BPEL Orchestration of Secure WebMail

Saket Kaushik, Duminda Wijesekera and Paul Ammann
Department of Information and Software Engineering,
George Mason University,
Fairfax, VA 22030 USA
{skaushik|dwijesek|pammann}@gmu.edu

Abstract

Web Services offer an excellent opportunity to redesign and replace old and insecure applications with more flexible and robust ones. WSEmail is one such application that replaces conventional message delivery systems with a family of Web Services that achieve the same goal. In this paper we analyze the existing WSEmail specification against the standard set of use cases (and misuse cases) supported (resp. prevented) by SMTP implementations – the current default message delivery infrastructure – and augment it with several missing pieces. In addition, we show how the WSEmail family of Web Services, specified in WSDL, can be orchestrated using BPEL. Finally, we provide a synchronization analysis of our WSEmail orchestration and show its correctness.

1. Introduction

Increasing misuse of conventional email systems by `spammers', bulk email senders and fraudsters has raised concerns about the security and trustworthiness of the existing message delivery infrastructure. Many have suggested replacing the existing system with a more secure system. WSEmail [17] is one such proposal that promotes the use of Web Services for the purpose of email delivery, while claiming to prevent several *misuse cases*. Additionally, for backward compatibility among the user community, any replacement system must support the standard *use cases* of the conventional distributed software systems that implement SMTP [14] protocol for email delivery, while ensuring secure operations. Therefore, an implicit requirement is that the replacement system must provide specific security guarantees for the supported use cases that their conventional counterparts do not. However, to the best of our knowledge, such a study has not yet been provided. This missing piece is provided by this paper.

Supporting standard use cases for email delivery implies that in addition to supporting message routing, middleware functionality for achieving persistent asynchronous message transmission[19] must be provided. Since message transmission consists of several sub processes running in parallel or sequence, the WSEmail family of web services for email delivery are best expressed in a process specification and integration framework, like BPEL[2], the approach taken here. In our approach, we specify common Web Services required for email delivery, and analyze their composition for the satisfaction of security goals.

Incidentally WSEmail [17] focuses on providing a flexible means of communication, such as, being expected to dynamically discover and negotiate a communication protocols, as evidenced by using extensions such as Instant Messaging (IM), *etc*. However, the authors don't provide the details of the negotiation language, or the process by which dynamic configurations are made possible. In addition, it is not clear whether and how standard set of use cases for email delivery are supported. Finally, designing new sets of protocol for each use case may be prohibitively expensive (for instance, authors report existence of 68 interfaces and 343 classes organized in 30 projects). We take an alternate approach of supporting standard, but configurable, protocols for message delivery and provide theoretical analyses for security verification of processes that is missing in earlier proposals.

AMPol [1] extends WSEmail by separating policies from delivery mechanisms. However, this system suffers from lack of adequate specification and a formal methodology for interoperation. In other words, policy rules are not given a formal meaning, and consequently they can be interpreted differently by the WSEmail partners. This causes well known problems like ambiguity over satisfaction of contracts, or establishing failure, *etc*. In contrast, email feedback approach[12] for conventional SMTP implementations does not suffer from this problem. Here, we apply this solution to WSEmail, and leverage on the results shown earlier for security properties. In addition, we provide synchronization and verification analyses, which are missing from both approaches cited above.

2. Use cases and misuse cases

We enumerate use cases supported by the existing email delivery system built around the SMTP protocol and misuse cases that have been raised during the recent past against the system. In addition, we cover various mechanisms that help prevent these misuse cases. This clearly demarcates the difference between the WSEmail/AMPol and this work (since they don't consider all conventional use cases and misuses). Standard SMTP use cases are as follows:

Use Case 1: Best effort transmission of a text message from a sender (the principal actor) to a recipient (the secondary actor) through intermediate mail servers.

A message transmission is considered complete only if the message is routed to the recipient's machine or mail account (mailbox). Transmission is broken down into three logical steps: Transmission from senders to their email service providers (SESP); transmission from sending email service provider (SESP) to recipient email service provider (RESP); and finally from RESP to the message recipient. However, in practice multiple mail servers may be involved and are subsumed under the logical entities – SESP and RESP.

Use Case 2: Error reporting on transmission failure.

Message transmission is a transaction that is initiated by a sender and completes when the message reaches its destination. This transaction can fail due to a variety of reasons. The standard error reporting convention is to generate report on failure. Error messages are generated on the point of failure and the infrastructure tries to deliver them to the senders. Together, the two use cases are said to provide *reliable transmission*.

Specialization of use cases: Above two use cases can be specialized for a variety of message types and properties of transmission channels. Standard use cases supported by SMTP implementations are:

1. Reliable transmission of a text message.
2. Reliable transmission of a multipart MIME message [5].
3. Reliable transmission of an authenticated text/MIME message.
4. Reliable transmission of a message over an encrypted channel.
5. Reliable transmission of message receipts.

Specializations, described above, involve reliable asynchronous transfer of messages across hosts on the internet. Since message transmission is asynchronous – a recipient process may not be active when the sender process sends the message – hence, transmission infrastructure provides only best effort delivery. If transmission fails at any stage of the message processing pipeline, an error message is constructed at that stage and delivered (with best effort delivery) to the source. In addition, SMTP extensions [18, 9] includes commands and replies for source authentication and negotiations for establishing a secure channel for synchronous transmission. Finally, SMTP allows message delivery receipts. *i.e.*, a mail message that acknowledges receipt of message sent back to the sender from the recipient. To support this use case, sending and receiving mail servers reverse their *roles* or the recipient initiates a new SMTP session for transmitting a standard reply in response to the original message.

Email delivery is subject to many misuse cases. These include lack of authentication, loss of privacy and integrity of content, vulnerability to unsolicited commercial email (spam), email bombs [4], *etc*. We describe these cases next.

1. *Integrity and privacy of data*: in spite of availability of STARTTLS command, most email messages are sent in cleartext over the wire. Messages may be stored in cleartext at a mail server though message transmission may be encrypted. This misuse case can be prevented through additional caution.
2. *Absence of sender authentication*: SMTP AUTH command is insufficient for end-to-end sender authentication, since it requires prior exchange of secret data between senders and recipients. As a result, sender address spoofing is possible, due to which, non-repudiation of message initiation cannot be guaranteed.
3. *Vulnerability to email bombs*: This is a variation of DoS attack on email networks. Mail servers are vulnerable to being overwhelmed by a large number of incoming messages, leading to denial of email service.
4. *Vulnerability to unsolicited commercial email (spam)*: Because of recipient's lack of control over which messages are delivered to their mailbox, they become vulnerable receiving unsolicited commercial or fraudulent mail.

Of the above, vulnerability to spam has seriously dented the utility of email service. To counter this problem, automated recipient controls are added during the transmission process to control delivery of messages. Several such control mechanisms are in use, and we cover important ones here. Because of these controls, we add an additional use case to the standard list of SMTP enabled use cases, which is, provision of feedback about rejected messages due to failure of acceptance criteria [12]. A drawback of delivery controls is the introduction leakage channels through which sensitive information can be lost [13]. We also add this misuse case to the list of standard misuse cases and cover it in section 7.

2.1 Our contribution

Our main contribution is a specification, design and verification of WSMail, an end-to-end, web-based mail service using WSDL and BPEL that supports all stated use cases and prevents all stated misuse cases.

While authors consider transmission of MIME messages in [17] in parts, a new requirement, they don't provide details of standard use cases considered here. In addition, misuse cases discussed here are alluded to (in [1]) but the authors don't provide sufficient evidence that their solution will prevent the cases discussed here. We fill these gaps, as shown later.

The rest of the paper is organized as follows. In section 3, we give a brief overview of conventional email transmission and transmission using Web Services. Next, we present WSDL specifications of the family of WSEmail Web Services. Process integration in BPEL is presented in section 5, while section 6 addresses coverage of use case and prevention of misuse cases. We tackle process integrity in section 7. In section 8 we discuss a new misuse case and how we prevent it.

3. Overview of message delivery

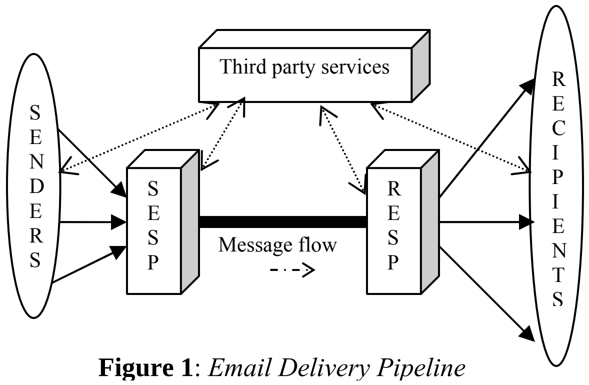

Figure 1: *Email Delivery Pipeline*

As shown in figure 1, a conventional email message begins its journey at the (the principal actor) sender's machine and is initially routed to the sender's email service provider (ESP). Email service provider then transmits the message on behalf of the sender to the recipient's service provider. From here the recipient picks up the delivered message [15]. Email service providers help scale email messaging to the level of internet users in addition to providing several important services. First, ESPs dispense with the need for senders and recipients to be online for communicating with each other, *i.e.*, they enable persistent asynchronous communications. Also, they are well placed to provide value added services for their subscribers. For example, commercial unsolicited messages filtration, removal of malicious code commonly bundled with messages, *etc.*, can be undertaken by the ESP. In addition to these four principals, several other actors may be involved with message transmission. These include third party information sources, like, reputation services (DCC [11], Cloudmark [10]) or escrow services (for attention bonds [16]), *etc.* Other principles consult third parties during transmission to ascertain properties of messages – like, authenticity of bonds or whether a message is a `bulk' email message or not. These interactions are represented by dotted double arrows in figure 1.

In a Web Services based message transmission, we replace each actor by one or more Web Services. Together these Web Services form a family referred to as the *WSEmail family*. Here we show different orchestrations of these Web Services providing many *flavors* of email transmissions. We also show that earlier solutions engineered for conventional systems can be readily adapted for the Web Services environment and possibly improved upon.

4. Web Services for message transmission

In this section we begin with the basic technical details of our model. Three basic components are considered for our specifications. First, we describe the types and parts of messages that are exchanged between Web Services. Then, we specify various Web Services that constitute the WSEmail family. Finally, we specify various orchestrations of the WSEmail family using BPEL process specifications.

4.1 Message types

Message types define the protocol used for communication, *i.e.*, service interfaces are understood in terms of their input and output message. Here, we limit the types of transported objects, however, our list is extensible and it is possible to include the complete set of MIME[8] objects. Basic types are described in Table 1, and complex (i.e. structural) types are described in table 2. We give these type definitions for completion. We don't intend to leverage on their type structure for the purposes of this paper. Our code (shown later) can be modified to be used with other typed structures as well. For instance, several techniques use custom structures for `time' or `credential', *etc.*, so we simply refer to them using an XML namespace element. Please note that we use the characters `*', `?', and `+' in the same sense of use as in BPEL manual [2], *i.e.*, .`*' means zero or more repetitions, `?' meaning zero or one occurrence and `+' means one or more repetitions.

Type Name	Primitive Type	Example
MIME	ASCII string	Application/PDF
PKISignature	ASCII String	463hfd$&47654
Message ID	Long Int	239809832092
MType	Character string	Urgent, Personal, …
WantAck	Boolean	Yes/No
Number	Positive Int	100
Nonce	Positive Int	10000
Email Address	ASCII string	abc@xyz.com
Password	ASCII string	******
Answer	ASCII string	Xy3

Table 1: *Basic types of message elements*

Type Name	Type Structure	Example
Time	XmlNS=URI#Time	10:00 A.M EST
Key Pair	IntXInt	(53,97)
Credential	XmlNS=URI#Cred	Credential struct
Image	XmlNS=URI#Jpeg	JPEG struct
AObject	Application/Type	PDF file
Credential Chain	Credential*	Cred1, …, CredN
Currency	Enum: {$, £}	$, £
Bond	XmlNS=URI#Bond	$3.5 Cred 1
Turing test	Image	10101..01,
Turing test reply	ImageXAnswer	(10101..01, xy3)
Content	String?, AObject*	"Example", Image

Table 2: *Complex types of message elements*

```
<types>
 <schema targetNS="uri1" xmlns="schema1">
   <element name="Content" type="String">
   </element>
      .
      .
      .
   <element name="Turing Test">
    <complexType>
        <all>
        <element name="Image"
                 type="Application/JPEG"/>
        <element name="Answer"
type="String"/>
        </all>
   </complexType>
   </element>
  </schema>
</types>
```

Listing 1: *Basic types in WSDL*

Elements described in table 1 are expressed in Web Services Description Language (WSDL) [7] in the syntax shown in listing 1 (we omit all the details here).

4.2 Messages

Next, we describe message types that are transmitted between Web Services. First, we detail the structure of a *mail message*; that is, a message that is initiated by the message sender and is delivered to the intended recipient. This message consists of routing information, objects to be transmitted and additional attributes that aid the delivery of the message. Additional attributes are added by senders to signal the utility of a message to the recipient. They are used by downstream processes to make routing decisions [12]. Mail message is described in WSDL format in listing 2.

```
<message name="MailMessage">
 <part name="From"
       element="Email Address"/>+
 <part name="To" element="Email Address"/>+
 <part name="Date" element="Time"/>+
 <part name="ID" element="Message ID"/>+
 <part name="Surety" element="Bond"/>?
 <part name="Pass" element="Password"/>*
 <part name="Ack" element="WantAck"/>*
 <part name="Sign"
       element="PKISignature"/>*
 <part name="RTT reply" element="Turing
                         Test Reply"/>*
 <part name="MType" element="String"/>?
 <part name="Subject" element="String"/>?
 <part name="Body" element="Content"/>?
</message>
```

Listing 2: *WSDL Mail Message*

In addition to mail messages, clients and servers transmit several other types of message enable underlying communication protocols by informing the status of the communication, properties of the transmission (QoS,) *etc*. (listed in table 3). Their WSDL syntax is shown in listings 3 and 4.

Message Type	Utility
Mail Message	Message to be delivered
Receipt notice	Notice of receipt and acceptance for delivery of a mail message
FailNotice	Notice of delivery failure
RejectNotice	Notice of delivery rejection
RefinementMessage	Changes desired in a mail message
RefinementFailure	Desired changes not possible
InformationMessage	Third party message evaluations
MailIntent	Indication of transmission intent
Service Level Accord	QoS for invocations
AcceptancePolicy	Acceptance rules advertisement
PKICertificate	Proof of identity and data secrecy

Table 3: *Types of messages*

```
<message name="ReceiptNotice">
 <part name="Date" element="Time"/>+
 <part name="ID" element="Message ID"/>+
 <part name="Sign" element="PKISignature"/>*
</message>

 <message name="FailNotice">
 <part name="Date" element="Time"/>+
 <part name="ID" element="Message ID"/>+
 <part name="Error" element="String"/>+
 <part name="Sign" element="PKISignature"/>*
</message>

<message name="RejectNotice">
 <part name="Date" element="Time"/>+
 <part name="ID" element="Message ID"/>+
 <part name="Eval Policy" element="Policy"/>+
 <part name="Sign" element="PKISignature"/>*
</message>

<message name="RefinementMessage">
 <part name="Date" element="Time"/>+
 <part name="ID" element="Message ID"/>+
 <part name="Sign" element="PKISignature"/>*
 <part name="Surety" element="Bond"/>*
 <part name="MType" element="String"/>?
 <part name="RTT" element="Turing Test"/>*
 <part name="Body" element="Content"/>*
</message>

<message name="RefinementFailure">
```

```
 <part name="ID" element="Message ID"/>+
 <part name="RError" element="String"/>+
</message>

<message name="InformationMessage">
 <part name="Date" element="Time"/>+
 <part name="ID" element="Message ID"/>+
 <part name="Information" element="String"/>+
 <part name="Sign" element="PKISignature"/>*
</message>
```

Listing 3: *WSDL Application Data*

Message definitions in listing 3 determine the application data or the payload for the message communications. Listing 4 defines protocol data exchanged for effectively completing the task at hand. In particular, *Mail Intent*, message expresses the intent to send messages, *Service Level Agreement*, message is a response to mail intent message indicating number of messages allowed; while *Mail Acceptance Rule* message states acceptable message attributes.

```
<message name="MailIntent">
 <part name="Date" element="Time"/>+
 <part name="NoOfMsgs" element="Number"/>+
 <part name="Sign" element="PKISignature"/>*
</message>

 <message name="ServiceLevelAgreement ">
  <part name="Date" element="Time"/>+
  <part name="AllowedNo" element="Number"/>+
 <part name="Sign" element="PKISignature"/>*
</message>

<message name="AcceptancePolicy">
 <part name="Date" element="Time"/>*
 <part name="Surety" element="Bond"/>*
 <part name="Sign"
       element="PKISignature"/>*
 <part name="RTT reply"
       element="Turing Test"/>*
 <part name="MType" element="String"/>*
 <part name="Body" element="Content"/>*
 <part name="Sign" element="PKISignature"/>*
</message>

<message name="PKICertificate ">
 <part name="Key" element="Credential"/>+
 <part name="Session" element="Nonce"/>*
</message>
```

Listing 4: *WSDL Control Data*

4.3 WSEmail family of Web Services

Next, we design a family of Web Services that perform various tasks to aid delivery of email messages. We list the set of externally callable methods for each principal involved in message delivery.

Sender's ESP (SESP): Sender's email service provider is designed to receive messages, route them to the destination, examine and *repair messages* before sending them, refine messages, *etc.*

1. SESPConnectPT
2. SESPReceiveMsgPT
3. SESPAuthPT
4. SESPDeliveryPT
5. SESPMsgCallbackPT
6. SESPImprovementPT
7. SESPVirusExaminationPT
8. SESPVirusRemovalPT

Sender: Sender's may need to expose a callback interface to receive rejection notices or notices for improving messages

1. SenderMsgCallbackPT
2. SenderMsgRefinementPT
3. SenderPasswdCallbackPT

Recipient's ESP (RESP): Recipient's ESP provides the following set of services.

1. RESPHeloPT
2. RESP-TLSPT
3. RESPReceiveMsgPT
4. RESPVirusScanPT
5. RESPFilterPT
6. RESPControlPT
7. RESPSanitizationPT
8. RESPDeliveryPT
9. RESPStoragePT
10. RESPImprovementPT

Recipient: A recipient need not expose any service; however, some recipients may allow their service providers to "push" messages to the recipient's host through the following service:

1. RReceiveMsgPT

In addition to services provided by the SESP and RESP, third party services may be invoked during message transmission to check their desirability. Here we restrict to two Web Services, though this list could easily be extended.

Third party services: RESP may invoke a distributed checksum service to verify if a message is a bulk message. Similarly, calls to escrow service to determine the validity of attached bonds is also possible.

1. CheckSumPT
2. BondVerificationPT

WSDL definitions of the Web Services, described above, are presented in listings 5, 6 and 7.

```
<portType name="SESPReceiveMsgPT">
 <operation name="GetMessage">
   <input message="Mail Message"/>
   <output message="Receipt Notice"/>
   <fault name="Fail" message="FailNotice"/>?
 </operation>
</portType>

```

```
<portType name="SESPConnectPT">
 <operation name="GetSLA">
   <input message="MailMessage"/>
   <output message=" IntentMessage"/>
   <fault name="Fail" message="SLAFail"/>?
 </operation>
 </portType>

<portType name="SESPAuthPT">
 <operation name="AUTH">
   <output message="PKICertificate"/>
 </operation>
</portType>

<portType name="SESPDeliveryPT">
 <operation name="SendMessage">
   <input message="MailMessage"/>
   <output message="ReceiptNotice"/>
   <fault name="Fail"
       message="FailNotice"/>?
 </operation>
</portType>
<portType name="SESPCallbackPT">
 <operation name="MessageCallBack">
   <input message="RefinementMessage"/>
 </operation>
</portType>

<portType name="SESPImprovementPT">
 <operation name="Refinement">
   <input message="RefinementMessage"/>
   <output message="MailMessage"/>
   <fault name="Fail"
       message="FailNotice"/>?
 </operation>
</portType>

<portType name="SESPExaminationPT">
 <operation name="VirusScan">
   <input message="Mail Message"/>
   <output message="Information Message"/>
 </operation>
</portType>

<portType name="SESPVirusRemovalPT">
 <operation name="VirusRemoval">
   <input message="Mail Message"/>
   <output message="Mail Message"/>
 </operation>
</portType>
```

Listing 5: *WSDL portType specs for SESP services*

```
<portType name="RESPHeloPT">
 <operation name="SLAevaluation">
   <input message=" MailIntent"/>
   <output message="ServiceLevelAccord"/>
 </operation>
</portType>

<portType name="RESP-TLSPT">
 <operation name="STARTTLS">
   <input message="PKICertificate"/>
   <output message="PKICertificate"/>
 </operation>
</portType>

<portType name="RESPReceiveMsgPT">
 <operation name="GetMessage">
   <input message="MailMessage"/>
   <output message="ReceiptNotice"/>
   <fault name="Fail”
       message="FailNotice"/>?
   <fault name="Reject" message="Reject
                          Notice"/>?
 </operation>
</portType>

<portType name="RESPVirusScanPT">
 <operation name="VirusScan">
   <input message="MailMessage"/>
   <output message="InformationMessage"/>
 </operation>
</portType>

<portType name="RESPFilterPT">
 <operation name="BayesianFiltering">
   <input message="MailMessage"/>
   <output message="InformationMessage"/>
 </operation>
</portType>

<portType name="RESPControlPT">
 <operation name="SenderRep">
   <input message="Sender"/>
   <output message="InformationMessage"/>
 </operation>
</portType>

<portType name="RESPSanitizationPT">
 <operation name="Sanitization">
   <input message="Mail Message"/>
   <output message="Mail Message"/>
 </operation>
</portType>

<portType name="RESPDeliveryPT">
 <operation name="SendMessage">
   <input message="MailMessage"/>
   <output message="ReceiptNotice"/>
   <fault name="Fail"
       message="FailNotice"/>?
 </operation>
</portType>

<portType name="RESPStoragePT">
 <operation name="StoreMessage">
   <input message="MailMessage"/>
   <fault name="Fail"
       message="FailNotice"/>?
 </operation>
</portType>

<portType name="RESPImprovementPT">
 <operation name="RefineMsg">
   <input message="MailMessage"/>
   <output message="RefinementMsg"/>
   <fault name="Fail"
       message="FailNotice"/>?
   <fault name="Reject" message="Reject
                          Notice"/>?
 </operation>
</portType>
```

Listing 6: *WSDL portType specs for RESP services*

```
<portType name="CheckSumPT">
 <operation name="DCC">
   <input message="MailMessage"/>
   <output message="InformationMessage"/>
 </operation>
</portType>

<portType name="bondVerificationPT">
 <operation name="VerifyBond">
   <input message="MailMessage"/>
   <output message="InformationMessage"/>
 </operation>
</portType>
```

Listing 7: *WSDL portType specs for third party services*

5. BPEL orchestration of WebMail

In this section, we begin with a basic set of synchronized Web Service invocations for message delivery. We present their interfaces – synchronous or asynchronous – for communication with other distributed processes. We illustrate typical activities, in the notation borrowed from BPEL specification manual by Andrews, Curbera [1], *et al*. SESP is described in figure 2 and RESP in figure 3, followed by their process descriptions (resp. listings 8 and 9). In the BPEL process specifications (listings 8 and 9) of processes we assume that `<partnerLink>` elements, identifying the roles of involved services, are already specified. Because of space limitations we omit namespace elements, variable declarations, *etc.*, and depend on the context for clarity.

5.1 SESP process specification

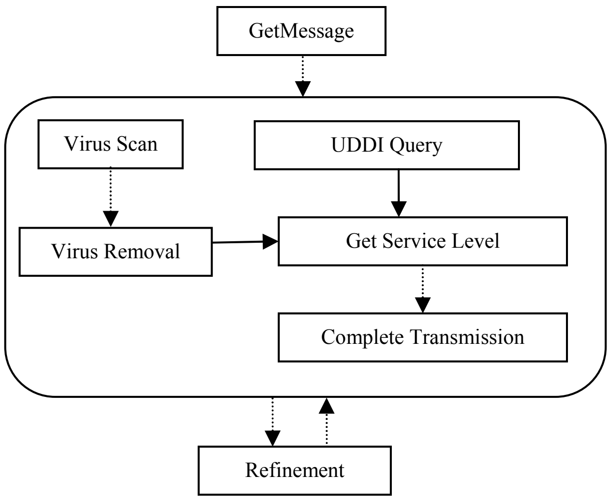

Figure 2: An *SESP Orchestration*

Dotted lines in figure 2 (and 3) indicate *sequential executions* and solid lines indicate *control dependencies for synchronizing concurrent activities*. Note that the diagram does not give details about exception handling. These cases are showcased in code later; and are ignored here for the sake of clarity.

In figure 2, SESP process waits for messages from senders. Senders invoke SESP's ReceiveMsgPT. Once the message is received, two concurrent threads of execution begin, *viz.*, scanning the received message's body for viruses and a UDDI query to locate the recipient's email service provider (RESP). If a message is found to be infected, the virus removal process is run *after* the scan is completed. Next, the SESP invokes the *HeloPT service* of the recipient to begin message delivery. Assuming that the RESP allows SESP to transmit messages through a service level agreement (SLA), SESP invokes RESP's message receiving operation RecieveMsgPT.

```
<process name="SESPProcess">
  <partnerLinks>
      <partnerLink name="transmission"
          partnerLinkType="…"
          myRole="ReceiveMsgSrv" />
              .
              .
              .
  </partnerLinks>

  <faultHandlers>
      .
      .
      .
   </faultHandlers>

  <sequence>
     <flow>
    <sequence> // New message from sender
     <receive partnerLink="transmission"
       portType="SESPReceiveMsgPT"
       operation ="SendMessage"
       variable ="M">
      </receive>
    </sequence>
    <sequence> // Refined message retransmission
     <receive partnerLink="self-transmit"
       portType="SESPReceiveMsgPT"
        operation ="SendMessage"
        variable ="M">
     </receive>
    </sequence>
    <sequence> // Call back service
        <receive partnerLink="RESP-SESP-CB">
          portType=" SESPCallbackPT">
          operation="MessageCallback"
          Variable="RefinementMsg">
        </receive>
        <invoke partnerLink="self">
          portType=" SESPImprovementPT">
          operation="MessageCallback"
          inputVariable="RefinementMsg"
          outputVariable="M">
          <throw "FailureFault"
              faultVariable="RefinementMsg">
        </invoke>
       <reply partnerLink="self-transmit">
           portType="SESPReceiveMsgPT"
            operation="SendMessage"
           variable="M">
        </reply>
    </sequence>
    <flow> // message preparation
      <links>
          <link name="fix-deliver"/>
```

```
56          <link name="UDDI-resn">
57      </links>
58      <sequence>
59        <invoke partnerLink="scanner"
60          portType=" SESPExaminationPT"
61          operation=" VirusScan"
62          inputVariable="M"
63          outputVariable="result">
64        </invoke>
65        <switch>
66        <case condition="result=true">
67        <invoke partnerLink="scanner"
68          portType="SESPVirusRemovalPT"
69          operation="VirusRemoval"
70          inputVariable="M"
71          outputVariable="M">
72          <source linkName="fix-deliver" />
73        </invoke>
74        </case>
75        <otherwise>
76              <empty />
77        </otherwise>
78        </switch>
79      </sequence>
80      <sequence> // where to send?
81        <invoke partnerLink="nameReslv"
82          portType="UDDIService"
83          operation="GetAddress"
84          inputVariable="From"
85          outputVariable="IPAddress">
86          <source linkName="UDDI-resn" />
87        </invoke>
88      </sequence>
89      <sequence> // send message to RESP
90        <invoke partnerLink="outbound"
91          portType="SESPConnectPT"
92          operation name="GetSLA"
93          inputVariable="M"
94          outputVariable="SLA">
95          <target linkName="UDDI-resn" />
96          <target linkName="fix-deliver" />
97        </invoke>
98        <while condition="number &lt; SLA">
99        <flow>
100       <sequence>
101       <invoke partnerLink="destination"
102         portType="SESPDeliveryPT"
103         operation name="SendMessage"
104         inputVariable="M"
105         outputVariable="R">
106         <catch "RejectionFault"
107             faultVariable="RejectNotice">
108            <reply partnerLink="SenderCB">
109             portType=" SenderMsgCallbackPT">
110             operation="Rejection"
111             variable="RejectNotice">
112         </catch>
113       </invoke>
114       </sequence>
115       </while>
116     </sequence>
117    </flow>
118 </sequence>
119 </process>
```

Listing 8: Example SESP Process

Listing 8 shows a typical SESP process in BPEL syntax. The code has four main blocks: headers and type declarations (lines 1 – 15), message reception (line 19–52); message preparation (lines 53–88); and message delivery (lines 89–116). The first part accepts messages from a sender, to be delivered to some recipient. In addition, the SESP process allows its message callback service to retransmit an earlier rejected (but now revised) message. In other words, messages rejected earlier, say for lack of authentication or other attributes desired by RESP, are repaired with the help of this feedback loop. Next, each message enqueued for delivery is subject to checks (like virus scan, *etc.*) to ensure good quality of each message. Finally, the message is sent across to the RESP.

5.2 RESP process specification

Next, we define an RESP process that enforces a sample service level agreement (SLA) and a reasonable message acceptance policy (AP), given informally as:

SLA	*Allow*		*10 messages per connection*
	Allow		*Feedback for rejected messages*
AP	*Accept message*	*IF*	*No virus/worm is attached*
		AND	*Filter allows receipt*
			OR
			Distributed checksum allows receipt
	Accept message	*IF*	*No virus/worm is attached*
		AND	*Message bonded with value > b*
		AND	*Bond is verified by an escrow service*

Table 4: *Sample Message Acceptance Policy*

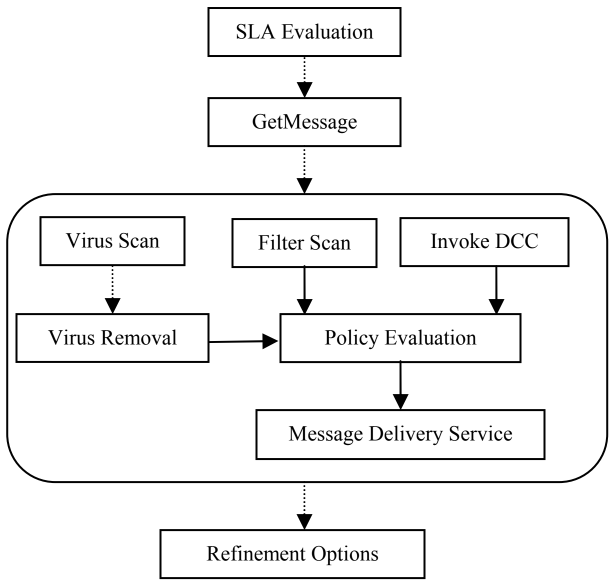

Figure 3: An *RESP Orchestration*

Upon invocation of RESP's RecieveMsgPT ("Get-Message" operation) the message is transmitted to RESP. For each received message, the RESP applies a message acceptance policy to accept or reject it. If the transmitted mail fails to satisfy this policy, the RESP either throws a *rejection notice* or a *refinement message*. The refinement message suggests changing some parts of the message that may make it acceptable to the RESP. As a result, refinement activity may begin at the SESP. Note that based on its own policy, an SESP may decide to ignore all advice, and consequently, the callback service interface may not be exposed (the current strategy used by existing SMTP implementations). On the other extreme, if neither party stops the refinement process, it may go on forever. Many such strategies have been studied by researchers in other contexts (like automated trust negotiation [20], *etc.*), and can be supported here. In the code presented next, we take the approach of refining a message up to a specified number of times (5 here). This is because we haven't found the need yet for a more complex strategy.

```
<process name="RESPProcess">
  <partnerLinks>
     <partnerLink name="ESPtransmission"
         partnerLinkType="…"
         myRole="ReceiveMsgSrv" />
      .
      .
      .
  </partnerLinks>

 <faultHandlers>
      .
      .
      .
 </faultHandlers>
  <sequence>
 // logic for generating SLA
  <switch> // Evaluate SLA
   <case condition="number &lt; 11">
     <receive partnerLink="RESPtransmission"
     portType="RESPReceiveMsgPT"
     operation ="GetMessage"
     variable ="M">
     </receive>
  <flow> // Invoke concurrent processes
     <links>
     </links>
     <sequence> // Virus scanning
       <invoke partnerLink="scanner"
         portType="RESPExaminationPT"
         operation=" VirusScan"
         inputVariable="M"
         outputVariable="result">
       </invoke>

       <switch>
       <case condition="result=true">
       <invoke partnerLink="scanner"
         portType="RESPVirusRemovalPT"
         operation="VirusRemoval"
         inputVariable="M"
         outputVariable="M">
         <source linkName="fixed" />
       </invoke>
       </case>
       <otherwise>
         <empty>
             <source linkName="empty" />
         </empty>
       </otherwise>
     </switch>
   </sequence>
   <sequence> // Distributed checksum
     <invoke partnerLink="TPDCC"">
       portType="CheckSumPT"
       operation=" DCC"
       inputVariable="M"
       outputVariable="checksumOK">
      <source linkName="dcc-deliver" />
     </invoke>
   </sequence>

   <sequence> // Verify bond
     <invoke partnerLink="TPEscrow"">
       portType="bondVerificationPT"
       operation="VerifyBond"
       inputVariable="M"
       outputVariable="verified">
       <source linkName="bond-verify" />
     </invoke>
   </sequence>
   <sequence> // Bayesian filtering
     <invoke partnerLink="RESPFilter"">
       portType="RESPFilterPT"
       operation="BayesianFiltering"
       inputVariable="M"
       outputVariable="filterOK">
       <source linkName="filtering" />
     </invoke>
   </sequence>
  </flow>
 <!— enforcing acceptance policy -->
  <sequence>
   <switch>
   <case condition="(fixed OR empty)
      AND (checksumOK OR filterOK))">
     <invoke partnerLink="RESP-Recipient">
       portType="RESPStoragePT"
       operation="StoreMessage"
       inputVariable="M"
       <switch>
       <case condition="Ack = YES">
        outputVariable="delivered">
       </case>
       <case condition="No space">
       <throw "FailFault">
       </case>
       <otherwise> <empty />
       </otherwise>
     </invoke>
   </case>

   <case condition="(fixed OR empty) AND
           (verified AND bond &gt; b)">
     <invoke partnerLink="RESP-Recipient">
       portType="RESPStoragePT"
       operation="StoreMessage"
       inputVariable="M"
       <switch>
       <case condition="Ack = YES">
        outputVariable="delivered">
       </case>
       <case condition="No space">
       <throw "FailFault">
       </case>
       <otherwise> <empty />
       </otherwise>
     </invoke>
   </case>
   <case condition="NOT fixed OR NOT
           (checksumOK AND filterOK)>
```

```
122      <throw “RejectionFault” faultVariable=
123             “RejectionNotice”>
124     </throw>
125    </case>
126    <otherwise>
127      <sequence>
128      <switch>
129      <case condition=“history &gt; 5”>
130      // 5: maximum invocations of improvement service
131      <invoke partnerLink=“self”>
132        portType=“RESPImprovementPT”>
133        operation=“RefineMsg”
134        inputVariable=“M”
135        outputVariable=“RefinementMsg”>
136        // outputVariable stores M’s refinement history
137      </invoke>
138      <reply partnerLink=“RESP-SESP-CB”>
139        portType=“ SESPCallbackPT”>
140        operation=“MessageCallback”
141        Variable=“RefinementMsg”>
142      </reply>
143      </case>
144      <otherwise> <empty />
145      </otherwise>
146    </otherwise>
147    </switch>
148  </sequence>
149 </switch>
150 </sequence>
151 </process>
```

Listing 9: Example RESP Process

The RESP process is made up of five main parts, as shown in listing 9, *viz*, headers, types and supported faults (lines 1—15), message reception from SESP (lines 19—24); invocation of helper services to gauge *message quality* (lines 25—81); acceptance policy evaluation based on message quality (lines 83—125) and finally, computing feedback for rejected messages (lines 128—143). The RESP waits for messages to arrive, and if the service level agreement is satisfied, messages are accepted (as shown in listing 9). Next, the RESP makes concurrent calls to several `helper' services, like Bayesian filtering service, bond verification service, distributed checksums, virus scans, *etc.*, to gauge the quality of incoming message. Once these processes terminate with an output, the RESP process starts evaluating the message based on its acceptance policy. During this stage a message may be accepted or rejected. Rejected messages may be returned to the SESP with feedback on some hints usable for resubmission, if the sender decides to do so (using the message improvement service).

Example 1: *Assume a mail message (M) that contains the following appropriately initialized parts: From, To, Date, ID, Subject and Body. We make the following assumptions:*

- *M does not contain any attached virus/worm*
- *M is the only message in queue*
- *RESP’s SLA accepts 10 messages per connection, and provides feedback for rejected messages.*
- *Acceptance policy requires that no virus be attached to a message, and either the message has a bond (“Surety”) or satisfies the Bayesian filter.*
- *Message content may contain prohibited words.*

According to the generic BPEL processes described, with the change that above policy instead of the one shown in table 4 is evaluated, M will not be accepted for delivery at the RESP (lines 83—125, listing 9). This is because it fails to satisfy both conditions – it doesn’t include a valid bond and it doesn’t satisfy the Bayesian filter on account of the prohibited words in its body. As in listing 9 (lines 126—142), the RESP process initiates a call to the message improvement service (to allow the sender to revise the message). The content of the refinement message would include the following parts: Date, ID, Sign, Surety and Body – the missing information that caused rejection. Essentially, this response provides the sender acceptable values for the parts Date, ID, Surety and Body. That is, the refinement message identifies the deficiencies in M: no valid bond (or surety) and presence of prohibited words. Once made aware, the sender may choose to alter the rejected message, so that it reaches its destination [12].

6. Coverage of use cases and misuse cases

We show next that the set of Web Service definitions, identified above, satisfy all stated use cases and avoid all mis-uses. We give our arguments in the form of (abbreviated) BPEL specifications as a proof of this claim.

6.1 Coverage of standard use cases

Line 22 in listing 8 (and line 22 in listing 9) sender invokes message delivery operation – “SendMessage” – for the SESP process (resp. SESP invokes “GetMessage” operation on RESP process). Clearly, the service invoked only accepts messages of type “MailMessage”. That is, input messages of type text or MIME messages (identified in the type declarations in lines 1—15) are *queued for delivery*. However, the SESP service interface (resp. RESP interface) *does not guarantee delivery* of the queued message, but only an assurance of best-effort delivery. As a result, if delivery fails at this stage, an error is generated – lines 106 to 112 in listing 8 (resp. lines 122 – 124 in listing 9). If all prerequisites for delivery are satisfied, then both SESP and RESP processes are guaranteed to attempt delivery. (Note, that the listings include only one delivery attempt, but multiple delivery attempts can be supported). Hence, the SESP and the RESP processes satisfy both the requirements of standard use cases – best effort transmission and error report on delivery failure. Consequently, the services defined here are sufficient for supporting standard use cases; additional proof is provided next.

Use Case: Authenticated message transmission
This use case is supported through invocations of the SenderPasswdCallbackPT and SESPAuthPT services. Due to space limitations, we follow the abbreviated BPEL syntax borrowed from [6].

SESP process modification
```
Begin Sequence
   Receive Message M
   Invoke SenderPasswdCallbackPT
   Switch
      Case: Password is correct
         ... // proceed to other delivery tasks
      Otherwise
         Throw <Failure Fault, message: incorrect password>
   End Switch
End Sequence
```

RESP Process modification
```
Begin Sequence
   Receive RESPHeloPT
   Receive Message M
   Invoke SESPAuthPT
   Switch
       Case: Credential verified
               ... // proceed to other delivery tasks
      Otherwise
         Throw <Failure Fault, message: invalid credential>
   End Switch
End Sequence
```

Code example above illustrates a simple (and scalable) way to support authenticated messages. Here, messages are authenticated in two tiers, *i.e.*, message senders are authenticated by their SESPs; while SESP is authentic-cated (using AuthPT service) by the RESP. It should be noted that this strategy provides only partial guarantees to sender authentication (since the sender is never directly authenticated by the RESP). More elaborate schemes, like, strong authentication based on PKI or secret key schemes like Kerberos are also possible, though we don't specify them here.

Use Case: Secure message transmission
This use case is supported through successive invocations of the RESP-TLSPT

RESP Process modification
```
Begin Sequence
   Receive RESPHeloPT
   Invoke RESP-TLSPT
   Switch
   Case: while SLA
      Receive Message M
          ... // proceed to other delivery tasks
   Otherwise
         Throw <Failure Fault, message: not allowed>
   End Switch
End Sequence
```

At each successive hop of a message, the sending agent can invoke transmission over TLS (or SSL) for privacy and integrity of data over the wire. This use case completes the set of standard use cases for email delivery

6.4 Preventing misuse cases

Here we show that the set of Web Services we define are adequate for preventing stated misuse cases. Again, we show coverage of all misuse cases with abbreviated BPEL specifications. We use listings 8 and 9 to give informal proof sketches of our claim. In addition, misuse cases like integrity, privacy, non-repudiation of message initiation are dependent upon more basic misuses like lack of sender authentication and absence of secure transmission. So, here we show how we prevent these basic misuses rather than the ones dependent on them.

Misuse Case 1: Denial of email service (email bombs)
This misuse is prevented using service level agreement for incoming mail connections. For instance, a service level agreement (SLA) can restrict number of concurrent connections from a particular domain and number of messages transmitted per connection (for instance, in listing 9 – lines 17 through 19 – restrict an SESP to only 10 messages per connection).

Misuse Case 2: Transmission in cleartext with no sender authentication
These misuses are prevented using acceptance policies for incoming messages. For instance, an acceptance policy requiring messages be authenticated and transmitted over a secure channel is easily encoded in BPEL as:

RESP Process modification
```
...
   Switch
      Case: "Password=correct AND channel= encrypted"
        Rnotice=  Invoke RESPStoragePT(Msg)
      Otherwise
         RMsg = Invoke RESPImprovmentPT(Msg)
         Reply SESPCallBack(RMsg)
   End switch
---
```

Consider lines 82 onwards in listing 9, where messages attributes are evaluated by the acceptance policy for the delivery session. The above policy that checks for password based authentication and encrypted channel can be applied *in conjunction* with other message acceptance requirements. That is, prevention of this misuse is possible by enforcing the correct acceptance policy.

Misuse Case 3: Controlling unwanted messages
Similar to the prevention of misuse case 2, this misuse is prevented using acceptance policies. The difference with the previous case is in the invocation of different Web Services like (FilterPT, DCC, *etc.*) during acceptance policy evaluation. For instance, a policy that requires the Bayesian filter and checksum service to approve a message is coded in BPEL as follows:

RESP Process modification

```
…
  Switch
     Case: Filter = false && checksum = false
       Rnotice=  Invoke RESPStoragePT(Msg)
     Otherwise
       RMsg = Invoke RESPImprovmentPT(Msg)
       Reply SESPCallBack(RMsg)
  End switch
---
```

As before, these conditions can be enforced *in conjunction* with other conditions (or otherwise) in listing 9 (lines 82 onwards).

7. Ensuring processes integrity

In this section we analyze SESP and RESP processes and informally argue that they exhibit several desirable properties. SESP and RESP processes include synchronized and parallel invocations of Web Services. For correctness of these calls, we show that the processes possess.

Deadlock freedom [3]: This property states that parallel invocations of Web Services are independent of each other, i.e., they do not block while waiting for the other to terminate or release a lock on synchronized resources.

Interference freedom [3]: This property states that execution of atomic steps of one component never falsify the properties enabled because of another component.

Distributed Termination [3]: This property states that a process terminates or stops executing after a finite amount of time.

Because of space limitations, we informally argue these properties, and work on formal proofs is in progress. In the following analysis, we categorize pairs (or sets) of programs according to following terms:

Parallel but disjoint [3]: A pair of programs is considered parallel but disjoint if one program cannot change variables accessed by other program.

Parallel with shared variables [3]: A pair of programs is parallel with shared variables if any one program can change variables accessed by the other.

Parallel with shared variables and synchronization [3]: Parallel programs with shared variables are also synchronized if they are able to suspend their execution while waiting on another program component to finish executing.

Before we begin arguing about the properties of our implementation of SESP and RESP processes, we give the abbreviated BPEL specification of the sender process.

Sender Process

```
Declarations: process, variables, faults
  Flow
     Invoke SESPReceiveMsgPT(M)
     Receive SenderCallbackPT
        Sequence
          // improve message
          invoke SESPReceiveMsgPT(M)
        End Sequence
  End Flow
```

Next, we argue about the correctness of SESP and RESP processes. Note that these processes fall in the third category stated above (parallel, synchronized processes, with shared variables). Also, we assume that individual Web Service components that are disjoint and recursion free and *always* satisfy their contracts. That is, assuming that their preconditions are met, they always terminate satisfying all their post conditions.

Proposition 1: Sender process exhibits deadlock freedom and interference freedom.

Proof: Sender process is defined (above) to be not synchronized with any other process. Therefore, there are no potential deadlock situations. Also, concurrent invocations (of call back interfaces) are disjoint and therefore they are also interference free.

Proposition 2: Sender process terminates.

Proof: Sender process is defined (above) to be disjoint from any other parallel process. Moreover, it is loop-free: that is, there is no recursive or mutually recursive call in its definition. Callback service does not invoke any additional Web Services, nor is the callback service synchronized with any additional process. Therefore it will terminate.

Proposition 3: SESP message transmission process is interference free and terminates

Proof: Message transmission by the sender to the SESP is disjoint from all other processes. Next, multiple concurrent processes are invoked by SESP process, like, UDDI service, virus scan, *etc.,* (line 53—88 listing 8). Each of these sub processes is defined to be disjoint, without mutual or self recursion. Therefore, they are interference free and terminate. The synchronization point is the termination of all sub processes (line 89, listing 8). Consequently, SESP execution is guaranteed to reach this synchronization point.

At this stage, the SESP process begins message transmission to RESP by invoking its HeloPT service. RESP can either refuse connection or allow transmission. In the first case, SESP process terminates (lines 98 –115, listing 8) and is, consequently, interference free. In the second case, the process makes synchronized calls to RESP for message delivery. In the first case, the message is accepted and therefore it terminates (lines 98 –115, listing 8). If the message is rejected, the RESP may invoke SESP call back interface for message improvement. This is done sequentially, after the transmission of original message is rejected. Thus mutual recursion is introduced, but RESP terminates recursion after a maximum number of invocations that are statically bound at compile time (here, by 5 – lines 130 to 136 in listing 9). Therefore, recursion terminates.

Proposition 4: The SESP process is deadlock free.

Proof: The SESP process invokes related Web Services that evaluate message before delivery, and RESP Web Services for message transmission. SESP Web Services are defined to be disjoint, with no mutual or self recursion; so, there are no potential deadlock situations. Recursion can occur in SESP and RESP process when messages are rejected. However, invocations of processes are sequential as evident in the code (lines 98 –115, listing 8). Consequently, there are no potential deadlock situations.

Proposition 5: The RESP process is deadlock free.

Proof: The RESP process, like the SESP process, invokes various helper services to evaluate messages during acceptance policy evaluation. Because all concurrent invocations are mutually disjoint, no potential deadlocks occur. RESP is deadlock free with SESP due to proposition 4. Hence, RESP is deadlock free.

Proposition 6: RESP message transmission process is interference free and terminates

Proof: This proof is similar to that of proposition 3. On receiving a request for mail delivery, the RESP process chooses a service level agreement for the session (lines 16–17 listing 9). Next, while SLA conditions are true, message transmission takes place. For each message various email control mechanisms (like *filterPT*, *checksumPT*, *bondVerification*, *etc.*) are called (lines 25—81 listing 9). Each of these sub-processes is defined to be disjoint and recursion free, therefore, running in parallel, they are interference free. Our assumption of disjoint-ness implies that these processes terminate. After termination of sub processes, acceptance policy is enforced. As all sub-processes terminate, policy evaluation proceeds without blocking (lines 83—150, listing 9). These lines include sequential synchronized calls from SESP to RESP and possibly from RESP to SESP. This mutual recursion was shown to terminate in proposition 3. Therefore, RESP is interference free and terminates.

8. Privacy leakages due to feedback

Example 1 shows that providing feedback not only reveals to the sender the policy that is being evaluated at the RESP, but also leaks several other types of information. For instance, in example 1, the sender could determine the expressions rejected by the RESP's Bayesian filter. This information can be misused by the sender to send undesirable messages to the recipient by simply camouflaging the `flagged' expressions – using `HTML` tags, insertion of spaces and other similar techniques. Other types of leakages [13] that compromise recipient's private information are also possible with WSEmail.

Leakages are categorized into two classes [13], *viz*, those due to feedback provided in-band with the transmission channel, and those due to out of band feedback channels. In the case of example 1, the leakage of information occurs due to in band feedback channel. These can by simply prevented in the SLA by prohibiting feedback. Consequently, the message improvement service will not be invoked. However, leakage is still possible, as shown next. Consider a scenario where an acceptance policy requires that a message satisfy the Bayesian filter *and* include a valid bond. Because of this policy whenever the bond is seized by a recipient, causing out of band monetary flow, it reveals the *strength* of the filter to the sender as the sender gets the confirmation that the message satisfied the Bayesian filter. Clearly, strength of the filter is sensitive information that must be protected, as argued above.

Authors in [13] develop methods for preventing out of band privacy leakages. These are directly applicable to the BPEL processes described here. We translate their solution for logic programs to our imperative programs. In addition, we show how process synchronization can be used to enforce their solution, a study missing in their work. First, we illustrate the problem with an original (unsafe) policy and its BPEL specification.

Policy 1 [Original (Unsafe) Policy]: *Consider the following acceptance policy for accepting messages:*

Accept	*IF*	*Sender is not blacklisted and bond $\geq a$*
message	*OR*	*Sender blacklisted and bond $\geq b$ $(b>a)$*

As shown earlier[13], this is an unsafe policy since it introduces an out of band feedback channel. For instance, if a sender sends a message bonded with value $c \in (a,b)$ and the bond is seized, then money transfer indicates to the sender that he or she is not blacklisted by the particular recipient. An (abbreviated) BPEL specification of this policy enforcement is as follows:

Policy evaluation block in RESP process

```
Begin Sequence
  Switch
      Case: Sender $\notin$ blacklist AND bond > a
        Rnotice=  Invoke RESPDeliveryPT(Msg)
      Case: Sender blacklist AND bond > b
        Rnotice= Invoke RESPDeliveryPT(Msg)
```

```
    Otherwise
      RMsg = Invoke RESPImprovmentPT(Msg)
      Reply SESPCallBack(RMsg)
  End switch
End Sequence
```

8.1 Policy transformation

Out of band leakages described above are harder to prevent without discontinuing the use of Web Services that introduce the leakage channel. That is, protection against privacy leakages requires that recipients and RESPs disable the use of such Web Services. However, this condition is too strict; an alternate solution exists that achieves the same goal without requiring the recipients to write truncated acceptance policies. This is done be automatically generating two safe policies from the original: the *necessary* and the *sufficient* policy.

Intuitively, the necessary policy is a weaker policy (truncated form of original policy) that does not invoke *leaky* Web Services. On the other hand, the sufficient policy is a strictly stronger policy that does not invoke leaky Web Services. With the ability to automatically construct these policies, a policy author can still enforce the original policy with a trusted client; and use the necessary and sufficient policies in tandem with a suspicious or an unknown client. Their construction and use is detailed next.

For the transformation procedures below we assume that a policy can be represented as a logical formula in disjunctive normal form (DNF), *i.e.*, it can be represented as $d_1 \vee d_2 \vee \ldots \vee d_n$ where each d_i is a conjunction of Boolean conditions.

```
NecessaryTransform(Policy, private):
Input: A set of policy rules
Input: A set of sensitive information attributes
Output: A set of policy rules that protect sensitive information
  if (Policy rules contains $p \in private$)
    Repeat till Policy does not contain any $p \in private$
1.    choose a rule $\in$ Policy | rule=$\vee_i d_i$ and some $d_i$ contain p
2.    modify each such $d_i$ such that it does not contain p
  else
    return
```

Policy 2 [Necessary Policy]: *Consider the original policy, discussed in Policy 1.*

Accept message	*IF*	*Sender is not blacklisted and bond ≥ a*
	OR	*Sender blacklisted and bond ≥ b (b>a)*

Applying the NecessaryTransform procedure to the original unsafe policy yields the following necessary policy:

Accept message	*IF*	*Bond ≥ a*

In this particular example, the contents of a blacklist are considered sensitive. Consider the evaluation of this policy at RESP:

Policy evaluation block in RESP process

```
Begin Sequence
  Switch
    Case: bond > a
      Rnotice= Invoke RESPDeliveryPT(Msg)
    Otherwise
      RMsg = Invoke RESPImprovmentPT(Msg)
      Reply SESPCallBack(RMsg)
  End switch
End Sequence
```

As is evident from the code above, this policy accepts messages with a minimum bond value, and assuming recipient will seize bonds for all unwanted messages, the only information that this policy leaks is that the recipient requires a bond value of a for messages to be accepted. No information about content of recipient's blacklist can be deduced.

SufficientTransform procedure is presented next.

```
SufficientTransform(Policy, private):
Input: A set of policy rules
Input: A set of sensitive information attributes
Output: A set of policy rules that protect sensitive information
  if (Policy rules contains $p \in private$)
    Repeat till Policy does not contain any $p \in private$
1.    choose a pair of rules $\in$ Policy | rule1=$\vee_i d_i$ and some $d_i$ contain
      p and rule2=$\vee_j D_j$ and some $D_i$ contain NOT(p)
2.    remove rule1 and rule2 and construct a new rule such that
      rule=$(\vee_i d_i)\vee(\vee_j D_j)$ except the disjuncts containing p
  else
    return
```

Policy 3 [Sufficient Policy]: *Consider the original policy, discussed in Policy 1. Applying the SufficientTransform procedure to the original unsafe policy yields the following necessary policy:*

Sufficient policy:

Accept message	*IF*	*Bond ≥ b*

Consider the evaluation of this policy at RESP:

Policy evaluation block in RESP process

```
Begin Sequence
  Switch
    Case: bond > b
      Rnotice= Invoke RESPDeliveryPT(Msg)
    Otherwise
      Throw RejectFault(Msg)
  End switch
End Sequence
```

As in the previous case, the sufficient policy enforcement can only reveal to the sender that the message requires a minimum bond value of b. No information about the contents of the blacklist is divulged.

9. Related work

Lux, May, *et al* in [17] introduce WSEmail, *i.e.*, transmission of email messages using Web Services. The advantage of using Web Services is that it lends additional flexibility to the message transmission process, while avoiding standard pitfalls, like, lack of sender authentication, susceptibility to spam, *etc.* However, the authors restrict to previewing their prototype instead of considering the standard SMTP use cases for message delivery, or the orchestration of related Web Services. Here we fill these gaps.

Next closely related work is by Afandi [1], where the author discusses *adaptive* policies for messaging systems (like WSEmail). The central idea is to separate policies from the mechanism to allow flexibility in the behavior of network components involved in message transmission; however, this work emphasizes on design and architecture of such a system. Here, we complement AMPol by providing a simple implementation using BPEL.

Kaushik, Winsborough *et al* in [12, 13] solve similar problems in conventional systems, and provide several alternative solutions. We consider the applicability of their solutions, appropriately tailored, to the new domain. In addition, we show how process synchronization is used to enforce their solution, the piece missing in all earlier works. Finally, we give informal proofs of correctness of our implementation that uses parallel concurrent process for achieving message transmission.

Chafle, Chandra *et al* [6] present an analysis for decentralized orchestration of Web Services using BPEL. Though the problem we consider here is not directly related, but our analysis takes a leaf out of their synchronization analysis of BPEL orchestration.

10. Conclusion

In this paper we have analyzed an emerging Web Services based application for internet messaging known as WSEmail and compared it to the conventional messaging systems. Since the existing specifications for WSEmail don't consider all the standard use cases of existing message delivery infrastructure or the set of misuse cases that must be prevented, we augment their architecture with our additions. We provide a formal specification of each Web Service considered and show that standard use cases are supported with the family of Services we have identified; and all misuse cases can be prevented with the same (extensible) set. We next show how to orchestrate this family of services securely to achieve the goal of secure transmission of email messages, with no privacy leakages, a piece missing in most other works, and reason as to why our specification is correct.

References

[1] R. N. Afandi, *AMPOL: Adaptive Messaging Policy Based System*, *Master's Thesis in Computer Science*, University of Illinois at Urbana-Champaigne, 2005.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, *Business Process Execution Language for Web Services*, 2003.

[3] K. R. Apt and E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1997.

[4] T. Bass, A. Freyre and D. Gruber, *E-Mail Bombs and Countermeasures:Cyber Attacks on Availability and Brand Integrity*, IEEE Network, 12 (1998), pp. 10--17.

[5] N. Borenstein and N. Freed, *RFC 1521 - MIME (Multipurpose Internet Mail Extensions)*, 1993.

[6] G. Chafle, S. Chandra, V. Mann and M. G. Nanda, *Decentralized Orchestration of Composite Web Services*, *Thirteenth international world wide web conference (WWW 2004)*, 2004.

[7] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, 2001.

[8] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions*, *RFC 2045*, 1996.

[9] P. Hoffman, *SMTP Service Extension for Secure SMTP over Transport Layer Security*, *RFC 3207*, 2002.

[10] http://www.cloudmark.com/, *Cloudmark*.

[11] http://www.rhyolite.com/anti-spam/dcc/, *Distributed Checksum Clearinghouse*.

[12] S. Kaushik, W. Winsborough, D. Wijesekera and P. Ammann, *Email Feedback: A Policy-Based Approach to Overcoming False Positives*, *3rd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FMSE 2005)*, Fairfax, VA, 2005, pp. 73--82.

[13] S. Kaushik, W. Winsborough, D. Wijesekera and P. Ammann, *Policy Transformations for Preventing Leakage of Sensitive Information in Email Systems*, in E. Damiani and P. Liu, eds., *20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Springer Berlin / Heidelberg, Sophia Antipolis, France, 2006, pp. To appear.

[14] J. Klensin, *Simple Mail Transfer Protocol*, *RFC 2821*, 2001.

[15] J. F. Kurose and K. W. Ross, *Computer Networking : A Top-Down Approach Featuring the Internet*, Addison Wesley, 2004.

[16] T. Loder, M. V. Alstyne and R. Walsh, *An Economic Answer to Unsolicited Communication 5th ACM conference on Electronic Commerce*, 2004, pp. 40-50.

[17] K. D. Lux, M. J. May, N. L. Bhattad and C. A. Gunter:, *WSEmail: Secure Internet Messaging Based on Web Services*, *2005 IEEE International Conference on Web Services (ICWS 2005)*, Orlando, FL, 2005, pp. 75-82.

[18] J. Myers, *SMTP Service Extension for Authentication*, *RFC 2554*, 1999.

[19] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.

[20] T. Yu, X. Ma and M. Winslett:, *PRUNES: an efficient and complete strategy for automated trust negotiation over the Internet.* , *7th ACM Conference on Computer and Communications Security (CCS '00)*, Athens, Greece, 2000, pp. 210-219.