Security Policy Composition for Composite Services

Fumiko Satoh[†1,2]
[†1]*IBM Tokyo Research Laboratory,*
1623-14 Shimo-tsuruma, Yamato-shi,
Kanagawa, 242-8502, Japan
sfumiko@jp.ibm.com

Takehiro Tokuda[†2]
[†2]*Department of Computer Science,*
Tokyo Institute of Technology,
Meguro, Tokyo, 152-8552, Japan
{sfumiko, tokuda}@tt.cs.titech.ac.jp

Abstract

An application based Service-Oriented Architecture (SOA) consists of an assembly of external services and the application is called as a composite service. A composite service could be implemented by other composite services hence the application could have a recursive structure, which is one of the features of SOA application. Securing an SOA application is an important non-functional requirement. However, specifying a security policy of a composite service is not so easy because the policy should keep the consistency with other policies of external services which are invoked in the process. We need the way to assure the consistency of policies, but the concrete way is not developed yet to specify a consistent policy for a composite service. Therefore, this paper proposes a security policy composition mechanism from existing policies of external services. Our contribution is creating a security policy of a composite service automatically based on predicate logic, with support for two approaches of policy composition: bottom-up and top-down. Also, we focus on three kinds of security policies, such as a Data Protection Policy, an Access Control Policy, and a Composite Process Policy, and propose the policy composition rules for each policy. Our mechanism makes it possible to validate the consistency of policies by inference without increasing a developer's workload, even if a composite service has a recursive structure.

1. Introduction

Service-Oriented Architecture (SOA) is a concept of building applications by assembling services that are components of business functions. Typically, an SOA application is implemented as a composite service that invokes external services in the process. The process and service invocations are defined in a process language such as BPEL [1], and a user can rebuild an application by changing only the process definitions without updating the service implementations. The benefit of an SOA application is flexibility to adapt to changing business processes.

SOA is convenient for satisfying functional requirements, but it is more difficult to satisfy the non-functional requirements such as security. The security requirements are specified as security policies for a composite service, but actually the way to specify policies for the composite services is not discussed clearly. For example, when an application developer assembles existing services that have their own security policies, how can the composite policies be defined and assured so that there are no inconsistencies with those existing policies? Also, there are several kinds of security policies, such as for data protection and for access control, and hence we need to compose these policies separately to create policies for a composite service. Currently, a developer needs to define the composite policies by hand by referring to the policies of the external services invoked in the composite process. However it is very hard to complete a policy composition without any inconsistency, because the process definitions and security policies may be complex and it is not clear how to compose policies to maintain consistency.

We propose a security policy composition mechanism to resolve these difficulties. Our approach is based on predicate logic, and generates a composite security policy by inference. We define the logic for the policy representation and composition process, and clarify the policy composition rules. In this paper, three kinds of security policies are discussed: Data Protection Policies, Access Control Policies, and Composite Process Policies. The composite process definitions written in BPEL and security policies written in WS-SecurityPolicy [2] are transformed into a logic representation, and they are executed as a prolog

978-0-7695-3261-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWE.2008.23

program to infer a security policy for the process.

Our mechanism can apply two approaches to generate a composite security policy: bottom-up or top-down. The bottom-up policy composition can compose policies from existing policies for external services, and the consistent composite policies are generated by inference. In contrast, a developer can specify composite policies regardless of existing policies. Our mechanism verifies the specified security policies are consistent with the existing policies, and the developer can confirm that the specified composite policy will work properly. Our main contributions are the automatic composition and verification of security policies from existing XML representations while reducing the developer's workload.

The rest of this paper is organized as follows. Section 2 explains the motivating example of our study and clarifies the problems with policies for composite services. Section 3 provides definitions of composite services and their security policies. We propose our security policy composition architecture and composition rules in Section 4. The policy composition is demonstrated in Section 5. Section 6 provides related work and we conclude our study in Section 7.

2. Motivating Example

2.1. Scenario: Travel Reservation Service

First, we explain the application scenario which motivated us to clarify the problem. The travel reservation service shown in Figure 1 is a composite service that consists of the process invoking the airline reservation service and the hotel reservation service, which are external services. These external services are invoked symmetrically in the process of the travel reservation. The external services could be also composite services which invokes other external services.

Here we suppose that the travel reservation service must be secure. A security policy will be necessary such that the exchanged messages should be signed and encrypted, only an employee of the travel agency can invoke the service, and so on. However, both the airline reservation service and the hotel reservation service might have their own security polices. In this situation, to define a security policy for the composite service in a bottom-up way, the composite security policy and the external security policies should be consistent. This means that the policies of external services will also be satisfied as long as the composite policy is satisfied. In contrast to the bottom-up approach, we could define a composite security policy in a top-down approach. If a

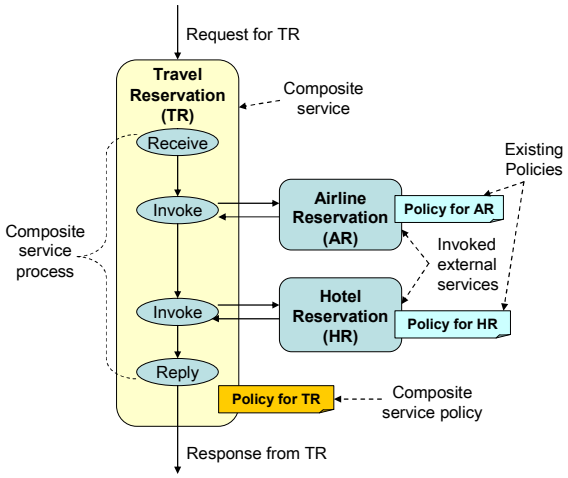

Figure 1. Composite Service Example: Travel Reservation Service

composite service developer assembles some existing services which have their own policies, then the developer could define a new policy for the overall composite service. In this top-down policy definition, it is not clear how we could validate if all of the security policies of the composite service and the invoked external services are consistent.

The goal of this study is to provide a way to compose security policies for a composite service, so that the policies have no inconsistencies from either the bottom-up or the top-down perspective. In the next section, we clarify some possible inconsistencies in security policies that should be addressed by this work.

2.2. Problems of Security Policies in Composite Processes

We define three categories of security policies to focus on this study: (1) Data Protection Policies (DPP), (2) Access Control Policies (ACP), and (3) Composite Process Policies (CPP).

A DPP is a security policy specifying how to protect exchanged data, i.e. data of "username" type should be signed and encrypted. An ACP defines which role has authority to invoke a service. The CPP is a special policy defined when services are assembled as a composite service. For example, a service developer could apply a constraint such as service A and service B should be executed by users who have different roles. The details of these policies are defined in Section 3.2, but here we explain the possible problems for these security policies.

The problems for a composite security policy are the inconsistencies between a composite service policy

and invoked external service policies. We clarify the inconsistencies to be resolved for each policy as follows:

- **DPP**
  Data may be unprotected or less protected in a composite service even if the same data would need high protection in external services.
- **ACP**
  If the external services specify authorized roles who can invoke the service, then those policies should be preserved in the composite service policy. For example, we cannot execute the composite service if the external service allows the user who has a roleA, but the composite service does not allow the user of roleA.
- **CPP**
  If a CPP requires that external service A and service B should be executed by the separate roles, but these external services allow the same roles in both ACP of service A and service B, so these ACP are not conformed to the CPP.

These inconsistencies of a composite service policy and external service policies would be common, and they seem to be resolved easy by a developer. However, if the external services that are invoked in the composite service process are implemented by other composite services, the problems are not easy. A developer should analyze the policies of external services recursively, and then resolve the inconsistencies manually by referring these policies. It is so hard to resolve them correctly because the composite service structure and policies might be complex and a developer could not be a security expert.

We propose a logic-based approach to resolve these problems and provide a way to create consistent security policies for a composite service. We define logic of the policies, processes, and rules for policy compositions. This logic can be executed as a Prolog program, and the inference will provide the result if any inconsistencies exist in a composite service policy. Also, the inferences can lead to a correct policy if there are any inconsistencies in the process.

3. Security Policies for Composite Services

3.1. Composite Service Definition

A composite service process which invokes external services can be represented using BPEL [1]. BPEL has great flexibility to express complicated invocations or conditions, so we focus on a typical process to simplify our presentation. Figure 2 shows a typical composite process expressed in BPEL representation and a corresponding diagram generated by the tool such as eclipse of BPEL project [4] or WebSphere Integration Developer [5]. The composite service process has both *receive* and *reply* actions. The incoming parameters are received at *receive*, and the outgoing parameters are returned at *reply*. Here we assume that the external services are invoked by symmetric invocation in the composite process. In this case, when a process invokes an external service, the process will not go on to the next action until it has received the response parameters from the external service. This kinds of invocation is represented using *invoke* actions.

Table 1. Variable Assignments of Travel Reservation Process

From		To	
Message	**Variables**	**Message**	**Variables**
Travel Request	mileageNo airlineInfo cardInfo	Airline Request	mileageNo airlineInfo cardInfo
Travel Request	customerID hotelInfo cardInfo	Hotel Request	customerID hotelInfo cardInfo
Airline Response	result	Travel Response	airlineResult
Hotel Response	result	Travel Response	hotelResult

An *assign* action specifies copying a variable from a variable specified by a *from* element into a variable specified by a *to* element. Figure 2 has an *assign* action to copy from the *result* value of *airlineResponse* which is a response message of the airline reservation service, to the *airlineReservationResult* value of *agencyResponse* which is a response message of the travel reservation service. Here only one example of a variable assignment is shown, and Table 1 shows all of the services' variables and variable assignments executed in the travel reservation service. The request variables of the travel reservation service are assigned to the request variables of the external services, and then the two external services, airline reservation service and hotel reservation service each return a true as a result value when the reservation succeeds. The result values are assigned to the response variables of the travel reservation service and returned to the client by the *reply* action.

This process is a typical composite process in BPEL. In this study, we focus on this process that mainly involves these actions: *receive*, *invoke*, *reply*, and *assign*. The relations between the composite service variables and the external service variables are defined by *assign* actions as shown in Figure 2, and these relations are leveraged to create the security policies. In the next section, the three types of security policies we discuss in this study are explained.

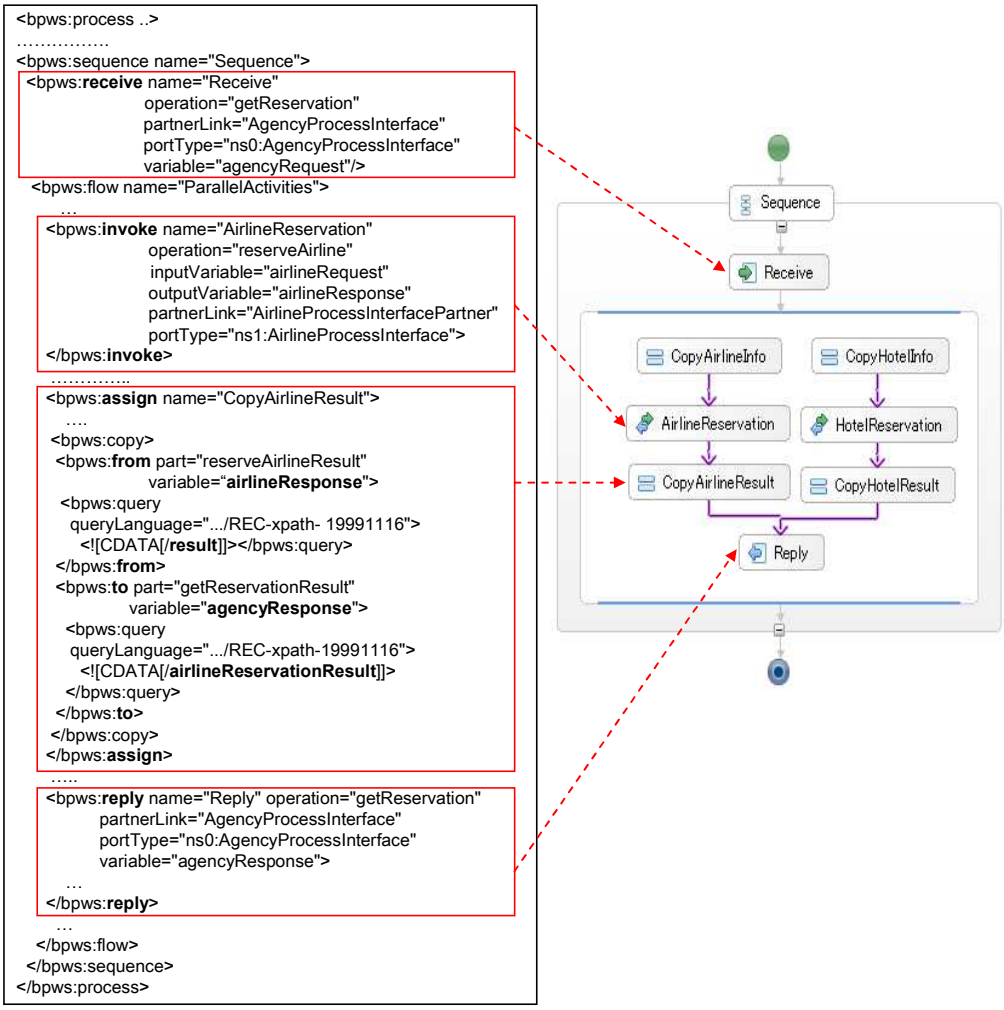

Figure 2. A Travel Reservation Composite Process in BPEL

3.2. Security Policy Types

Here we define the security requirements that are discussed in this study, and specify the security policies for these security requirements. An SOA application consists of assemblies of services, and there are two types of services: an atomic service and a composite service. An atomic service is a service which does not invoke external services in the process. A composite service invokes external services in the process, and also consists of service assemblies. Here we classify the security policies in these types of services.

A security policy for an atomic service is classified as a Data Protection Policy (DPP) and an Access Control Policy (ACP). The security policies for a composite service consist of a composite DPP and ACP of the atomic services, and a Composite Process Policy (CPP). A CPP has special requirements for a series of services in the composite process. Figure 3 shows the security policy classification of atomic services and composite services. The following sections explain details of these security policies.

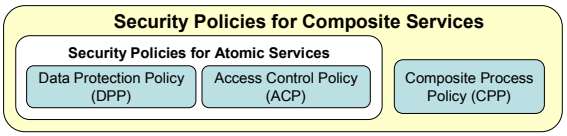

Figure 3. Security Policy Classification

3.2.1 Data Protection Policy. The Data Protection Policy (DPP) describes data protection, such as integrity or confidentiality, during message exchanges. Web Services Security (WS-Security) [3] can protect the messages exchanged between clients and providers

```
<!-- Endpoint Policy -->
<wsp:Policy ….>
  <sp:AsymmetricBinding>
    <sp:InitiatorToken>
      <sp:X509Token />
    </sp:InitiatorToken>
    <sp:RecipientToken>
      <sp:X509Token />
    </sp:RecipientToken>
    <sp:AlgorithmSuite>
      <sp:Basic256 />
    </sp:AlgorithmSuite>
    <sp:IncludeTimestamp />
    <sp:ProtectTokens />
  </sp:AsymmetricBinding>
  <sp:SignedSupportingTokens>
    <sp:UsernameToken />
  </sp:SignedSupportingTokens>
</wsp:Policy>

<!-- Message Policy -->
<wsp:All …>
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
  <sp:EncryptedParts>
    <sp:Body/>
  </sp:EncryptedParts>
</wsp:All>
```

Figure 4. Example of Data Protection Policy in WS-SecurityPolicy

by using XML signature or XML encryption. Web Services Security Policy (WS-SecurityPolicy) [2] is a specification to express security policies for WS-Security. Web Services is a typical technology to implement a service invocation, and therefore we assume that a DPP is a security policy written in WS-SecurityPolicy.

Figure 4 is an example of the DPP written in WS-SecurityPolicy. The security policy represents security requirements using a set of assertions which are XML elements to specific security properties. For example, *SignedParts* element is one of security policy assertions for specifying the signed portion in the SOAP message.

This policy requires a signature and encryption on a SOAP Body by using an X.509 certificate. The signed and encrypted portions in the SOAP message are specified by the *SignedParts* and *EncryptedParts* assertions, and these elements have a *Body* element in this example. In WS-SecurityPolicy, sets of algorithms are defined as algorithm suites assertions. This example specifies the *Basic256* algorithm suite in the *AlgorithmSuite* assertion. The algorithms corresponding to *Basic256* are defined in [2], where HmacSha1 algorithm is used for the signature method, and Sha1 algorithm is used for the digest method.

In the XML signature and encryption, the key for the signature and encryption is represented as a security token, which is an XML element with security-related information. An X509 token is a security token for a Base64-encoded X.509 certificate. This policy has the *X509Token* assertion in the *AsymmetricBinding* assertion, which means an X509 security token is used for the signature and encryption of the SOAP Body. A *ProtectTokens* assertion requires a signature on the security token that was used to sign the SOAP Body. A *SignedSupportingTokens* assertion specifies a requirement of additional token, in this example a signed username token is required.

This example specifies very simple requirements, but WS-SecurityPolicy is a quite complicated and flexible specification. For users without detailed knowledge of the related specifications, it is too difficult to understand all of the security requirements.

3.2.2. Access Control Policy. The Access Control Policy (ACP) restricts who can access a service. For example, a travel reservation service should be invoked only by travel agency employees. This requirement is for a service operation itself, not for data. Therefore, an ACP is sets of an operation name and list of roles that are allowed to access the service. The ACP can be defined as follows: ACP := (*operation name, role list*). The ACP for the operation *getReservation* of the travel reservation service can be defined as (getReservation, [agencyEmp]), where *agencyEmp* is a role for a travel agency employee.

As for a policy representation, there are standard specifications for ACPs. XACML [5] is a typical specification for access control policy, and we could use it for ACP. However we may need some extensions for XACML to express the ACP for a service itself. In this study, defining the expression of the ACP is not the main focus, so the representation of the ACP is not discussed here.

3.2.3. Composite Process Policy. As for the security policy of a composite service, we can define a DPP and an ACP for a composite service in the same way as for an atomic service. Additionally, we need to introduce a process policy for a composite process itself: the Composite Process Policy (CPP). In this paper, we define the CPP to specify the following requirements:

1. Special ACP for services invoked in a process
2. Separation of Duties: Services which should be executed by different roles
3. Order of services invoked in a process

The CPP should be defined by a developer who assembles services or interpreted from the business

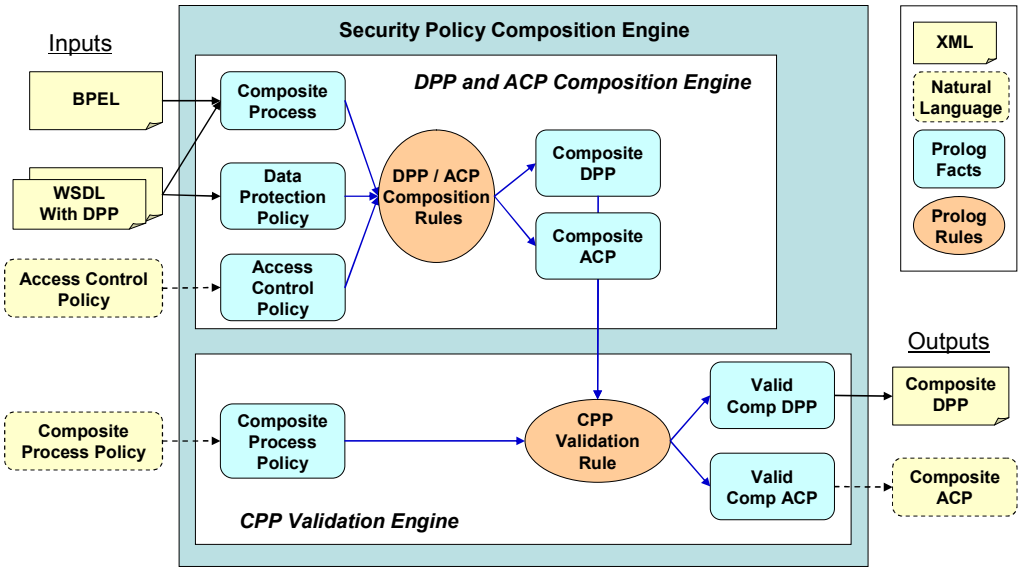

Figure 5. Security Policy Composition Architecture

requirements. However, these requirements tend to be written as a document in a natural language. There is no standardized representation for CPP which can be processed by software such as XML. As with the ACP, the representation of CPP is out of the scope of this paper.

This paper proposes a way to compose the security policies for a composite process from the policies of the external services invoked in the process. Our approach is based on predicate logic, and the policy composition rules are explained in the next section.

4. Security Policy Composition

4.1. Policy Composition Architecture

First, we explain the basic idea of policy composition. An atomic service has two kinds of security policies, DPP and ACP. A DPP can be regarded as security properties for the data itself. For example, suppose that the travel reservation service requests a customer ID, and a security policy requires that the request message should be signed and encrypted by a high level algorithm. We regard this as signifying that the customer ID itself is highly confidential data, so another service which uses this customer ID should maintain the high security by using the same high level algorithm. Therefore, a DPP can be defined from the security properties of the data used in the composite service.

The composition of the ACP is done in a similar way. An ACP defines requirements for a service operation itself, and then the ACP is considered as the properties of the service operation. We can say that an ACP for a composite service should be consistent with the properties of the operations of the atomic services.

We define this idea in predicate logic to provide a policy composition mechanism by using inference. Figure 5 illustrates our policy composition architecture. Our system has three main technical points: (1) Transformation into logic from the composite process and policy representation, (2) DPP and ACP composition rules, and (3) CPP validation rule. The inputs are process definitions in BPEL, service descriptions of a composite service in WSDL, and DPP written in WS-SecurityPolicy, and they can be transformed into predicates in our system. The policy composition is executed by two inferences using the DPP and ACP composition rule, and the CPP validation rule. In the first inference, we can get a composite DPP and ACP without inconsistencies among the policies of external services. These composite DPP and ACP if they have no violations of the CPP. The inference results are transformed into a description in WSDL of a composite service attached a security policy written in WS-SecurityPolicy, and then we can generate the concrete composite security policies.

In addition to this bottom-up policy composition, our system infers inconsistencies of composite policies from the top-down. If a security policy for a composite service is defined by a service developer, the composite policy can be used as an input to our system to check that there are no inconsistencies with the policies of the

atomic services. The two-way policy composition is an advantage of logic-based inference, and hence we take this approach for the security policy composition.

The following sections show the definitions of predicates for security policies and composition rules. Due to limited space, we show only parts of the predicates.

4.2. Process and Security Policies in Predicate Logic

The DPP written in WS-SecurityPolicy is attached in WSDL, and the BPEL process representation imports WSDL of both the composite service and the external services. Our system transforms these WSDL, BPEL, and DPP into predicates that are used as Prolog facts in the policy composition inference. The following are parts of the predicates for WSDL, BPEL, and DPP. Here, uppercase letters show the types of variables.

- **WSDL**

```
portType(i:Inter, o:Operation).
operation(o:Operation, req:RequestMsg,
          res:ResponseMsg).
variable(req:RequestMsg, reqvar:List).
variable(res:ResponseMsg, resvar:List).
```

The predicates for WSDL are defined straightforwardly from the XML elements in WSDL. The predicate *portType* has an interface *i* and an operation *o*, and the operation *o* exchanges a request message *req* and a response message *res*. Each message consists of a list of variables defined by a list *reqvar* and a list *resvar*.

- **BPEL**

```
receive(name:String, o:Operation, pl:PartnerLink,
        pt:PortType, req:RequestMsg).
invoke(name:string, o:Operation, pl:PartnerLink,
        pt:PortType, req:RequestMsg,
        res:ResponseMsg).
reply(name:String, o:Operation, pl:PartnerLink,
      pt:PortType, res:ResponseMsg).
assign(from:RequestMsg, fromvar:List,
        to:RequestMsg, tovar:List).
link(source:Action, target:Action).
```

We defined predicates for some actions in BPEL. The predicates *receive*, *invoke*, and *reply* are transformed from the corresponding actions. The variables in these predicates correspond to the attributes of an XML element for each action. Also, the variable assignment in the composite process is specified by the predicate *assign*, where the specified variable assignment from the variable *fromvar* of the message *from* is to the variable *tovar* of the message *to*. The predicate *link* means that an action *source* and an action *target* are linked directly in the process. There are no BPEL actions corresponding to the *link*, so the orders of actions which are specified in the composite process are transformed in the predicate *link*.

- **DPP**

```
dpp(m:Msg, sigIds:List, endIds:List, tokenIds:List,
    optionIds:List).
signature(m:Msg, sigId:String, var:List,
          tokenId:String, calgo:C14NM,
          salgo:SigM, talgo:TransM, dalgo:DigM).
encryption(m:Msg, endId:String, var:List,
           tokenId:String, kalgo:KeyEncM,
           dalgo:DataEncM).
token(m:Msg, tokenId:String, t:TokenType).
protectToken(m:Msg, optionId:String, sigId:String).
signedSupportingToken(m:Msg, optionIId:String,
                      tokenId:String).
```

DPP predicates are transformed from the WS-SecurityPolicy description. The WS-SecurityPolicy specification defines many security policy assertions to specify security requirements. However, by considering the WS-Security semantics, the security properties specified by WS-SecurityPolicy are classified into four types of security requirements: Signature, Encryption, Token, and OptionalRequirements. The predicate for DPP is *dpp*, which is applied to a message *m* and has Id lists for the four security requirements. The signature requirement is specified by a predicate *signature*, where s*igId* is the Id of this signature on a variable in a variable list *var*, and the *tokenId* is the Id of a security token used for this signature, using the canonicalization algorithm *calgo*, the signature algorithm *salgo*, the transform algorithm *talgo*, and the digest algorithm *dalgo*. Similarly, the predicate *encryption* is for the encryption requirements, where the key encryption algorithm is *kalgo* and the data encryption algorithm is *dalgo*. The predicate for the security token is *token*, which has variables for the token Id *tokenId* and the token type *t*, such as X509v3. Here, the token types must be predefined. In the WS-SecurityPolicy specification, there are many kinds of optional requirements. The two examples shown here are *protectToken* and *signedSupportingToken*. The predicate *protectToken* requires that a security token be used for the signature whose Id is *sigId*, which should also be signed by itself. The predicate

signedSupportingToken requires a signature on the security token whose Id is *tokenId*. Additional properties are not included due to the limited space, but our predicates are defined from the XML representation of WS-SecurityPolicy, so we can easily transform security policies into these predicates.

We also defined the predicates for ACP and CPP.

- **ACP**

```
acp(name:String, roles:Set).
available(name:String, o:Operation).
role(roleName:String).
level(role1:String, role2:String).
```

The predicate *acp* is an access control policy named *name* and the roles in *roles* are allowed to access the service. The ACP named *name* is applied to the service operation *o*, which is specified by the predicate *available*, and therefore the roles in *roles* can access the service operation *o*. The role names are defined by the predicate *role*. If a role *role2* has all of the rights to access the services for a role *role1*, we say that the role *role2* is a higher level role than the role *role1*. This relationship between two roles can be specified by the predicate *level*.

For a CPP, there are three kinds of process policies defined in Section 3.2, and they are specified in logic as follows:

- **CPP**

```
allowedRolesByProcess(o:Operation, roles:Set).
sod(o1:Operation, o2:Operation).
ordered(comp:Operation, o1:Operation, o2:Opreation).
```

The first predicate *allowedRolesByProcess* specifies that the roles in *roles* can access the operation *o* in the composite process. The second predicate *sod* specifies that the operations *o1* and *o2* should be executed by different roles in the process. And the predicate *ordered* is an order of service operation invocation for the composite operation *comp*. It means the operation *o1* should be invoked before the operation *o2*.

We define these three security policies in predicate logic, and we can execute them as Prolog facts. The composite security policies are inferred from these facts by using the policy composition rules defined in the next section.

4.3. Policy Composition Rules

As shown in Figure 5, our system executes two inferences to compose a security policy. In the first inference, the composite DPPs and ACPs are generated and in the second inference, they are validated by using the CPP validation rule. This section describes the composition rules as logic for the three types of security policies.

4.3.1. DPP composition. As explained in Section 4.2, the requirements of DPP are considered as data properties. Therefore, the composite DPP will be consistent with an external service DPP if the same security properties of variables used in both a composite service and an external service are equivalent.

- **DPP composition rule**

```
isIntegrityConsistent(comp:Operation, cVar:String,
                      ext:Operation, eVar:String,
                      calgo:C14NM, salgo:SigM,
                      talgo:TransM, dalgo:DigM,
                      t:TokenType) :-
  assignedToVar(comp:Operation, cVar:String,
                ext:Operation, eVar:String),
  requestIntegrity(ext:Operation, eVar:String,
                   calgo:C14NM, salgo:SigM,
                   talgo:TransM, dalgo:DigM,
                   t:TokenType),
  not(requestIntegrity(comp:Operation, cVar:String,
                       calgo:C14NM, salgo:SigM,
                       talgo:TransM, dalgo:DigM,
                       t:TokenType)).
```

The predicate *isIntegrityConsistent* is a constraint for consistency of data integrity. It returns true when the variable *cVar* of the composite operation *comp* and the variable e*Var* of the external operation *ext* have the same signature requirements, where the canonicalization algorithm is *calgo*, the signature algorithm is *salgo*, the transform method is *talgo*, and the digest method is *dalgo*, and the security token *t* which is used for the signature. The predicate *assignedToVar* infers a variable assignment from *cVar* to e*Var*. Based on our idea of DPP composition, the security properties of the assigned variables should be consistent. The predicate *requestIntegrity* returns true if a variable requires integrity, and the two predicates *requestIntegrity* for both *cVar* and *eVar* should be true. However, the predicate *isIntegrityConsistent* has a contradiction with the *requestIntegrity* for *cVar*, so it should return false if the composite DPP is consistent with DPPs of external services.

For the other DPP requirements, encryption and tokens, we define similar predicates, but they are omitted here due to the space limitations.

4.3.2. ACP composition. The ACP can be regarded as an operation property, so the composite ACP will be valid if properties of composite operation and external operations are consistent.

- **ACP composition rule**

```
isACPConsistent(comp:Operation, ext:Operation,
                roles:Set) :-
  invokedOperation(comp:Operation, ext:Operation),
  allowedRoles(ext:Operation, roles:Set),
  not(allRolesIncluded(comp:Operation, roles:Set)).
```

The predicate *isACPConsistent* is a constraint on the ACP's consistency between the composite service operation *comp* and the external service operation *ext*. The predicate *invokedOperation* means that a composite operation *comp* invokes an external service operation *ext* in the process. The predicate *allowedRoles* signifies that the external service operation *ext* is allowed for the roles specified in *roles*. The composite and external ACPs are consistent if the roles allowed by *ext* are included in the roles allowed by *comp*. The predicate *allRolesIncluded* returns true when all of the roles in *roles* are included in the allowed roles list of *comp*. The predicate *isACPConsistent* returns false if the composite and external ACPs are consistent. When true is returned, we can redefine the consistent ACP by referring a counterexample.

4.3.3. CPP validation. Here we define the constraints on CPP consistency for each of the three requirements defined in Section 3.2.3.

- **Validation rule for composite ACP**

```
isProcessACPSatisfied (o:Operation, roles:Set) :-
   allowedRolesByProcess(o:Operation, roles:Set),
   not(allowedRoles(o:Operation, roles:Set)).
```

The predicate *isProcessACPSatisfied* returns false if the operation *o* is allowed for the roles in *roles* by both the composite ACP and the CPP.

- **Validation rule for Separation of Duties**

```
isSODSatisfied(o1:Operation, o2:Operation) :-
   sod(o1; Operation, o2; Operation),
   allowedRoles(o1: Operation, roles1:Set),
   allowedRoles(o2: Operation, roles2:Set),
   not(allMembersNotIncluded(role1:Set, role2:Set)).
```

The predicate *isSODSatisfied* means that the allowed roles for operation *o1* are in *roles1* and for operation *o2* are in *roles2*, and no roles are required for both *roles1* and *roles2* if a separation of duty policy is specified by the predicate *sod*.

- **Validation rule for order of services**

```
isOrderSatisfied(comp:Operation,
                 p1:Process, p2:Process) :-
  ordered(comp:Operation, p1:Process, p2:Process),
  not(postInvokeProcess(comp:Operation,
                        p1:Process, p2:Process)).
```

The last predicate *isOrderSatisfied* is a rule to control the invocation order of the processes *p1* and *p2* in the composite operation *comp*. This predicate returns false when *p1* is invoked before *p2* if the predicate *ordered* is specified in the CPP.

We defined these constraints on policy consistency as rules to validate the composite DPP, ACP, and process. These predicates returns false when the rule is satisfied and invalid policies are shown as a counterexample when they return true. We can correct the invalid composite policy to satisfy the CPP validation rules by referring the counterexample.

Thanks to the way Prolog inference works, our system can work for both the top-down and bottom-up policy composition approaches. The consistent composite policies are inferred using the bottom-up approach, and the composite policies can be validated using the top-down policy definition approach. Also, we can compose policies using the same predicates even if external services are also composite services, not atomic services. In the next section, policy composition is demonstrated for the travel reservation service shown in Section 2.1.

5. Policy Composition Example

We are proposing a logic-based approach for security policy composition, and define the facts and rules for composite policy inference. The core part of our system as shown in Figure 5 is the security policy composition engine which is implemented in Prolog. Here we demonstrate our policy composition results from the core engine for the travel reservation scenario of Section 2.1.

In our system, composite process definitions and security policies for external services are assumed to be predefined and are transformed into the corresponding predicates. Here we give examples of the logic representation for the WSDL and BPEL of the composite process, the DPPs and the ACPs of the external services of the travel reservation service as follows.

Travel Reservation composite service

- **WSDL**

```
portType(agencyProcessInterface, getReservation).
operation(getReservation,
          agencyRequest, agencyResponse).
variable(agencyRequest,
         ['agp:airlineInfo', 'agp:hotelInfo',
          'agp:customerID', 'agp:mileageNo',
          'agp:cardInfo']).
variable(agencyResponse,
         ['agp:airlineResult', 'agp:hotelResult']).
```

- **BPEL**

```
receive(receive, getReservation,
        agencyProcessInterfacePartner,
        agencyProcessInterface, agencyRequest).
invoke(airlineReservation, reserveAirline,
        airlineProcessInterfacePartner,
        airlineProcessInterface,
        airlineRequest, airlineResponse).
reply(reply, getReservation,
      agencyProcessInterfacePartner,
      agencyProcessInterface, agencyResponse).
assign(agencyRequest, 'agp:mileageNo',
      airlineRequest, 'api:mileageNo').
link(receive, airlineReservation).
```

Some predicates for the travel reservation composite service are shown here. These predicates are transformed from WSDL and BPEL. The WSDL description defines that the operation of the composite service, its portType and variables. The operation is *getReservation* which receives a message *agencyRequest*. The predicates for BPEL specify actions in the operation *getReservation*. The predicates *invoke* mean that the operation *reserveAirline* is invoked. The predicates *link* specifies that the action *receive* links to the action *airlineReservation*. This composite process has eight assign actions, and one of them is shown here. This predicate *assign* means that the variable 'api:*mileageNo'* in the message *agencyRequest* is assigned to the variable 'api:mlieageNo' in the message *airlineRequest*.

Here we show some predicate examples for the composite process, and similar predicates for external services, airlineReservation and hotelReservation services, are necessary. They are omitted due to the space limitations here.

Airline Reservation service

- **DPP**

```
signature(airlineDpp, 'api:sigID1',
    ['api:mileageNo', 'api:airlineInfo', 'api:cardInfo'],
    'api:x509ID', exc14n, hmacsha1, exc14n, sha1).
token(airlineDpp, 'api:x509ID', x509V3).
token(airlineDpp, 'api:unID', username).
protectToken(airlineDpp, 'api:optionID1', 'api:sigID1').
signedSupportingToken(airlineDpp,
                      'api:optionID2', 'api:unID').
```

- **ACP**

```
available(airlineACP, reserveAirline).
acp(airlineACP, [agentEmp, airlineEmp]).
```

Here a simplified DPP and ACP are defined for the external services to explain the policy composition example. The DPP of the airline reservation service specifies that all request variables should be signed with a signed X509v3 security token, and a signed username token is required. The service can be used by a user who has a role as an *agentEmp* and an *airlineEmp*. The variables and IDs are as specified in QName to distinguish among the variables of different services.

Hotel Reservation service

- **DPP**

```
signature(hotelDpp, 'hpi:sigID1',
    ['hpi:customerID', 'hpi:hotelInfo', 'hpi:cardInfo'],
    'hpi:samlID', exc14n, hmacsha1, exc14n, sha1).
token(hotelDpp, 'hpi:samlID', saml).
protectToken(hotelDpp, 'hpi:optionID1', 'hpi:sigID1').
```

- **ACP**

```
available(hotelACP, reserveRoom).
acp(hotelACP, [agentEmp, hotelEmp]).
```

For the hotel reservation service, a similar DPP is defined. All of the request variables should be signed with a signed Saml security token. The service can be used by a user whose role is *agentEmp* and *hotelEmp*.

To generate the integrity requirements for the composite DPP, the predicate *isIntegrityConsistent* is used. Six solutions are inferred for the composite service the travel reservation service. One of the solutions is:

```
Comp = getReservation, CVar = 'agp:hotelInfo',
Ext = reserveRoom, EVar = 'hpi:hotelInfo',
C14NM = exc14n, SigM = hmacsha1,
TransM = exc14n, DigM = sha1,
TokenType = saml
```

This solution can be interpreted as saying that the variable '*agp:hotelInfo*' of the composite operation *getReservation* is equivalent to the variable '*hpi:hotelInfo*' of the external operation *reserveRoom* and these variables should be signed with a *saml* token and the inferred algorithms, i.e. the canonicalization

method and the transformation method are exclusive c14n, the signature method is HmacSha1, and the digest method is Sha1. The travel reservation service has six request variables, and the solutions correspond to each variable. The composite DPP will be generated by merging these solutions.

We can infer the composite ACP by executing the predicate *isACPConsistent*. Here there are no allowed roles defined for the travel reservation service, and we can get lists of the required roles for the service. Two solutions are inferred, and one of them is:

Comp = getReservation,
Ext = reserveAirline,
Roles = [agentEmp, airlineEmp]

This solution means that the composite operation *getReservation* needs to be allowed for *agentEmp* and *airlineEmp*, which are roles in the ACP of the external operation *reserveAirline*. The composite service developer can correct the ACP of *getReservation* by referring to this solution.

Here we demonstrated a security policy composition for a simple scenario which the external services are atomic, and it seems that we got just the results that were expected. However, the BPEL and WS-SecurityPolicy representations are quite complex and it is quite hard for developers to understand them and compose the security policies by hand. Also, invoked external services could be also composite services. Our proposed system can handle recursive service invocations. Therefore, a developer only needs to think of the top-level composite service for the policy composition even if the invoked external services are also implemented by another composite service, which is one of the benefits of our approach. Our main contribution is assuring consistency of the security policies among the invoked external and composite services without increasing the developer's workload.

This is a first step in our study of security policy composition, so there are some limitations and remaining issues. Now we are focusing on some of the specific actions in BPEL processes, such as *receive*, *invoke*, *replay*, and *assign*. Our work so far assumes that the external services are invoked symmetrically, but asymmetric invocations are important in practice, especially for large applications. Our current implementation is a core policy composition engine, so it is necessary to extend the implementation to support the transformations between the XML representations of BPEL and WS-SecurityPolicy. We will work on these aspects in the future.

6. Related Work

We proposed a logic-based approach for security policy composition. There is some earlier work for Web Services Security and policy using logic-based approaches.

Tziviskou and Nitto [7] proposed a formal specification for the requirements in WS-Security, and validate if the exchanged messages satisfy the requirements. Their approach is similar to ours, both in representing the security policy written using WS-SecurityPolicy and in using predicate logic. However, the goal to be achieved is clearly different. They focus on comparisons between two policies or messages. In contrast, we propose security policy composition rules using logic, and validate the consistency of the invoked external policies and the composite policies. Lee et al. [8] also worked to compose security policies, and they apply the concept of logically defeasible events to test the security policies written in WS-SecurityPolicy. Their motivation is a policy composition for different departments, and the composition preferences need to be defined by the policy writers. In our approach, the policy composition rules are predefined and will be executed according to the BPEL process. The policy translation from XML representation into logic can be processed automatically, and therefore our approach can compose policies without the efforts of policy writers.

There have been studies of policy representations using predicate logic, not only for WS-SecurityPolicy. Wang and Yuan [9] applied predicate logic to workflow security management. Their security policy focus is access control for a specific workflow, but the transformation from workflow into logic is not discussed in their work. Halpern and Weissman [10] proposed reasoning about policies by using first-order predicate logic. Their example of security policies are for access control policy to a document, and are simpler than our policies. Glasgow et al. [11] proposed a formal framework for security policy called Security Logic, and they apply modal logic to generic security policy considerations. Our work is more specific to SOA application security.

Security policy composition has been studied in a variety of fields. Li et al. [12] studied policy consistency in Web service compositions written in BPEL, with motivations similar to ours. They studied privacy policies and approaches to validate policies in graph transformations. He and Yang [13] analyzed security policy integration between different application domains. They provide requirements for security integration and patterns. Their targeted policy

is access control policy, but they do not mention the concrete policy representation. Srivatsa el al. [14] presented an access control model and techniques for specifying and enforcing access control rules for Web service compositions. They introduced composite roles and principles and specify access control policies using pure-past linear temporal logic. Charfi and Mezini [15] proposed an aspect-oriented approach to specify security policies for Web service compositions. They implement a set of aspects in AO4BPEL which is an aspect-oriented extension to BPEL. Bertino et al. [16] proposed an extension of WS-BPEL syntax with an authorization model. They specify authorization information and constraints using XACML [5] for the BPEL process, and can specify the Separation of Duty as with our approach. There are many related projects discussing security policies for processes, especially access control policies. Our study supports three kinds of security policies: DPP, ACP, and CPP, which is one of our advantages.

7. Conclusion

This paper proposes a security policy composition mechanism for composite services. To define security policies for a composite service which invokes some external services in the process, we need to maintain consistency with the security policies of the external services. However, it is quite hard for developers to define composite policies by hand because the composition rules are not clear for maintaining consistency and the composite processes and policies are themselves complicated. We propose a logic-based approach to compose security policies automatically from a composite process definition and existing security policies of external services. The composite processes and policies are represented using predicate logic, and the composite policies are inferred according to the composition rules we defined. The advantage of this approach is that two composition approaches can be supported: bottom-up and top-down policy composition. Also, our approach can handle a composite service which has a recursive structure. We considered three kinds of security policies for composite services: DPP, ACP, and CPP. We demonstrated the policy composition approach using a travel reservation service, and showed that consistent policies are inferred by our approach. This technology can contribute to assure SOA application security without increasing the developer's workload. In this study, some restrictions remain, such as implementations of transformations from concrete policy representations into predicates, and in supporting additional actions in BPEL, but we now begin work to address those restrictions.

References

[1] Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] WS-SecurityPolicy 1.2, http://www.oasis-open.org/committees/download.php/23821/ws-securitypolicy-1.2-spec-cs.pdf.

[3] Web Services Security: SOAP Message Security 1.1, http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[4] Eclipse BPEL project, http://www.eclipse.org/bpel/.

[5] WebSphere Integration Developer, http://www.ibm.com/software/integration/wid/.

[6] eXtensible Access Control Markup Language (XACML) Version 2.0, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[7] Tziviskou, C. and Nitto, E.D., Logic-based Management of Security in Web Services. Proc. of IEEE International Conference on Service Computing, 2007, pp. 228--235.

[8] Lee, A.J., Boyer, J.P., Olson, L.E. and Gunter, C.A., Defeasible security policy composition for web services. Proc. of the 4th ACM workshop on Formal methods in security, 2006, pp.45--54.

[9] Wang, H.J. and Yuan, M., Predicate Logic and its Application in Workflow Security Policy Management. http://math.arizona.edu/~ksimic/ming.doc, 2005.

[10] Halpern, J.Y. and Weissman, V., Using first-order logic to reason about policies. Proc. of 16th IEEE Computer Security Foundations Workshop, 2003, pp. 187--201.

[11] Glasgow, J., Macewen and G., Panangaden, P., A logic for reasoning about security. ACM Transactions on Computer Systems, vol. 10, Issue 3, 1992, pp. 226--264.

[12] Li, Y.H., Paik, H., Benatallah, B. and Benbernou, S., Formal Consistency Verification between BPEL Process and Privacy Policy. Privacy Security and Trust conference, 2006.

[13] He, D.D. and Yang, J., Security Policy Specification and Integration in Business Collaboration. Proc. of IEEE International Conference on Service Computing, 2007, pp. 20--27.

[14] Srivatsa, M., Iyengar, A., Mikalsen, T., Rouvellou I. and Yin, J., An Access Control System for Web Service Compositions. Proc. of IEEE International Conference on Web Services, 2007, pp. 1--8.

[15] Charfi, A. and Mezini, M., Using Aspects for Security Engineering of Web Service Compositions. Proc. of IEEE International Conference on Web Services, 2005, pp. 59--66.

[16] Bertino, E., Crampton, J. and Paci, F., Access Control and Authorization Constraints for WS-BPEL. Proc. of IEEE International Conference on Web Services, 2006, pp. 275--284.