEL engine. At a minimum, supporting context variables requires the following:

- Validating WS-BPEL process files with the context variable extension

- Generating runtime structures for the extended activities and variables
- Intercepting and retrieving context parameters from extended web service responses
- Creating code structures to receive notifications
- Subscribing to context sources
- Linking subscriptions for a context type to a specific process and variable
- Routing notifications to the recipient process
- Propagating changes in context-type values to the appropriate context variable

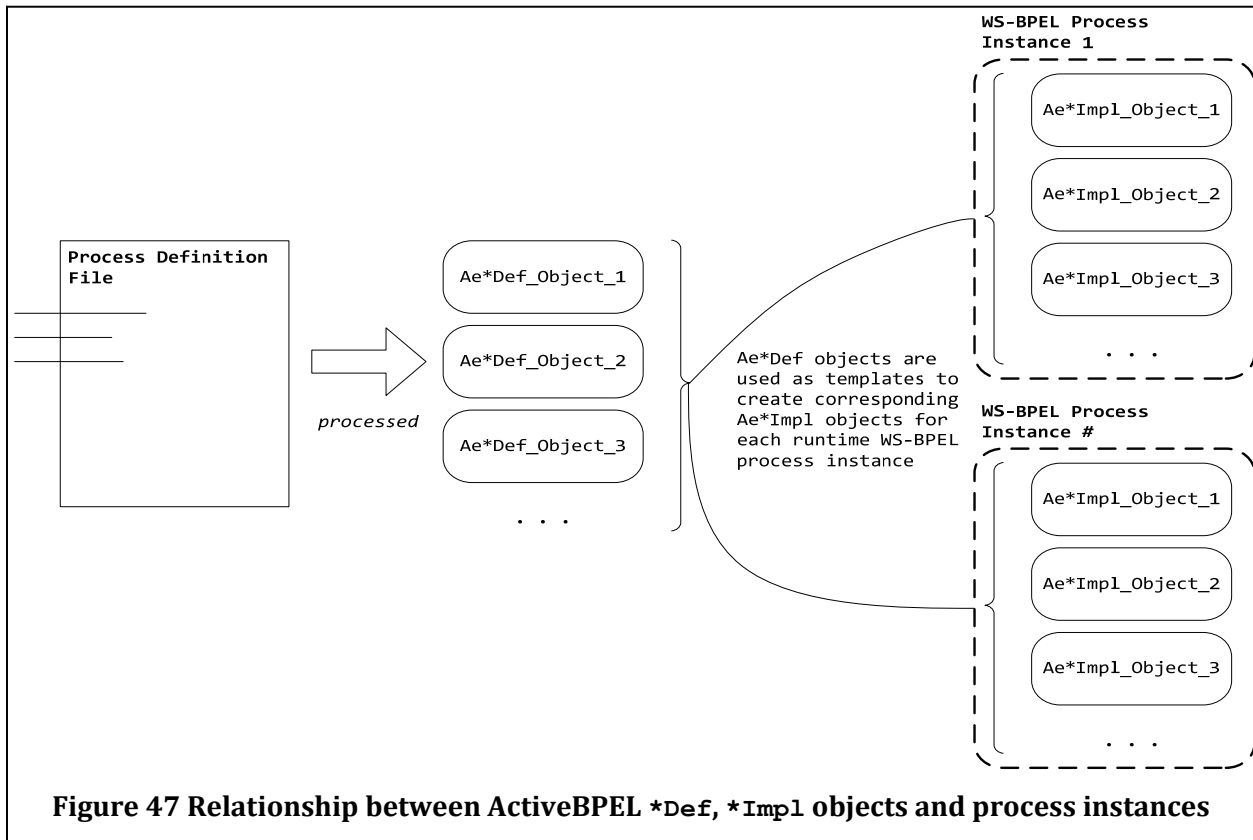
These changes are not implementation specific, and have to be made to any WS-BPEL engine that implements the context-variable extension described in this thesis.

### 5.3.1 ACTIVEBPEL 4.1

The open-source ActiveBPEL engine was modified for the prototype. Written in Java, ActiveBPEL 4.1 supports both BPEL4WS 1.1 and WS-BPEL 2.0. It is implemented using over 2300 classes and interfaces comprising several million lines of code. ActiveBPEL is meant to run as a web-application in a variety of popular servlet containers including Apache Tomcat 5.5.x, JBoss 4.0.5 and Websphere 6.1. The prototype configuration had the ActiveBPEL engine running as the `active-bpel` webapp on Apache Tomcat 5.5.25. Our modifications required changes and/or additions of over 10,000 lines of code.

When Tomcat loads the `active-bpel` webapp, a number of startup tasks take place. Key among them is the creation of a singleton ActiveBPEL engine (`AeBusinessProcessEngine`) that manages WS-BPEL process instances. The engine functions as the interface between a process and the external world, isolating it from the logistics of communication, message sequencing and routing, persistence, failure handling, etc. To create a WS-BPEL process the programmer first defines it in a `.bpe1` process definition file. The process definition is then packaged into a Business Process Archive (`.bpr`) along with ActiveBPEL-specific deployment artifacts. When this `.bpr` is deployed in the ActiveBPEL webapp the engine parses the process definitions it contains and creates a set of activity-definition objects (not to be confused with WS-BPEL activities) for each WS-BPEL process. There is an activity definition object for each WS-BPEL construct – for example, a WS-BPEL `<variable>` element has a corresponding `AeVariableDef`, a WS-BPEL `<invoke>` element has an `AeActivityInvokeDef`, a `<process>` element has an `AeProcessDef`, and





**Figure 47 Relationship between ActiveBPEL \*Def, \*Impl objects and process instances**

so on. Together, these activity-definition objects form the model of a WS-BPEL process. After all these objects are created the engine validates them, checking for unrecognized attributes and extensions. Validation failures abort `.bpr` deployment.

A WS-BPEL process instance is created when one of its start activities is triggered. When this occurs the engine treats the activity-definition object model as template and uses it to create a congruent activity-implementation object model. For example, returning to `<variable>`, `<invoke>`, and `<process>`, their corresponding implementation objects are `AeVariable`, `AeActivityInvokeImpl`, and `AeBusinessProcess`, respectively. While there is only one set of activity definition objects, there may be many sets of activity-implementation objects – one per process instance. This relationship is shown in Figure 47.

The engine itself does not directly handle tasks like messaging or persistence – instead, it delegates these other constructs in lower layers or to handlers. ActiveBPEL has a variety of these and the relevant ones will be discussed as necessary.

The modified ActiveBPEL engine was exercised using a simple WS-BPEL process with five activities and a single context variable. This process assumed the role of the manufacturer in the motivating scenario and was exposed through a `purchaseProcessLT` partner link type that had the following definition:

```
<wsdl:definitions>
  . . .
  <wsdl:portType name="InternalPurchase">
    <wsdl:operation name="issuePurchaseOrder">
      <wsdl:input message="tns:issuePurchaseOrderRequest" />
    </wsdl:operation>
  </wsdl:portType>

  <plnk:partnerLinkType name="purchaseProcessLT"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype">
    <plnk:role name="purchaseProcess" portType="tns:InternalPurchase" />
  </plnk:partnerLinkType>
</wsdl:definitions>
```

Only a single role, `purchaseProcess`, is defined, and this is assumed by the process itself.

The scenario process offers a single one-way operation: `issuePurchaseOrder`. When invoked by an external entity – the test framework in our case – the process invokes the supplier’s web service, receives context parameters in the response, subscribes for notifications from the relevant context sources, and then waits for incoming notifications. This is a concrete realization of the ideas expressed in Figure 16. In the following sections we describe how ActiveBPEL 4.1 was modified to run the scenario process. Note that all operation descriptions apply to ActiveBPEL’s processing of WS-BPEL 2.0 process definitions. Class names may differ when the engine deals with BPEL4WS 1.1 process definitions.

### 5.3.2 EXTENSION VALIDATION

The WS-BPEL specification requires using an `<extension>` element in a process definition to signal the presence of an extension. This declaration alone does not change the behavior of a WS-BPEL process or activity – it simply advertises the extension namespace; it is the use of attributes or elements belonging to the extension’s namespace within a WS-BPEL construct that changes the process’ or activity’s behavior. Our process definition indicates its use of context variables through the following declaration:

```
<bpel:extensions>
  .
  .
  .
  <bpel:extension mustUnderstand="yes"
    namespace="http://ece.uwaterloo.ca/aag/statefulbpel"/>
</bpel:extensions>
```

The WS-BPEL specification allows an engine considerable leeway in dealing with extensions. Each extension declaration includes a `mustUnderstand` attribute. Based on the value of this attribute (`yes/no`) and whether the engine supports the extension, the engine has the option of:

1. Rejecting the process definition
2. Accepting the process definition but ignoring any usages of the extension
3. Accepting the process definition and applying the rules associated with the extension

ActiveBPEL simply ignores the use of any extensions marked optional – i.e., extension declarations with `mustUnderstand="no"`. On the other hand, process definitions with unrecognized, yet mandatory, extensions, are immediately rejected. Since we did not want to change how ActiveBPEL processed optional extensions, we simply marked the extension declaration in the scenario process as mandatory and worked on implementing the necessary functionality within the engine.

ActiveBPEL uses an `AeWSBPELDefReaderRegistry` to associate WS-BPEL constructs in a process definition with the appropriate `*Def` class. It then uses double dispatch and the visitor pattern to create a set of activity-definition objects for this process definition. When `AeDispatchReader` calls `configureChild()` on each definition object, they in turn call the `visit()` method of `AeWSBPELReaderVisitor`. `AeWSBPELReaderVisitor` has a `visit()` method for each type of `*Def` object. It uses the `getAttribute()` or `getAttributeBooleanNS()` methods to consume all recognized attributes in the WS-BPEL construct and add them to a set of consumed attributes. After the method completes, this set is compared with the list of attributes present in the original `*Def` object. If any unconsumed attributes are present they are each added as an `AeExtensionAttributeDef` to the `*Def` object. After this phase the definition objects are traversed again using a validation visitor – `AeWSBPELDefToValidationVisitor` – to create a validator for each `*Def` object. These validators are then executed. If any `*Def` object has an `AeExtensionAttributeDef` its associated `AeWSBPELExtentionAttributeValidator` immediately flags that extension usage as not understood, causing validation to fail and aborting deployment.

The context-variable extension described in Section 4.3 requires that the `isStateful="yes"` attribute be added to:

- The root `<process>` element
- The `<variable>` element denoting the context variable

To support this, both the `AeProcessDef` and `AeVariableDef` objects were modified to include fields denoting the presence of this attribute. `AeWSPBELReaderVisitor` was also changed so that its `visit()` methods for `AeProcessDef` and `AeVariableDef`:

1. Consumed the `isStateful` attribute using the `getAttributeBooleanNS()` method
2. Set the corresponding fields in the `*Def` objects on consuming this attribute

Consuming the `isStateful` extension attribute prevented `AeExtensionAttributeDef` objects from being created, and thus, avoided automatic failure during validation. Finally, the `validate()` methods in `AeWSBPELProcessValidator` and `AeVariableValidator` were changed to recognize usages of the context variable extension. Once the `*Def` objects were created and validated, it was possible to deploy a scenario process definition that used the context-variable extension.

### 5.3.3 GENERATING RUNTIME STRUCTURES

ActiveBPEL uses activity implementation objects to represent a runtime WS-BPEL process instance. Each set of implementation objects is isolated from all others and has its own collective state. To support context variables these implementation objects have to be changed:

- `<variable>`
- `<process>`
- `<invoke>`

Standard WS-BPEL variables are implemented using the `AeVariable` class. A context variable has functionality beyond that of a regular variable: it has to be able to update its value based on the contents of a notification message, maintain a mapping between a subscription EPR and a child element within the variable, etc.

The implementation object that corresponds to a WS-BPEL process is `AeBusinessProcess`. A process containing context variables needs to maintain a list of these variables and may even be

responsible for enforcing the rules concerning their modification. It is also the entity that receives notifications for context types. Having the process perform these tasks (as opposed to the variable implementation objects) is a logical choice. In ActiveBPEL the process and engine are tightly connected, with the process using the engine as its interface to the outside world and the engine forwarding results to the process. In fact, the engine has no knowledge of the implementation objects within a process. By having processes receive notifications and handle subsequent routing to context variables, we maintain this relationship, and prevent the engine from knowing anything about the WS-BPEL structures and activities within a process. Moreover, some optimizations are easier to implement, since the process has a global view of the type and state of its internal structures. For example, if two context variables subscribe to the same context type at the same context source the process can recognize this and use the same subscription EPR for both, instead of subscribing a second time.

An `AeActivityInvokeImpl` is the implementation object for an invoke activity. In ActiveBPEL invoke responses are routed to, and processed by, the invoke implementation object making the request. If an invoke activity has as its output variable a context variable, it has to process responses differently: when a web service responds with context parameters the implementation object needs to determine which context types have to be subscribed to, initiate the subscriptions, handle subscription responses and failures, etc.

Although it would have been possible to simply add the necessary functionality to `AeVariable`, `AeBusinessProcess` and `AeActivityInvokeImpl`, the decision was made to create separate context-variable aware objects whenever possible. This is the standard OO practice of subclassing. Each context-variable aware implementation object:

1. Extends the standard implementation object
2. Implements new interfaces with methods specific to context variables

The context-enabled versions of `AeVariable` and `AeActivityInvokeImpl` are `AagStatefulVariable` and `AagActivitySubscriptionInvokeImpl` respectively. As described earlier in this section, `AeBusinessProcess` and `AeBusinessProcessEngine` are intimately linked. Although subclassing was considered, the extra logic involved and interface changes required were extremely high. As a result, simplicity dictated that context-variable-specific changes were made directly

to `AeBusinessProcess`. Whenever appropriate however, context-variable specific methods were listed in a new interface.

To generate this set of implementation objects ActiveBPEL followed the same double-dispatch and visitor patterns used with definition-model creation and validation. Specifically, an `AeDefToWSPBELImplVisitor` is used to visit the set of definition objects and create a set of corresponding implementation objects. The `visit()` methods in `AeDefToWSPBELImplVisitor` for the process, variable and invoke definition objects were modified to create context-aware versions of the implementation objects. The following rules were enforced before a context-variable-aware implementation object was created:

- A context-variable-aware `AeBusinessProcess` is only created if `AeProcessDef` has an `isStateful="yes"` attribute
- An `AagStatefulVariable` is created only if the variable and its containing process element contain the `isStateful="yes"` attribute
- An `AagActivitySubscriptionInvokeImpl` is created only if its containing process element and its output variable contain the `isStateful="yes"` attribute

Note that the presence of the `isStateful="yes"` attribute is not checked by examining the process definition XML. Instead, each definition object has a field denoting the presence of this attribute.

There are opportunities to improve this implementation-object creation. For example, it would be wise to create a context-variable-aware `AeBusinessProcess` object only if the process definition contained variables with the context-variable extension – regardless of whether the process element itself had the `isStateful="yes"` attribute. Alternatively, uses of `isStateful` in a `<variable>` without the attribute also appearing in `<process>` could be caught during validation. That said, the simpler approach above was taken for the purposes of prototyping.

Once a set of implementation objects is created the WS-BPEL process instance can start executing. The following sections detail the tasks these new implementation objects have to perform to support context variables.

#### 5.3.4 PROCESSING EXTENDED RESPONSES

When a web service returns context parameters in its response message there are a variety of ways in which a WS-BPEL engine can process them. Engines with no context-variable support can simply receive the response, validate it, and transfer all its elements to the invoke activity's output variable. It does not treat the context-source reference parameters specially. If the engine supports context variables but either the process or the invoke activity's output variable is missing the `isStateful="yes"` attribute, then again, the above behavior applies. An argument can be made that in the second case the WS-BPEL engine should remove the context parameters, but this is flawed. It may be that the programmer does not want to use context variables – in this case parameter removal is a non-issue; but it could also be that they have opted for a “roll-your-own” approach to dealing with context sources. Since programmer control was a key criterion in designing context variables, we decided that it was best to leave the parameters in. Finally, we must consider the case where the WS-BPEL engine receives a response with context parameters for an invoke activity whose output variable is a context variable. In this case the engine cannot just copy the response contents to the output variable. Instead, it has to perform the following tasks:

- Determine which response parameters have to be copied to the output variable
- Determine if any of the context parameters correspond to these copied parameters
- For each context-parameter and response-parameter match, subscribe to the appropriate context type/context source

ActiveBPEL invoke implementation objects use a callback pattern to receive invocation responses. Each `AeActivityInvokeImpl` object implements the `IAeMessageReceiver` interface which includes an `onMessage()` method. The process's execution queue calls `onMessage()` on the appropriate implementation object (in this case an `AeActivityInvokeImpl`) whenever an incoming web service message is received. In an `AeActivityInvokeImpl` object, `onMessage()` performs the following tasks:

- It validates the message
- It initiates or validates correlation sets
- It uses an object of type `IAeMessageDataConsumer` (specifically `AeVariableMessageDataConsumer`) to copy the contents of the message to the output variable

- It completes its own execution and queues the next implementation object for execution

Remember that when an invoke activity's output variable is a context variable, an `AagActivitySubscriptionInvokeImpl` is created instead of an `AeActivityInvokeImpl`. `AagActivitySubscriptionInvokeImpl` implements the `onMessage()` method, overriding the one in `AeActivityInvokeImpl`. It, like the default implementation, validates the message, initiates/validates correlation sets, and uses a message data consumer to copy the message contents to the output variable. It also does the following:

- Determines if the response message contains context parameters
- For each context parameter in the response, uses the `queueSubscribe()` in `AeBusinessProcess` to queue a `Subscribe` request to the relevant context source

Unlike `AeActivityInvokeImpl`, the `onMessage()` method in `AagActivitySubscriptionInvokeImpl` only terminates the invoke activity if the response message contains no context parameters. If these parameters exist, the invoke implementation object is 'live', and it remains on the execution queue until all subscriptions are completed.

### 5.3.5 SUBSCRIBING TO CONTEXT SOURCES

Using context variables in a WS-BPEL process requires the WS-BPEL engine to act as a WSN subscriber and the process instance to act as a WSN consumer. To subscribe, the engine sends out a WSN `Subscribe` message to a WSN notification producer. An abbreviated version of this message is shown below:



```

<soapenv:Envelope . . .>
  <soapenv:Header>

    <!-- EPR parameters to identify the WSN producer being subscribed to -->
    <wsa:To . . .>http://localhost:5050/TestDest</wsa:To>
    <wsa:Action . . .>
      http://docs.oasis-open.org/wsn/bw-2/NotificationProducer/SubscribeRequest
    </wsa:Action>
    <wsa:MessageID . . .>uuid:c7e5654c-7cde-7d19-d7a8-c27b12812dc9</wsa:MessageID>
    <wsa:From . . .>
      <wsa:Address>http://testaddr</wsa:Address>
      <wsa:ReferenceParameters>
        <aag:statefulSubscribe . . .>TestRefParam</aag:statefulSubscribe>
      </wsa:ReferenceParameters>
    </wsa:From>
    <!-- End of EPR parameters -->

  </soapenv:Header>
  <soapenv:Body>
    <wsnt:Subscribe . . .>

      <!-- WSN Consumer EPR -->
      <wsnt:ConsumerReference>
        <wsa:Address . . .>http://testaddr</wsa:Address>
        <wsa:ReferenceParameters . . .>
          <aag:statefulSubscribe . . .>TestRefParam</aag:statefulSubscribe>
        </wsa:ReferenceParameters>
      </wsnt:ConsumerReference>
      <!-- End of WSN Consumer EPR -->

      <!-- WS-Topic the consumer is subscribing to -->
      <wsnt:Filter>
        <wsnt:TopicExpression
          xmlns:="http://ece.uwaterloo.ca/aag">testProperty</wsnt:TopicExpression>
      </wsnt:Filter>
      <!-- End of WS-Topic the consumer is subscribing to -->

    </wsnt:Subscribe>
  </soapenv:Body>
</soapenv:Envelope>

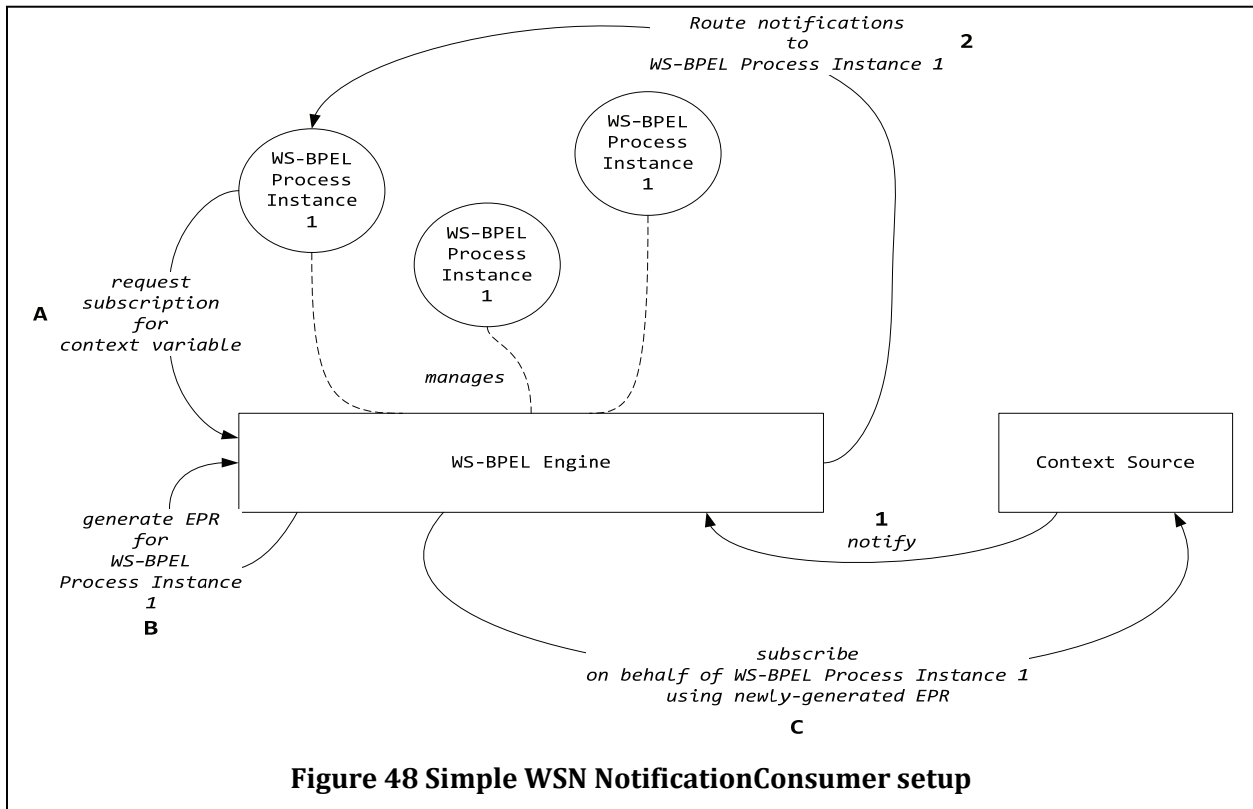
```

Note that the `Subscribe` message requires a `ConsumerReference`, i.e. the EPR of the WSN consumer. Generating this EPR depends on how WSN consumers are implemented within a WS-BPEL engine. Below, we describe two possible implementations, and then fully detail the one used in our prototype.

### 5.3.5.1 CREATING WSN NOTIFICATIONCONSUMERS

This thesis considers two alternatives to implementing WSN consumers and dealing with notifications in a WS-BPEL engine. The first requires the engine to:

1. Generate an EPR for each WS-BPEL process instance

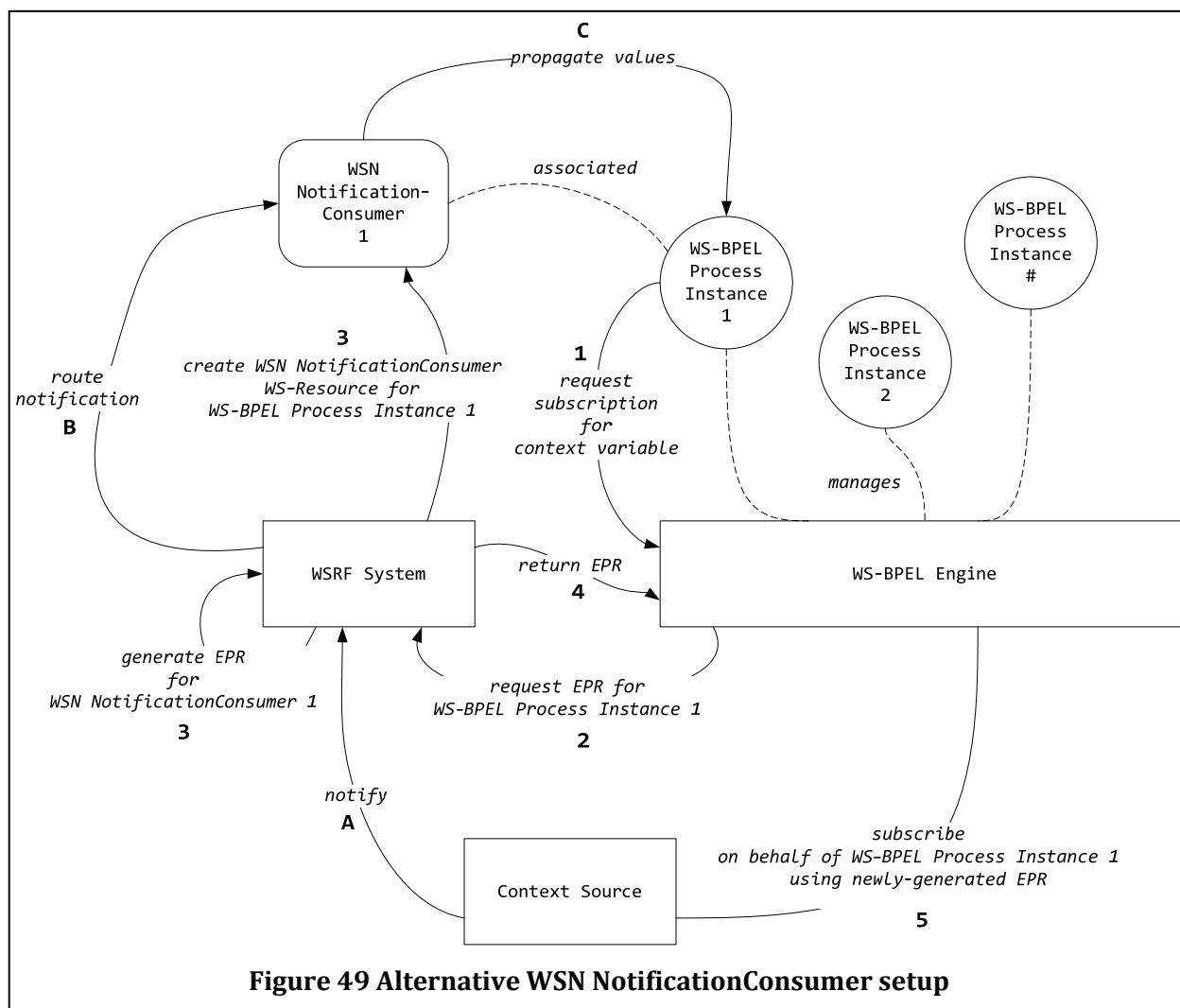


2. Route notifications directly to the process instance

This setup is shown in Figure 48. An advantage of this approach is its conceptual simplicity. Implementing it however, requires the WS-BPEL engine to:

- Generate a valid and unique EPR for each process instance
- Maintain a mapping between each EPR and its corresponding process instances
- Handle WSA headers for incoming and outgoing SOAP messages. This requires the engine to handle other tasks like Universally-Unique-Identifier (UUID) generation, message sequencing, etc.
- Function as a router, directing SOAP messages with WSA headers to the appropriate process instances
- Implement MEPs for the WSN `Subscribe` and `Notify` operations
- Handle WSA and WSN related faults

What at first seems like a simple design decision actually requires turning the WS-BPEL engine into an EPR-based resource router. It also requires that the engine support (at minimum) the



**Figure 49 Alternative WSN NotificationConsumer setup**

WSN Subscribe and Notify MEPs, along with their associated fault handling logic. This is a significant amount of code.

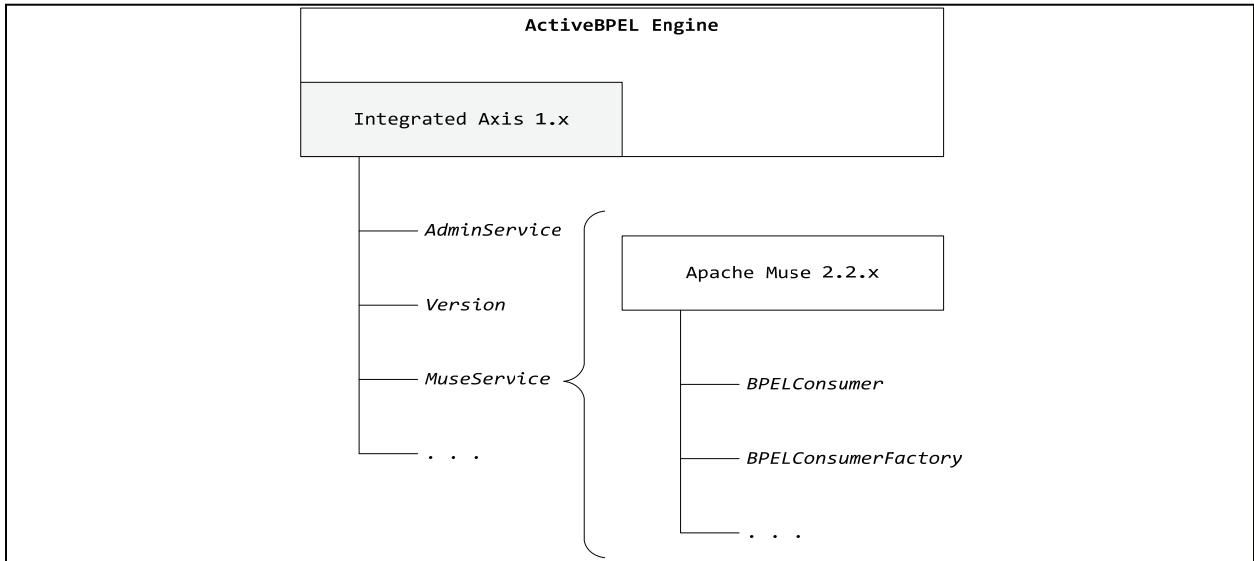
There is an alternative that uses Apache Muse. Muse already functions as an EPR-based resource router and supports the WSN and WSRF MEPs. It has been developed for a number of years and is used in a variety of production environments. Moreover, since Muse was used to implement the prototype's context sources, we have a reasonable understanding of its architecture, major code artifacts and development process. To incorporate a WSRF system like Muse into a WS-BPEL engine, a design is required that:

1. Explains how the Muse webapp and the `active-bpel` webapp can be integrated

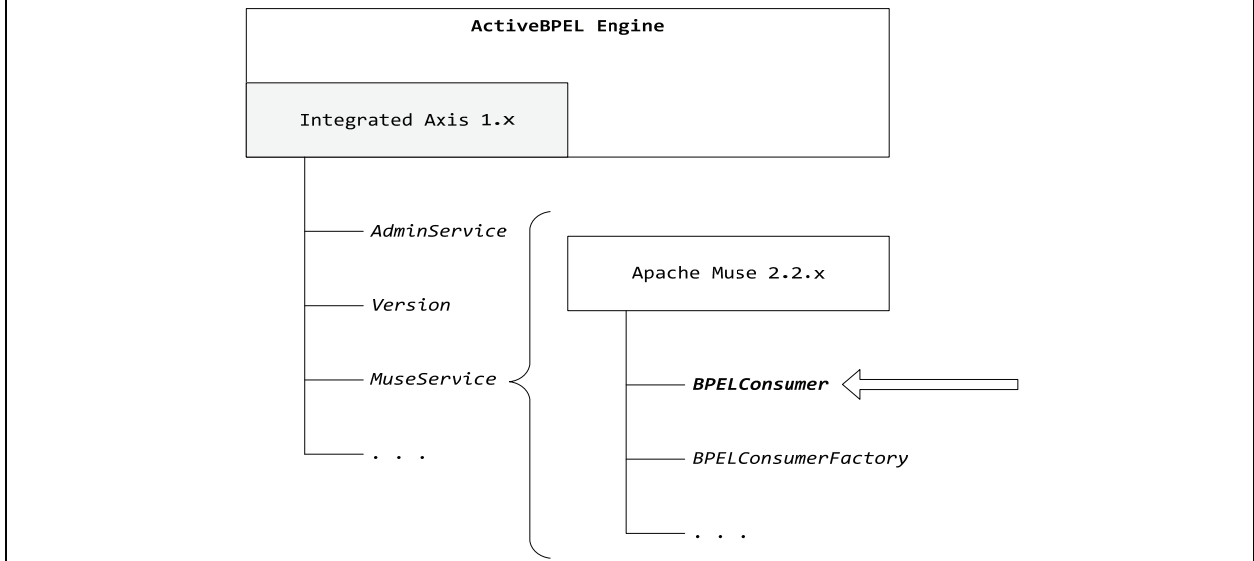
2. Shows what entities are created for a WSN subscription and how these are linked to a process instance
3. Shows which entities receive notifications and how notification messages or values are routed to a process instance

A design that addresses these concerns is shown in Figure 49. Here, the process-implementation object does not act as a WSN consumer. Instead, a separate Muse-managed resource – a WSN-consumer resource type – is created and associated with the process implementation object. Muse automatically assigns an EPR to this WSN consumer resource, and SOAP messages with this EPR are routed by Muse, not the ActiveBPEL engine. When it receives such a SOAP message, Muse first determines which resource should process it. After the message is processed by the resource in question, it forwards the message's contents to its associated process instance. The difference in routing is clear when considering notification messages. In the first approach the WS-BPEL engine would receive these notifications and route them to the process instance itself. In the second, Muse would receive these notifications, route them to a WSN consumer resource, which would then forward them to its associated process instance.

Although this design is simple its implementation is not. The first challenge was integrating both web applications. In their default configuration both ActiveBPEL and Muse run as separate webapps in a servlet container, each using their own SOAP stacks. In the prototype the ActiveBPEL engine runs as the `active-bpel` webapp on Tomcat. On startup the engine initiates its integrated Axis 1.x SOAP stack. It then deploys each available process definition as a web service on top of this stack. When SOAP messages are sent to a WS-BPEL process they are first routed to ActiveBPEL by Tomcat, then processed by Axis 1.x to retrieve all the headers and set up messaging structures, and finally, routed to the appropriate process instance by the engine. Apache Muse uses the same webapp and SOAP stack setup as ActiveBPEL, but with Axis2 instead of Axis 1.x. This posed a problem: first, it was not clear if two SOAP stacks could coexist in the same web application; second, it was unclear if different servlets in a web application shared the same classloader. The second is more serious because it prevents Muse-managed resources from accessing ActiveBPEL classes like an `AeBusinessProcess`. This would make it impossible for us to use method calls to propagate notifications from a WSN-consumer resource to a process implementation object.



**Figure 50 Relationship between Axis services and Muse resources**



**Figure 51 The BPELConsumer Muse resource in the service/resource heirarchy**

Given these concerns it was decided to:

1. Reimplement the Muse IsolationLayer module on top of the ActiveBPEL Axis 1.x SOAP stack
2. Use the ActiveBPEL startup framework to load Muse as a web service in parallel with deployed WS-BPEL process definitions

This ensured that ActiveBPEL and Muse resources shared the same classloader and removed the potential for conflicts between the two SOAP stacks.

Modifying Muse to use ActiveBPEL's SOAP stack required fairly localized changes – namely, implementing its `SoapClient`, `Environment`, and `IsolationLayer` interfaces on Axis 1.x. Having ActiveBPEL's integrated Axis 1.x SOAP stack load Muse as a web service required changing the `Axis-web-service-deployment-descriptor` `ae-server-config.wsdd` to include the following stanza:

```
<service name="MuseService" provider="java:MSG">
  <!-- Class that implements the web service -->
  <parameter name="className"
    value="ece.uwaterloo.ca.aag.platform.abaxis.server.AagABAxisIsolationLayer"/>
  <!-- Method in that class to which the incoming message should be sent -->
  <parameter name="allowedMethods" value="handleRequest"/>
</service>
```

This stanza indicates that any URI with the service path `MuseService` should be routed to the `handleRequest()` method in the `AagABAxisIsolationLayer` class for processing. `AagABAxisIsolationLayer` is the new Axis-1.x-based isolation layer class we built for Muse. While this should have worked, it did not – in fact, it resulted in an unanticipated problem. As shown in Figure 50, Muse resources are not exposed as Axis-1.x-managed services, but as Muse-managed ones. To route SOAP messages to these resources Muse requires that the destination URI include the name of the target resource. Consider the `BPELConsumer` `WSN-consumer-resource` in Figure 51. For SOAP messages to be received by the correct `BPELConsumer` instance both the transport URI and the `WSA To` header need the following form:

```
<transport URI>/<context path>/<servlet path to AXIS 1.x>/
<Muse service name>/<Muse-hosted resource>
```

A real-world example of such a URI is shown below:

<http://localhost:9090/active-bpel/services/MuseService/BPELConsumer>

Axis 1.x also routes SOAP messages web services using the invocation URI. It however, uses the last part of the URI as the name of the *target service*. Axis 1.x inspects its list of services, finds the service that matches this name and routes the SOAP message to it. While this routing behavior works in many situations it fails here, since Axis determines the target service to be `MuseSer-`

vice/BPELConsumer, while we would simply like it to be `MuseService`. This problem was solved by inserting a handler into Axis 1.x handler chain (Handlers and the Message Path in the Axis Architecture Guide [5]) that inspects the transport URI for all incoming SOAP messages. If a message's transport URI contains the name of the integrated Muse service (`MuseService`), its `MessageContext` object's `targetService` field is simply set to `MuseService`, ensuring that the message is sent to Muse for further routing. This handler was added to the Axis handler chain by adding the following stanza to `ae-server-config.wsdd`:

```
<transport name="http">
  . . .
  <requestFlow>
    <handler type="java:ece.uwaterloo.ca.aag.platform.abaxis.handlers.
      AagABAxisMuseRedirectionHandler"/>
  </requestFlow>
</transport>
```

The second challenge in implementing the combined Muse-and-ActiveBPEL design involved creating the WSN-consumer resource type itself. This resource type was modeled using a very simple WSDL interface with the WS-BaseNotification `NotificationConsumer` port type. It had a single operation – `Notify` – and was implemented using two capabilities: a Muse-provided `NotificationConsumer`, and a custom capability that implemented the `NotificationMessageListener` interface. The original plan envisioned that the engine created this resource by calling a `createConsumerResource()` method in the integrated Muse service. It would supply the `ID` and `QName` of the process instance with which the WSN-consumer resource would be associated, and receive an EPR in return. Unfortunately, two problems scuttled this:

1. Muse needs an incoming SOAP message to initialize the framework
2. The ActiveBPEL project structure and build process resulted in a circular build dependency between the resource factory, `AeBusinessProcessEngine`, and `AeBusinessProcess` that could not be resolved

Given this situation only one solution sufficed: turn the resource factory into a web service and have the WS-BPEL engine invoke a WSDL operation on it (by sending it a SOAP message) whenever it wanted to create a new WSN-consumer resource for a process instance.

Having the resource factory act as a web service meant modeling it as a Muse resource type. Named `BPELConsumerFactory`, the WSN-consumer-resource factory was implemented as a singleton resource type using only one custom capability. It exposed a very simple WSDL with a single operation: `CreateBPELConsumer`. As input this operation took a `ProcessID` as `xsd:long` and a `ProcessQName` as `xsd:QName`. It returned `BPELConsumerEPR` – the EPR of the newly created WSN-consumer resource.

Now that WSN-consumer resources can be created, we present a brief overview of the steps the modified ActiveBPEL engine takes to accomplish this. When `AagActivitySubscriptionInvokeImpl` calls `queueSubscribe()` on `AeBusinessProcess`, the process checks if it has an EPR for its associated WSN consumer resource. If it does not, the method blocks, and a request to create this resource is sent using the `addCreateConsumer()` method in `AeBusinessProcessEngine`. The engine propagates this request into the lower layers, where a custom Axis handler constructs and sends the required SOAP message to `BPELConsumerFactory`. When `BPELConsumerFactory` responds with the EPR of the WSN consumer resource, the EPR is returned to the process instance that requested it. Since each WSN consumer resource is linked to a process instance, the returned EPR reflects this by including two reference parameters: the `ProcessID` and `ProcessQName`.

Closely tied to the issue of creating these consumer resources was the issue of getting the transport URI for:

- Messages to the Muse consumer-resource resource factory
- the `WSA To` header

The ActiveBPEL engine insulates process instances from many low-level concerns (such as the engine's transport URI). This is a logical and defensible design choice, but it causes problems when trying to address SOAP messages to the integrated Muse service. After investigation it was determined that the transport URI was present in an incoming message's `MessageContext` object all the way through the Axis handler chain. However, while most of the information in the `MessageContext` object was transferred into an `AeExtendedMessageContext` object on reaching the ActiveBPEL engine, the transport URI was not. This necessitated:



```

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <PurchaseItemResponse . . .>
      <Return xsi:type="ExtendedReturn" . . .>
        <ItemCode>Stapler-101</ItemCode>
        <PurchaseStatus>SUCCEEEDED</PurchaseStatus>
        <ShipmentStatus>IN PROGRESS</ShipmentStatus>
        <Context>
          <StatefulParameterName>ShipmentStatus</StatefulParameterName>
          <NotificationTopicNamespace>
            http://delivery.company/shipping/Shipment
          </NotificationTopicNamespace>
          <NotificationTopic final="false" name="ShipmentCondition"/>
          <ServiceEPR>
            <ns3:Address>
              http://#.#.#.#/deliverycompany/services/ShipmentTracker
            </ns3:Address>
            <ns3:ReferenceParameters>
              <muse-wsa:ResourceId . . .>MuseResource-10</muse-wsa:ResourceId>
            </ns3:ReferenceParameters>
          </ServiceEPR>
        </Context>
      </Return>
    </PurchaseItemResponse>
  </S:Body>
</S:Envelope>

```

**Figure 52 PurchaseItemResponse response message**

1. Changing AeBpelHandler to place the transport URI into a new field in AeMessageContext
2. Retrieving the contents of that field inside ActiveBPEL's AeAbstractReceiveHandler and using it to set the base transport URI for the engine

Once this base URI was set, the URI for the integrated Muse service could easily be composed as <base URI>/MuseService.

### 5.3.5.2 SUBSCRIBING

Once the WSN consumer resource is created, the subscription can be initiated within the ActiveBPEL engine. As described in Section 5.3.4, an invoke implementation object receives response messages in its `onMessage()` method. If this implementation object's output variable is a context variable it initiates a subscription for each context parameter in the response. Although WSN allows an entity to subscribe for notifications from multiple topics in a single `Subscribe` message, the prototype uses a single `Subscribe` message per context parameter. This is done for simplicity: the engine's subscription logic can remain the same regardless of whether the context parameters share the same context source or not. Also, this allows the logic for propagating notification val-

ues to be simplified. Note however, that nothing in the high-level design stipulates this, and implementers can optimize subscriptions as appropriate. The prototype's default behavior can be seen in the example below. Consider the response in Figure 52. This message has three response elements and one context parameter. If the output variable is of type message, it contains a copy of all three elements, one of which has to be linked to the context type and source in the context parameters. So, for parameter `ShipmentCondition`, a subscribe request is made to the WS-Topic `ShipmentCondition` at the context source with EPR:

```
<ServiceEPR>
  <ns3:Address>
    http://#.#.#.#/deliverycompany/services/ShipmentTracker
  </ns3:Address>
  <ns3:ReferenceParameters>
    <muse-wsa:ResourceId . . .>MuseResource-10</muse-wsa:ResourceId>
  </ns3:ReferenceParameters>
</ServiceEPR>
```

To make this subscribe request the `AagActivitySubscriptionInvokeImpl` calls the `AeBusinessProcess` object's `queueSubscribe()` method, and passes it the above EPR and context type, as well as a reference to itself. When the subscribe succeeds (or faults) this reference is used to call the initiating `AagActivitySubscriptionInvokeImpl` object's `onSubscribeResponse()` or `onSubscriptionFault()` methods. Note that this callback pattern parallels the one used by `AeActivityInvokeImpl` to receive invocation responses.

Once `queueSubscribe()` is called, the subscription phase is initiated; the invoke implementation object has nothing to do with the rest of the subscription process. `AeBusinessProcess` queues the subscription request with `AeBusinessProcessEngine` using the `addSubscribe()` method. The engine then sends this request to its `QueueManager` – a construct for sequencing requests and responses to and from external entities. This `QueueManager` locates the class handling subscribe requests and hands it the request for asynchronous processing. After `QueueManager` hands off this request, control returns to the process instance which can execute other, parallel activities, while waiting for the response. Subscribe messages are sent, received and queued for return by two classes: an `AagSubscriber` and an `AagSubscribeHandler`. The latter uses a `Muse NotificationProducerClient` to subscribe to the context source. Since Muse supports the WSN standard, the XML for the `Subscribe` messages and the MEPs involved in this op-

eration are generated and handled by Muse. The ActiveBPEL classes use Java method calls to invoke the `Subscribe` operation and receive its response. A `SubscribeResponse` has the form:

```
<soapenv:Envelope . . .>
  <soapenv:Header>
    <!-- EPR for the WSN subscriber -->
  </soapenv:Header>
  <soapenv:Body>
    <wsnt:SubscribeResponse . . .>

      <!-- EPR of the subscription resource representing the subscription -->
      <wsnt:SubscriptionReference>
        <wsa:Address>http://#.##.##/Service/SubscriptionManager</wsa:Address>
        <wsa:ReferenceParameters>
          <muse-wsa:ResourceId . . .>MuseResource-300</muse-wsa:ResourceId>
        </wsa:ReferenceParameters>
      </wsnt:SubscriptionReference>
      <!-- End of subscription reference EPR -->

      <wsnt:CurrentTime>2008-02-26T22:14:35-05:00</wsnt:CurrentTime>
    </wsnt:SubscribeResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

`AagSubscribeHandler` retrieves the EPR of the subscription resource, the Muse Java class representing the subscription, and the response parameter name to which these are linked, and returns it to the process instance that requested the subscription. When `AeBusinessProcess` receives this response in its `dispatchSubscribeData()` method it locates the invoke implementation object that initiated the subscription request and uses its execution queue to call this invoke object's `onSubscribeResponse()` callback method. When `onSubscribeResponse()` is executed, the invoke implementation object:

1. Notifies the output variable of this subscription
2. Notifies the process of this subscription

When the process implementation object is notified it adds an entry to a table that maps subscription EPRs to context variables. This is later used to propagate changed values from notifications to the appropriate context variables. When the output variable is notified, it registers a mapping between the same subscription EPR and the specific element within its structure that the subscription refers to. Once these two mappings – in both the process and variable implementation objects – are complete, the subscription phase is finished. Once all outstanding subscriptions have completed the `AagActivitySubscriptionInvokeImpl` object terminates.

### 5.3.6 PROCESSING NOTIFICATIONS

Context sources send notifications to the consumer EPR included in the `Subscribe` request. In the prototype this EPR is that of a WSN consumer resource associated with a process instance. WS-BaseNotification states that a notification consumer may receive two types of notifications:

1. One containing application-specific notification content only

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <wsa:To . . .>
      http://#.#.#.#/active-bpel/services/MuseService/BPELConsumer
    </wsa:To>
    . . . Other WSA parameters
  </soap:Header>
  <soap:Body>
    <wsnt:Notify . . .>
      <wsnt:NotificationMessage . . .>

        <!-- EPR of the subscription associated with this notification -->
        <wsnt:SubscriptionReference>
          <wsa:Address>http://#.#.#.#/ Service/SubscriptionManager</wsa:Address>
          <wsa:ReferenceParameters>
            <muse-wsa:ResourceId>MuseResource-300</muse-wsa:ResourceId>
          </wsa:ReferenceParameters>
        </wsnt:SubscriptionReference>

        <!-- WS-Topic this notification is being generated for -->
        <wsnt:Topic . . .>ship:ShipmentCondition</wsnt:Topic>

        <!-- EPR of the WSN producer generating the notification -->
        <wsnt:ProducerReference>
          <wsa:Address>http://#.#.#.#/ Service/ShipmentTracker</wsa:Address>
          <wsa:ReferenceParameters>
            <muse-wsa:ResourceId>MuseResource-Ship1</muse-wsa:ResourceId>
          </wsa:ReferenceParameters>
        </wsnt:ProducerReference>

        <!-- Content of the notification message -->
        <wsnt:Message>
          <wsrf-rp:ResourcePropertyValueChangeNotification>
            <wsrf-rp:OldValues>
              <ship:ShipmentCondition>Condition-300-1</ship:ShipmentCondition>
            </wsrf-rp:OldValues>
            <wsrf-rp:NewValues>
              <ship:ShipmentCondition>Condition-300-2</ship:ShipmentCondition>
            </wsrf-rp:NewValues>
          </wsrf-rp:ResourcePropertyValueChangeNotification>
        </wsnt:Message>

      </wsnt:NotificationMessage>
    </wsnt:Notify>
  </soap:Body>
</soap:Envelope>
```

**Figure 53 Format of a WSN notification message**

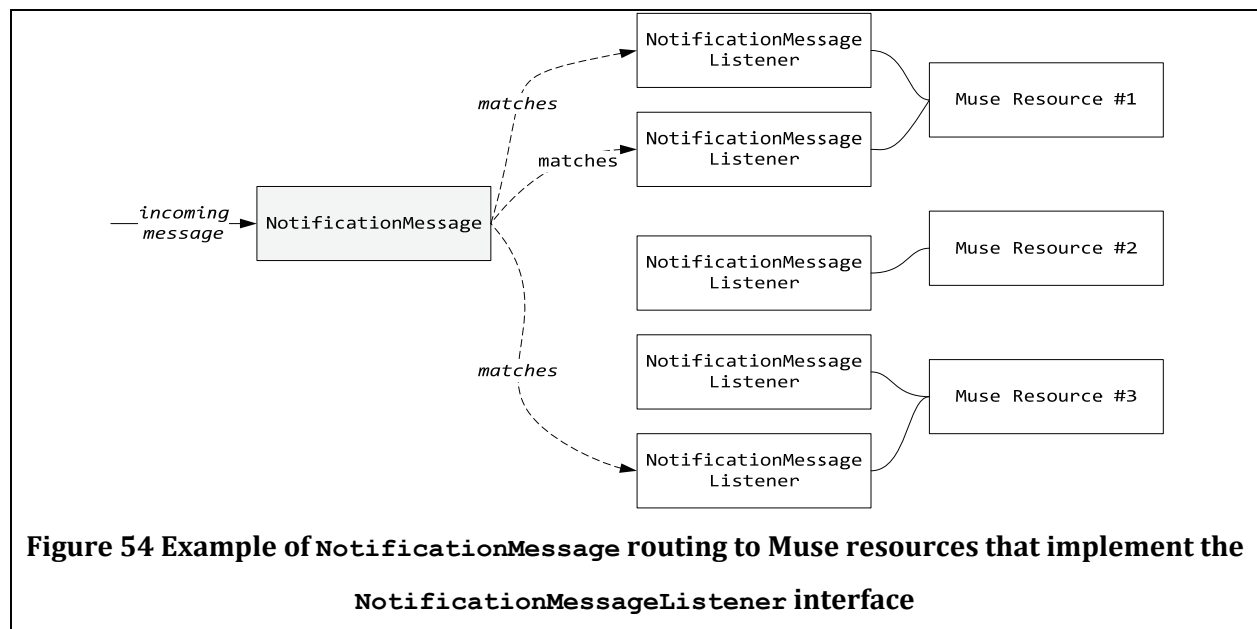
## 2. Notification data using the `NotificationMessage` format

The first approach complicates the implementation of a generalized notification receiving and processing framework within the WS-BPEL engine, since each context variable would require specific data types and formats. We took the second approach, which uses a standardized message, and allows us to implement a generalized notification receiver in the engine.

The format of a notification message is shown in Figure 53. In this message, the `NotificationMessage/SubscriptionReference` element contains the EPR of the subscription resource for which the notification was generated. Note that this EPR is the same one returned during the subscription phase described in Section 5.3.5.2. The actual notification content is embedded in the `NotificationMessage/Message` element. The following sections describe how the prototype routes notifications to a process instance, and how the changed value in this notification is used to update a context variable.

### 5.3.6.1 PROPAGATING NOTIFICATIONS TO A PROCESS INSTANCE

Notification messages are routed by Muse to a WSN-consumer resource based on the EPR in the message header. As detailed in Section 5.3.5.1, this resource is implemented using two capabilities: a Muse-provided `NotificationConsumer`, and a custom capability that implements the `NotificationMessageListener` interface. The `NotificationConsumer` capability allows a resource to



**Figure 54 Example of `NotificationMessage` routing to Muse resources that implement the `NotificationMessageListener` interface**

register one or more listeners, each of which executes whenever it receives a notification matching its criteria. Listeners implement the `NotificationMessageListener` interface, which provides the following methods:

- `public boolean accepts(NotificationMessage message)`
- `public void process(NotificationMessage message)`

Criteria matching is performed in the `accepts()` method. When Muse determines which resource instance should receive a notification, it calls `accepts()` on each capability within that resource instance that implements the `NotificationMessageListener` interface. If a capability is interested in processing the message it signals its interest by returning `true`, after which its `process()` method is called. As shown in Figure 54, this allows programmers considerable flexibility in determining how to respond to incoming notification messages. They can separate message processing code into manageable portions – each in a different class – or they can define multiple listeners, each for a different `NotificationMessage`. The prototype processes all notifications. The content of the `NotificationMessage/Message` element is as follows:

```
<wsnt:Message>
  <wsrf-rp:ResourcePropertyValueChangeNotification>
    <wsrf-rp:OldValues>
      <!-- Content of old value of resource property -->
    </wsrf-rp:OldValues>
    <wsrf-rp:NewValues>
      <!-- Content of new value of resource property -->
    </wsrf-rp:NewValues>
  </wsrf-rp:ResourcePropertyValueChangeNotification>
</wsnt:Message>
```

The context type's new and old values are contained in the `ResourcePropertyValueChangeNotification` element. The `NewValues` element contains the serialized version of the new value of the context type. Although the `NewValues` element can have more than one child element (corresponding to multiple new values) we do not support this. This is because there is a 1:1 correspondence between subscription and context type, so `NewValues` should only ever have a single child element per notification message.

When the listener capability receives a notification in its `process()` method it extracts the content of the `NewValues` element. It does not deserialize the changed value into a Java type. This choice is implementation-specific: ActiveBPEL variable implementations can be updated using

XML DOM, so this approach minimizes data-conversion problems. The capability then sends this updated value to the process instance with the `ID` and `QName` with which it is associated. To do so, it uses the `updateStatefulVariable()` in `AeBusinessProcessEngine`, passing it the:

1. Subscription reference for this notification
2. Changed values for the context type
3. Process ID

The engine uses the process ID to locate the appropriate process instance. It then calls the process implementation object's `updateStatefulVariable()` method, passing it the changed values and the subscription EPR. Note that a process instance has only one WSN-consumer resource. This means that all context-variable related notifications for that process instance are handled by a single resource.

#### 5.3.6.2 UPDATING THE CONTEXT VARIABLE

When `AeBusinessProcess` receives a notification update in its `updateStatefulVariable()` method it uses its table of subscription EPR to context-variable mappings to determine which context variable to update. Since there is a 1:1 correspondence between subscription EPR and context variable, only one variable will be changed per notification. Once this variable is located, `AeBusinessProcess` then calls the variable's `updateValue()` method. Each `AagStatefulVariable` contains a mapping between the subscription EPR and the child element to which it applies. When the variable's `updateValue()` method is called, the following takes place:

1. The variable's mappings are checked to determine the child element to which the subscription EPR is mapped
2. The variable's content is converted into a normalized DOM document
3. The contents of the referenced child element are replaced by the new value

Once the last step is complete, the value of the context variable has changed – without intervention on the part of the programmer, or extra process logic.

#### 5.3.7 DEVELOPMENT IMPRESSIONS

Adding the above functionality to ActiveBPEL was a non-trivial task because its size, and complexity of its internal structure. Also, as is common in many large pieces of software, the architec-

ture documentation describing major functionality was limited. As a result, before any changes could be made it was necessary to reverse engineer ActiveBPEL's operation, and understand the major classes, method flows and architectural constructs being used. In addition, ActiveBPEL's separation into multiple projects and the resulting build sequence limited where new classes could be created and how new layers could be integrated. It was also unclear at times which artifacts (deployment descriptors, *etc.*) to modify to achieve certain goals, and where they were located in the project structure.

#### 5.4 TEST SETUP

The open source soapUI [20] tool was used extensively during prototype development. soapUI is a web-service-testing tool with support for functional testing, service mocking, load testing, *etc.* It allows a developer to import WSDL interfaces and exercise the operations in their port types by creating sample requests and responses and defining test cases. soapUI also allows a developer to create mock services based on these WSDL interfaces. This allows, among other things, for mock services to return scripted responses when their operations are invoked. Only the very basic features of soapUI were used to test the prototype.

The WSDLs for the WS-BPEL process in the scenario and the supplier web service were imported into soapUI, and sample requests generated. This allowed each entity to be tested independently. Mock services proved very useful during development. First, running three different application servers on a development machine taxed hardware resources. It also slowed down development during initial stages since we did not have full control over response contents. Moreover, it caused complications, since three entities – some with incomplete functionality – were interacting. Given these issues mock services were substituted for the architectural entities not under test. They were created for the supplier web service and `ShipmentTracker`. The supplier web service was set up to respond to invocations with a range of context parameters (none, one, and many), each with EPRs pointing to the `ShipmentTracker` mock service. The `ShipmentTracker` mock service acted as a sample context source and implemented the WSN `Subscribe` and `Notify` operations; it allowed a WS-BPEL process to subscribe to a context type and receive notifications. Although these notifications were pre-defined they were sent manually. This was a major limitation of our test setup – that it required human intervention. For example, the endpoints



that `Subscribe` messages were sent to were hardcoded; they should, instead, be linked to the transport URI in the `Subscribe` request that was sent to the `ShipmentTracker` mock service. Dependencies like this can be automated by using soapUI's Groovy [18] scripting support. For example, a script could be written that extracts the transport URI from a `Subscribe` message, stores it, and uses it as the endpoint in future notifications. It would also be advisable to implement actual test cases to simplify further development and flag regressions.

## 5.5 PROTOTYPE LIMITATIONS

As can be imagined the prototype has a number of limitations. First, it implements only minimal fault handling. In a real-world environment subscription requests could fail, context sources may disappear, or the integrated `MuseService` might shut down. The WS-BPEL engine must be able to recover from these failures and propagate an appropriate exception or notification to the process instance if required. Next, context parameters can only be retrieved from document/literal wrapped responses. While this decision was made for WS-I compliance, many web services do not conform to this WSDL style. It would be useful to use context parameters with web services written in other WSDL styles. There is also no way to advertise that a WSDL operation may return context variables – it would be advisable to add a WSDL extension attribute that denotes this. As for ActiveBPEL itself, currently only WS-BPEL variables of type `Message` are allowed to be context variables. Since WS-BPEL supports other variable types, both context variables and parameter subscription should be made to work with these variable types as well. Finally, adding context variable functionality to ActiveBPEL required adding significant amounts of new code, changing internal interfaces, *etc.*, and some of these changes do not make the best use of the ActiveBPEL architecture. Also, parts of the context-variable functionality are not cleanly separated from that of 'normal' WS-BPEL operation. Both these factors could complicate further extension of the prototype engine. Finally, the `ShipmentTracker` context source does not generate notifications itself. Instead, another Java program changes its context type through the web-service interface, utilizing Muse's default behavior to generate notifications. This should be simplified.

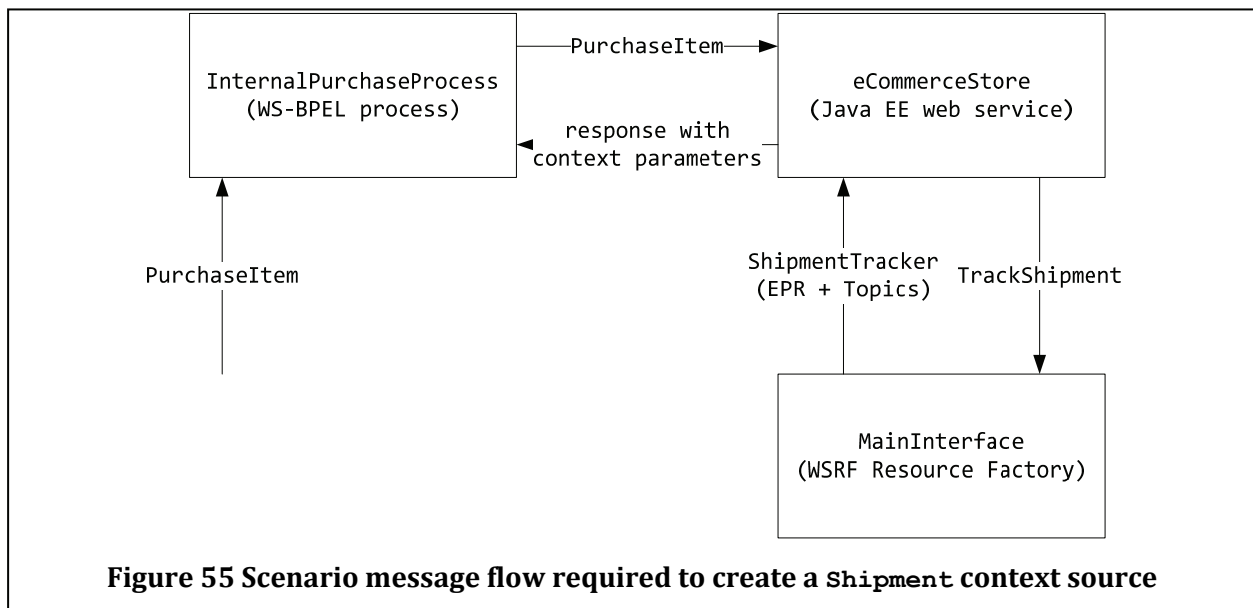
## Chapter 6

### EVALUATION

As described in the previous chapter, we have implemented the scenario described in Section 3.1 to exercise our proposed architecture. It consists of three components:

1. A WS-BPEL business process that represents the manufacturer's purchase process
2. A Java EE web service that serves as the component supplier
3. A WSRF-based context-source system that functions as the shipping company

The scenario is initiated when we invoke the `PurchaseItem` operation on the WS-BPEL business process. The process then invokes the `PurchaseItem` operation on the Java-EE web service, which invokes the `TrackShipment` operation on the context-source factory. This factory creates a new `ShipmentTracker` context source that represents the shipment containing the order and returns it to the Java EE web service. The message flow we have just described is shown in Figure 55 below.



When it receives the information pertaining to the `ShipmentTracker`, the Java-EE web service responds to the WS-BPEL process with the message in Figure 56. The process copies this response

```

<S:Envelope . . .>
  <S:Body>
    <PurchaseItemResponse . . .>
      <Return xsi:type="ExtendedReturn" . . .>
        <ItemCode>ItemCode_1</ItemCode>
        <PurchaseStatus>SUCCEEEED</PurchaseStatus>
        <ShipmentStatus>IN PROGRESS</ShipmentStatus>
        <Context>
          <StatefulParameterName>ShipmentStatus</StatefulParameterName>
          <NotificationTopicNamespace>
            http://delivery.company/shipping/Shipment
          </NotificationTopicNamespace>
          <NotificationTopic final="false" name="ShipmentCondition"/>
          <ServiceEPR>
            <ns3:Address>
              http://#.#.#./deliverycompany/services/ShipmentTracker
            </ns3:Address>
            <ns3:ReferenceParameters>
              <muse-wsa:ResourceId . . .>MuseResource-10</muse-wsa:ResourceId>
            </ns3:ReferenceParameters>
          </ServiceEPR>
        </Context>
      </Return>
    </PurchaseItemResponse>
  </S:Body>
</S:Envelope>

```

**Figure 56 Java-EE web service response message**

```

<Context>
  <StatefulParameterName>ShipmentStatus</StatefulParameterName>
  <NotificationTopicNamespace>
    http://delivery.company/shipping/Shipment
  </NotificationTopicNamespace>
  <NotificationTopic final="false" name="ShipmentCondition"/>
  <ServiceEPR>
    <ns3:Address>
      http://#.#.#./deliverycompany/services/ShipmentTracker
    </ns3:Address>
    <ns3:ReferenceParameters>
      <muse-wsa:ResourceId . . .>MuseResource-10</muse-wsa:ResourceId>
    </ns3:ReferenceParameters>
  </ServiceEPR>
</Context>

```

**Figure 57 Response-message context parameter**

to its `purchaseItemResponse` message-type variable. This response contains an extended return type with the single context parameter shown in Figure 57. This context parameter contains:

1. The EPR of the newly-created `Shipment` context source
2. A context-type – `ShipmentCondition`
3. The response-message parameter that `ShipmentCondition` is linked to, namely `ShipmentStatus`

The WS-BPEL engine uses the contents of the `<ContextParameter>` element to subscribe the scenario process's `purchaseItemResponse` variable to the `ShipmentCondition` context type. After this, `purchaseItemResponse` is transparently updated when `ShipmentCondition` changes.

The `purchaseItemResponse` variable can also be updated using polling or `onEvent` message handlers. With polling, programmers would first have to specify a `partnerLinkType` and a `partnerLink` that codifies the relationship between the `ShipmentTracker` context-source's `ShipmentTrackerPortType` and the purchase process's `InternalPurchase` port type. They would then have to implement the polling loop using one each of the `<repeatUntil>`, `<wait>`, `<invoke>`, and `<assign>` activities. In this polling loop, the `<wait>` activity is of type duration – but the choice of update interval is not obvious. To receive updates for `ShipmentCondition` the `<invoke>` activity would have to invoke the `GetResourceProperty` WSDL operation on the `ShipmentTracker` WS-Resource, using `ShipmentCondition` as the input parameter. The `<assign>` activity would be used to copy the value of `ShipmentCondition` to the `purchaseItemResponse/ShipmentStatus` element. As noted earlier, using polling to implement state update requires an additional six constructs. It also increases coupling since the name of the port type and the operation to be invoked on the context source have to be known in advance. Note that this is the case when the scenario process depends on only one piece of externally-driven environment state; if the process depended on state from multiple context sources the number of state-maintenance constructs required would increase linearly.

As with polling, using an `onEvent` handler also requires a `partnerLinkType` and `partnerLink` that links the process' and the context-source's port types. The process also needs an `onEvent` handler, which has the following form:

```
<process name="purchaseProcess" ...>
  . . .
  <eventHandlers>
    <onEvent partnerLink="purchasing" operation="findOrderStatus" ...>
      <scope>. . . Activities for event handler logic . . .</scope>
    </onEvent>

    <onEvent partnerLink="purchasing" operation="terminateOrder" ...>
      <scope>. . . Activities for event handler logic . . .</scope>
    </onEvent>
  </eventHandlers>
  . . .
</process>
```

The operation attribute is the name of the operation that *the external entity invokes on the process* to deliver state updates. This is stated in Section 12.7.1 of the WS-BPEL specification: “the port-Type and operation attributes define the port type and operation that is invoked by the partner in order to cause the event.” Thus, to support this one onEvent message handler, the programmer has to modify the process’ WSDL interface and define a new operation, its input and/or output messages, and the corresponding XML-Schema types for their contents. Note also, that the prototype `ShipmentTracker` context source does not support processes using onEvent handlers for notifications. This is because `ShipmentTracker` does not contain custom code for invoking different operations for different recipient processes. As for the onEvent handler itself, it contains a `<scope>` activity and a child `<assign>` activity; `<scope>` is required because the specification demands it, and `<assign>` is required to copy the current value of `ShipmentCondition` to the `purchaseItemResponse/ShipmentStatus` element. Considering the factors above, we see that for a programmer to update a single externally-driven state variable using WS-BPEL onEvent message handlers, they have to:

1. Make significant changes in the process’ WSDL
2. Ensure that the context source is modified to call the correct operation on the process to deliver notifications
3. Add an additional five constructs to the process logic

With context variables none of the above state-maintenance activities, process interface changes, or increased coupling is required. Also, in-process state-maintenance logic is significantly reduced – from six constructs in the commonly used polling version to none when using our approach. The programmer also does not have to deal with issues like extending state-source WSDLs with partner link types, choosing a wait interval, accounting for state dynamism, or concurrent variable-modification semantics, all of which are orthogonal to the business logic. Note also that we have only considered a very simple scenario. In a more complex one the reduction in state maintenance code is much higher, as the complexity of maintaining external state information grows with the number of state items being maintained.

## Chapter 7

### ENHANCEMENTS

The full potential of context variables cannot be realized using existing WS-BPEL activities and constructs. This is because the WS-BPEL language was designed around process-updated state and variables to store them in, not on variables containing state that is updated by external entities. This thesis proposes three WS-BPEL extensions that would:

1. Allow programmers to write shorter, simpler process logic based on context variables
2. Open simpler ways to share externally-driven state with other business processes

These extensions are: a `<conditionWithTimeout>` activity, context handlers and context variable handoff. In addition, this thesis also proposes an enhanced context source that incorporates semantic information for each of its context types.

#### 7.1 THE `<CONDITIONWITHTIMEOUT>` EXTENSION ACTIVITY

Programmers cannot fully realize the power of context variables if they use the standard conditional activity - `<if>`. This is because `<if>` evaluates the variables in its condition expression immediately; it cannot block until the condition becomes true. This behavior is consistent with the mental model underlying WS-BPEL, *viz.* that the environment state and the variables representing it are changed by the business process itself. This assumption no longer holds when context variables are used. The consequences of this design decision can be demonstrated by returning to the sample scenario, where part of the process logic can be expressed using the pseudocode shown in Figure 58. This logic can be described as a “*blocking condition with timeout*,” a common programming construct in many I/O-dependent systems. Examples of such conditions

```
IF
  Shipment.EstimatedDeliveryDate >
  InternalDeliveryCutoffDate
OR
  CurrentDate =
  InternalDeliveryCutoffDate
THEN
  // execute compensatory business logic
```

**Figure 58 Pseudocode for "blocking condition with timeout" scenario process logic**

```

<!-- Wait until the shipment is late or cutoff date is reached -->

<repeatUntil>
  <condition>
    <!-- Main condition -->
    ($currShip/ns1:Shipment/EstimatedDeliveryDate >
      $InternalDeliveryCutoffDate)
    or
    <!-- Timeout -->
    ($CurrentDate > $InternalDevliveryCutoffDate)
  </condition>

  <sequence>
    <wait>
    <!-- Polling delay -->
    </wait>

    <!-- Invoke web service and update value of
      Shipment.EstimatedDeliveryDate -->
    <invoke/>
  </sequence>
</repeatUntil>

<!-- Check for timeout or late shipment and execute compensatory logic -->
<if>
  <condition>
    ($currShip/ns1:Shipment/
      EstimatedDeliveryDate >
      $InternalDeliveryCutoffDate)
  </condition>
  ...
  <elseif>
    <condition>...</condition>
  </elseif>
  ...
</if>

```

**Figure 59 Scenario process logic using polling**

include socket sends and reads, condition waits in threads, *etc.* This construct cannot be expressed cleanly using standard WS-BPEL activities; `<wait>` is the only activity that explicitly blocks, but unlike `<if>` or `<while>` it cannot evaluate arbitrary boolean expressions. Without context variables, the desired behavior can be approximated using the polling code shown in Figure 59. This approach allows the process to respond to whichever situation occurs first – a change in estimated delivery date or the timeout.

With `Shipment` as the context variable the code shown in Figure 60 can be used. Note that we no longer have to explicitly update the state variable using `<invoke>`; in fact, if a state update ar-

```

<!-- Wait until the shipment is late or cutoff date is reached -->

<repeatUntil>
  <condition>
    <!-- Main condition -->
    ($currShip/ns1:Shipment/EstimatedDeliveryDate >
     $InternalDeliveryCutoffDate)
    or
    <!-- Timeout -->
    ($CurrentDate > $InternalDevliveryCutoffDate)
  </condition>

  <wait>
    <!-- delay -->
  </wait>
</repeatUntil>

<!-- Check for timeout or late shipment and execute compensatory logic -->
<if>
  <condition>
    ($currShip/ns1:Shipment/EstimatedDeliveryDate >
     $InternalDeliveryCutoffDate)
  </condition>
  ...
<elseif>
  <condition>...</condition>
</elseif>
  ...
</if>

```

**Figure 60 Scenario process logic using context variables and standard activities only**

rives while the process is waiting, the process engine transparently updates the variable's value. Consequently, the logic no longer looks like polling, but more like a condition wait in a thread. Although this is an improvement, since only the standard WS-BPEL activities have been used, the semantics of "block until condition met or timeout" still cannot be expressed cleanly. Note also that while the context variables' values can be updated transparently prior to the execution of a conditional activity, once the activity is reached, its condition expression is evaluated immediately and the appropriate code branch taken – the process cannot block *within* the activity itself.

Given the limitations of using only the standard activities, this thesis proposes adding a blocking condition and timeout construct to WS-BPEL. The proposed approach uses the `<extensionActivity>` activity type defined in Section.10.9 of the WS-BPEL specification. An `<extensionActivity>` wraps non-standard activities and places minimum restrictions on the semantics and grammar of its contained elements, making it ideal for the proposed construct; an example of its



```

<conditionWithTimeout standard-attributes>
  standard-elements
  <blockingCondition expressionLanguage="anyURI"?>boolean-
expression</blockingCondition>
  <timeout>
    (
      <after expressionLanguage="anyURI"?>duration-expr</after>
      |
      <at expressionLanguage="anyURI"?>deadline-expr</at>
    )
  </timeout>
  activity
  <else?>
    activity
  </else>
</conditionWithTimeout>

```

**Figure 61 Design of <conditionWithTimeout> extension activity**

use is the <peopleActivity> activity type defined in the BPEL4People[3] specification. The new activity we propose is loosely based on the standard <if> (Section 11.2, WS-BPEL specification [39]) and <wait> (Section 10.7, WS-BPEL specification [39]) activities and has the structure shown in Figure 61. It consists of a timed conditional branch defined using the <conditionWithTimeout> element, followed by an optional <else> element. The timed conditional branch has two parts:

1. The blocking condition itself
2. Its associated timeout

The former is similar to the <if> activity's <condition> and is defined in the <blockingCondition> element.

This extension activity blocks until the condition is true or the timeout defined in the <timeout> element is reached. If the condition is met the activity under <conditionWithTimeout> is executed; if the timeout is reached and the blocking condition is still false, the activity in the <else> branch is executed. Timeouts can be expressed as a delay (using the <after> element) or a deadline (using the <at> element). As in Section 10.7 of the WS-BPEL specification, if the specified duration in <after> is zero or negative, or a specified deadline in <at> has already been reached or passed, then the <blockingCondition> is immediately evaluated and the appropriate code branch executed.

```

<conditionWithTimeout>
  <blockingCondition>
    $currShip/ns1:Shipment/EstimatedDeliveryDate >
    $InternalDeliveryCutoffDate
  </blockingCondition>

  <timeout>
    <at>$InternalDeliveryCutoffDate</at>
  </timeout>

  <!-- Execute this activity if condition met before timeout -->
  activity

  <else>
    <!-- Execute this activity if condition not met before timeout -->
    activity
  </else>
</conditionWithTimeout>

```

**Figure 62 Scenario process logic using context variables and <conditionWithTimeout>**

Using <conditionWithTimeout> the pseudocode in Figure 58 can be expressed using the code in Figure 62. Note that this proposed extension activity significantly reduces the number of activities required to model the process logic and captures its semantics correctly. Since blocking conditions with timeouts are fairly common in current I/O-dependent programs, <conditionWithTimeout> allows WS-BPEL to model a much wider range of I/O behavior.

## 7.2 CONTEXT HANDLERS

A *context handler* is an event handler that executes on context changes. Unlike WS-BPEL message-based event handlers, context handlers are not exposed as a WSDL operation in a process' port type. They are also not executed by external web services. Instead, they are similar to event listeners in GUI toolkits. But while event listeners execute code on user or system events (like clicking a mouse button or refreshing a window), a context handler executes when there is a

```

<contextHandlers>?
  <contextHandler contextVariable="BPELVariableName"*
    <condition expressionLanguage="anyURI"?>bool-expr</condition>?
    <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
    </correlations>
    <scope . . . > . . . </scope>
  </contextHandler>
</contextHandlers>

```

**Figure 63 Design of the <contextHandler> activity**

change in the context variable to which it is linked. It has the structure shown in Figure 63.

Each scope (any `<scope>` and the `<process>` itself) can have its own set of context handlers. Each handler references, or is linked to, a single context variable through the `contextVariable` attribute. Programmers can define multiple context handlers for a single context variable. They can also limit the changes to which each handler responds by specifying a boolean condition in its optional `<condition>` element. This affords them a fair bit of flexibility in designing these handlers, allowing them to specify which ones should be executed on a context change. Context handlers can contain child activities. Like WS-BPEL event handlers, the single child of a context handler **MUST** be a `<scope>` activity, which can itself contain other activities. A context handler can also define and use correlation sets in its `<correlations>` and `<correlation>` elements. When it comes to correlations, context handlers follow the same guidelines as those defined for WS-BPEL `onEvent` handlers in Section 12.7.1 of the WS-BPEL specification [39].

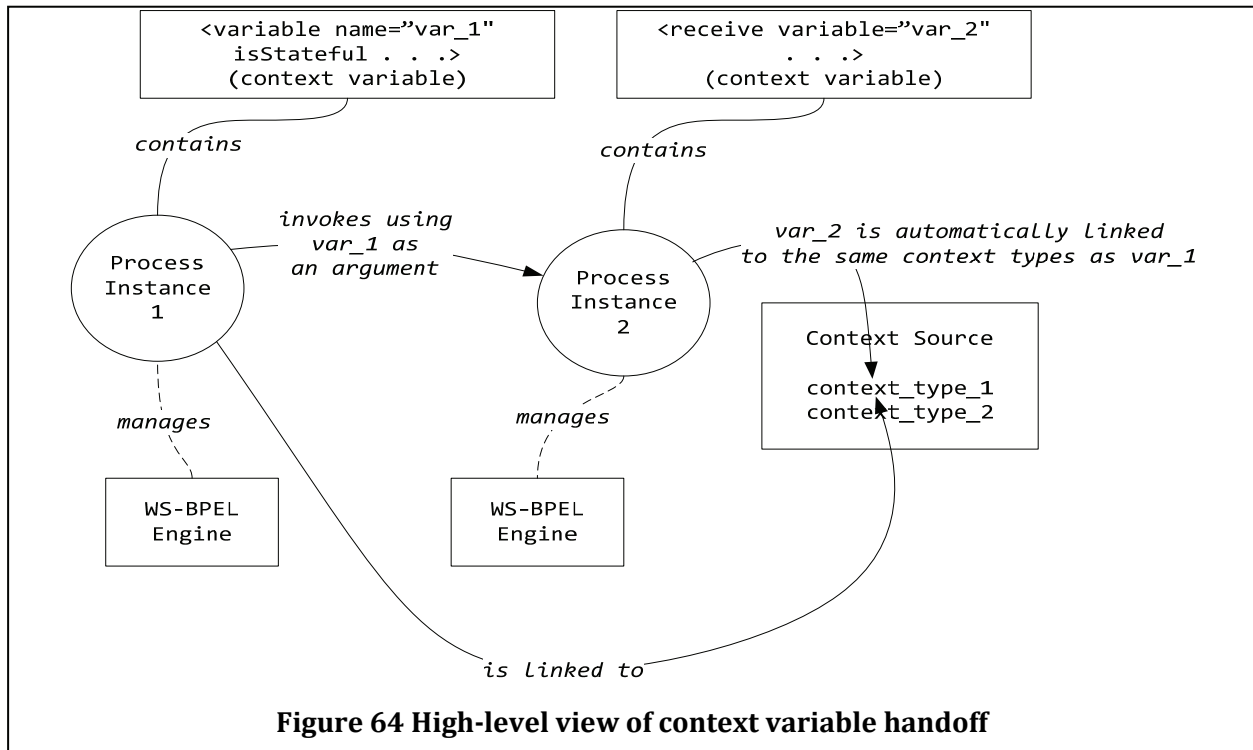
The design of context handlers diverges from that of WS-BPEL event handlers in two significant ways:

1. Restrictions on the access and use of in-process variables
2. Concurrent execution of handlers

First, each context handler can query or modify all the variables in all its ancestor scopes. It is up to the WS-BPEL engine to use good concurrency practices to ensure that this variable access is done properly. Note that this differs from the rules governing variable access in WS-BPEL `onEvent` handlers. When it comes to `onEvent` handlers, the WS-BPEL specification states that “variable references are resolved to the associated scope only and **MUST NOT** be resolved to the ancestor scopes”. Our approach to variable access has two advantages:

1. It is in line with how other WS-BPEL activities treat process variables
2. It increases the utility of context handlers

The second divergence in behavior concerns the execution of context handlers. WS-BPEL allows multiple event handlers and multiple instances of the same event handler to execute simultaneously. Although this approach is extremely open-ended, it makes it difficult for process programmers to visualize how their processes will react to incoming messages. Consequently, we propose an alternative: only a single context handler instance should execute at a time when a



context variable changes. If multiple context handlers can execute per change, then their executions should be serialized. This however, raises the following issues:

- How should handler execution be serialized if multiple context handlers fire on a context-variable change?
- How much control over serialization order should we give programmers?
- How should context handlers respond if the variable they're linked to changes multiple times during a handler execution sequence?

Clearly much more research into current designs in GUI toolkits, event-based systems and event-based WS-BPEL is required before we can fully specify this aspect of context-handler execution.

### 7.3 CONTEXT HANDOFF

Another enhancement is *context variable handoff*. This occurs when a WS-BPEL process transfers or copies its subscription to a context source by sending a context variable as input in a WSDL operation invoked on another process. The high-level operation of context variable handoff is shown in Figure 64.

Consider the following use case for this enhancement. Imagine a teleradiology business process where radiology tests can be sent for analysis to a variety of external partners offering different levels of service based on a patient's condition. A business process that orchestrates interactions with these partners can view the patient's health status as context, and use a context variable to represent this health status within the process; it can then upgrade or downgrade the service level based on changes to this variable. Imagine that once analysis results are received from the external service, they are forwarded to another business process in the patient's local health unit. If context variable handoff is implemented, this second process does not have to initiate another subscription to the context type representing the patient's health status. Instead, the invoking process simply transfers its own subscription by including the context variable in the invocation message.

Although describing context variable handoff is simple, it is not obvious how to implement it in a standards-compliant manner. Remember that the link between a context source and a WSN consumer (the WS-BPEL engine) is represented using a subscription resource. The WSN specification does not support a standardized way to change the consumer endpoint in a subscription, so this would require the specification to be extended. Alternatively, context variable handoff could be implemented by reusing the "web service parameter extension" detailed in Section 4.2. Since the WS-BPEL engine already has the context source addresses and context types used in the context variable, it can embed these details as context parameters in the outgoing message. This, as with regular web services, would require changes in the process' WSDL schema.

Context variable handoff turns a WS-BPEL process and engine into more than a consumer of context information – it creates another web service that provides links to context sources, and thus expands the number of context-enabled services available. This builds a context-aware web-services environment from the ground up.

#### 7.4 CONTEXT SOURCES WITH SEMANTICS

A major advantage to using WS-Resources to model context sources is that it places few restrictions on the format of resource properties. Resource properties do not have to conform to a hierarchy or ontology, are not limited to pre-existing types, and are not required to have machine-interpretable semantics. Beyond a few limitations on how the Resource Properties Document is

```

<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://delivery.company/shipping/Shipment"
  xmlns:sawSDL="http://www.w3.org/ns/sawSDL">

  <xsd:element name="ShipmentCondition" type="xsd:string"
    sawSDL:modelReference="http://www.retailer.com/ontology/shipment#status"/>

  <xsd:element name="ShippingResourceProperties">
    <xsd:complexType>
      <xsd:sequence>
        .
        .
        .
        <xsd:element ref="ship:ShipmentCondition" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

**Figure 65 Use of modelReference attribute with ShipmentCondition context type**

declared, the WS-ResourceProperties specification gives programmers considerable flexibility in defining the structure and content of each resource property. This allows a WS-Resource to act as a base on which to build further enhancements. One such enhancement is the idea of semantically-marked-up context sources.

Our context source design does not describe how machine-interpretable semantics can be associated with a context source's context types. Instead, each context type's meaning is communicated in the same way as with many APIs – through the use of developer documentation. As services become more flexible however, and the variety and complexity of the compositions they participate in increases, developer documentation alone will not suffice. Instead, explicitly indicating the semantics of data and having the underlying architecture automatically check it (akin to type checking) will become increasingly important. This would prevent semantic mismatches when composing services, reduce programmer error, and open the door to some automated service composition.

We draw on research from METEOR-S [31] to create semantically-marked-up context sources. METEOR-S outlines a method of adding semantic annotations to WSDL using the WSDL extensibility mechanism; their annotations are later codified in the SAWSDL specification [63]. SAWSDL describes how semantic annotations can be added to WSDL operations and WSDL-contained XML-Schema types using the `modelReference` and `schemaMapping` extension attributes. Of the two we are interested in `modelReference`, full details of which are in Section 4 of the SAWSDL specification [63]. The `modelReference` attribute allows WSDL designers to annotate a schema-

defined element, complex type or simple type with one or more URIs linking them to existing concepts in a semantic model. It would be a simple step to combine the `modelReference` attribute with the context types in a context source's Resource Properties Document. The code sample in Figure 65 shows how `modelReference` can be used to add semantics to the `Shipment-Condition` context type used in our scenario. For each context type, the `modelReference` attribute would contain URIs linking that context type to concepts in various ontologies. This would allow automated reasoning systems to match context sources to WS-BPEL processes, compare context types from different context sources, *etc.*

Although this method of adding semantics to context sources is not as expressive as alternatives built on OWL-S [62] or WSMO [65], it has a number of advantages. First, it requires minimal changes to existing WSDL interfaces. Current tooling can also be used. Moreover, developers do not have to learn a completely new interface language like OWL-S or WSMO; they can use the development toolchain they're comfortable with, and can build on their existing knowledge of web-service standards. Finally, semantic annotations can be used piecemeal, since not all context types require semantic information. Consequently, developers can add the `modelReference` attribute on an "as needed" basis, linking context types to concepts in new and existing ontologies as the situation demands.

## 7.5 OTHER ENHANCEMENTS

There are, of course, many possible enhancements that utilize context variables. One is context-variable history where, through the use of an expression language, the programmer could retrieve snapshots of a context variable's previous value. Another enhancement is a WSDL extension attribute for an operation's output message which, when used, would indicate that the message contained context parameters. All the enhancements in this chapter can be implemented and trialed on the prototype developed for this thesis.

## Chapter 8

### CONCLUSIONS

As organizations are tasked with speedily responding to changes in their business environment, their WS-BPEL business processes will have to account for more and more externally-driven environment state, or context. Current methods of sourcing and maintaining context in WS-BPEL are limited: they intersperse too many state maintenance activities within the process, require substantial interface changes, and introduce excessive coupling between the process and the context source. They also require *a priori* knowledge of the context source or the process interface. This causes a number of problems:

1. It limits a developer's ability to design one context source and have it be used by multiple business processes
2. It is difficult to write WS-BPEL processes that interact with context sources discovered at runtime
3. As the amount of context used by a process increases, the state-maintenance activities required to source and update this context obscures the business logic, making the process harder to read and maintain

These problems make it substantially harder for WS-BPEL programmers to implement concise, yet highly-adaptable business processes.

This thesis presents a solution to this problem based on the idea of a context variable. Context variables are designed using the WS-BPEL language extension mechanism. Each context variable represents a piece of context in a WS-BPEL process. Context variables use the pub/sub MEP for updates: each one is subscribed to a context type at an external context source; when the value of that context type changes, a notification is delivered to the variable and its value is automatically updated – without intervention from the programmer or the use of additional in-process activities. This allows WS-BPEL programmers to write business processes that depend on many pieces of external state while minimizing the use of a of state-maintenance activities. Furthermore, this thesis shows how context sources can be built in a standards-compliant manner using:

1. Constructs from WS-Resource Framework



## 2. Pub/sub as provided by WS-Notification

It then presents a standards-compliant context-parameter extension to web-service messages that allows other web services to send context-source references to WS-BPEL processes. Together, context variables, context sources and context-parameter extensions form an integrated system that allows developers to build generic context sources for use by different context-dependent WS-BPEL processes, while simplifying the development of these processes.

This thesis concludes by detailing the prototype used to validate the proposed solutions. Built using a modified, standards-compliant WS-BPEL engine, a JaveEE web service and a WSRF framework, this prototype demonstrates that the integrated system can be realized using existing technologies. A simple three-actor scenario with a manufacturer's WS-BPEL process, a supplier's web service, and a shipping company's context-source system, was used to exercise this prototype. The scenario shows that using context-variables in WS-BPEL processes requires substantially fewer state-maintenance activities and WSDL interface changes than either polling or onEvent message handlers. It also demonstrates how the proposed architecture allows state maintenance to be automated in WS-BPEL, making it easier for processes to interact with context sources without programmer intervention.

The approach outlined in this thesis offers a number of advantages: it is based on a simple conceptual model, builds on current frameworks, languages and standards, and substantially reduces in-process state delivery and maintenance logic when compared to either polling or onEvent message handlers. These factors make it much easier for business-process programmers to write concise, yet flexible WS-BPEL business processes. Finally, since it is standards compliant, the outlined approach can be easily used with existing toolkits and frameworks.

## BIBLIOGRAPHY

- [1] Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., and Pinkerton, M.: Cyberguide: A mobile context-aware tour guide, *Wireless Networks*, vol. 3, iss. 5, pp. 421-433, 1997.
- [2] Abowd, G.D., and Mynatt, E.D.: Charting Past, Present, and Future Research in Ubiquitous Computing, *ACM Transactions on Computer-Human Interaction*, vol. 7, iss. 1, pp. 29-58, 2000.
- [3] Active Endpoints Inc., A.S.I., BEA Systems Inc., IBM Corp., Oracle Inc., and SAP AG. WS-BPEL Extension for People (BPEL4People), Version 1.0.  
<http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/>, 2007.
- [4] Andrews, T., Curbera, F., Dholakia, H., Golan, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. Business Process Execution Language for Web Services version 1.1. <http://www.ibm.com/developerworks/library/specification/ws-bpel/>, 2003.
- [5] Apache Axis. Axis Architecture Guide. <http://ws.apache.org/axis/java/architecture-guide.html>, 2005.
- [6] Apache Muse. Apache Muse - Deployment Descriptor.  
<http://ws.apache.org/muse/docs/2.2.0/manual/architecture/deployment-descriptor.html>, 2007.
- [7] Apache Muse. Apache Muse - Programming Model.  
<http://ws.apache.org/muse/docs/2.2.0/manual/architecture/programming-model.html>, 2007.
- [8] Apache Muse. Apache Muse - WSDL2Java Tool.  
<http://ws.apache.org/muse/docs/2.2.0/manual/tools/wsdl2java.html>, 2007.
- [9] Baresi, L., and Guinea, S.: Towards Dynamic Monitoring of WS-BPEL Processes, *ICSOC*, 2005, pp. 269-282.
- [10] BEA Systems. Web Services Metadata for the Java Platform.  
<http://jcp.org/en/jsr/detail?id=181>, 2005.
- [11] Box, D., Cabrera, L.F., Critchley, C., Curbera, F., Ferguson, D., Geller, A., Graham, S., Hull, D., Kakivaya, G., Lewis, A., Lovering, B., Mihic, M., Niblett, P., Orchard, D., Saiyed, J., Samdarshi, S., Schlimmer, J., Sedukhin, I., Shewchuk, J., Smith, B., Weerawarana, S., and Wortendyke, D. Web Services Eventing (WS-Eventing). <http://www.w3.org/Submission/WS-Eventing/>, 2004.
- [12] Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F., Lovering, B., Lucco, S., Millet, S., Mukhi, N., Nottingham, M., Orchard, D., Shewchuk, J., Sindambiwe, J., Storey, T., Weerawarana, S., and Winkler, S. Web Services Addressing (WS-Addressing). <http://www.w3.org/Submission/ws-addressing/>, 2004.
- [13] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., and Shan, M.-C.: Adaptive and Dynamic Service Composition in eFlow, *CAiSE*, 2000, pp. 13-31.
- [14] Casati, F., and Shan, M.-C.: Event-Based Interaction Management for Composite E-Services in eFlow, *Information Systems Frontiers*, vol. 4, iss. 1, pp. 19-31, 2002.
- [15] Chandy, K.M., Aydemir, B.E., Karpilovsky, E.M., and Zimmerman, D.M.: Event-driven architectures for distributed crisis management, *PDCS*, 2003, pp. 803-808.
- [16] Chen, G., and Kotz, D.: Solar: An Open Platform for Context-Aware Mobile Applications, vol., iss.
- [17] Chen, H., Finin, T., and Joshi, A.: An ontology for context-aware pervasive computing environments, *The Knowledge Engineering Review*, vol. 18, iss. 3, pp. 197-207, 2003.
- [18] Codehaus Foundation. Groovy. <http://groovy.codehaus.org>, 2008.

- [19] Dey, A.K.: Understanding and Using Context, *Personal and Ubiquitous Computing*, vol. 5, iss. 1, pp. 4-7, 2001.
- [20] eviWare. The Web Service, SOA and SOAP Testing Tool - soapUI. <http://www.soapui.org>, 2008.
- [21] Fidler, E., Jacobsen, H.A., Li, G., and Mankovski, S.: The PADRES Distributed Publish/Subscribe System, *ICFI*, 2005, pp. 12-30.
- [22] Gelernter, D.: Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, vol. 7, iss. 1, pp. 80-112, 1985.
- [23] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., and Weihl, B. Publish-Subscribe Notification for Web Services. <http://www.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf>, 2004.
- [24] Gu, T., Pung, H.K., and Zhang, D.Q.: A service-oriented middleware for building context-aware services, *Journal of Networking and Computing Applications*, vol. 28, iss. 1, pp. 1-18, 2005.
- [25] IBM, and SAP AG. WS-BPEL 2.0 Extensions for Sub-Processes. <http://www.ibm.com/developerworks/library/specification/ws-bpelsubproc/>, 2005.
- [26] Jacco Brok, B.K.E.M.H.J.B.: Enabling new services by exploiting presence and context information in IMS, *Bell Labs Technical Journal*, vol. 10, iss. 4, pp. 83-100, 2006.
- [27] Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F.A.L.F., and Buchmann, A.A.B.A.: Extending BPEL for run time adaptability, *EEDOC*, 2005, pp. 15-26.
- [28] Keidl, M., and Kemper, A.: Towards context-aware adaptable web services, *WWW*, 2004, pp. 55-65.
- [29] Klingemann, J., Wäsch, J., and Aberer, K.: Adaptive Outsourcing in Cross-Organizational Workflows, *CAiSE*, 2000, pp. 417-421.
- [30] Korpipaa, P., Mantyjarvi, J., Kela, J., Keranen, H., and Malm, E.J.: Managing context information in mobile devices, *Pervasive Computing, IEEE*, vol. 2, iss. 3, pp. 42-51, 2003.
- [31] Large Scale Distributed Information Systems (LSDIS). METEOR-S: Semantic Web Services and Processes. <http://lsdis.cs.uga.edu/projects/meteor-s/>, 2005.
- [32] Lemlouma, T., and Layaida, N.: Context-aware adaptation for mobile devices, *MDM*, 2004, pp. 106-111.
- [33] Li, G., Muthusamy, V., and Jacobsen, H.A.: NIÑOS: A distributed service oriented architecture for business process execution, Middleware Systems Research Group, Toronto, ON, Technical Report, July 2007.
- [34] Lucchi, R., and Zavattaro, G.: WSSecSpaces: a secure data-driven coordination service for Web Services applications, *ACM Symposium on Applied Computing*, 2004, pp. 487-491.
- [35] Maamar, Z., Narendra, N.C., and Sattanathan, S.: Towards an ontology-based approach for specifying and securing Web services, *Information and Software Technology*, vol. 48, iss. 7, pp. 441-455, 2006.
- [36] Martin, D.: Putting Web Services in Context, *Electronic Notes in Theoretical Computer Science*, vol. 146, iss. 1, pp. 3-19, 1/24, 2006.
- [37] McIlraith, S.A., Son, T.C., and Zeng, H.: Semantic Web Services, in Editor (Ed.)^(Eds.): 'Book Semantic Web Services' (2001, edn.), pp. 46-53
- [38] OASIS. Web Services Base Notification 1.3. [http://docs.oasis-open.org/wsn/wsn-ws\\_base\\_notification-1.3-spec-os.pdf](http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf), 2006.

- [39] OASIS. Web Services Business Process Execution Language Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, 2007.
- [40] OASIS. Web Services Business Process Execution Language Version 2.0 Primer. <http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.htm>, 2007.
- [41] OASIS. Web Services Context Specification Version 1.0. <http://docs.oasis-open.org/ws-caf/ws-context/v1.0/wsctx.html>, 2007.
- [42] OASIS. Web Services Resource 1.2 (WS-Resource). [http://docs.oasis-open.org/wsrp/wsrp-ws\\_resource-1.2-spec-os.pdf](http://docs.oasis-open.org/wsrp/wsrp-ws_resource-1.2-spec-os.pdf), 2006.
- [43] OASIS. Web Services Resource Framework 1.2. [http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=wsrf](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf), 2006.
- [44] OASIS. Web Services Resource Framework - Primer v1.2. <http://docs.oasis-open.org/wsrp/wsrp-primer-1.2-primer-cd-02.pdf>, 2006.
- [45] OASIS. Web Services Resource Metadata 1.0 (WS-ResourceMetadataDescriptor). [http://docs.oasis-open.org/wsrp/wsrp-ws\\_resource\\_metadata\\_descriptor-1.0-spec-cs-01.pdf](http://docs.oasis-open.org/wsrp/wsrp-ws_resource_metadata_descriptor-1.0-spec-cs-01.pdf), 2006.
- [46] OASIS. Web Services Resource Properties 1.2 (WS-ResourceProperties). [http://docs.oasis-open.org/wsrp/wsrp-ws\\_resource\\_properties-1.2-spec-os.pdf](http://docs.oasis-open.org/wsrp/wsrp-ws_resource_properties-1.2-spec-os.pdf), 2006.
- [47] OASIS. Web Services Service Group 1.2 (WS-ServiceGroup). [http://docs.oasis-open.org/wsrp/wsrp-ws\\_service\\_group-1.2-spec-os.pdf](http://docs.oasis-open.org/wsrp/wsrp-ws_service_group-1.2-spec-os.pdf), 2006.
- [48] OASIS. WS-Topics 1.3. [http://docs.oasis-open.org/wsn/wsn-ws\\_topics-1.3-spec-os.pdf](http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf), 2006.
- [49] Pawan, C., An, L., Jeng, J.J., and Chen, S.K.: Enterprise integration and monitoring solution using active shared space, *ICEBE*, 2005, pp. 665-672.
- [50] Plale, B., Gannon, D., Huang, Y., Kandaswamy, G., Pallickara, S.L., and Slominski, A.: Cooperating Services for Data-Driven Computational Experimentation, *Computing in Science & Engineering*, vol. 7, iss. 5, pp. 34-43, 2005.
- [51] Ranganathan, A., and McFaddin, S.: Using workflows to coordinate Web services in pervasive computing environments, *ICWS*, 2004, pp. 288-295.
- [52] Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., and Nahrstedt, K.: Gaia: a middleware platform for active spaces, *SIGMOBILE Mobile Computing Communications Review*, vol. 6, iss. 4, pp. 65--67, 2002.
- [53] Schilit, B., Adams, N., and Want, R.: Context-Aware Computing Applications, *IEEE Workshop on Mobile Computing Systems and Applications*, 1994, pp. 85-90.
- [54] Stal, M.: Web services: Beyond Component-Based Computing, *Communications of the ACM*, vol. 45, iss. 10, pp. 71-76, 2002.
- [55] Sun Microsystems. The Java API for XML-Based Web Services (JAX-WS) 2.1. <http://jcp.org/en/jsr/detail?id=224>, 2007.
- [56] Sun Microsystems. The Java™ Architecture for XML Binding (JAXB) 2.0. <http://jcp.org/en/jsr/detail?id=222>, 2006.
- [57] Sun Microsystems. The Java™ Architecture for XML Binding (JAXB) 2.1. <http://jcp.org/en/jsr/detail?id=222>, 2006.
- [58] Sun Microsystems. Java™ Platform, Enterprise Edition (Java EE) Specification, v5. <http://jcp.org/en/jsr/detail?id=244>, 2006.
- [59] Sun Microsystems. SOAP with Attachments API for Java™ (SAAJ) 1.3. <http://jcp.org/en/jsr/detail?id=67>, 2005.
- [60] Sun Microsystems. Web Services for Java EE, Version 1.2. <http://jcp.org/en/jsr/detail?id=109>, 2006.

- [61] Vukovic, M., and Robinson, P.: Adaptive, planning based, web service composition for context awareness, *International Conference on Pervasive Computing*, 2004, pp.
- [62] W3C. OWL-S: Semantic Markup for Web Services. <http://www.w3.org/Submission/OWL-S/>, 2004.
- [63] W3C. Semantic Annotations for WSDL and XML Schema. <http://www.w3.org/TR/sawSDL/>, 2007.
- [64] W3C. Simple Object Access Protocol (SOAP) 1.1 <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>, 2000.
- [65] W3C. Web Service Modeling Ontology (WSMO). <http://www.w3.org/Submission/WSMO/>, 2005.
- [66] W3C. Web Services Addressing 1.0 - SOAP Binding. <http://www.w3.org/TR/ws-addr-soap/>, 2006.
- [67] W3C. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>, 2001.
- [68] W3C. XML Schema Part 0: Primer. <http://web4.w3.org/TR/xmlschema-0/>, 2004.
- [69] W3C. XML Schema Part 1: Structures. <http://web4.w3.org/TR/xmlschema-1/>, 2004.
- [70] Want, R., Hopper, A., Falcão, V., and Gibbons, J.: The active badge location system, *ACM Transactions on Information Systems (TOIS)*, vol. 10, iss. 1, pp. 91-102, 1992.
- [71] Yau, S.S., Karim, F., Wang, Y., Wang, B., and Gupta, S.K.S.: Reconfigurable Context-Sensitive Middleware for Pervasive Computing, *Pervasive Computing, IEEE*, vol. 1, iss. 3, pp. 33-40, 2002.
- [72] Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., and Chang, H.: QoS-Aware Middleware for Web Services Composition, *IEEE Transactions on Software Engineering*, vol. 30, iss. 5, pp. 311-327, 2004.