A Framework and Language Support for Dynamic Security Policy in Service-Oriented Architecture

CHI WU-LEE AND GWAN-HWAN HWANG*
Department of Computer Science and Information Engineering
National Taiwan Normal University
Taipei, 106 Taiwan

In today’s global network-based environment, where mission-critical applications typically run on highly distributed systems, customers expect reliable, available, and secure services. Supporting security becomes an important issue in service-oriented architecture (SOA). This paper describes how to simultaneously support both dynamic security policies and separation of concerns when developing an SOA application. We propose the DPSL (dynamic policy specification language) for managing and controlling the security according to the dynamic behavior of the workflow in SOA. The operation model is compatible with existing SOA standards, such as the WSDL, WS-Policy, WS-Security-Policy, WS-ReliableMessaging, and the BPEL. As a result, existing standard Web-services engines and BPEL engines can be employed directly to support dynamic policies in SOA. The implementation and experimental results demonstrate the feasibility of the proposed architecture.

Keywords: SOA, web services, BPEL, workflow, security policy

1. INTRODUCTION

Service-oriented architecture (SOA) is a paradigm for organizing and utilizing distributed services that may be under the control of different ownership domains [1]. SOA provides a uniform means of offering, discovering, and interacting with and using services to produce desired effects consistent with measurable preconditions and expectations. Being service-oriented requires services to be only loosely coupled to operating systems and other technologies that underlie applications. SOA separates functions into distinct units, or services [2], which developers make accessible over a network so that users can combine and reuse them when producing applications. These services communicate with each other by exchanging data or by coordinating an activity between multiple services.

The SOA can be implemented by a software system such as Web services [2]. Web services make functional building blocks accessible over standard Internet protocols independently of platforms and programming languages. These services can be new applications or wrapped around existing legacy systems to make them network-enabled. One of the goals of SOA is to allow users to combine various functionalities to form ad-hoc applications that are almost entirely developed from existing software services. Orchestration is normally required to produce a new application [3]. A high-level language such as the BPEL extends the service concept by providing a method for defining and supporting the orchestration of fine-grained services into more coarse-grained business services [4]. Orchestration involves ensuring that Web services interact with each other.
at the message level, including the business logic and execution order of the interactions from the perspective and under the control of a single endpoint. This involves an executable business process that may result in a long-lived, transactional, multi-step process model. In fact, the BPEL incorporates a workflow model to describe the orchestration of Web services.

As SOA services are opened up, it is necessary to consider how to combine these services securely [5]. In many business domains, Web services must exhibit quality attributes such as robustness, security, and maintainability. Several emerging technologies and standards address different aspects of the problem of security of services in SOA. For example, standards such as WS-Security [6], SAML [7], WS-Trust [8], WS-SecureConversation [9], and WS-SecurityPolicy [10] focus on the security and identity management of SOA implementations that use Web services. These standards have been created to address message-level security and provide the ability to satisfy security requirements within an SOA environment. However, these standards only support the static specification or description of the security requirements in an SOA environment.

As mentioned above, services require orchestration to produce a new application. The BPEL demonstrates that a workflow model is necessary when performing the orchestration. However, the execution of a workflow-based system is intrinsically dynamic. For example, the branching that occurs during workflow execution may depend on the values of certain variables in a process instance, and the presence of branching in a previous execution may influence the subsequent execution. This makes it necessary to investigate if statically specifying the security policy requirement fulfills the requirements of the workflow-based SOA environment.

In this paper we first present several motivating examples to demonstrate that if the security policy requirement cannot be specified according to dynamic behavior in a workflow-based SOA system, then the resulting overly tight coherence between the Security policy requirement and the flow structure will increase the difficulty, overhead, and cost of system development and maintenance. According to the separation-of-concerns principle, implemented Web services should only provide the core required functionality. Herein we (1) propose the concept of dynamic policies for SOA systems, (2) describe the architecture to support dynamic policies in the BPEL, and (3) define a language called the dynamic policy specification language (DPSL) to support the proposed architecture. Although the DPSL is designed to cooperate with previously proposed standards for SOA architecture, such as the WSDL, the BPEL, WS-Security, WS-Policy, WS-SecurityPolicy, and WS-ReliableMessaging, we believe the method of incorporating dynamic policies into SOA using the DPSL can be applied to any SOA architecture. Therefore, we can separate the specification of the security policy requirement and service orchestration so as to reduce the software development and maintenance cost of an SOA system. The implementation and experimental results presented here demonstrate the feasibility of the proposed architecture.

This paper is organized as follows. In Section 2 we present examples to show that statically specifying the security policy requirement is insufficient. Section 3 surveys previous work on Web-services framework and related work which tried to extend it. In Section 4 we present the syntax and semantics of DPSL. The architecture to support DPSL in BPEL is given in Section 5. In Section 6 we show the implementation details and experimental results. Conclusions are drawn in Section 7.
2. MOTIVATING EXAMPLES

In this section we present examples to demonstrate that statically specifying the security policy requirement is insufficient. Without loss of generality, we use the BPEL, WS-Security, and WS-SecurityPolicy to construct our demonstration example. Fig. 1 shows the first example, which is a BPEL process. Assume that it is part of the military supply system. The “Client”, “Inventory Query Service”, and “Registration Service” are external Web services. The basic security requirement in preliminary system development is that the communication between the process and external Web services should be protected by the mechanism supported by WS-SecurityPolicy. Assume that the security policies between the three external Web services are defined by SP1, SP2, and SP3 as shown in Fig. 1. SP1, SP2, and SP3 define the communication security requirements for Web-services transaction in WS-SecurityPolicy, and are bound to the three bindings of the three Web services separately. The process consists of several activities:

- **Activity A1** This receives the supply requirement from the “Client”, and contains the amount and number of items.
- **Activity A2** This checks the inventory of the item received in activity A1 by invoking an external Web service.
- **Activity A3** This sends the actual supply requirement to the supply depot and registers it.
- **Activity A4** This replies to the “Client”.

Assume that the security requirement has to be changed after its first deployment. Since inventory information is confidential in a military supply system, the authority decides that activity A2 should not communicate with “Inventory query services” with security policy SP1 directly. The new requirement is that the security policy should be selected according to the category of the item obtained in activity A1. The items are classified into two categories, each with different security policies in communicating
with the “Inventory query service”. Assume that the two security policies are SP1’ and SP1”. Different security policies have different cryptographic algorithms and keys. WS-Policy supports policy alternatives, which allows us to specify a set of policies from which the service client can choose one policy arbitrarily [11]. However, since WS-Policy does not support switching security policies according to the dynamic behavior of the BPEL process directly, the system developer needs to change the original BPEL process according to the new security requirements. Fig. 2 (a) shows the modified BPEL process, which adds an “if” activity B1 that decides how to pick the security policies according to the item category obtained in activity A1. If the appropriate security policy is SP1’, B1 controls the flow to execute activity C1; otherwise B1 controls the flow to proceed to C2. Note that C1 and C2 are activities that specify the corresponding binding$^1$ of the partnerLink of A2. Furthermore, if the authority needs to add more security policies for the communication to the “Inventory query service”, the system developer needs to add more activities to the BPEL process, as shown in Fig. 2 (b). We can see that the security requirement depends on the dynamic behavior of the BPEL process since it depends on the data received in activity A1. Also, the structure of the BPEL process itself needs to be modified when the security requirement changes. The resulting tight coherence between the structure of the BPEL process and the security requirement makes the system difficult to design and maintain. Below we give another example to demonstrate another kind of dynamic behavior that the security requirement may depend on.

The second example is similar to the first example. However, we now have two “Inventory query service” Web services for different items received in activity A1. Referring to, the process consists of several activities:

- **Activity A1** This receives the supply requirement from the “Client”, and contains the amount and number of items.

---

$^1$ Actually, it specifies the endpoint reference to the partner of the partnerLink. The endpoint reference belongs to a port and the port belongs to a binding that is associated with the portType of the partner [12-15].
• **Activity D1**  This checks the inventory of the item received in activity A1 and decides to branch to either A2 or A3 to invoke the applicable external Web service.
• **Activities A2 and A3**  These invoke different external Web services.
• **Activity A4**  This sends the actual supply requirement to the supply depot and registers it.
• **Activity A5**  This replies to the “Client”.

Similar to the previous example, assume that the security requirement has to change after its first deployment. The new requirement is that activity A4 can communicate with the register services using different security policies according to the previous flow control of the BPEL. This depends on the execution of activity D1. If activity A2 was executed, then activity A4 uses security policy SP2'; otherwise it should use policy SP2''. Fig. 4 shows the modified BPEL process. Note that C1 and C2 in D2 are used to select an appropriate binding of the partnerLink for A4.

In the two motivating examples, the appropriate security policy cannot be determined until after the BPEL starts executing. In the first example, applying the security policy depends on the parameters received in activity A1, whereas in the second example the security policy should be enforced according to the flow control of the BPEL process. The situation is more complicated since the security policy depends on the computation result of the BPEL process, given that activities can perform sophisticated computations. An intuitive solution is to encode the security requirement in the implementation of the BPEL process, but then the BPEL process would have to be modified when the security requirement changes, and vice versa. This tight coherence between the BPEL process and its security policy increases the cost of implementing and maintaining the SOA application.

### 3. RELATED WORK

The main work related to the present study is the Web-services policy framework (WS-Policy) [11], which is an industrial specification standardized by the World Wide...
Web Consortium (W3C). WS-Policy defines a policy as a collection of policy alternatives, each of which is a collection of policy assertions. New policies can be defined to satisfy the needs of specific domains if these are not already included. Tosic et al. extended WS-Policy by introducing WS-Policy4MASC, which uses four new types of policy assertions: “goal”, “action”, “utility”, and “meta” [16]. The primary intent of this approach is to specify monitoring and control policies. The policy assertions define actions to be taken – such as the removal, addition, replacement, skipping, or rerunning of a subprocess, or process termination – as soon as certain conditions are met. However, it does not support controlling policies according to the dynamic behavior of process execution in SOA. Liang et al. proposed a set of customized policy assertions in WS-Policy, called WS-CustomizationPolicy, which do not provide for the management of dynamic properties [17].

WS-Policy can only specify static properties, and hence it is implicit that these properties will never change when the service is defined. Several proposals had been published to support dynamic policies in Web services. Yee and Korba proposed a flexible security personalization approach that allows the Web-service provider and customer to derive the desired security policy via negotiation [18]. They also discussed how to extend the proposed approach to WS-Policy. Mathes et al. proposed WS-TemporalPolicy, which allows the service developer to attach a validity period to the properties described in WS-Policy [19]. Hollunder proposed a new WS-Policy operator, the if-operator, to enable conditional assertions [20]. This operator can choose assertions dynamically according to the value of a service parameter, the response time, or the cost of services. Baresi et al. proposed the Web-service constraint language (WS-CoL), which specifies user requirements for the execution of certain Web services. WS-attachment and WS-Policy are extended so that applying a policy can be constrained by the WS-CoL expression [21]. The original syntax and semantics of WS-Policy have also been extended by others [19-21], but the disadvantage that the standard Web-services engine is unable to carry out the new policy or assertion remains. Our work avoids this limitation by not needing to extend WS-Policy; instead, we employ the standard Web-services engine and the BPEL engine to support dynamic policies according to DPSL documents in the BPEL run-time system.

4. THE DPSL

To separate the implementation of the BPEL and the setting of its security policy, Web services, the BPEL, WS-Security, and WS-SecurityPolicy are standalone standards. Web services and the BPEL are proposed first, followed by the corresponding standards for their security such as WS-Security and WS-SecurityPolicy. Since Web services and the BPEL can operate without considering security requirements, WS-Security and WS-SecurityPolicy cannot interfere with the operation of Web services and the BPEL. Thus, the application designer always first defines the Web services and the BPEL required for the application, and then tests the semantics of the defined application without adding the security mechanism. Finally, after the semantics is tested, the designer defines general security policy assertions and applies Web-Services security, including SOAP Message security, WS-Trust, and WS-SecureConversation. WS-SecurityPolicy can de-
fine only a static security policy, and hence the most flexible way to include multiple
security policies in WS-SecurityPolicy is to specify them as possible alternatives [11].
The client can then choose one of these policies to make a particular Web-service trans-
action. It is obvious that statically providing policy alternatives is not sufficient to im-
plement the motivating examples presented in Section 2, since in a real application the
security policy may depend on the dynamic behavior of the BPEL execution. In the first
motivating example, the security requirement depends on the category of the item ob-
tained in activity A1, which is represented as a variable in the BPEL process. In the sec-
ond example, the security policy changes according to the result of branching in activity
D1. Our goal is to separate the BPEL process implementation and security policy, which
means that the system designer should not need to change the implementation of BPEL
processes when the security policy is altered.

Here we consider an SOA application to be a software system consisting of multiple
BPEL processes and standalone Web services. Also, we define the dynamic behavior of
a workflow process like the BPEL to be the dynamic state under actual operating condi-
tions that is described by various parameters, including variables, the execution history,
and the control flow. To synchronize the enforcement of security policies with the dy-
namic behaviors of an SOA system, the system designer has to embed detectors of the
dynamic behavior into the workflow system of an SOA application. The detectors recog-
nize specified behaviors and then instruct the workflow system to perform appropriate
tasks to implement certain security policies. Thus, the semantics of the security require-
ment is closely related to the implementation of the SOA application, and so changing
the security requirement always requires the implementation of the SOA application to
be modified. This makes the maintenance both difficult and expensive, and so it would
be preferable to separate the implementation of the SOA application and the security
requirement. Furthermore, an SOA application usually contains multiple workflow sys-
tems. For example, the BPEL is designed to implement an SOA application with multi-
ple processes, and the security requirement in one BPEL process may depend on the dy-
namic behavior of another BPEL process. In this case the dynamic behavior of one
BPEL process needs to be detected by another BPEL process.

In this paper we propose the concept of dynamic policies, where the enforcement of
service policies can depend on the dynamic behavior of the workflow systems in the
SOA application. Although here we only discuss how to implement this concept in the
BPEL-based SOA architecture, it can be applied in any SOA system. A BPEL-based
SOA application initially consists of some BPELs and their corresponding WSDLs (note
that a standalone Web service is considered to be a BPEL process that only receives re-
quests and replays the execution result). Referring to Fig. 5, BPEL and WSDL, 0≤i<N,
consist of an SOA application without considering security requirements. During soft-
ware development, we usually first design an application without a security policy re-
quirement. After its functionality is tested, we then attempt to add the security policy
requirement to the application. We need to consider how to define the security policy
requirement in the DPSL.

Before we provide the details of the DPSL syntax, we first present the general syn-
tactic form of a DPSL document (Fig. 6). A DPSL document, which is defined as being
an XML document, consists of the following three sections:
1. **The header section.** Since a DPSL document is also an XML document, a DPSL document should begin with an XML declaration that specifies the version of XML being used (e.g., `<xml version="1.0"/>`). This section also contains required namespace declarations.

2. **The WS-Policy description section.** This section defines certain policies according to WS-Policy defined by the W3C. These policies are set to be activated dynamically.

3. **The dynamic policy specifier section.** This section contains many dynamic policy specifiers, each of which has a designated portType name that is the portType name in the original SOA application. According to the WS-Policy attachment [14], we can associate a policy to a binding, a portType, an operation, or even an input/output message of an operation. Note that the original SOA application does not specify a WS-Policy. Without loss of generality, we can refer to the portType to specify the dynamic policies since it is the most important WSDL element describing a Web ser-
vice, the operations that can be performed, and the messages that are involved. Each dynamic policy specification can be defined so as to apply different policies under different dynamic behaviors. A dynamic behavior descriptor corresponds to a WS-Policy link that points to a WS-Policy – it describes how to select a WS-Policy in the designated portType. A dynamic behavior descriptor contains a Boolean predicate. When it is True, the corresponding WS-Policy will be applied in the designated portType. Since we can only choose one out of multiple WS-Policies in the WS-Policy description section for a portType, the preceding WS-Policy has a higher priority. We can define the scope within which the specified WS-Policies will be applied in the <scope> element. If this element is absent, then the specified WS-Policies will be applied to all the operations contained in the designated portType; otherwise these WS-Policies are only applied to the listed operations or messages.

![Fig. 7. The syntax of the dynamic policy specifier.](image)

Fig. 7 shows the syntax of a dynamic policy specifier. We specify syntax definitions in the Backus-Naur Form [22] in this paper. The XML element `<Designated_portType>`
is the root node of the specifier and the attribute name="portType Link" is used to point to the portType name in the original SOA application. The dynamic selection of security policy will be applied in this portType. Each Dynamic behavior descriptor element specifies a security policy selection in which the policy specified in policyURI link will be applied when the evaluation of Inspector predicate is True in the execution point noted in element <Position>. The execution point is labeled according to the names of the BPEL process and activity. When the execution of the process reaches the execution point, the Boolean predicate will be evaluated by the run-time system before or after the execution of an activity according the value of the when attribute.

We now define the following expressions and functions that are used in the Inspector predicate to verify the dynamic behavior of the BPEL processes:

- **[BPEL process name]:BPEL variable Xpath.** This expression represents the current value of a BPEL variable. BPEL process name and BPEL variable Xpath are the names of the BPEL process and the Xpath that points to the variable, respectively.

- **CheckFlow.ActivityExecuted([BPEL process name]:BPEL activity name).** This Boolean function checks if an activity has been executed. The parameter [BPEL process name]:BPEL activity name is the name of a BPEL activity.

- **CheckFlow.ActivityExecutedTimes([BPEL process name]:BPEL activity name).** This function checks the number of times that an activity is executed, and returns an integer.

- **CheckFlow.BranchResult([BPEL process name]:BPEL activity name).** This function checks the result of branching in a structure activity in BPEL, and returns a set of activities that are executed after a structure activity named [BPEL process name]:BPEL activity name.

These functions and the expressions defined in XML Path Language (XPath) 2.0 can be used to construct an Inspector predicate. XPath 2.0 allows the use of “and” and “or” in logical expressions. The value-comparison operators are “eq”, “ne”, “lt”, “le”, “gt”, and “ge”. For example, the statement "($VariableName.parameters/V1 ≥ 100) && (CheckFlow.BranchResult(BPEL:D1) &eq; BPEL:I2)" checks if the value of a BPEL variable “Variable_V1” is greater than or equal to 100 and if the result of branching in structure activity “BPEL:D1” is “BPEL:I2”. Figs. 8 (a) and (b) show two DPSL documents for the motivating examples presented in Section 2. Fig. 8 (a) defines the dynamic security policy requirement of our first motivating example. The requirement is that there are two different security policies that should be selected according to the category of the item obtained in activity A1 in Fig. 1. Referring to Fig. 8 (a), lines 7 to 12 define two security policies SP1’ and SP1”. The dynamic policy specifier of the designated portType “Inve:Inventory-1_Original_PT” at lines 13 to 32 contains two dynamic behavior descriptors. The first descriptor defines that if the value of the BPEL variable “Moti:SBPEL-1_OPRequest.parameters/item_NO” is less than or equal to 100, the policy SP1’ should be applied in the designated portType. Similarly, another descriptor defines that if its value is greater than 100, the policy SP1” should be applied. Fig. 8 (b) shows the security requirement of our second motivating example shown in Fig. 3. The requirement is that the security policy changes according to the result of branching in activity D1. Similar to Fig. 8 (a), lines 7 to line12 in Fig. 8 (b) define two security policies SP2’ and SP2”. The dynamic policy specifier of the designated portType “Reg:Register-2_Original_PT” at lines
13 to 32 contains two dynamic behavior descriptors. The first descriptor defines that if the branch result of the BPEL activity “Moti:D1_Invoke” is the BPEL activity “Moti:A2_Invoke”, the policy SP2’ should be applied in the designated portType. The other descriptor defines that if the branch result is the BPEL activity “Moti:A3_Invoke”, the policy SP2″ should be applied.

```
<DynamicSecurityPolicy name="NCName"
  targetNamespace="anyURI"
  xmlns:Inve="http://www.example.org/Inventory-1_Original/"
  xmlns:Moti="http://Motivation-1_Original/"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns="anyURI">
  <wsp:Policy wsu:Id="SP1'">
    ...
  </wsp:Policy>
  <wsp:Policy wsu:Id="SP1">
    ...
  </wsp:Policy>
  <Designated_portType name="Inve:Inventory-1_Original_PT">
    <Dynamic_behavior_descriptors>
      <Dynamic_behavior name="Value<=10" policyURI="#SP1'">
        <Position when="before">
          Moti:A2_Invoke
        </Position>
        <Inspector>
          Moti:$BPEL-1_OPRequest.parameters/item_NO ≤ 100
        </Inspector>
      </Dynamic_behavior>
      <Dynamic_behavior name="Value>10" policyURI="#SP1" >
        <Position when="before">
          Moti:A2_Invoke
        </Position>
        <Inspector>
          Moti:$BPEL-1_OPRequest.parameters/item_NO > 100
        </Inspector>
      </Dynamic_behavior>
    </Dynamic_behavior_descriptors>
  </Designated_portType>
</DynamicSecurityPolicy>
```

(a)

```
<DynamicSecurityPolicy name="NCName"
  targetNamespace="anyURI"
  xmlns:Reg="http://www.example.org/Register-2_Original/"
  xmlns:Moti="http://Motivation-2_Original/"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns="URI">
  <wsp:Policy wsu:Id="SP2'">
    ...
  </wsp:Policy>
  <wsp:Policy wsu:Id="SP2">
    ...
  </wsp:Policy>
  <Designated_portType name="Reg:Register-2_Original_PT">
    <Dynamic_behavior_descriptors>
      <Dynamic_Behavior name="Branch=A2" policyURI="#SP2'">
        <Position when="before">
          Moti:A4_Invoke
        </Position>
      </Dynamic_Behavior>
      <Dynamic_behavior name="Branch=A2" policyURI="#SP2" >
        <Position when="before">
          Moti:A4_Invoke
        </Position>
      </Dynamic_behavior>
    </Dynamic_behavior_descriptors>
  </Designated_portType>
</DynamicSecurityPolicy>
```

(b)
5. ARCHITECTURE TO SUPPORT THE DPSL IN THE BPEL RUN-TIME SYSTEM

Unlike WS-Security and WS-SecurityPolicy, the DPSL is not an accepted standard, and the existing run-time system of the BPEL cannot support the programming model shown in Fig. 5 to specify SOA application and dynamic policies. We therefore need to modify or enhance the existing run-time system of the BPEL. The first and direct solution is to modify or design a new BPEL engine so that it can parse the DPSL and then communicate with other BPEL engines or Web-services servers according to the dynamic policies defined in DPSL documents. However, since implementing or modifying a BPEL engine is expensive, we design an architecture to support the DPSL in the BPEL run-time system. A major advantage is that it complies with the existing BPEL engine that follows W3C standards. Referring to Fig. 9, we assume that BPEL \(_i\) and WSDL \(_i\), \(0 \leq i < N\), consist of an SOA application without considering security requirements, and \(D\) is a DPSL document that specifies the dynamic policies for this SOA application. A DPSL translator reads the security requirements in \(D\) and then translates the original SOA application, BPEL \(_i\) and WSDL \(_i\), into a new SOA application, BPEL’ \(_i\) and WSDL’ \(_i\), \(0 \leq i < N\). Note that this new application also complies with the W3C standards, and hence we can use the existing BPEL engine (which is compatible with W3C standards) to execute the new application. Therefore, using the DPSL translator removes the need to implement a new BPEL engine.

![Fig. 9. The operation of the DPSL translator for the BPEL.](image-url)
Algorithm 1 in Fig. 10 shows how the DPSL translator generates BPEL processes that implement the dynamic policies specified by a DPSL document. In Step (1), we copy the original BPEL and WSDL documents. BPEL' and WSDL', 0 ≤ i ≤ N, are identical to the original application. In Steps (2) and (3), we parse the DPSL document D and extract all the dynamic behavior descriptors and store them in a set Z. Step (4) is to install BPEL activities in the original application to evaluate inspector predicates in Z. Note that the result of evaluation of each inspector predicate is stored in a newly created BPEL variable with an unique name. In Step (5) we make translation for all the dynamic policy specifiers separately. For each dynamic policy specifier, we first install some BPEL variable with an unique name. In the sending process, we have to install some “if condition” and “assign” activities which are responsible to assign an appropriate endpoint reference to its partnerLink so that it can use the appropriate policy.

Algorithm 1: DPSL transformation for BPEL processes

Input: The original SOA documents: BPEL, and WSDL, 0 ≤ i ≤ N, and a DPSL document D.
Output: The transformed documents: BPEL', and WSDL', 0 ≤ i ≤ N – 1.

1. Let BPEL=BPEL, and WSDL'=WSDL, 0 ≤ i ≤ N – 1.
2. Parse D. Assume that there are k dynamic policy specifiers dps0, dps1, ..., dpsk, in D.
   - Let dps0-(designated portType name) be the specified portType in the dynamic policy specifier dps0.
   - Let dps0-(set of dynamic behavior descriptors) be the set of dynamic behavior descriptors defined in dps0. If x ∈ dps0-(set of dynamic behavior descriptors), then x-(position), x-(inspector predicate), and x-(WS-Policy) are the defined position, inspector predicate, and corresponding WS-Policy in dynamic behavior descriptor x respectively.
3. Let Z be the set of all defined dynamic behavior descriptors in D; that is, Z = dps0-(set of dynamic behavior descriptors) ∪ dps1-(set of dynamic behavior descriptors) ∪ ... ∪ dsk-(set of dynamic behavior descriptors).
4. FOR each d in Z
   - According to dp-d(positions), insert a new activity “before” or “after” the position at BPEL.dps, to evaluate dp-d(inspector predicate). Note that the result of the evaluation is stored in a variable with a unique name that, for convenience, is denoted as $Result_{dp-d(inspector predicate)}$.
      END FOR
5. FOR j = 0 to k – 1
   - Let i be the “invoke” activity whose partnerLink is linked to the portType defined in dpsj. Assume that i is an activity of BPEL.
   - Let R = dpsj-(set of dynamic behavior descriptors). Assume that there are m dynamic behavior descriptors in R. Assume that R=R0, R1, ..., Rm-1.
   - Add m new bindings and m new ports to receiving process BPEL-r.
     - Binding, which is within the port, 0 ≤ i ≤ m-1, is a binding in the WSDL that associates the WS policy specified in ri by policyURI_link, that is, ri-(WS-Policy). Note that all these bindings are associated with the same portType according to dpsj.
     - Portj, 0 ≤ i ≤ m-1, is a port whose address is an unique endpoint reference and is associated to Binding.
   - Insert one “if” activity that contains m “if conditions” and m “assign” activities before i in BPEL/r, as shown in Fig. 11.
     - B is a structured activity (i.e., and “if” activity) that checks if $Result_{ri-(inspector predicate)}$, 0 ≤ i ≤ m-1, is True or False.
     - i, 0 ≤ i ≤ m-1, is an “if condition” that checks the result of the associated inspector predicate of ri.
**End For**

Fig. 10. The algorithm which performs the DPSL transformation.

The architecture of the translated BPEL programs of the two motivating examples is shown in Figs. 12 and 13 according to the DPSL documents shown in Figs. 8 (a) and (b). The real BPEL codes are available at http://www.csie.ntnu.edu.tw/~ghhwang/DPSL/DP-SL_Motivating_Examples.rar.

**6. IMPLEMENTATION AND EXPERIMENTS**

We constructed a simple implementation of the DPSL translator that omitted some of the functions mentioned above (e.g., the scope). The implementation was translated
into the two motivating examples presented in this paper. The software packages used to conduct our experiments included Apache AXIS2 1.5.1 [23], Apache AXIS2 Module: SOAP Monitor [24], JDK 1.5.0_21, Eclipse WTP [24] with soapUI plug-in [25], and ActiveBPEL Designer [27]. For the original BPEL document, we first employed Eclipse WTP to edit the WSDL documents, and then used the Apache AXIS2 Code Generator to generate the Java code template from the WSDL, with this template being used to design the operation codes. Finally, the service was deployed on the AXIS2 engine. The BPEL document was constructed using ActiveBPEL Designer and deployed to an ActiveBPEL engine to test its functionality.

We employed the DPSL translator to generate the BPEL and WSDL documents. Tables 1 and 2 list the numbers of lines of the original and translated BPEL and WSDL documents for the two motivating examples. The translated codes were deployed to the AXIS2 and ActiveBPEL engines to verify their correctness. The translated codes were significantly larger, which is due to them containing additional codes to activate the dynamic policies defined in the DPSL. Compare the original codes with translated codes, there is a thirty percent increase in the code sizes on average. In the two examples, there is only one dynamic policy specifier in the DPSL document. There will be more increase in the code sizes if there are more dynamic policy specifiers in DPSL documents.

In the second part of our experiments, we conduct experiments to evaluate the execution time of the original and translated BPEL processes. To evaluate the overhead of the translated BPEL processes, we arrange the BPEL processes and invoked services in the same machine. Since translated BPEL processes need to add some “if” and “assign” activities, it needs more time to finish the execution. Referring to Table 3. The original BPEL process of Motivation-1 needs about 148 ms to finish the execution which contains invocation of two Web services. The translated BPEL processes need about 156 ms to finish. The overhead is generated because the extra execution of added “if” and “assign” activities. However, if we translate the code manually, it also has to add some “if” and “assign” ac-

<table>
<thead>
<tr>
<th>Table 1. Code sizes of the Motivation-1 WSDL and BPEL documents.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before translation</td>
</tr>
<tr>
<td>Motivation-1 Original.bpel</td>
</tr>
<tr>
<td>Inventory-1 Original.wsdl</td>
</tr>
<tr>
<td>BPEL-1 Original.wsdl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Code sizes of the Motivation-2 WSDL and BPEL documents.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before translation</td>
</tr>
<tr>
<td>Motivation-2 Original.bpel</td>
</tr>
<tr>
<td>Register-2 Original.wsdl</td>
</tr>
<tr>
<td>BPEL-2 Original.wsdl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Execution time of the original and translated BPEL processes of Motivation-1 and Motivation-2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before translation</td>
</tr>
<tr>
<td>Motivation-1 BPEL process</td>
</tr>
<tr>
<td>Motivation-2 BPEL process</td>
</tr>
</tbody>
</table>
activities because the security policies should be changed according the dynamic behavior of the BPEL processes.

7. CONCLUSION

This paper has addressed the issue of supporting dynamic policies in an SOA environment so as to satisfy security requirements. We have described two motivating examples to demonstrate that the security in SOA depends on the dynamic behavior of the workflow system. We have also proposed a solution that employs the DPSL to specify the security policy requirements in an SOA system. This architecture can fulfill the requirement for the separation of concerns by separately defining the service implementation and the policy specification. The use of a DPSL translator supports dynamic policies in SOA using standard SOA engines. The implementation and experimental results demonstrate the feasibility of the proposed architecture.

In the future we will investigate issues related to how to implement dynamic access control, fault tolerance, failure recovery, exception handling, and process instance security according to the dynamic behavior in SOA.

REFERENCES


Chi Wu-Lee (吳李騏) is a Ph.D. candidate in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. His research interests include service-oriented architecture, access control, cloud computing and work-flow management System.

Gwan-Hwan Hwang (黃冠寰) is a Professor and Chairman in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. He received the B.S. and M.S. degrees while in the Department of Computer Science and Information Engineering at National Chiao Tung University, in 1991 and 1993, respectively, and the Ph.D. degree while in the Department of Computer Science at National Tsing Hua University, Hsinchu, Taiwan, in 1998. His research interests include Internet security, concurrent software testing, groupware, service-oriented architecture, and cloud computing.