Syntactic Validation of Web Services Security Policies

Yuichi Nakamura[1], Fumiko Sato[1], and Hyen-Vui Chung[2]

[1] IBM Research, Tokyo Research Laboratory
1623-14 Shimo-tsuruma, Yamato, Kanagawa
242-0001 Japan
{nakamury, sfumiko}@jp.ibm.com
[2] IBM Software Group, Web Service Security Development
11501 Burnet Road, Austin, TX
78758-3400 USA
hychung@us.ibm.com

Abstract. The Service-Oriented Architecture (SOA) makes application development flexible in such a way that services are composed in a highly distributed manner. However, because of the flexibility, it is often hard for users to define application configurations properly. Regarding the security concerns we address in this paper, though WS-SecurityPolicy provides a standard way to describe security policies, it is difficult for users to make sure that the defined policies are valid. In this paper, we discuss the validation of WS-SecurityPolicy in the context of Service Component Architecture, and propose a method called *syntactic validation*. Most enterprises have security guidelines, some of which can be described in the format of Web services security messages. There also exist standard profiles for Web services such as the WS-I Basic Security Profile that also prescribes message formats. Since those guidelines and profiles are based on accepted best practices, the syntactic validation is sufficiently effective for practical use to prevent security vulnerabilities.

1 Introduction

Many enterprises are undertaking development using the Service-Oriented Architecture (SOA) [1] because their business models are changing more frequently. SOA makes application development easier because technology-independent services can be coupled over intranets and via the Internet. Meanwhile, the underlying computing environments on which the applications are running are becoming complex, because computers can be networked in complicated topologies, including firewalls and intermediate servers. Consequently, the proper configuration of non-functional aspects such as security requires a fairly deep understanding of such complex environments.

We believe that security must be unified with the software engineering process from the beginning, and thus security engineering [2, 3] is important. Unfortunately, security is considered as an afterthought in most actual development in the sense that security is added after the functional requirements are implemented. It is well known that finding defects downstream greatly increases the costs of removal and repair.

Recently, Service Component Architecture (SCA) [4] is being standardized as a component model for SOA. More importantly, the SCA Policy Framework [5] is also being discussed in which intentions for non-functional requirements such as security

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 319–329, 2007.
© Springer-Verlag Berlin Heidelberg 2007

and transaction are specified at an abstract level, and the intentions are later mapped onto concrete policies such as WS-SecurityPolicy [6]. The concept of the SCA Policy Framework is quite similar that of the Model-Driven Security (MDS) architecture we have been developing [7, 8]. Where we added security intentions to UML [9] constructs such as classes and methods in MDS, we instead add intentions to SCA components and composites.

In this paper, we describe *syntactic validation* of WS-SecurityPolicy in the context of the SCA Policy Framework. According to the SCA Policy, we need to prepare in advance a collection of WS-SecurityPolicy documents so that we will retrieve the policies from the security intentions attached to the SCA composites. Therefore, it is important to define *valid* policy documents. Most enterprises have security guidelines, some of which can be described in the format of Web services security messages. There also exist standard profiles on Web services security [10] such as WS-I Basic Security Profile [11] that also prescribe message formats. Based on those guidelines and profiles, we think that we can prevent the security vulnerabilities in a highly practical way by means of syntactic validation which performs syntax checks of policies against guidelines and profiles.

Our main contribution is to show a practical way to validate WS-SecurityPolicy based on a solid foundation of predicate logic. While semantic validation which includes formal security analysis is often too complicated for practical situations, syntactic validations based on best practices can be realistic and sufficiently useful in many situations. We also describe a framework to transform WS-SecurityPolicy into predicate logic rules in an orderly fashion.

The rest of this paper is organized as follows: Section 2 introduces Web services security and Web services security policy, and discusses the problems in defining security policies. In Section 3, we begin by SCA Policy Framework, and describe the details of syntactic validation and show examples. Section 4 discusses related work. In Section 5, we conclude this paper.

2 Reviewing Web Services Policy

Here we introduce Web policy (WS-Policy) [12] and discuss the issues of defining WS-Policy, mainly focusing on Web services security (WSS) [10]. We begin by introducing WSS, giving a summary of the concepts and showing its XML message format. Then WS-Policy is presented, including a security-specific policy language called WS-SecurityPolicy [6]. Since the message format of WSS is complex, WS-SecurityPolicy naturally tends to become complex. As a result, it is often hard for policy developers to define security policies using WS-SecurityPolicy. Some problems are discussed in more detail in Section 2.3.

2.1 Web Services Security

The WSS specification [10] defines a format including security tokens and mechanisms to protect SOAP messages. Digital signatures serve as integrity checks to ensure message protection, and encryption guarantees confidentiality. In addition, WS-Security provides a flexible mechanism to include various claims in SOAP messages using security tokens. With message protection and security tokens, WSS can provide a basis for other specifications such as WS-Trust [13] and WS-SecureConversation [14].

WSS messages includes three types of elements: a Signature element (defined in the XML Digital Signature specification [15]), an encryption-related element such as EncryptedKey (defined in the XML Encryption specification [16]), and security tokens such as UsernameToken (defined in WSS UsernameToken Profile [17]). Listing 1 shows an example of a WSS message that includes an X.509 certificate as a security

Listing 1. WSS Message Example. In this example, we omit namespace declarations, and use abbreviated notations for URIs such as algorithm names in order to save space

```
<soap:Envelope>
<soap:Header>
 <wsse:Security>
  <wsse:BinarySecurityToken
    ValueType="X509v3" wsu:Id="X509Token" EncodingType="Base64Binary">
     MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
  </wsse:BinarySecurityToken>
    <ds:Signature>
      <ds:SignedInfo>
         <ds:CanonicalizationMethod Algorithm="xml-exc-c14n"/>
          <ds:SignatureMethod Algorithm=" rsa-sha1"/>
        <ds:Reference URI="#body">
        <ds:Transforms>
           <ds:Transform
              Algorithm=" xml-exc-c14n"/>
        </ds:Transforms>
        <ds:DigestMethod
              Algorithm="sha1"/>
        <ds:DigestValue>LyLsF094hPi4wPU...</ds:DigestValue>
     </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>Hp1ZkmFZ/2kQLXDJbchm5gK...</ds:SignatureValue>
  <ds:KeyInfo>
     <wsse:SecurityTokenReference>
         <wsse:Reference URI="#X509Token"/>
     </wsse:SecurityTokenReference>
  </ds:KeyInfo>
 </ds:Signature>
 </wsse:Security>
</soap:Header>
 <soap:Body wsu:Id="body">
  <tru:StockSymbol xmlns:tru="http://fabrikam123.com/payloads">
     QQQ
  </tru:StockSymbol>
</soap:Body>
</soap:Envelope>
```

2.2 WS-Policy and WS-SecurityPolicy

WS-Policy [12] provides a framework to describe policies which are associated with particular services. It defines a set of logical operators such as conjunction, All[. . .],

and disjunction, OneOrMore[. . .] so as to formulate domain-specific assertions. WS-SecurityPolicy is a domain-specific language to represent policies for message protection based on WSS and SSL. For example, we can describe our desired policy in such a way that a signature is required on a particular element or so that a particular element must be encrypted.

Listing 2 shows an example of WS-SecurityPolicy [6] that can be used for verifying or generating the WSS message shown in Listing 1. WS-SecurityPolicy has a number of sections for integrity and confidentiality assertions, bindings, and supporting tokens. Integrity and confidentiality assertions indicate which particular parts of the message should be signed and encrypted, respectively. A binding specifies detailed information to sign and encrypt some parts of messages such as signatures and gives encryption algorithms, security token information, and a layout for the WSS elements. Supporting tokens are additional tokens that are not described in a binding section. Listing 2 only includes an integrity assertion which appears as a SignedParts element, and a binding section which appears as an AsymmetricBinding element.

Listing 2. WS-SecurityPolicy Example. Actual representation requires inserting logical operators such as All and ExactlyOne that are all omitted here

```
<sp:AsymmetricBinding>
  <sp:InitiatorToken>
    <sp:X509Token sp:IncludeToken="AlwaysToRecpt">
      <sp:WssX509V3Token10/>
    </sp:X509Token>
  </sp:InitiatorToken>
  <sp:RecipientToken> .....  </sp:RecipientToken>
  <sp:AlgorithmSuite>
    <sp:Basic256/>
  </sp:AlgorithmSuite>
  <sp:Layout>
    <sp:Strict/>
  </sp:Layout>
</sp:AsymmetricBinding>

<sp:SignedParts>
 <sp:Body/>
</sp:SignedParts>
```

2.3 Issues in Defining Policies

Since WS-SecurityPolicy is extremely flexible, it is often hard for users to define *valid* policies with it. We can consider two kinds of validations: *syntactic* and *semantic* validations. Syntactic validation is concerned with validating the format of the messages. For example, we may have a syntactic rule such that a BinarySecurityToken must appear before Signature element. On the other hand, semantic validation means formal methods prove that the defined policy ensures no security vulnerability exists. For example, we may want to guarantee that attackers cannot alter messages during message transmission.

In our research, we focus on syntactic validation. Obviously, semantic validation is important since one of the ultimate goals is to prevent security vulnerabilities in a systematic manner. However, considering the complexity of WS-SecurityPolicy, it is hard to establish a theoretical foundation for semantic validation. In contrast, while syntactic validation does not provide security analysis in a theoretical sense, it can still be useful for security validation in real situations.

Most enterprises have security guidelines, which often describe detailed security requirements. For example, a requirement might say that when customer information is sent over a network, it should be encrypted with the RSA Encryption Standard Version 1.5 using a 1,024-bit key. This rule can be checked against the WS-SecurityPolicy by checking a certain element. Also, the format of WSS will be included in such guidelines, again taking account of various security considerations. With a good set of rules for WSS formats, syntactic validation can be sufficiently effective.

WS-I Basic Security Profile (BSP) provides several good examples. The following is one of them, C5543:

- When the signer's SECURITY_TOKEN is an INTERNAL_SECURITY_TOKEN, the SIGNED_INFO MAY include a SIG_REFERENCE that refers to the signer's SECURITY_TOKEN in order to prevent substitution with another SECURITY_TOKEN that uses the same key

This indicates that Listing 1 may not be secure since the BinarySecurityToken is not signed. This rule can be checked with syntactic validation.

In addition to security checking based on guidelines, interoperability is a major concern for WSS. Since WS-I BSP provides a set of rules for message formats, we can also improve interoperability by means of syntactic validation.

3 Policy Validation

Here, we describe syntactic policy validation. We first introduce a framework for developing applications for the Service Component Architecture (SCA), and explain how policies are defined and used in the framework. Then we discuss a theoretical foundation for performing syntactic validation using WS-SecurityPolicy. In our research, we use predicate logic to represent WS-SecurityPolicy, profiles such as WS-I BSP, and security guidelines. On the basis of the predicate logic, we can perform validations as inferences over the predicates. In this framework, we must transform WS-SecurityPolicy expressions to predicates, so the transformation is also described.

3.1 Policy Development for Service Component Architecture

Security should be considered from the beginning, though in most actual development it is considered as an afterthought. Our thesis is that users should be able to specify abstract security intentions at an initial stage, and the intentions can be refined toward a detailed security policy. This idea can be easily implemented based on the Service Component Architecture (SCA) [4] and the SCA Policy Framework [5].

Figure 1 illustrates how abstract policies are specified and refined. The assembler creates the SCA composite, combining the primitive components, and adds the abstract security intentions such as confidentiality and integrity to that composite. Policy developers define concrete policies typically represented in WS-Policy, specifying which intentions can be realized with each concrete policy. In our framework, a Policy deployer deploys the concrete policies to the SCA runtime in advance. When the SCA composites are deployed in the SCA runtime, appropriate concrete policies are retrieved based on the intentions attached to the SCA composites..

In our framework, it is important to make sure that valid concrete policies are deployed on the SCA runtime. Otherwise, even if the security intentions are appropriately added to SCA composite, the intentions will no be realized correctly, and security vulnerabilities may result.

3.2 Validation Based on Predicate Logic

We understand that a WS-Policy document prescribes a set of Web services messages based on predicate logic. For example, the WSS message in Listing 1 is a representative of the WSS messages that are prescribed by the WS-SecurityPolicy in Listing 2. Extending this notion, we see that security guidelines and profiles can be represented using predicate logic, and thus we can prescribe sets of messages.

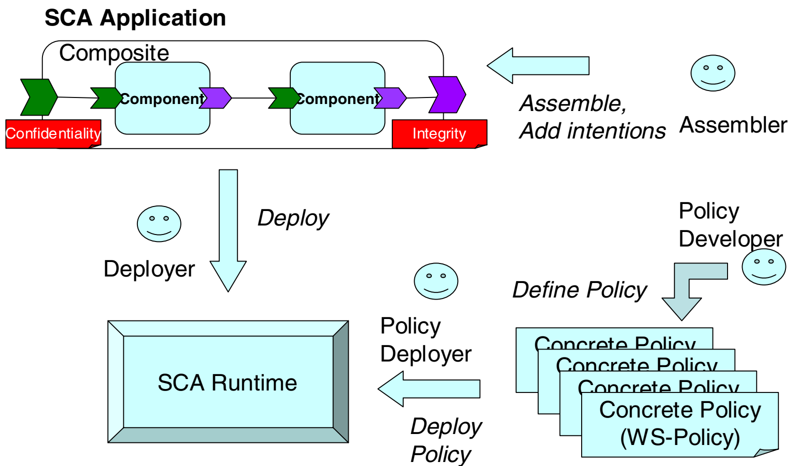

Fig. 1. Policy Configuration and Development for SCA

Figure 2 shows that validation can be viewed as a set operation between predicates. In the figure, WSSP1, WSSP2, and WSSP3 are WS-SecurityPolicy documents. Because the sets of WSSP1 and WSSP2 are both included in the set of BSP, we can say that WSSP1 and WSSP2 conform to BSP. In contrast, WSSP3 does not conform to BSP, since WSSP3 is not included in BSP.

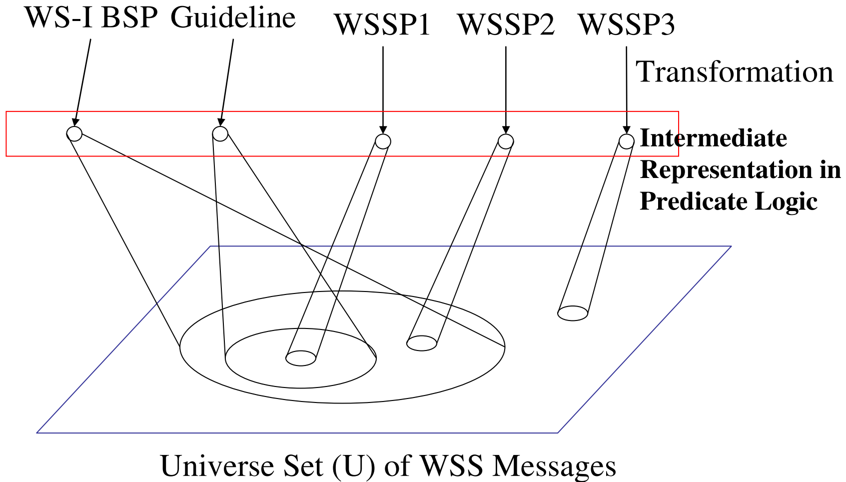

Fig. 2. Concept of WS-Policy Formalization Based on Predicate Logic

In our approach, WS-SecurityPolicy is represented using predicate logic, and threfore we can adopt Prolog [12] as a concrete representation and a calculation foundation. Listing 3 shows a Prolog program that is equivalent to the WS-SecurityPolicy

Listing 3. Prolog Program for a WS-SecurityPolicy Document

```
    myPolicy0(E):-
        E=env(H,B),
        H=h(Sec),
        Sec=
        sec(
            bst('@ValueType'('#X509v3'),
            '@EncodingType'('#Base64Binary'),
            '@id'(TokenID),
            bstValue),
            sig(
            sigInfo(
            c14nMethod('@Algorithm'('xml-exc-c14n#')),
            sigMethod('@Algorithm'('xmldsig#rsa-sha1')),
            ref('@URL'(BodyID),
                transforms(
                  transform(
                  '@Algorithm'('xml-exc-c14n#')),
                  digestMethod('@Algorithm'('xmldsig#sha1')),
                  digestValue(dVal))),
          sigValue(sVal),
          keyInfo(
            str(reference('@URI'(TokenID))))))),
      B=body('@id'(BodyID),bodyValue).
```

example in Listing 2. The right hand side primarily represents the structure of the WSS messages. We introduce a notation in which each tree structure is represented with a functor and its arguments. For example, “env” indicates an Envelop element, and it has the child elements header and body that are represented by the variables “H” and “B” in the program.

3.3 Transformation to Predicate Logic

WS-SecurityPolicy must be transformed into a predicate logic representation as indicated by the internal representation in Fig 2. Because of the complexity of WS-SecurityPolicy, the transformations are complex. In our approach, we classify the transformation into three types of rules, *primitive rules*, *structure rules*, and *merging rules*. Here are examples:

Primitive rules transform policy assertions into Prolog fragments. For example, the X509Token assertion in Listing 2 is transformed into this fragment in Listing 3:

```
bst('@ValueType'('#X509v3'),
    '@EncodingType'('#Base64Binary'),
    '@id'(TokenID),
    bstValue),
```

In the same manner, SignedPart assertion is transformed into “sig” and its child elements in Listing 3.

Structure rules order the elements of the header elements, and optionally change the order of processing. For example, a Layout assertion defines the order of elements in a SOAP header, and an EncryptBeforeSigning assertion requires that encryption must be performed before signing.

Merging rules define how to merge the Prolog fragments created by primitive rules. With only primitive and structure rules, the constructed messages may have redundant elements or may lack necessary associations between elements. Figure 3 illustrates how a transformation is performed. The Primitive rules for X509Token and SignedPart construct “bst” and “sig” elements, respectively. In addition, two merging rules are applied. First, Basic256 under AlgorithmSuite is used to specify an algorithm in the signature. Second, we associate an X.509 token and the signature, applying the rule that the signature element created by SignedPart must refer to a token created by InitiatorToken.

We have defined a set of rules classified into these three categories. Using these rules, we can transform WS-SecurityPolicy documents into our internal representation as Prolog programs.

3.4 Performing Validation

Since profiles and guidelines can be represented in Prolog, validation can be performed by executing the Prolog formulas. Let’s consider C5443 of WS-I BSP as introduced in Section 2.3. Listing 4 shows a Prolog program for C5443. Since this is similar to Listing 3, most of it is omitted and the crucial difference is emphasized in bold. The key difference is that C5443 requires signing on a security token, and therefore the reference to the token is included in the signature element.

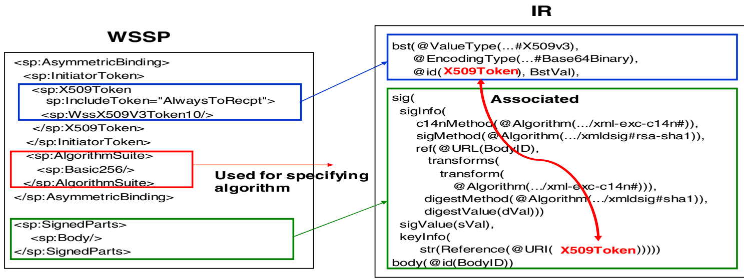

Fig. 3. Merging Token and Signature Elements

Listing 4. Predicate for the C5443 of WS-I BSP

```
c5443(E):-
    ....
    sec(
        sig(...
            ref('@URL'(BodyID), ...)
            ref('@URL'(TokenID), ..)
            ….
          ),
    B=body('@id'(BodyID),bodyValue).
```

Let us consider validation using myPolicy0 in Listing 3 and c5443 in Listing 4. One of the easiest ways is to perform the following formula:

~c5443(E),myPolicy0(E).

The result must always be “false” if myPolicy0 conforms to C5443. This formula indicates that there exists an envelope E that does not satisfy C5443, but satisfies myPolicy0. In this way, once the predicates have been represented as Prolog, the validations can be performed easily.

In addition to the validations, we can derive counterexamples by executing the formula. If the formula returns true, we should receive a substitution value, an envelope instance. The returned envelope is a counterexample in the sense that it can be derived from myPolicy0, but does not conform to C5443. As we mentioned, it is difficult for users to check if a WS-SecurityPolicy document is valid. On the other hand, counterexamples are often helpful for users seeking to understand the nature of bugs in the policy definitions.

Here is an example of how a user might apply this function. Listing 1 is shown as a counterexample and the user is informed that C5443 is not satisfied. When she compares the listing and C5443, she can see that the security token is not signed. In order to fix the bug, she needs to add a ProtectToken assertion that indicates the security token should be signed. Though we cannot yet programmatically offer suggestions about how to fix problems, such counterexamples can be good hints for users to help them fix such bugs by themselves.

4 Related Work and Discussion

While there has only a small amount of work on SCA security, there are several approaches for including security in the application models, especially for UML. SecureUML [20] is an attempt to integrate security into business application development. Addressing Role-Based Access Control (RBAC), it demonstrates a means to combine application models with security annotations. Security annotations are access controls on particular classes and additional support for specifying authorization constraints. This approach is quite different from ours, since we assume that application developers only add abstract intentions to the application models.

Deubler et al. [21] proposed an interesting approach to developing secure SOA applications. Using state transition diagrams and system structure diagrams (both similar to UML diagrams), they built application models, including security functions such as authentication. Then they perform model checking [22] to find security problems. Compared to our approach, they represent the mechanisms for security explicitly. For example, authentication and permission services are defined, specifying their behaviors. On the other hand, we think that such detailed security mechanisms should not be a concern of the application developers. In our tooling architecture, application developers only add security intentions that are associated with detailed security policies during deployment.

Bhargavan et al. [23] proposed formal semantics for WS-SecurityPolicy[1]. Concerned with XML rewriting attacks, their tool can automatically check whether or not the security goals of the formal model are vulnerable to any XML rewriting attacks. We regard such validation as semantic validation, since the model represents how to send, receive, and process security primitives such as tokens, timestamps, nonces, signatures, and encryption requests. While such semantic validation can prove that a given policy is secure, there are limitations in practical situations. For example, a formal model may address only limited types of security attacks, or may not represent all of the semantics of the security processing due to the complexity of WSS. As long as we cannot provide a complete solution, we think that the syntactic validation we are proposing should be useful. Since they provide a collection of guidelines and profiles, we can leverage them in order to reduce security risks.

5 Concluding Remarks

Since applications are becoming more complex in the SOA environment, it is becoming harder to configure their security. Addressing this issue, we introduced a security configuration framework based on the SCA policy concept, and discussed how to define valid WS-SecurityPolicy documents, since such valid definitions are critical when preparing valid security policies.

S*yntactic validation* of WS-SecurityPolicy is the key idea in this paper. Since most enterprises have security guidelines and best practices, we can leverage them to validate the security policies. Because guidelines can be described in the format of Web services security messages, we can eliminate security vulnerabilities in a highly practical way by means of syntax checking of the security policies against the guidelines, what we call syntactic validation. We can implment this idea using predicate logic,

[1] This work is based on an older version of WS-SecurityPolicy [24].

where the policies and guidelines are represented as Prolog programs. We also have described a framework to transform WS-SecurityPolicy into predicate logic rules in an orderly fashion.

While semantic validation is effective in theory, it requires formal security analysis that is often too complicated for practical situations. Still, syntactic validations based on best practices can be realistic and sufficiently useful in many situations. We will continue investigating this approach, accumulating and representing more guidelines using predicate logic.

References

1. A CBDI Report Series – Guiding the Transition to Web Services and SOA, http://www.cbdiforum.com/bronze/downloads/ws_roadmap_guide.pdf
2. Devanbu, P., Stubblebine, D.: Software Engineering for Security: a Roadmap. In: ICSE 2000 (2000)
3. Anderson, R.: Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley, Chichester (2001)
4. SCA Service Component Architecture: Assembly Model Specification, Version 1.00, (March 15, 2007)
5. SCA Policy Framework: Version 1.00 (March 2007)
6. WS-SecurityPolicy v1.2, Committee Specification (April 30, 2007), http://www.oasis-open.org/committees/download.php/23821/ws-securitypolicy-1.2-spec-cs.pdf
7. Tatsubori, M., Imamura, T., Nakamura, Y.: Best Practice Patterns and Tool Support for Configuring Secure Web Services Messaging. In: ICWS 2004 (2004)
8. Nakamura, Y., Tatsubori, M., Imamura, T., Ono, K.: Model-Driven Security Based on a Web Services Security Architecture. In: International Conference on Service Computing (2005)
9. Unified Modeling Language, http://www.omg.org/technology/documents/formal/uml.htm
10. Web Services Security: SOAP Message Security 1.1
11. Basic Security Profile Version 1.0, Final Material (March 30, 2003)
12. W3C Candidate Recommendation "Web Services Policy 1.5 –Framework" (February 28, 2007), http://www.w3.org/TR/2007/CR-ws-policy-framework-20070228/
13. WS-Trust 1.3 OASIS Standard (March 19, 2007)
14. WS-SecureConversation 1.3 OASIS Standard (March 1, 2007)
15. Eastlake, D., Solo, J.R., Bartel, M., Boyer, J., Fox, B., Simon, E.: XML Signature Syntax and Processing, W3C Recommendation (February 12, 2002)
16. XML Encryption Syntax and Processing, W3C Recommendation (December 10, 2002)
17. Web Services Security, UsernameToken Profile 1.1
18. Web Services Security: X.509 Certificate Token Profile 1.1
19. Prolog :- tutorial, http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html
20. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: Proceedings of UML2002 (2002)
21. Deubler, M., Grünbauer, J., Jürjens, J., Wimmel, G.: Sound Development of Secure Service-based Systems. In: ICSOC (2004)
22. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers, Boston (1993)
23. Bhargavan, K., Fournet, C., Gordon, A.D.: Verifying policy-based security for web services. In: CCS '04. Proceedings of the 11th ACM conference on Computer and communications security, pp. 268–277. ACM Press, New York (2004)
24. Web Services Security Policy Language (WS-SecurityPolicy) (December 18, 2002) http://www-106.ibm.com/developerworks/library/ws-secpol/