

# 國立交通大學

資訊學院 資訊學程

碩士論文

以有限狀態自動機達成XML簽章驗證的線性化  
Linear XML Signature Verification Scheme Based on Finite Automata



研究生：李建賢

指導教授：謝續平 教授

葉義雄 教授

中華民國九十七年七月

以有限狀態自動機達成XML簽章驗證的線性化

Linear XML Signature Verification Scheme Based on Finite Automata

研究生：李建賢

Student：Chien-Hsien Lee

指導教授：謝續平 博士

Advisor：Shiuh-Pyng Shieh

葉義雄 博士

Yi-Shiung Yeh

國立交通大學

資訊學院 資訊學程



Submitted to College of Computer Science  
National Chiao Tung University  
in partial Fulfillment of the Requirements  
for the Degree of  
Master of Science  
in  
Computer Science  
July 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年七月

# 以有限狀態自動機達成 XML 簽章驗證的線性化

研究生：李建賢


指導教授：謝續平

葉義雄

國立交通大學

資訊學院 資訊學程 碩士班

## 摘要



XML 簽章的主要瓶頸在於 Canonical XML, Canonical XML 是 W3C 定義正規化 XML 文件的方法, 此正規化流程稱為 Canonicalization (C14n), 降低 C14n 的複雜度就能大幅提升 XML 簽章的效能, 此研究提供一個轉換模型來降低 C14n 的複雜度, 在執行 XML 正規化的過程中透過以有限狀態自動機為基礎的轉換模型將每一個 XML 節點轉換成非遞迴的序列, 在簽章驗證的過程中透過一個有限狀態轉換器就能在線性時間內將此序列還原回 Canonical XML, 本研究提出得方法可以在 XML 簽章驗證時將正規化 XML 的複雜度降至  $O(n)$ , 此方法同時具有 streaming 的特性, 可以大幅降低記憶體使用率及提升運算速度, 適合使用於如防火牆及行動裝置這類低資源及低運算能力的裝置上。

# *Linear XML Signature Verification Scheme Based on Finite Automata*

**Student: Chien-Hsien Lee**

**Advisor : DR.Shiuh-Pyng Shieh**

**DR. Yi-Shiung Yeh**

Degree Program of Computer Science  
National Chiao Tung University

## **Abstract**

XML Signature main bottleneck is Canonical XML. Canonical XML is a formalization method for XML document defined in W3C Canonical XML [9]. This formalize process called Canonicalization (C14n). Reducing the complexity of C14n can also significantly improve the performance of XML Signature. This research provides a transformation model to reduce the complexity of C14n. In the processing of C14n, the results of node operations are converted to non-recursive binary sequence by the transformation model based on Finite Automata. For signature verification, the binary sequence can be restored into Canonical XML by Finite Automata in linear time. The proposed scheme can reduce the complexity of Canonical XML to  $O(n)$  and streaming characteristics. The characteristics of streaming can also substantially reduce memory usage and improve computing speed. This scheme is suitable for applications such as firewall or mobile devices with limited -resource or low computing capability.

## 誌 謝

本篇論文之所以能夠完成首先要感謝我的兩位指導教授葉義雄老師及謝續平老師，感謝葉義雄老師開放式的教學方式循序引導我的研究方向，葉教授已於2007年7月辭世，他為人處世的態度及對研究的熱忱將永存我心，感謝續平老師的大力幫忙在葉義雄老師過世後協助我繼續完成論文，謝謝教授總是不厭其煩的指導我論文寫作的方法，並以你豐富的經驗及卓越的學術素養提供我寶貴意見，循序引導我完成論文研究。其次要感謝我的父母及家人，在工作及學業的壓力下能不斷支持及鼓勵我，讓我能繼續堅持下去。感謝陳登吉老師及李鎮宇學長在葉教授過世後提供我諮詢及協助，再次感謝您及黃世昆老師擔任我的口試委員。感謝陳耀良特助，有你的鼓勵及協助我才能再次踏上求學之路，感謝呂宜薰副理、史素珍技術長及洪錦坤老大，有你們的幫助及體諒我才能兼顧工作及學業，特別感謝陳振楠總經理的關心及幫忙，我才能以在職進修的方式來交大唸書，也再次感謝您在百忙之中撥空擔任我的口試委員，謝謝品保組長楊博雄大哥常常提供諮詢，在我為了論文頭痛的時候不厭其煩的聽我發牢騷。最後要感謝交大及所有指導過我的教授給了我再一次求學的機會及美好的回憶。

# Table of Contents

|                    |  |           |
|--------------------|--|-----------|
| <b>1.</b>          | <b>Introduction .....</b>  | <b>1</b>  |
| <b>2.</b>          | <b>Overview.....</b>   | <b>3</b>  |
| 2.1.               | <b>XML-Signature Syntax .....</b>  | <b>3</b>  |
| 2.2.               | <b>Canonical XML .....</b>   | <b>7</b>  |
| 2.3.               | <b>XPath transformation .....</b>  | <b>8</b>  |
| 2.4.               | <b>XML Signature Processing .....</b>                                    | <b>9</b>  |
| <b>3.</b>          | <b>Related Work.....</b>   | <b>10</b> |
| 3.1.               | <b>Efficient Implementation of XML Security for Mobile Devices .....</b> | <b>10</b> |
| 3.2.               | <b>A Streaming Validation Model for SOAP Digital Signature.....</b>      | <b>10</b> |
| <b>4.</b>          | <b>Proposed Scheme .....</b>   | <b>12</b> |
| 4.1.               | <b>Methodology Analysis .....</b>  | <b>12</b> |
| 4.2.               | <b>Conversion of Structure and Content in Canonical Processing.....</b>  | <b>14</b> |
| 4.3.               | <b>Constructing Finite Automata of Canonical XML.....</b>                | <b>18</b> |
| 4.4.               | <b>Output Function for Generate Canonical Form.....</b>                  | <b>32</b> |
| 4.5.               | <b>Example for CFA .....</b>   | <b>33</b> |
| 4.6.               | <b>Support for XPath Transformation .....</b>                            | <b>35</b> |
| 4.7.               | <b>Complexity Analysis for CFA .....</b>                                 | <b>36</b> |
| <b>5.</b>          | <b>Performance Analysis .....</b>  | <b>38</b> |
| <b>6.</b>          | <b>Conclusion .....</b>  | <b>42</b> |
| <b>Appendix I</b>  | <b>The Algorithm of C14n Conversion.....</b>                             | <b>43</b> |
| <b>Appendix II</b> | <b>The Sample of XML Document For Performance Testing.....</b>           | <b>46</b> |
| <b>References</b>  | <b>.....</b>   | <b>48</b> |

# List of Figures

|   |           |
|---|-----------|
| <b>Figure 2-1 Example of XPath transformation .....</b>           | <b>8</b>  |
| <b>Figure 2-2 XML Signature Process flow.....</b>                 | <b>9</b>  |
| <b>Figure 4-1 Canonical XML Conversion.....</b>                   | <b>13</b> |
| <b>Figure 4-2 DOM tree traversing.....</b>                        | <b>15</b> |
| <b>Figure 4-3 Definition of CFA .....</b>                         | <b>19</b> |
| <b>Figure 4-4 Diagram of CFA .....</b>                            | <b>31</b> |
| <b>Figure 4-5 Example of CFA .....</b>                            | <b>33</b> |
| <b>Figure 5-1(a) Comparison without XPath transformation.....</b> | <b>41</b> |
| <b>Figure 5-1(b) Comparison with XPath transformation .....</b>   | <b>41</b> |



# List of Tables

|   |    |
|---|----|
| Table 4-1 Transition Table of CFA .....       | 30 |
| Table 5-1 testing environment .....           | 38 |
| Table 5-2 results of performance testing..... | 40 |





# 1. Introduction

XML (Extensible Markup Language) [10] is a structure text standard derived from SGML [19]. It realized the platform independence for the digital information exchange for Web or everywhere. The e-commerce technologies such as Web Services [20] or ebXML [18] are all developed based on XML. In order to preserve the characteristics of xml, the W3C define the XML Signature [8] standard to guarantee that data integrity and non-repudiation for XML message exchange. The difference with XML Signature [8] and traditional Digital Signature [14] is that XML Signature is protecting application content, not binary Data. The traditional digital signature [14] if there is one bit different will have completely different results; XML Signature [8] is not the case. For example, the serialization result of one XML document may have two or more encoding, but they are equivalent. Because XML Parser convert the XML to Objects[11], the content encoding of each node is Unicode(UTF-16)[13].The other characteristics such as comment processing, character escape, external resource , reference resolve, attribute sorting[9] and so on are all increasing the complexity of XML Signature implementation.

In addition, XML Signature supports partial Sign Based on XPath [12] technology. The XPath can address the node in the DOM tree to determinate that which one must be added in signature computing. In Multi-Hop [18] transmission process, this technology supports Multi-Sign but does not affect the previous Signature.

Canonical XML (C14n) [9] is the core technology for XML Signature. C14n is designed to formalize XML document. Because the processing of XML is application-oriented, therefore application concern is the contents of the documents rather than presentation view. C14n ensure that the document is logically equivalent and not physically.C14n is a special-purpose compiler, but it generates formalize octet stream (UTF-8 encoding) [9] [15] instead of executable program.

C14n computing includes many complex processing such as attribute and namespace sorting, etc. The lower bound of complexity for sorting is  $n \log n$ [2], so total cost for all elements in document is about  $O(n(D+m \log m))$  ( $D$  is the depth of node,  $m$  is the number of attribute or namespace[7]). Other operations of Canonical XML such as XPath transformation [8] [12] all its complexity is about  $O(n^2)$  or more complex. Therefore, the lower bound of XML Signature Complexity can be defined in  $O(n^2)$ . Thus, the performance of XML Signature main bottleneck can be defined in C14n. Reducing the complexity of C14n can also significantly improve the performance of XML Signature.

Recently, the performance research for XML Signature still depends on XML Parsing Model [11] [17] and XML Signature Syntax Processing [8] improvements. Thus, they cannot reduce the complexity anymore.

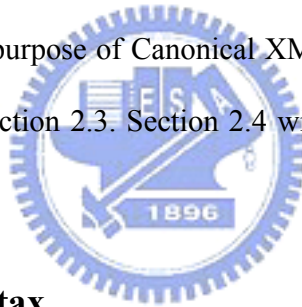
However, the C14n effort for Sign Processing is not avoided, because these operations such as parsing [11] [17], searching and sorting operations [9] cannot be eliminate. Since these efforts are unavoidable, why do not use these operations potential to improve the performance of verification. Therefore, the objective of this research is to reduce the complexity of C14n for XML Signature verification.

The rest of the paper is organized as follows. Section 2 describes the concept of XML Signature. The related work and limitation are introduced in Section 3. Section 4 describes the detail of proposed scheme, and the analysis of performance is given in Section 5. The conclusion is in Section 6.

## 2. Overview

XML Signature is a digital signature scheme designed for XML document signing. It depends on Document Object Model (DOM) for preserve XML document feature. DOM is a platform and language independent interface. XML Document can be converting to a DOM tree by XML parser. It allow program or script to access and update the content , structure and style dynamically by tree operation. Canonical XML is the core technology for XML Signature. It is a formalize method for XML by traversing DOM tree. In addition, XML Signature is also supporting partial sign by XPath transformation. It provides more flexibility to process XML document after document sign.

In this chapter, we will explain the concept of XML Signature. Section 2.1 introduces the syntax of XML Signature. The purpose of Canonical XML (C14n) and XPath transformation are illustrated in Section 2.2 and Section 2.3. Section 2.4 will describe the detail of XML Signature Processing.



### 2.1. XML-Signature Syntax

XML Signature format is also a XML document after signing. In this section, we will introduce the syntax and its element. The XML Signature syntax is showed as follows.

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI? >
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
```

```
(<KeyInfo>)?
(<Object ID?>)*
</Signature>
```

The document root is Signature element. It contains SignedInfo, SignatureValue, KeyInfo and Object four elements. These elements will be explained in the following.

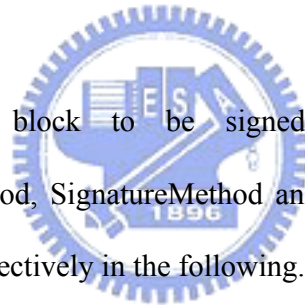
#### ■ **Signature :**

Signature element is the document root of XML Signature. It contains an attribute of id to identifier the Signature block. The example is showed as follows.

```
<ds:Signature Id="SIG20061208100303765"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
```

#### ■ **SignedInfo :**

SignedInfo is the block to be signed for XML Signature. It contains CanonicalizationMethod, SignatureMethod and Reference 3 element. These elements will be explained respectively in the following.



##### ● CanonicalizationMethod :

This element presents the algorithm of Canonicalization (C14n). C14n is the formalization method for XML. We will illustrate C14n in Section 2.2. The example is showed as follows.

```
<ds:CanonicalizationMethod
Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"></ds:Ca
nonicalizationMethod>
```

##### ● SignatureMethod :

This element presents the algorithm of Digital Signature. The example is showed as follows.

**<ds:SignatureMethod**

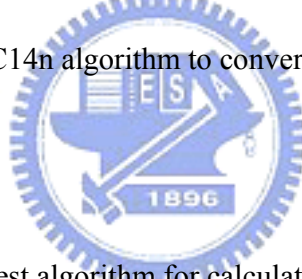
**Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"></ds:SignatureMethod>**

- **Reference :**

This element contains the content reference for document sign. The content reference can be contained in the same XML document or exist in external resource everywhere. It has the URI attribute to identifier the location of content. It also contains Transforms, DigestMethod and DigestValue 3 elements. We will explain these elements in the following.

**Transforms :**

This element is optional. It contains one or more Transform element. The Transform element presents the algorithm for content transformation. The transform algorithm convert the content of reference by URI attribute to another format. Such as C14n algorithm to convert a XML document to its canonical form.



**DigestMethod :**

It present the digest algorithm for calculate the reference content digest value after transformation.

**DigestValue :**

It presents the digest value for reference content after transformation.

The example of Reference is showed as follows.

**<ds:Reference URI="#xxx">**

**<ds:Transforms>**

**<ds:Transform**

**Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"></ds:Transform>**

**</ds:Transforms>**

**<ds:DigestMethod**

**Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></ds:DigestMethod>**

```
<ds:DigestValue>w+EVhhaShIexMB0P7acWmdjnuIw=</ds:DigestValue>
</ds:Reference>
```

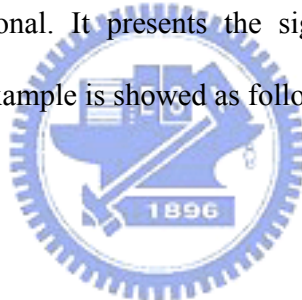
■ **SignatureValue :**

This element presents the signature value for XML Signature process. The source for sign is the SignedInfo block. The flow has two-step. First, Canonical the SignedInfo block to get an octet stream. Second, calculate the Signature for the octet stream by signature algorithm and key. The example is showed as follows.

```
<ds:SignatureValue>Okt8tnVm5HFoH0WvKiehm2eEZ0+rmFM2RsIVXOhPewvSsVus52/XeObN
S+YRoSoa5b6hiOpDovGgzvUFl8vU1K+2llleaJe+XeAcqo1sZGWQY15dcKCXH/EmvPk4mtp5/e
j0KQ93JHmOyM1RBvluX03IWmO6HohRjQa8DFOW1k=</ds:SignatureValue>
```

■ **KeyInfo :**

This element is optional. It presents the sign key information and certificate for document sign. The example is showed as follows.



```
<ds:KeyInfo>
  <KeyValue>
    <RSAKeyValue>
      <Modulus>qXckWQKfZVBSuUxxGWpeMj/3ROF0atV9Q9RKnKmz+GcbNx99zyMJUeYcs
y11TQbpaqBVi0Qw/naIEHnV6TThY+lyZOxZUSID3Yd0GNBWqvK40fdm8V511GoKVaR
Zze4iS6EdkAYKtVRQD/kTdESn8MeLPmJar11Xqs2RbbSGX0=</Modulus>
      <Exponent>AQAB</Exponent>
    </RSAKeyValue>
  </KeyValue>
  <ds:X509Data>
    <ds:X509IssuerSerial>
      <ds:X509IssuerName>CN=Hello Test XML CA, OU=Evaluation Only,
O=HELLO-CA.COM Inc., C=TW</ds:X509IssuerName>
      <ds:X509SerialNumber>1163646748</ds:X509SerialNumber>
    </ds:X509IssuerSerial>
    <ds:X509SubjectName>CN=12345678-01-001, OU=TST, OU=12345678-RA-001,
OU=Hello Test XML CA, O=Finance, C=TW</ds:X509SubjectName>
    <ds:X509Certificate>MIIEIjCCA4ugAw.....BupPdB</ds:X509Certificate>
  </ds:X509Data>
</ds:KeyInfo>
```

■ **Object :**

This element is optional. It contains the extra information or a XML document for XML Signature. The example is showed as follows.

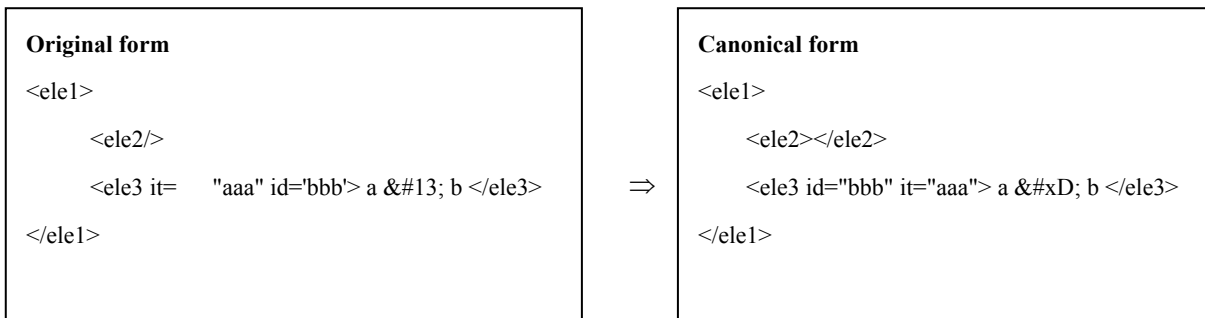
```
<dsig:Object Id="Res0" xmlns="" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"><ele1>  
<ele2 id="xxx">xxx</ele2>  
</ele1></dsig:Object>
```

## 2.2. Canonical XML

XML Canonicalization (C14N) is a formalization process for XML document. It defines a set of rules to convert each Node in the XML DOM tree to canonical form. Because XML document comparison is based on the principle of logically equivalent, the document must be formalized before comparison. In this section, we will describe the outline of C14n.

The rules of C14n contain the operations of syntax and structure. The rules of syntax formalize define the lexical formatting for content display with each node. The rules of structure formalize define the serialize order for node traversing. The node types for syntax formalize contain Document Root, Element, Attribute Node, Text Node, Processing Instruction (PI) Node and Comment Node. For example, the text node value, except all ampersands are replaced by `&amp;`; all open angle brackets (`<`) are replaced by `&lt;`; all closing angle brackets (`>`) are replaced by `&gt;`; and all `#xD` characters are replaced by `&#xD;`. The rules of structure formalize include the traversing order, namespace inheritance and attribute sorting. The document order is defined by the location of node and its relationship. The node traversing is following this order. In addition, the lexicographic order defines that which namespace or attribute is greater another by lexical. For example, the attribute “it” is greater “id”, because the first character code is equal, but the second character code “t” is greater “d”. Therefore, the attributes in element will be sorted by lexicographic order. Otherwise, the qualifier name (QName) for attribute is combining its local name and namespace prefix name. Thus, we must resolve the namespace value for prefix before sorting. The example of Canonical XML is showed as follows. The empty element `ele2` converts

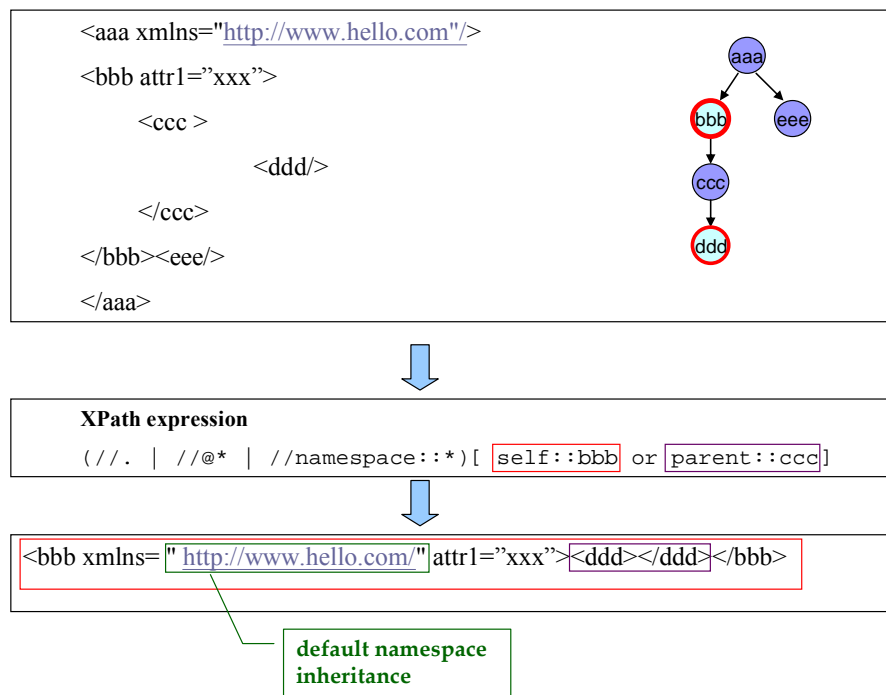
to start tag and end tag of ele2. The attributes it and id of ele3 are sorted by lexicographic order (ascending). The decimal code “13” in text node converts to hex code “D”.



### 2.3. XPath transformation

XPath is a language for addressing parts of XML document. XPath filtering is a process to transform a XML node set to others with XPath expression. The XPath expression is a predicate for node testing. If the node has satisfied the predicate, will be added to result set. Finally, we can get a node set for XPath transformation. The XPath transformation is very complex operation. Its complexity is  $O(n^2)$  or more complex(exponentiation).

Figure 2-1 Example of XPath transformation

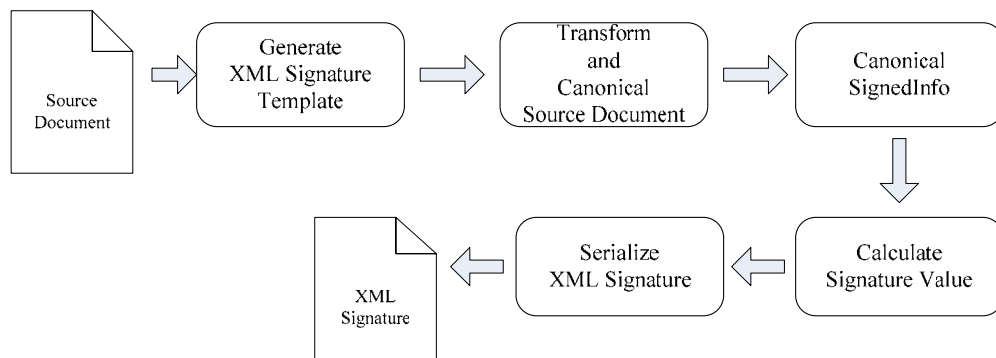




The example of XPath transformation is presented in Figure 2-1. The xml document is converted into a document tree of nodes. We test each node in document tree to determine which one satisfy the XPath predicate. The XPath expression is meaning that select all nodes which itself is “bbb” or its parent is “ccc”. We can find that node “bbb” and node “ddd” is satisfying the predicate. However, the node “bbb” also inherited the default namespace from node “aaa”. Because the result of XPath transformation is just a subset of original document, the definition of default namespace will be missing. To inherit the default namespace of ancestor node will fix this problem.

## 2.4. XML Signature Processing

XML Signature has three computing process in its processing flow. First, the digest value in the reference element must be calculated. Secondly, the SignedInfo block must be formalize by canonical algorithm. Finally, we calculate the signature value for the canonical form of SignedInfo. Figure 2-2 presents the processing flow of XML Signature calculation. The first step is generating the template document of XML Signature. Next, we can execute the computing processes in previous illustrate and filled the empty value in this template. Finally, the XML Signature can be output by serialize the Signature document. The process of verification is similar to signing. First, verify the digital signature for SignedInfo. Secondly, verify the digest value for each reference element.



**Figure 2-2 XML Signature Process flow**

## 3. Related Work

Efficient Implementation of XML Security for Mobile Devices [4] and A Streaming Validation Model for SOAP Digital Signature [1] are two researches to improve the performance of XML Signature. These two scheme are focused on XML Parsing Model [11] [17] and XML Signature Processing flow improvements. The scheme analysis and limitation will be illustrated in the following.

### 3.1. Efficient Implementation of XML Security for Mobile Devices

Efficient Implementation of XML Security for Mobile Devices [4] implements XAS API to process XML Signature all operations in memory. This scheme improves processing flow to avoid duplicate node traversing. First, it generates XML Signature template in memory. Secondly, calculates the digest value for each reference node. Finally, calculate the signature with “Signedinfo” node. The actor emphasized that this processing flow is streaming, but the performance cannot get significantly improvement. This skill is commonly used in XML Signature Syntax Processing since the specification was released in 2001, but it still depends on the original methods of Canonical XML and XML Syntax Processing. With process flow improvement, the key point to improve performance is reducing the number of DOM [11] tree traverse in C14n. In 2002, we have a study to complete all operations in C14n such as namespace Inheritance and attribute sorting in one pass, and then get substantial performance improvement. Now the apache XML Security Project [21] also uses this skill. However, this is the limit of the original method for C14n; the low bound of its complexity still exists.

### 3.2. A Streaming Validation Model for SOAP Digital Signature

A Streaming Validation Model for SOAP Digital Signature [1] implements a SAX-Like [17] XML parser to eliminate the effort of DOM Tree construction. The scheme is a streaming parsing

model to optimize parser. Therefore, it can reduce the time of traversing XML. However, this paper eliminates some C14n feature such as XPath transformation [8] [12] support etc., these feature can only support in DOM Parser. SAX can only traverse forward, it cannot search node in XML. Only DOM model supports node search with ancestor/descendent or sibling nodes. Some operations such as PI/Comment [10] locate or namespace [7] inheritance and attribute sorting [9] become difficult or impossible. Therefore, the actor was limiting this paper to deal with general cases. The complexity of C14n still has not reduced.

Since the limit of C14n processing always exists, the only way to defeat the low bound of complexity in C14n is abandoning the original C14n methodology. The proposed scheme is a new methodology to defeat this limit.



## 4. Proposed Scheme

Finite Automata is commonly using to process lexical or syntax [6]. We design a scheme based on finite automata to convert the Canonical XML into a sequence of structure and content. By this scheme, we can reduce the complexity to linear. In this chapter, the organization is described as follows. We begin with a methodology analysis in Section 4.1. Section 4.2 introduces the conversion of Canonical XML. Section 4.3 describes the definition of finite automata to accept the language defined in Section 4.2. Section 4.4 defines the output function call defined in finite automata for proposed scheme. Section 4.5 provides an example for implementation. XPath transformation supporting will be illustrated in Section 4.6. Finally, the complexity analysis will be discussed in Section 4.7.

### 4.1. Methodology Analysis

To reduce the complexity of C14n must to find the key point of bottleneck in C14n. In the computation model of C14n, it contains parsing, document tree traversing and node operations three main process. We will analysis the issue of each process and its solution in the following.

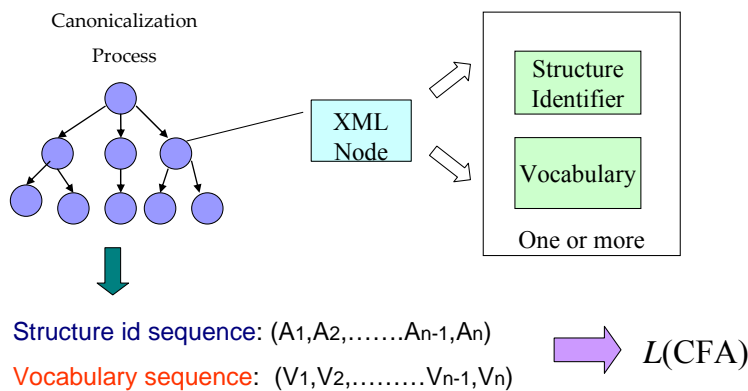
**Parsing:** XML is a text document. XML Parser parsing the XML document to build an object tree. The tree is named DOM. Each node in the DOM tree has its type and value. DOM tree presents the relationship of node in the document structure. The tasks of parser contain syntax validation, node generation and tree building. To eliminate the parsing process is to preserve the structure of object node and its relationship in the DOM tree.

**Document Tree traversing:** C14n is a process of DOM tree traverse. The operations of C14n apply to each node to generate its canonical form. The DOM Tree is a recursive structure. To traverse DOM tree must have stack to record the parent node of current node. The amount of memory need is also huge. To preserve the relationship of node without stack is impossible for original model. The Finite Automata is seems a good model to satisfy theses requires. It can store

the relationship of node by state and execute in the environment with limited-resource.

**Node operation:** C14n defines some complex operations for each node type. These operations formalize each node to generate its canonical form. To avoid these operations is to preserve the canonical form of node after formalize.

By previous analysis, we can begin to construction an efficient model for C14n based on Finite Automata. First, we convert canonical XML into a new language in canonicalization process. In Figure 4-1, each XML node is converted into a new form with structure and content. In the end of process, we can get two sequences structure identifier and vocabulary. Structure identifier defines the structure of each part in node. Vocabulary defines the canonical form of variable value in node. These two sequences form a new language  $L(CFA)$ .



**Figure 4-1 Canonical XML Conversion**

Secondly, we need a machine to accept the new language  $L(CFA)$ . CFA is the finite state transducer that accepts the language  $L(CFA)$  and generates Canonical XML.



In Section 4.2, we will describe the algorithm of language conversion and define the structure identifier and its vocabulary for each node type. The finite state transducer CFA will be

constructing in Section 4.3.

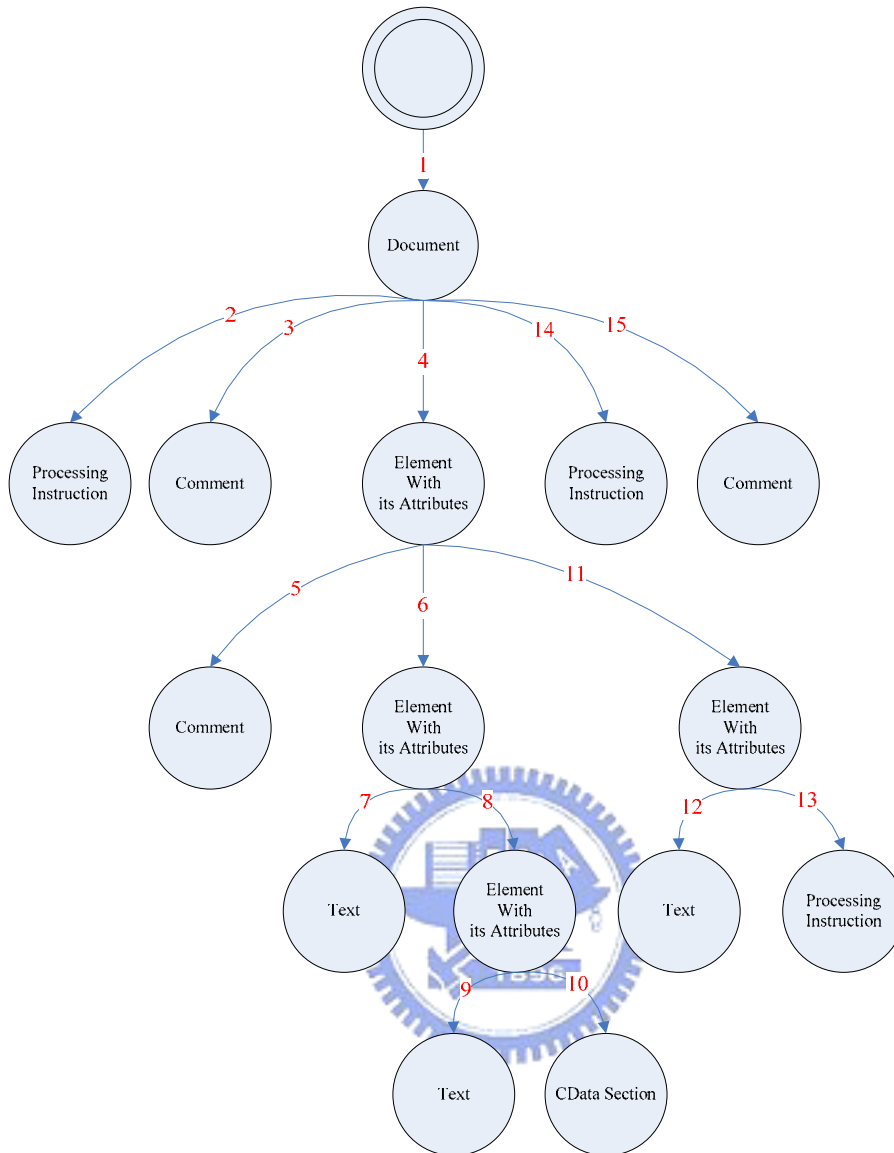
## 4.2. Conversion of Structure and Content in Canonical Processing

We want to convert canonical form of each node to its structure and content in the processing of Canonical XML. Canonical XML is a process of XML traversing; it is a recursive process flow. The process is similar to the operation of DFS (Depth First Search) [2], so it needs stack to record ancestor node. Our model depends on Finite Automata and streaming processing model, so we cannot store previous state. If we want to generate a non-recursive sequence after transformation, we must decomposition of one node to a number of structures. We will explain the transformation rule for different types of nodes in the following .

The engine of Canonical XML processing each node in DOM tree traversing. The processing flow of this traversing is showed as Figure 4-2. Each node is processing in document order defined in XPath. The document order define that the order of each node occurs in the XML representation after expansion. The traverse order is marked in each age of document tree.

For proposed scheme, the tree structure must to convert into a linear sequence. To reach this purpose can be regenerated the structure of nodes in the tree. Some nodes such as Document or Element in Figure 4-2 have descendents. The process of these nodes is traversing as deep as possible along the tree path. For go back to ancestor, the traversing need stack to record the path. To convert this recursive structure to a linear sequence is just to preserve the relationship of adjacent nodes. In addition, the Attribute node is contained in Element node. To convert the Element node to a simple structure must to extract Attribute from Element. For node converting, the set of node type is defined in the following.

{Document, Element, Attribute, Processing Instruction, CDATA Section, Text, Comment}



**Figure 4-2 DOM tree traversing**

Then, we define the transformation rule of each node type. The rules include how to convert node to structure identifier and its content. Because each node has fixed symbols and variable contents, therefore the meaningful content of each structure is just the variable content. The variable content of node is named **vocabulary**. For this transformation, we can convert Canonical XML into two sequences of structure identifier and its vocabulary.

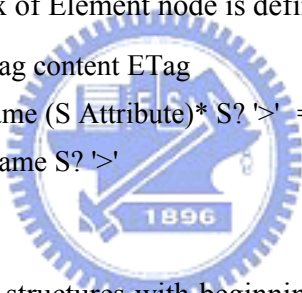
In following, the transformation rule will be explained for all node type. In addition, its structure identifier and vocabulary will be defined.

**Document:** The processing of document fragment and document is the same, so there is no separate definition for document fragment. Document node is the root of the tree, the others types of node are all in its context. Document node is converted into two structures with start document and end document. The syntax of Element node is defined as follows.

$$\begin{aligned} \langle \text{Document} \rangle &\rightarrow \text{DS } \langle \text{Context} \rangle \text{ DE} \\ \langle \text{Context} \rangle &= \{ \text{Element, Attribute, PI, CDATA Section, Text, Comment} \} \end{aligned}$$

The identifier of document start is defined as DS. The identifier of document end is defined as DE. Both two structures do not contain any vocabulary.

**Element:** The element node consists of Stag and Etag in its syntax. The Stag consists of start tag, attributes and end tag. The syntax of Element node is defined as follows.



$$\begin{aligned} \text{element} & ::= \text{STag content ETag} \\ \text{STag} & ::= \langle ' \text{ Name (S Attribute)* S? ' \rangle \Rightarrow \langle ' \text{ Name} \rightarrow \text{STB} \quad \text{and} \quad \rangle \rightarrow \text{STE} \\ \text{ETag} & ::= \langle / \text{ Name S? ' \rangle \Rightarrow \text{ETag} \rightarrow \text{ET} \end{aligned}$$

STag is converted into two structures with beginning and ending of start tag. The identifier of the start tag beginning is defined as STB, and the identifier of the start tag ending is defined as STE. Then, Etag is converted into one structure. The identifier of Etag is defined as ET. The vocabulary of the structures of STB and ET is the element name. The structure of STE is no vocabulary, it just identify the ending of start tag.

**Attribute:** The attribute node is composed of name and value. The syntax of Attribute node is defined as follows.

$$\text{Attribute} ::= \text{Name Eq AttValue} \Rightarrow \text{Name} \rightarrow \text{ATTRN} \text{ and } \text{AttValue} \rightarrow \text{ATTRV}$$

Attribute node is converted into two structures with attribute name and attribute value. The identifier of attribute name is defined as ATTRN and the identifier of attribute value is defined



ATTRV. Therefore, the vocabulary of ATTRN is attribute name, and the vocabulary of ATTRV is the canonical form of attribute value.

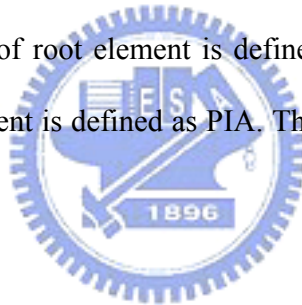
**Processing Instruction (PI):** PI is a simple and independent structure. The syntax of PI node is defined as follows.

```

PI ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
=>PI →PIC| PIB| PIA

```

PI is converted into to one structure by its location in document. The location of PI determines its structure identifier. It has tree different structure for its location. The structure identifier of PI node appears before the root element is defined as PIB. The structure identifier of PI node appears in the context of root element is defined as PIC. The structure identifier of PI node appears after the root element is defined as PIA. The vocabulary of PI is the canonical form of its target with data.



**CDATA Section:** It is a simple and independent structure. The syntax of CDATA Section node is defined as follows.

```

CDSect ::= CDStart CData CEnd
CDStart ::= '<![CDATA['
CData ::= (Char* - (Char* ']]>' Char*))
CEnd ::= ']]>'
=>CDSect → CDS

```

The CDSect consists of three parts CDStart, CData and CEnd. CDATA Section is converted into one structure. The identifier of CDATA Section is defined as CDS. The vocabulary of this structure is the canonical form of its CData value.

**Text:** Text is only a sequence of characters. The syntax of Text node is defined as follows.

$$\begin{aligned} \text{CharData} & ::= [^<\&]* - ([^<\&]* ''])> [^<\&]* \\ \text{Text} & \rightarrow \text{TN} \end{aligned}$$

Therefore, it converts to one structure. The identifier of Text node is defined as TN. The vocabulary of this state is the canonical form of CharData.

**Comment:** Comment is a simple and independent structure. The syntax of Comment Section node is defined as follows.

$$\begin{aligned} \text{Comment} & ::= '<!--' ((\text{Char} - '-') | ('-' (\text{Char} - '-')))* '-->' \\ \text{Comment} & \rightarrow \text{CNC} \mid \text{CNB} \mid \text{CNA} \end{aligned}$$

Comment is converted into one structure by its location in document. The location of Comment also determines its structure. It has tree different structure for its location. The structure identifier of Comment node appears before the root element is defined as CNB. The structure identifier of Comment node appears in the context of root element is defined as CNC. The structure identifier of Comment node appears after the root element is defined as CNA. The vocabulary of this tree state is the canonical form of Comment's data.

Now, we can add these transformation rules into the C14n process to convert Canonical XML into a new language CFA. The algorithm is described in **Appendix I**.

By this transformation, the each node in Canonical XML is converting to two sequences of structure identifier and its vocabulary. It is forming a new language. In next section, we will construct a Finite Automata to accept this language and generate Canonical XML for our purpose.

### 4.3. Constructing Finite Automata of Canonical XML

In previous section, Canonical XML is converted into a new language. Now we can begin to construction the Finite Automata [3] to accept the language and generate Canonical XML. Finite

automata have three major models generator, acceptor and transducer [22]. Generator only outputs and without input. Acceptor only receives input and without output. Transducer accepts input and generates output. For proposed scheme, we need a transducer to accept the structure sequence and generate Canonical XML. The transducer can be implement interpreter or compiler etc. Therefore, we want to construct a finite state transducer. A finite state transducer is a 7-tuple  $(Q, \Sigma, \delta, q_0, F, \Gamma, \lambda)$  [5]. The term  $Q$  is a finite set of states. The term  $\Sigma$  is the finite set of input symbols. The term  $\delta$  is transition function for states. The term  $q_0$  is the initial state in  $Q$ . The term  $F$  is the set of final states in  $Q$ . The term  $\Gamma$  is the finite set of output symbols. The term  $\lambda$  is the output function. The finite state transducer for proposed scheme is named Canonicalization Finite Automata (CFA). The definition of the Finite State Transducer CFA is defined as follows.

**CFA =  $(Q, \Sigma, \delta, q_0, F, \Gamma, \lambda)$**

**$Q$  is a finite set of states for CFA**

**$\Sigma$  is the input alphabet, it denotes the set of structure identifiers in the Canonical XML plus the end symbol**

**$q_0 \in Q$  is the initial state for finite automata CFA**

**$F \subseteq Q$  is the set of final states for finite automata CFA**

**$\delta : Q \times \Sigma \rightarrow Q$  is the state transition function for DOM tree structure**

**$\Gamma$  is the output alphabet, it denotes the set of functions call for generate canonical form each structure**

**$\lambda : Q \times \Sigma \rightarrow \Gamma$  is the output function**

**Figure 4-3 Definition of CFA**

Now, the term in CFA will be explained in the following. First, the set of alphabets is defined by structure of identifier in Section 4.2. Because the automata accepts the language defined in Section 4.2. If the automata accepts the sequence of structures, the structure validation

of XML document is correct. We collect all identifiers defined in Section 4.2 and plus an identifier EOS for the set of alphabet. The EOS is indicating the end of sequence. The list of all alphabets is defined in the following.

$$\Sigma = \{\text{All structure identifiers defined in Section 4.2}\} \cup \text{EOS}$$

$$\Rightarrow \{\text{DS,DE,STB,STE,ET,ATTRN,ATTRV,CDS,TN,PIC,PIB,PIA,CNC,CNB,CNA,EOS}\}$$

By the set of alphabet, we can define the set of states Q. The definition of state is that if a state  $S_i$  can accept alphabet A, the state is moving to  $S_A$ .  $S_A$  is label name of the state after transition. The definition is rewrite as follows.

$$\forall A \in \Sigma, \exists S_A \in Q$$

In addition, we define a state for the initial state and named “Start”. Because any state can accepts the alphabet “EOS” must move to the final state, so the state  $S_{\text{EOS}}$  is the final state. Now, we can define all states in Q.

$$Q = \text{Start} \cup \{S_A \mid \text{for all } A \in \Sigma\}$$

$$q_0 = \text{Start}$$

$$F = \{S_{\text{EOS}}\}$$



The set of output alphabet  $\Gamma$  is the function call for each structure. It depends on the alphabet  $\Sigma$ . If any state can accept alphabet A in  $\Sigma$ , the transition must output a function call  $G(A)$ . Therefore, the definition of output function can be rewrite as follows.

$$\lambda : \forall A \in \Sigma, \exists G(A) \in \Gamma$$

$$\Rightarrow \Gamma = \{G(A) \mid \text{for all } A \in \Sigma\}$$

The function call  $G(A)$  will be defined in Section 4.4. Finally, we will define the transition function  $\delta$  for each state in Q. Because the state presents the structure of Canonical XML in DOM traversing, the transition can be find in XML DOM structure.

In following, we will induct the transitions of state from the DOM structure and traversing order in C14n. First, we represent the DOM structure to a Context-Free Grammar in the following.

- ①  $\langle S \rangle \rightarrow \langle \text{Start} \rangle \langle \text{Document} \rangle \langle \text{End} \rangle \mid \langle \text{Start} \rangle \langle \text{Document fragment} \rangle \langle \text{End} \rangle$
- ②  $\langle \text{Document} \rangle \rightarrow \langle \text{Start Document} \rangle \langle \text{PC}_B \rangle \langle \text{Element} \rangle \langle \text{PC}_A \rangle \langle \text{End Document} \rangle$
- ③  $\langle \text{PC}_B \rangle \rightarrow \langle \text{PI}_B \rangle \langle \text{PC}_B \rangle \mid \langle \text{COMMENT}_B \rangle \langle \text{PC}_B \rangle \mid \epsilon$
- ④  $\langle \text{PC}_A \rangle \rightarrow \langle \text{PI}_A \rangle \langle \text{PC}_A \rangle \mid \langle \text{COMMENT}_A \rangle \langle \text{PC}_A \rangle \mid \epsilon$
- ⑤  $\langle \text{Document fragment} \rangle \rightarrow \langle \text{Element} \rangle \mid \langle \text{PI} \rangle \mid \langle \text{PI}_B \rangle \mid \langle \text{PI}_A \rangle \mid \langle \text{Comment} \rangle \mid \langle \text{COMMENT}_B \rangle \mid \langle \text{COMMENT}_A \rangle \mid \langle \text{Text} \rangle \mid \langle \text{CDATASection} \rangle$
- ⑥  $\langle \text{Element} \rangle \rightarrow \langle \text{Beginning of Start Element} \rangle \langle \text{Attrs} \rangle \langle \text{Ending of Start Element} \rangle \langle \text{Content} \rangle \langle \text{End Element} \rangle$
- ⑦  $\langle \text{Content} \rangle \rightarrow \langle \text{Element} \rangle \langle \text{Content} \rangle \mid \langle \text{PI} \rangle \langle \text{Content} \rangle \mid \langle \text{Comment} \rangle \langle \text{Content} \rangle \mid \langle \text{Text} \rangle \langle \text{Content} \rangle \mid \langle \text{CDATASection} \rangle \langle \text{Content} \rangle \mid \epsilon$
- ⑧  $\langle \text{Attrs} \rangle \rightarrow \langle \text{Attr} \rangle \langle \text{Attrs} \rangle \mid \epsilon$
- ⑨  $\langle \text{Attr} \rangle \rightarrow \langle \text{Attribute name} \rangle \langle \text{Attribute value} \rangle$
- ⑩  $\langle \text{Start Document} \rangle \rightarrow \text{DS}$
- ⑪  $\langle \text{End Document} \rangle \rightarrow \text{DE}$
- ⑫  $\langle \text{Beginning of Start Element} \rangle \rightarrow \text{STB}$
- ⑬  $\langle \text{Ending of Start Element} \rangle \rightarrow \text{STE}$
- ⑭  $\langle \text{End Element} \rangle \rightarrow \text{ET}$
- ⑮  $\langle \text{Attribute name} \rangle \rightarrow \text{ATTRN}$
- ⑯  $\langle \text{Attribute value} \rangle \rightarrow \text{ATTRV}$
- ⑰  $\langle \text{Text} \rangle \rightarrow \text{TN}$
- ⑱  $\langle \text{CDATASection} \rangle \rightarrow \text{CDS}$
- ⑲  $\langle \text{PI} \rangle \rightarrow \text{PIC}$
- ⑳  $\langle \text{PI}_B \rangle \rightarrow \text{PIB}$
- ㉑  $\langle \text{PI}_A \rangle \rightarrow \text{PIA}$
- ㉒  $\langle \text{Comment} \rangle \rightarrow \text{CNC}$
- ㉓  $\langle \text{COMMENT}_B \rangle \rightarrow \text{CNB}$
- ㉔  $\langle \text{COMMENT}_A \rangle \rightarrow \text{CNA}$
- ㉕  $\langle \text{Start} \rangle \rightarrow \epsilon$
- ㉖  $\langle \text{End} \rangle \rightarrow \text{EOS}$

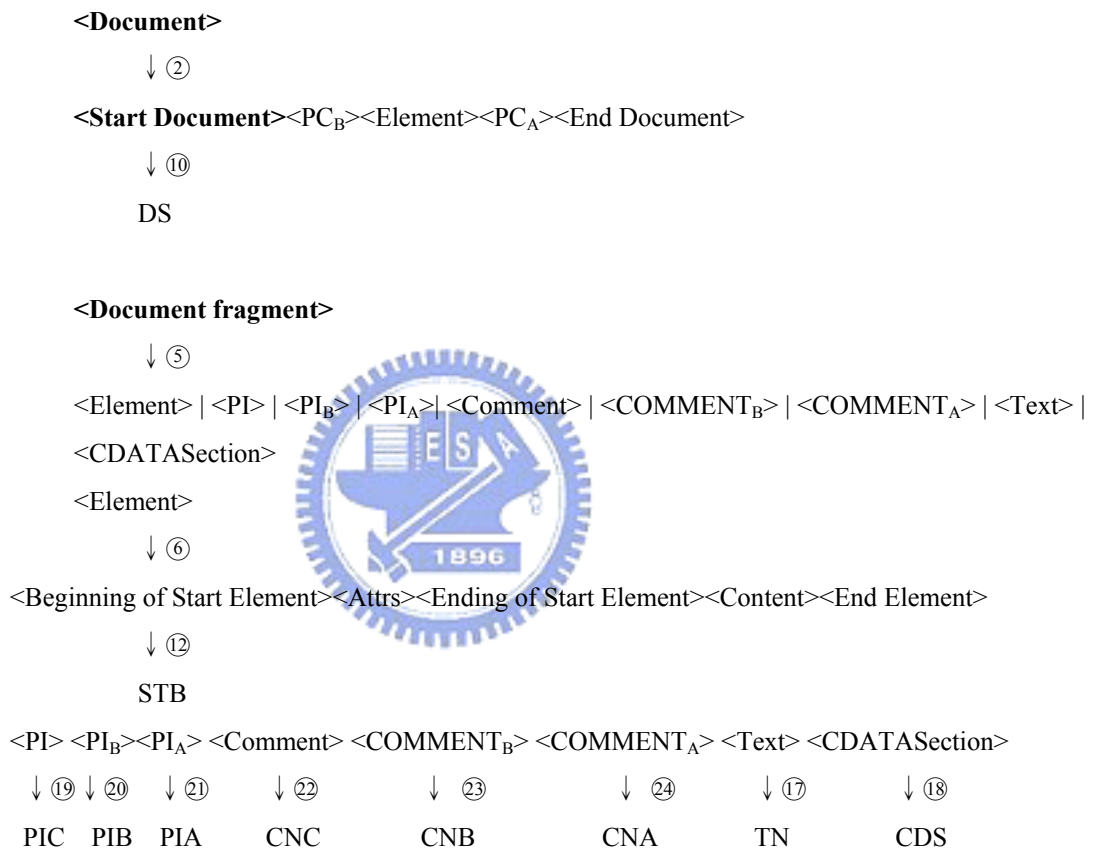


The transition of each state in Q is the set of alphabets it can be accepted. Because the state  $S_A$  is the result of state  $S_i$  accepted alphabet A, to find the next adjacent alphabet with alphabet A in this language can get the transition of  $S_A$ . We can find the next adjacent alphabet by derived from the CFG. The derive process are showed in the following.

**The transition of state “Start” is derived as follows :**

Because the Start is the initial state, the alphabet A is  $\epsilon$  for this state by previous description. The alphabet  $\epsilon$  can be map to variable <Start>. By rule 1, the next adjacent variables of “<Start>” are <Document> and <Document fragment>. Each arrow in the following indicates the derive process. The rule defined in context-free grammar in this section is applied in each derive process.

The rule number is appending in each arrow.



The state of “Start” can accept alphabets DS, STB, PIC, PIB, PIA, CNC, CNB, CNA, TN and CDS.

**The transition of state “S<sub>STB</sub>” is derived as follows :**

By rule 6, the alphabet STB is mapping to variable “<Beginning of Start Element>” for state S<sub>STB</sub>.

By rule 6, the next adjacent variable of “<Beginning of Start Element>” is <Attrs>.

<Attrs>  
 ↓ ⑧  
 <Attr><Attrs> |  $\epsilon$   
 ↓ ⑨  
 <Attribute name><Attribute value>  
 ↓ ⑮  
 ATTRN

If <Attrs> is  $\epsilon$ , the next adjacent variable of “<Beginning of Start Element>” in rule 6 is <Ending of Start Element>.

<Ending of Start Element>  
 ↓ ⑬  
 STE

The state of “ $S_{STB}$ ” can accept alphabets ATTRN and STE.

**The transition of state “ $S_{STE}$ ” is derived as follows :**

By rule 13, the alphabet is mapping to variable “<Ending of Start Element>” for state  $S_{STE}$ .

By rule 6, the next adjacent variable of “<Ending of Start Element>” is <Content>.

<Content>  
 ↓ ⑦

<Element><Content> | <PI><Content> | <Comment><Content> | <Text><Content> |  
 <CDATASection><Content> |  $\epsilon$

<Element>  
 ↓ ⑥  
 <Beginning of Start Element><Attrs><Ending of Start Element><Content><End Element>  
 ↓ ⑫  
 STB  
 <PI> <Comment> <Text> <CDATASection>  
 ↓ ⑰     ↓ ⑳     ↓ ⑱     ↓ ⑱  
 PIC     CNC     TN     CDS

If < Content > is  $\epsilon$ , the next adjacent variable of “<Ending of Start Element>” in rule 6 is <End Element>.

<End Element>  
 ↓ ⑭  
 ET

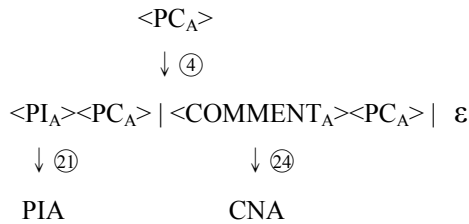
The state of “ $S_{STE}$ ” can accept alphabets STB, PIC, CNC, TN, CDS and ET.

**The transition of state “S<sub>ET</sub>” is derived as follows :**

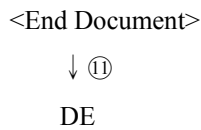
By rule 14, the alphabet ET is mapping to variable “<End Element>” for state S<sub>ET</sub>.

By rule 6, the <End Element> is in the end of this rule. Therefore, we want to find the next adjacent with <Element>.

By rule 2, the next adjacent variable of “<Element>” is <PC<sub>A</sub>>.

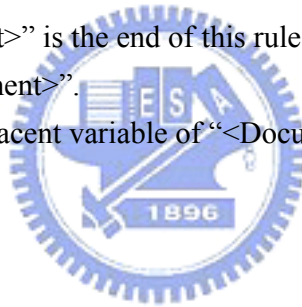
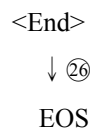


If <PC<sub>A</sub>> is ε, the next adjacent variable in rule 2 is “<End Document>”.

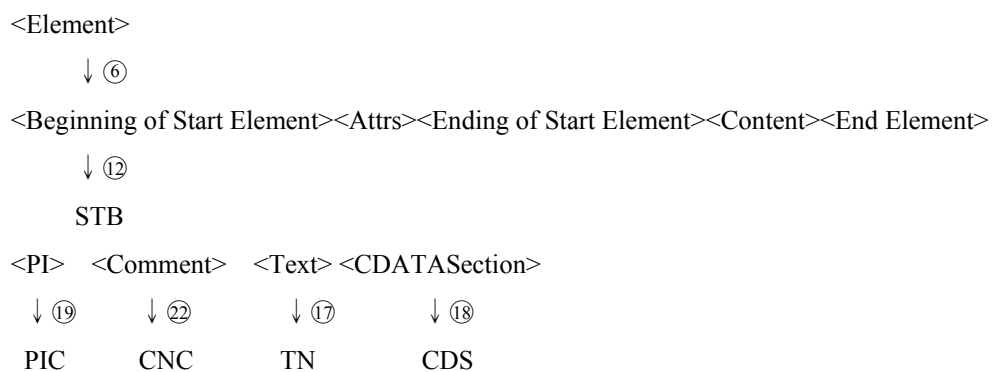
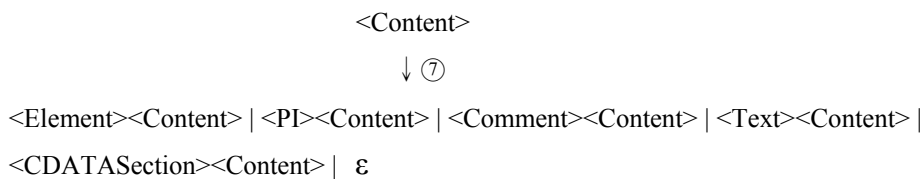


By rule 5, we find the “<Element>” is the end of this rule; we want to find the next adjacent variable with “<Document fragment>”.

By rule 1, the next adjacent variable of “<Document fragment>” is “<End>”.



By rule 7, the next adjacent variable of “<Element>” is <Content>.





If  $\langle \text{Content} \rangle$  is  $\epsilon$ , the next adjacent variable of “ $\langle \text{Ending of Start Element} \rangle$ ” in rule 6 is  $\langle \text{End Element} \rangle$ .

$\langle \text{End Element} \rangle$   
 $\downarrow$  ⑭  
 ET

The state of “ $S_{ET}$ ” can accept alphabets PIA, CNA, DE, EOS, STB, PIC, CNC, TN, CDS and ET

**The transition of state “ $S_{ATTRN}$ ” is derived as follows :**

By rule 15, the alphabet ATTRN is mapping to variable “ $\langle \text{Attribute name} \rangle$ ” for state  $S_{ATTRN}$ .

By rule 9, the next adjacent variable of “ $\langle \text{Attribute name} \rangle$ ” is  $\langle \text{Attribute value} \rangle$ .

$\langle \text{Attribute value} \rangle$   
 $\downarrow$  ⑯  
 ATTRV

The state of “ $S_{ATTRN}$ ” can accept alphabet ATTRV

**The transition of state “ $S_{ATTRV}$ ” is derived as follows :**

By rule 16, the alphabet ATTRV is mapping to variable “ $\langle \text{Attribute value} \rangle$ ” for state  $S_{ATTRV}$ .

By rule 9, the  $\langle \text{Attribute value} \rangle$  is the end of this rule. Therefore, we want to find the next adjacent variable with  $\langle \text{Attr} \rangle$ .

By rule 8, the next adjacent variable of “Attr” is “Attr”.

$\langle \text{Attr} \rangle$   
 $\downarrow$  ⑧  
 $\langle \text{Attr} \rangle \langle \text{Attr} \rangle \mid \epsilon$   
 $\downarrow$  ⑨  
 $\langle \text{Attribute name} \rangle \langle \text{Attribute value} \rangle$   
 $\downarrow$  ⑮  
 ATTRN

If  $\langle \text{Attr} \rangle$  is  $\epsilon$ , the next adjacent variable of “ $\langle \text{Attr} \rangle$ ” in rule 6 is  $\langle \text{Ending of Start Element} \rangle$ .

$\langle \text{Ending of Start Element} \rangle$   
 $\downarrow$  ⑬  
 STE

The state of “ $S_{ATTRV}$ ” can accept alphabets ATTRN and STE.

**The transition of state “ $S_{TN}$ ” is derived as follows :**

By rule 17, the alphabet TN is mapping to variable “ $\langle \text{Text} \rangle$ ” for state  $S_{TN}$ .

By rule 5, the variable “<Text>” is the end of this rule. Therefore, we want to find the next adjacent variable with “<Document fragment>”.

By rule 1, the next adjacent variable of “<Document fragment>” is “<End>”.

<End>  
 ↓ ②⑥  
 EOS

By rule 7, the next adjacent variable of “<Text>” is <Content>.

<Content>  
 ↓ ⑦

<Element><Content> | <PI><Content> | <Comment><Content> | <Text><Content> |  
 <CDATASection><Content> | ε

<Element>  
 ↓ ⑥

<Beginning of Start Element><Attrs><Ending of Start Element><Content><End Element>

↓ ⑫

STB

<PI> <Comment> <Text> <CDATASection>

↓ ⑲

PIC

↓ ⑳

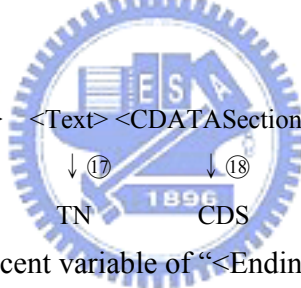
CNC

↓ ⑰

TN

↓ ⑱

CDS



If < Content > is ε , the next adjacent variable of “<Ending of Start Element>” in rule 6 is <End Element>.

<End Element>  
 ↓ ⑭  
 ET

The state of “S<sub>TN</sub>” can accept alphabet EOS, STB, PIC, CNC, TN, CDS and ET.

**The transition of states “S<sub>CDS</sub>”, “S<sub>PIC</sub>”, “S<sub>CNC</sub>” is derived as follows :**

By rule 18, the alphabet CDS is mapping to variable <CDATASection> for state S<sub>CDS</sub>.

By rule 19, S<sub>PIC</sub> is mapping to variable <PI>.

By rule 22, S<sub>CNC</sub> is mapping to variable <Comment>.

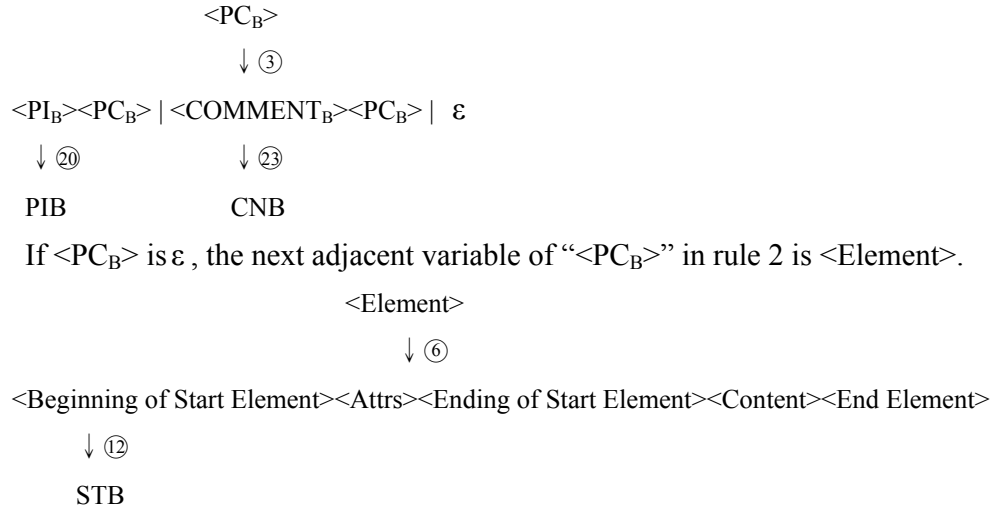
All three variables are in rule5 and rule7, the transition is same as “S<sub>TN</sub>”

The states of “S<sub>CDS</sub>,” S<sub>PIC</sub>” and “S<sub>CNC</sub>” can accept alphabets EOS, STB,PIC,CNC,TN,CDS and ET.

**The transition of state “S<sub>PIB</sub>” is derived as follows :**

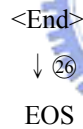
By rule 20, the alphabet PIB is mapping to variable <PI<sub>B</sub>> for state S<sub>PIB</sub>.

By rule 3, the next adjacent variable of <PI<sub>B</sub>> is <PC<sub>B</sub>>.



By rule 5, we find the “<PI<sub>B</sub>>” is the end of this rule; we want to find the next adjacent variable with <Document fragment>.

By rule 1, the next variable of “<Document fragment>” is <End>.

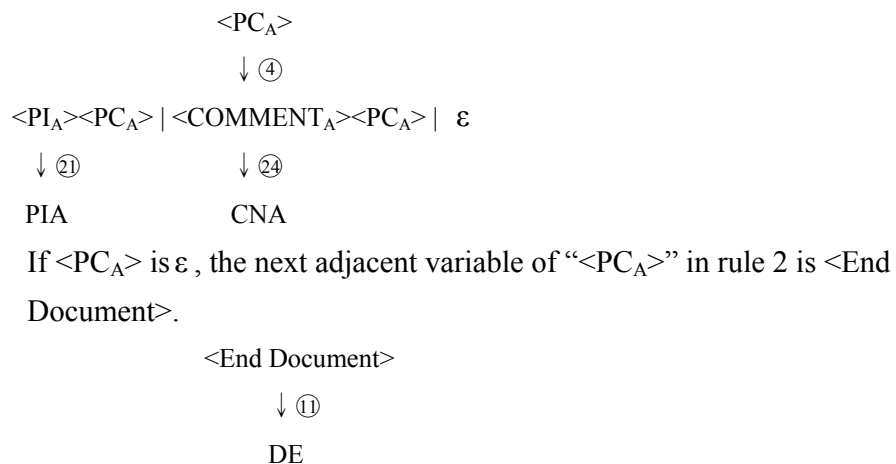


The state of “S<sub>PIB</sub>” can accept alphabets PIB, CNB, STB and EOS.

**The transition of state “S<sub>PIA</sub>” is derived as follows :**

By rule 21, the alphabet PIA is mapping to variable <PI<sub>A</sub>> for state S<sub>PIA</sub>.

By rule 4, the next adjacent variable of <PI<sub>A</sub>> is <PC<sub>A</sub>>.



By rule 5, the “<PI<sub>A</sub>>” is the end of this rule; we want to find the next adjacent variable with <Document fragment>.

By rule 1, the next adjacent variable of “<Document fragment>” is <End>.

<End>  
 ↓ ②⑥  
 EOS

The state of “S<sub>PIA</sub>” can accept alphabets PIA, CNA, DE and EOS.

**The transition of state “S<sub>CNB</sub>” is derived as follows :**

By rule 23, the alphabet CNB is mapping to variable <COMMENT<sub>B</sub>> for state S<sub>CNB</sub>.

By rule 3, the next adjacent variable of <COMMENT<sub>B</sub>> is <PC<sub>B</sub>>.

<PC<sub>B</sub>>  
 ↓ ③  
 <PI<sub>B</sub>><PC<sub>B</sub>> | <COMMENT<sub>B</sub>><PC<sub>B</sub>> | ε  
 ↓ ②⑩                      ↓ ②③  
 PIB                      CNB

If <PC<sub>B</sub>> is ε, the next adjacent variable of “<PC<sub>B</sub>>” in rule 2 is <Element>.

<Element>  
 ↓ ⑥  
 <Beginning of Start Element><Attrs><Ending of Start Element><Content><End  
 Element>  
 ↓ ⑫  
 STB

By rule 5, the “<COMMENT<sub>B</sub>>” is the end of this rule; we want to find the next adjacent variable with <Document fragment>.

By rule 1, the next adjacent variable of “<Document fragment>” is <End>.

<End>  
 ↓ ②⑥  
 EOS

The state of “S<sub>CNB</sub>” can accept alphabets PIB, CNB, STB and EOS.

**The transition of state “S<sub>CNA</sub>” is derived as follows :**

By rule 24, the alphabet CNA is mapping to variable <COMMENT<sub>A</sub>> for state S<sub>CNA</sub>.

By rule 4, the next adjacent variable of <COMMENT<sub>A</sub>> is <PC<sub>A</sub>>.

$\langle PC_A \rangle$   
 $\downarrow \textcircled{4}$   
 $\langle PI_A \rangle \langle PC_A \rangle \mid \langle COMMENT_A \rangle \langle PC_A \rangle \mid \epsilon$   
 $\downarrow \textcircled{21}$                        $\downarrow \textcircled{24}$   
 PIA                      CNA


If  $\langle PC_A \rangle$  is  $\epsilon$ , the next adjacent variable of “ $\langle PC_A \rangle$ ” in rule 2 is  $\langle \text{End Document} \rangle$ .

$\langle \text{End Document} \rangle$   
 $\downarrow \textcircled{11}$   
 DE

By rule 5, the “ $\langle COMMENT_A \rangle$ ” is the end of this rule; we want to find the next adjacent variable with  $\langle \text{Document fragment} \rangle$ .

By rule 1, the next adjacent variable of “ $\langle \text{Document fragment} \rangle$ ” is  $\langle \text{End} \rangle$ .

$\langle \text{End} \rangle$   
 $\downarrow \textcircled{26}$   
 EOS



The state of “ $S_{CNA}$ ” can accept alphabets PIA, CNA, DE and EOS.

**The transition of state “ $S_{DS}$ ” is derived as follows :**

By rule 10, the alphabet DS is mapping to variable  $\langle \text{Start Document} \rangle$  for state  $S_{DS}$ .

By rule 2, the next adjacent variable of “ $\langle \text{Start Document} \rangle$ ” is  $\langle PC_B \rangle$ .

$\langle PC_B \rangle$   
 $\downarrow \textcircled{3}$   
 $\langle PI_B \rangle \langle PC_B \rangle \mid \langle COMMENT_B \rangle \langle PC_B \rangle \mid \epsilon$   
 $\downarrow \textcircled{20}$                        $\downarrow \textcircled{23}$   
 PIB                      CNB

If  $\langle PC_B \rangle$  is  $\epsilon$ , the next adjacent variable of “ $\langle PC_B \rangle$ ” in rule 2 is  $\langle \text{Element} \rangle$ .

$\langle \text{Element} \rangle$   
 $\downarrow \textcircled{6}$   
 $\langle \text{Beginning of Start Element} \rangle \langle \text{Attrs} \rangle \langle \text{Ending of Start Element} \rangle \langle \text{Content} \rangle \langle \text{End Element} \rangle$   
 $\downarrow \textcircled{12}$   
 STB

The state of “ $S_{DS}$ ” can accept alphabets PIB, CNB, and STB.

The transition of state “S<sub>DE</sub>” is derived as follows :

By rule 11, the alphabet DE is mapping to variable <End Document> for state S<sub>DE</sub>.

By rule 2, the “<End Document>” is the end of this rule; we want to find the next adjacent variable with <Document>.

By rule 1, the next adjacent variable of “<Document>” is <End>.

<End>

↓ ②⑥

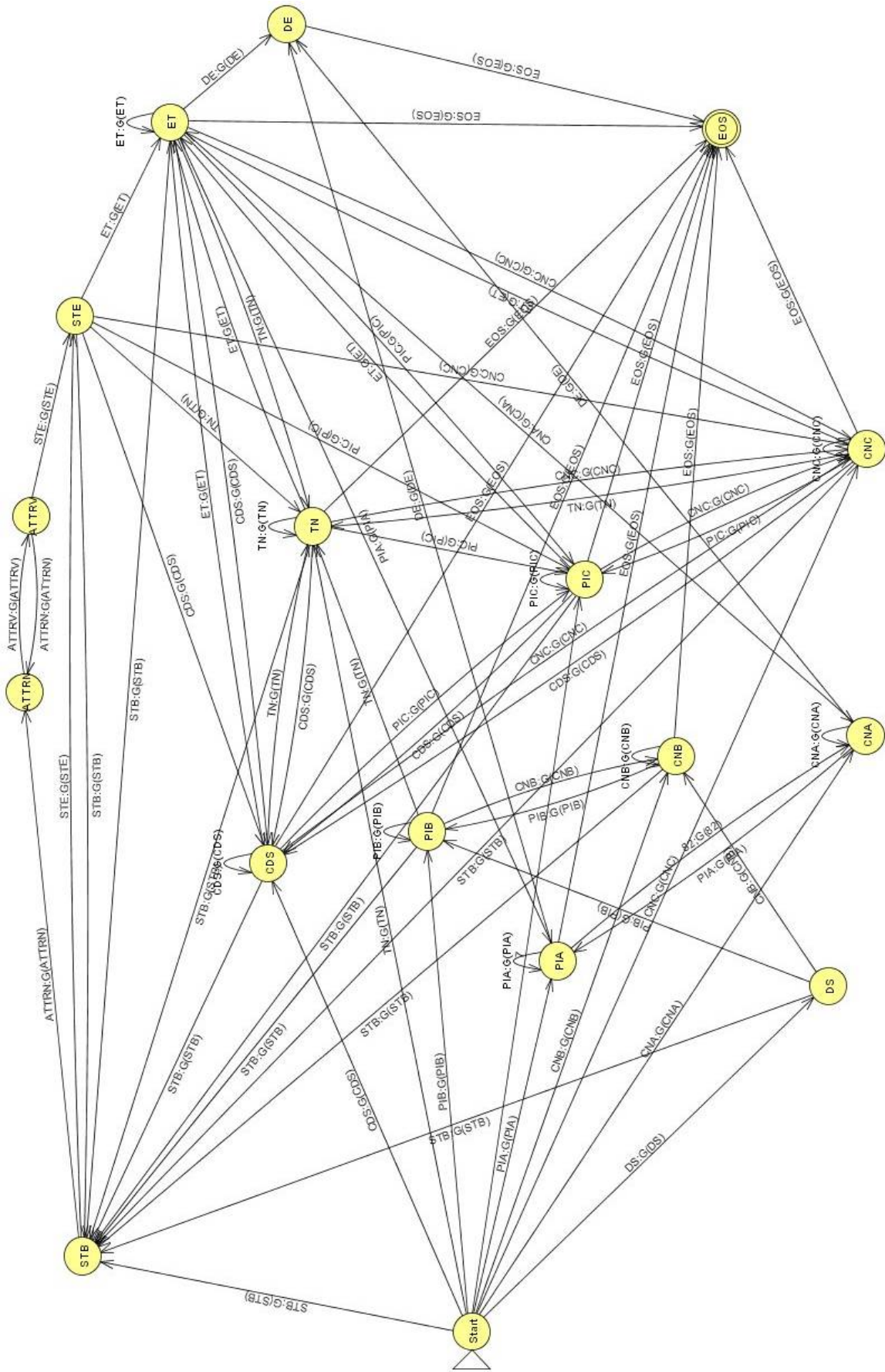
EOS

The state of “S<sub>DE</sub>” can accept EOS.

The S<sub>EOS</sub> is the final state. It cannot accept any alphabet. Finally, we get all the transitions in Table 4-1 and the diagram of finite state transducer in Figure 4-4.

| Input \ State | STB  | STE  | ET  | ATTRN  | ATTRV  | TN  | CDS  | SPIC | SPIB | SPIA | CNC  | SCNB | SCNA | DS  | DE  | EOS  |
|---------------|------|------|-----|--------|--------|-----|------|------|------|------|------|------|------|-----|-----|------|
| Start         | SSTB |      |     |        |        | STN | SCDS | SPIC | SPIB | SPIA | SCNC | SCNB | SCNA | SDS |     |      |
| SSTB          |      | SSTE |     | SATTRN |        |     |      |      |      |      |      |      |      |     |     |      |
| SSTE          | SSTB |      | SET |        |        | STN | SCDS | SPIC |      |      | SCNC |      |      |     |     |      |
| SET           | SSTB |      | SET |        |        | STN | SCDS | SPIC |      | SPIA | SCNC |      | SCNA |     | SDE | SEOS |
| SATTRN        |      |      |     |        | SATTRV |     |      |      |      |      |      |      |      |     |     |      |
| SATTRV        |      | SSTE |     | SATTRN |        |     |      |      |      |      |      |      |      |     |     |      |
| STN           | SSTB |      | SET |        |        | STN | SCDS | SPIC |      |      | SCNC |      |      |     |     | SEOS |
| SCDS          | SSTB |      | SET |        |        | STN | SCDS | SPIC |      |      | SCNC |      |      |     |     | SEOS |
| SPIC          | SSTB |      | SET |        |        | STN | SCDS | SPIC |      |      | SCNC |      |      |     |     | SEOS |
| SPIB          | SSTB |      |     |        |        |     |      |      | SPIB |      |      | SCNB |      |     |     | SEOS |
| SPIA          |      |      |     |        |        |     |      |      |      | SPIA |      |      | SCNA |     | SDE | SEOS |
| SCNC          | SSTB |      | SET |        |        | STN | SCDS | SPIC |      |      | SCNC |      |      |     |     | SEOS |
| SCNB          | SSTB |      |     |        |        |     |      |      | SPIB |      |      | SCNB |      |     |     | SEOS |
| SCNA          |      |      |     |        |        |     |      |      |      | SPIA |      |      | SCNA |     | SDE | SEOS |
| SDS           | SSTB |      |     |        |        |     |      |      | SPIB |      |      | SCNB |      |     |     |      |
| SDE           |      |      |     |        |        |     |      |      |      |      |      |      |      |     |     | SEOS |
| SEOS          |      |      |     |        |        |     |      |      |      |      |      |      |      |     |     |      |

Table 4-1 Transition Table of CFA



**Figure 4-4 Diagram of CFA**

## 4.4. Output Function for Generate Canonical Form

In this Section, we will describe the output function call defined in Section 4.3.

$$\Gamma = \{ G(A) \mid \text{for all } A \in \Sigma \}$$

In Section 4.2, the Canonical XML is converted into two sequences of structure identifier and its vocabulary. The vocabulary sequence is the input of function G. The function G may read a vocabulary from vocabulary sequence to generate its canonical form for each structure. All function call in  $\Gamma$  will be defined as follows.

Each function may be call the function ReadV() to generate Canonical XML. The function ReadV() is define to read one vocabulary from vocabulary sequence.

**G(DS)**{ do nothing }

**G(STB)** {  
     print '<';  
     print ReadV();  
   }

**G(STE)**{  
     print '>';  
   }

**G(ET)**{  
     print ReadV();  
     print "</";  
     print '>';  
   }

**G(ATTRN)**{  
     print 0x20; //a space  
     print ReadV();  
     print '=';  
   }

**G(ATTRV)**{  
     print "" ;  
     Print ReadV();  
     Print "" ;  
   }

**G(TN)**{  
     print ReadV();  
   }

**G(CDS)** {  
     print ReadV ;  
   }

**G(PIC)**{  
     print "<?";  
     print ReadV() ;  
     print ">";  
   }

**G(PIB)**{  
     print "<?";  
     print ReadV() ;  
     print ">";  
     print '0xA';  
   }

**G(PIA)**{ print '0xA';  
     print "<?";  
     print **ReadV()** ;  
     print ">";  
   }

**G(CNC)** {  
     print "<!--";  
     print ReadV();  
     print "-->";  
   }

**G(CNB)**{  
     print "<!--";  
     print ReadV();  
     print "-->";  
     print '0xA' ;  
   }

**G(CNA)**{ print '0xA' ;  
     print "<!--";  
     print ReadV();  
     print "-->";  
   }

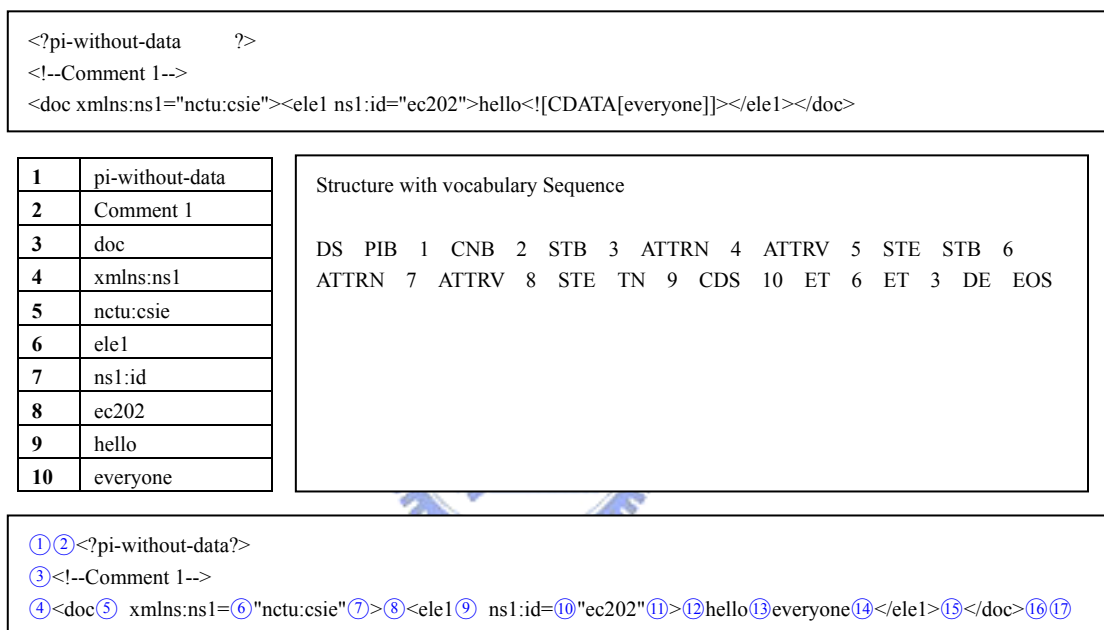
**G(DE)**{  
     do nothing  
   }

**G(EOS)** {do nothing}



## 4.5. Example for CFA

In Figure 4-5, a simple xml document is converted into two sequences of structure identifier and vocabulary. For implementation, we also convert vocabulary sequence into vocabulary table and its index sequence. Then we can combine structure sequence with vocabulary sequence. The vocabulary index follows structure alphabet. The process to generate canonical xml from this sequence by CFA will be illustrated in the following.



**Figure 4-5 Example of CFA**

Step1: We start with the Start state and read input DS. The state moves to  $S_{DS}$  from Start.

This step does not generate any output.

Step2: To read input PIB, the state moves to  $S_{PIB}$  from  $S_{DS}$ . The engine of CFA reads vocabulary index 1 and converts index 1 to real vocabulary “pi-without-data”. This step generates output “<?pi-without-data?>” before a newline.

Step3: To read input CNB, the state moves to  $S_{CNB}$  from  $S_{PIB}$ . The engine of CFA reads vocabulary index 2 and converts index 2 to real vocabulary “Comment 1”. This step generates output “<!--Comment 1-->” before a newline.

Step4: To read input STB, the state moves to  $S_{CNB}$  from  $S_{STB}$ . The engine of CFA reads vocabulary index 3 and converts index 3 to real vocabulary “doc”. This step generates output “<doc”.

Step5: To read input ATTRN, the state moves to  $S_{ATTRN}$  from  $S_{STB}$ . The engine of CFA reads vocabulary index 4 and converts index 4 to real vocabulary “xmlns:ns1”. This step generates output “xmlns:ns1=” after a space.

Step6: To read input ATTRV, the state moves to  $S_{ATTRV}$  from  $S_{ATTRN}$ . The engine of CFA reads vocabulary index 5 and converts index 5 to real vocabulary “nctu:csie”. This step generates output “nctu:csie”.

Step7: To read input STE, the state moves to  $S_{STE}$  from  $S_{ATTRV}$ . This step generates output “>”.

Step8: To read input STB, the step moves to  $S_{STB}$  from  $S_{STE}$ . The engine of CFA reads vocabulary index 6 and converts index 6 to real vocabulary “ele1”. This step generates output “<ele1”.

Step9: To read input ATTRN, the state moves to  $S_{ATTRN}$  from  $S_{STB}$ . The engine of CFA reads vocabulary index 7 and converts index 7 to real vocabulary “ns1:id”. This step generates output “ns1:id=” after a space.

Step10: To read input ATTRV, the state moves to  $S_{ATTRV}$  from  $S_{ATTRN}$ . The engine of CFA reads vocabulary index 8 and converts index 8 to real vocabulary “ec202”. This step generates output " ec202".

Step11: To read input STE, the state moves to  $S_{STE}$  from  $S_{ATTRV}$ . This step generates output “>”.

Step12: To read input TN, the state move to  $S_{TN}$  from  $S_{ATTRV}$ . The engine of CFA reads vocabulary index 9 and converts index 9 to real vocabulary “hello”. This step generates output “hello”.

Step13: To read input CDS, the state moves to  $S_{CDS}$  from  $S_{TN}$ . The engine of CFA reads

vocabulary index 10 and converts index 10 to real vocabulary “everyone”. This step generates output “everyone”.

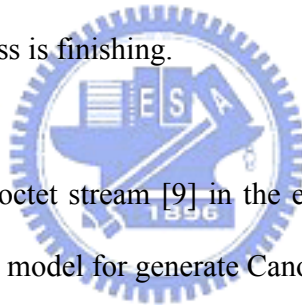
Step14: To read input ET, the state moves to  $S_{ET}$  from  $S_{CDS}$ . The engine of CFA reads vocabulary index 6 and converts index 6 to real vocabulary “ele1”. This step generates output “</ele1>”.

Step15: To read input ET, the state moves to  $S_{ET}$  from  $S_{ET}$ . The engine of CFA reads vocabulary index 3 and converts index 3 to real vocabulary “doc”. This step generates output “</doc>”.

Step16: To read input DE, the state moves state to  $S_{DE}$  from  $S_{ET}$ . This step does not generate any output.

Step17: To read input EOS, the state moves to  $S_{EOS}$  from  $S_{DE}$ . The engine of CFA reach final state and the process is finishing.

We get a Canonical XML octet stream [9] in the end of the process with the sequence. In addition, we also get a streaming model for generate Canonical XML.

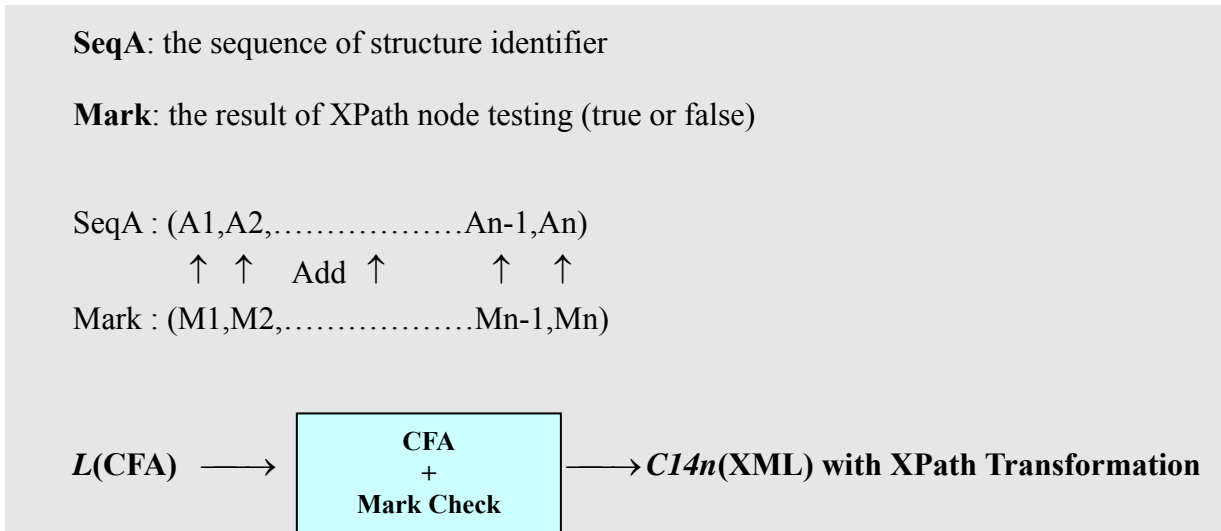


#### **4.6. Support for XPath Transformation**

For XPath transformation [8] [12], we must test each node in DOM to decide which node satisfy the XPath [12] expression. Finally, we can get a node set for this transformation. The node set is a partial of full document or document fragment. The serialization of the node set is not a valid xml, so we cannot parse it again. If we want to keep canonical form of original xml and hold down the XPath transformation result in this transformation model, the engine of CFA must decides which node is including in the node set after XPath transformation. For this purpose, the operations of canonicalization and XPath node testing will be combined in a process. XPath node testing is executed to check which node satisfies XPath expression before node formalize. To mark the structure identifier is an efficient method to identify which node is in the set after XPath

transformation. For implementation, the structure identifiers are all defined less than 128. Thus, the last bit of a byte can be used to mark the node for XPath transformation.

The engine of CFA only needs to check the mark with each structure identifier in sequence for XPath transformation. The complexity of each operation in state is not increasing. The process of CFA with XPath filtering is showed as follows.



#### 4.7. Complexity Analysis for CFA

In this section, we will analysis the complexity of CFA. The automata accept the sequence of structure identifier is always in linear time. Therefore, the complexity of output function call  $G(A)$  defined in Section 4.4 is the only factor for total complexity. The function calls  $G(A)$  define in Section 4.4 is just printing something. The complexity of each function call  $G(A)$  is  $O(1)$ . For the sequence of CFA, there are at most  $n$  function calls  $G(A)$ . The complexity of CFA is  $O(n)$ . If the XPath transformation is added to CFA, the complexity is not change. Because it only escape some structure identifier in sequence for function call by mark describe in Section 4.6. The sequence of function call is less then without XPath transformation. The complexity is still  $O(n)$ . In any cases, the CFA guaranteed that the complexity is  $O(n)$ . This analysis is showed as follows.

$S_n : (X_1, X_2, X_3 \dots X_{n-1}, X_n)$  is the sequence of structure identifier.

$$(X_1, X_2, X_3 \dots X_{n-1}, X_n) \xrightarrow{CFA} (G(X_1), G(X_2), G(X_3) \dots G(X_{n-1}), G(X_n))$$

There are  $n$   $G(A)$ .

Each  $G(A)$  function call is just print something. The complexity of  $G(A)$  is  $O(1)$ .

$$n \times O(1) \Rightarrow O(n)$$



## 5. Performance Analysis

The proposed scheme CFA guarantees that the complexity is always  $O(n)$  for any cases with Canonical XML and XPath transformation. To prove this improvement, the performance will be compared with original C14n scheme.

For performance testing, we make two test cases to show the difference of whether XPath transformation is enabled. Because XPath transformation is very complex, the huge performance gap is expected after XPath transformation enables. Therefore, the case1 is not including XPath transformation, and case2 is XPath transformation enabled. The testing document a small xml with all node type in C14n, and then clone the child nodes of the root element to expand size for samples. Therefore, the node number of test document is proportional to its size. The sample of xml document for performance testing is showed in **Appendix II**.

The C14n engine of Apache XML Security 1.4.1[21] is the implementation of original C14n scheme. The engine of CFA is also developing with the same language and platform from Apache XML Security 1.4.1. We also use the same I/O package in Apache XML Security 1.4.1 to serialize data. Thus, the performance issue only depends on algorithm.

The configuration of testing is showed in Table 5-1.

|  | Experimental Host         |
|--|---------------------------|
| CPU  | AMD Athlon XP 1700+       |
| RAM  | 1 GB                      |
| OS   | Windows XP                |
| JVM  | JDK1.4.2                  |
| Control Group<br>(Original C14n<br>scheme) | Apache XML Security 1.4.1 |

**Table 5-1 testing environment**

For XPath transformation testing, we design three XPath predicate as follows to filter nodes. The complexity of each XPath predicate is different, but the result set of nodes is equivalent.

※ the node is text

C14N-1 : self::text()

※ the node is text or the parent of this node is context node and it has xxx children

C14N-2 : self::text() | parent::node()/child::xxx

※the node is text or the node has bbb ancestor and its ancestor bbb also has xxx children

C14N-3 : self::text() | ancestor::bbb/descendant-or-self::node()/child::vvv

These predicates are sorted by complexity as follows.

C14N-1 < C14N-2 << C14N-3

The detail of testing result is showed in Table 5-2. The node number of each sample is also presented in Table 5-2. We can find the node number of test document is proportional to its size. In Table 5-2(a), it presents the processing time of original C14n scheme and CFA without XPath transformation. In Table5-2(b), it presents the processing time apply the XPath transformation in C14n and CFA with different XPath predicate. This table also presents the Input/output sequence length of CFA. The result set of nodes is same for all three XPath predicates. Therefore, the output sequence length is not change for these predicates. Because the time of CFA only depends on output sequence length after XPath transformation, the process time is also not change for different predicates. Figure5-1(a) is showed the performance comparison without XPath transformation. Figure5-1(b) is showed the performance comparison with XPath transformation. In Figure5-1(a), the CFA always 60 times faster than Apache XML Security. In Figure5-1(b), the time of Apache XML Security is increasing very huge following document size and XPath predicate complexity. However, the time of CFA is not increasing any more, on the contrary the

time is decreasing. Because the complexity of CFA is not change for XPath transformation [8] [12], but the sequence length of output is decreasing after XPath filtering. For 1MB document sample in Figure5-1(b), the CFA 6000 times faster than case C14N-1. Cases C14N-2 and C14N-3 is more complex than C14N-1, therefore the performance gap is bigger than C14N-1. CFA won the performance testing for all cases.

**non-XPath transformation**

| size(KB) / nodes \ Time(ms) | Apache XML Security | CFA |
|-----------------------------|---------------------|-----|
| 100 / 8660                  | 421                 | 7   |
| 200 / 17300                 | 562                 | 9   |
| 300 / 25780                 | 657                 | 12  |
| 400 / 43060                 | 860                 | 15  |
| 500 / 51700                 | 1156                | 20  |
| 600 / 43060                 | 1266                | 23  |
| 700 / 60180                 | 1531                | 25  |
| 800 / 68820                 | 1734                | 29  |
| 900 / 77460                 | 2046                | 32  |
| 1000 / 85940                | 2406                | 37  |

**(a)**

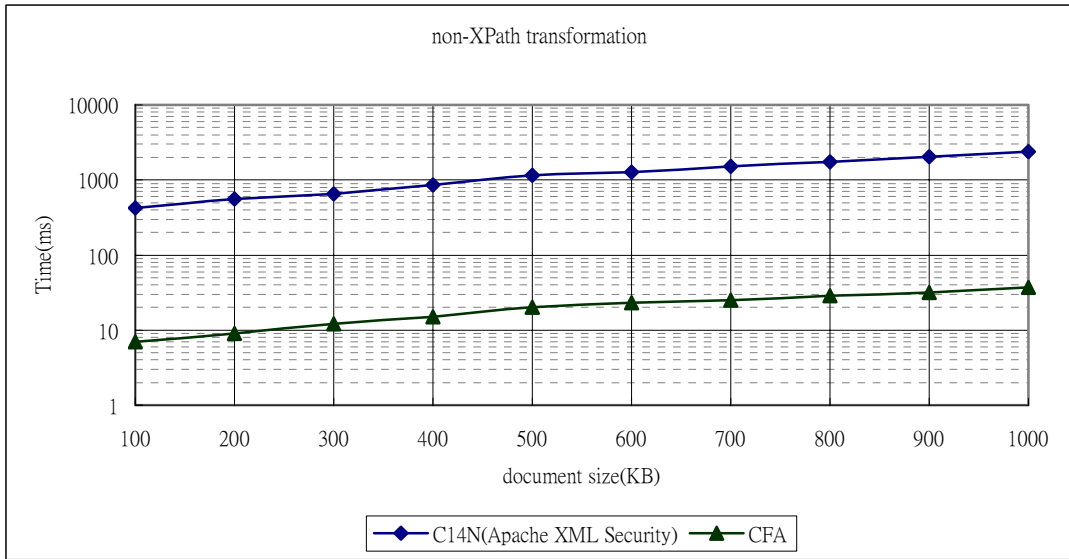
**XPath transformation**

| size(KB) / nodes \ Time(ms) | Apache XML Security C14N-1 | Apache XML Security C14N-2 | Apache XML Security C14N-3 | CFA | CFA Input/Output sequence length |
|-----------------------------|----------------------------|----------------------------|----------------------------|-----|----------------------------------|
| 100 / 8660                  | 2046                       | 2517                       | 67766                      | 4   | 15803/3406                       |
| 200 / 17300                 | 5828                       | 7969                       | 258516                     | 6   | 31571/6808                       |
| 300 / 25780                 | 11797                      | 17469                      | 572938                     | 7   | 47047/10147                      |
| 400 / 43060                 | 20312                      | 31156                      | 1042500                    | 7   | 62815/13549                      |
| 500 / 51700                 | 30766                      | 47984                      | 1499516                    | 10  | 78583/16951                      |
| 600 / 43060                 | 43796                      | 68265                      | 2121906                    | 14  | 94351/20353                      |
| 700 / 60180                 | 59813                      | 94484                      | 2843907                    | 15  | 109827/23692                     |
| 800 / 68820                 | 78438                      | 121312                     | 3751828                    | 15  | 125595/27094                     |
| 900 / 77460                 | 98375                      | 151031                     | 4759438                    | 15  | 141363/30496                     |
| 1000 / 85940                | 121438                     | 187421                     | 6627812                    | 18  | 156839/33835                     |

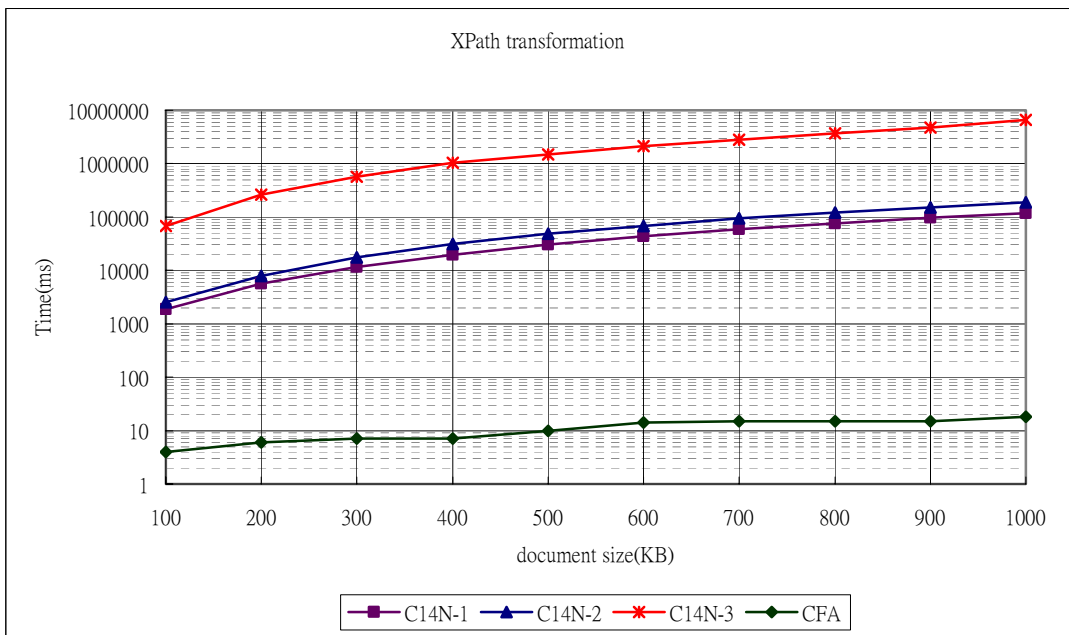
**(b)**

**Table 5-2 results of performance testing**





**Figure 5-1 (a) Comparison without XPath transformation**



**Figure 5-1(b) Comparison with XPath transformation**

## 6. Conclusion

XML is common use in computer and internet application. To protect application security, the XML security is designed for these purposes. XML Signature is the core technology in XML Security, but the performance issue is always the big problem. In proposed scheme, Canonical XML is converted into a new language of structure and its vocabulary. Then the finite automata CFA is constructed to process this language. This finite automata generates Canonical XML in linear time. This scheme provides a method to accelerate the XML Signature verification, and we can reduce the effect of XML Signature performance issue to minimize. Some device with limited-resource or low computing capability such as firewall or mobile device may be support XML Security but does not affect its performance by this scheme. W3C create a project to research efficient XML [16], but does not include XML Security. This scheme improves not only the performance of XML Signature but also exchanges XML. Canonical XML is same with original XML in logically. If we can improve Canonical XML processing, XML exchanging is also improved. This research may be providing a solution to improve performance issue for XML and XML Security concurrently.

# Appendix I The Algorithm of C14n Conversion

The sequences A and V are the transformation result. C14n-TO-CFA procedure converts Canonical XML to CFA.

A: structure identifier sequence

V: vocabulary sequence

```
C14n-TO-CFA(Node)
1  if Node.type = Document
2      then    add document start identifier(DS) into A
3              child ← getFirstChild(Node)
4              while child ≠ NIL
5                  do C14n-TO-CFA(child)
6                  child ← getNextSibling(child)
7              add document end identifier(DE) into A
8  if Node.type=Element
9      then    name ← getNodeName(Node)
10             add start tag beginning identifier(STB) into A
11             add name into V
12             attributes ← getAllAttributes(Node)
13             SortingAndProcess(attributes)
14             for i←0 to length(attributes)
15                 do C14n-TO-CFA(attributes[i])
16             add start tag ending identifier(STE) into A
17             child ← getFirstChild(Node)
18             while child ≠ NIL
19                 do C14n-TO-CFA(child)
20                 child ← getNextSibling(child)
21             add end tag identifier(ET) into A
22             add name into V
23  if Node.type=Attribute
24      then    name ← getNodeName()
25             Add attribute name identifier(ATRN) into A
26             Add name into V
27             attributeValue ← getNodeValue(Node)
```

```

28         attributeValue ← formalizeAttribute(attributeValue)
29         add attribute value identifier(ATTRV) into A
30         add attributeValue into V
31     if Node.type = Comment
32     then location ← SearchLocation(Node)
33         if location=BeforeRootElement
34         then add CNB into A
35             CommentValue ← getCommentValue(Node)
36             CommentValue ←FormalizeComment(CommentValue)
37             Add CommnetValue into V
38         if location=AfterRootElement
39         then add CNA into A
40             CommentValue ← getCommentValue(Node)
41             CommentValue ←FormalizeComment(CommentValue)
42             Add CommnetValue into V
43         if location=InnerRootElement
44         then add CNC into A
45             CommentValue ← getCommentValue(Node)
46             CommentValue ←FormalizeComment(CommentValue)
47             Add CommnetValue into V
48     If Node.type = PI
49     then location ← SearchLocation(Node)
50         if location=BeforeRootElement
51         then add PIB into A
52             PICContent ← getPICContent(Node)
53             PICContent ←FormalizePI(PICContent)
54             Add PICContent into V
55         if location=AfterRootElement
56         then add PIA into A
57             PICContent ← getPICContent(Node)
58             PICContent ←FormalizePI(PICContent)
59             Add PICContent into V
60         if location=InnerRootElement
61         then add PIC into A
62             PICContent ← getPICContent(Node)
63             PICContent ←FormalizePI(PICContent)
64             Add PICContent into V
65     if node.type=Text
66     then textValue ← getTextValue(Node)

```

```
67         textValue ←FormalizeText(textvalue)
68         add Text identifier(TN) into A
69         add textValue into V
70 if Node.type=CDATASection
71     then CDSectionValue ← getCDATASectionValue(Node)
72         CDSectionValue ←FormalizeCDataSection(CDSectionValue)
73         add CDATASection identifier(CDS) into A
74         add CDSectionValue into
```



# Appendix II The Sample of XML Document For Performance Testing

```
<?xml version='1.0' ?>
<!DOCTYPE doc [
<!ATTLIST e9 attr CDATA "default">
<!ATTLIST normId id ID #IMPLIED>
<!ATTLIST normNames attr NMTOKENS #IMPLIED>
<!ATTLIST e2 xml:space (default|preserve) 'preserve'>
<!ATTLIST e3 id ID #IMPLIED>
<!ATTLIST e9 attr CDATA "default">
]>
<?xml-stylesheet href="doc.xsl"
type="text/xsl" ?>
<!-- Hello every one -->
<aaa><kkk><NNN/></kkk>
<!-- Inner --><?pi-without-data ?>
<bbb id="xxx" xml:lang="xxx" xmlns="aaa:bbb:ccc" xmlns:aaa="xxx:aaa" xmlns:xxx="bbb:ccc">
<doc attr='&#xA;'>&#xD;&#xA;Hello, world<!-- Comment 1 --><![CDATA[value>"0" &&
value<"10" ?"valid":"error"]]></doc>

<!-- Comment 2 -->

<!-- Comment 3 -->

<doc>
  <clean> </clean>
  <dirty> A B </dirty>
  <mixed>
    A
    <clean> </clean>
    B
    <dirty> A B </dirty>
    C
  </mixed>
</doc>

<doc>
  <e1 />
  <e2 ></e2>
  <e3 name = "elem3" id="elem3" />
  <e4 name="elem4" id="elem4" ></e4>
  <e5 a:attr="out" b:attr="sorted" attr2="all" attr="I'm"
xmlns:b="http://www.ietf.org"
xmlns:a="http://www.w3.org"
xmlns="http://www.uvic.ca"/>
```

```

<e6 xmlns="" xmlns:a="http://www.w3.org" xml:lang="en">
  <e7 xmlns="http://www.ietf.org">
    <e8 xmlns="" xmlns:a="http://www.w3.org">
      <e9 xmlns="" xmlns:a="http://www.ietf.org"/>
    </e8>
  </e7>
</e6>
</doc>

<doc>
  <text>First line&#xD;Second line</text>
  <value>&#x32;</value>
  <compute></compute>
  <compute expr='value>"0" &amp;&amp; value&lt;"10" ?"valid":"error">valid</compute>
  <norm attr=' &apos;   &#x20;&#xD;&#xa;&#9;   &apos; '/>
  <normNames attr='  A   &#x20;&#13;&#xa;&#9;  B  '/>
  <normId id=' &apos;   &#x20;&#13;&#xa;&#9;   &apos; '/>
</doc>

<doc xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org">
  <e1>
    <e2 xmlns="">
      <e3 id="E3"/>
    </e2>
  </e1>
</doc>

<doc>
<e1/>
<e2></e2>
<e3 xmlns="http://www.kimo.com" xmlns:n1="http://xxx.tw" name="elem3" id="elem3" />
<e4 xmlns:n1="http://xxx.tw" name="elem4" id="elem4"></e4>
<e5 a:attr="out" b:attr="sorted" attr2="all" attr="I'm" xmlns:b="http://www.ietf.org"
  xmlns:a="http://www.w3.org" xmlns="http://example.org"/>
<e6 xmlns="" xmlns:a="http://www.w3.org">
<e7 xmlns="http://www.ietf.org">
<e8 xmlns="" xmlns:a="http://www.w3.org">
<e9 xmlns="" xmlns:a="http://www.ietf.org"/>
</e8>
</e7>
</e6>
</doc>
</bbb>
</aaa>
<?pi-without-data      ?><!-- The End -->

```

## References

- [1] W. Lu, K. Chiu, A. Slominski, and D. Gannon. A streaming validation model for SOAP digital signature. In 14th IEEE Symposium on High Performance Distributed Computing, pages 243–252, July 2005.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp.540–549.
- [3] Michael Sipser. Introduction to the theory of computation, Second Edition. Thomson Course Technology, 2006. ISBN 0-619-21764-2.
- [4] Kangasharju, Jaakko. Efficient Implementation of XML Security for Mobile Devices. In Web Services, 2007. ICWS 2007. IEEE International Conference, pages 134 - 141, July 2007.
- [5] Mehryar Mohri, Finite-state transducers in language and speech processing, Computational Linguistics, v.23 n.2, p.269-311, June 1997
- [6] Todd J. Green , Ashish Gupta , Gerome Miklau , Makoto Onizuka , Dan Suciu, Processing XML streams with deterministic automata and stream indexes, ACM Transactions on Database Systems (TODS), v.29 n.4, December 2004
- [7] Tim Bray, Dave Hollander, Andrew Layman and Richard Tobin. Namespaces in XML (Second Edition), W3C Recommendation. eds. 16 August 2006.
- [8] M. Bartel, J. Boyer, B. Fox, B. LaMacchia and E. Simon. XML-Signature Syntax and Processing. W3C Recommendation, 12 February 2002.
- [9] J. Boyer. Canonical XML 1.0. W3C Recommendation. March 2001.
- [10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation. eds. 16 August 2006.



- [11] A.L. Hors,P.L. Hegaret,L. Wood,G. Nicol,J. Robie,M. Champion and S. Byrne. Document Object Model (DOM) Level 3 Specification. W3C Recommendation., 7 April 2004.
- [12] J. Clark, S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation. October 1999.
- [13] P. Hoffman and F. Yergeau. UTF-16, an encoding of ISO 10646. rfc2781, February 2000.
- [14] FIPS PUB 186-2 . *Digital Signature Standard (DSS)*. U.S. Department of Commerce/National Institute of Standards and Technology, 27 January 2000.
- [15] F. Yergeau. *UTF-8, a transformation format of ISO 10646*. RFC 2279, January 1998.
- [16] G. White,J. Kangasharju,D. Brutzman,S. Williams. Efficient XML Interchange Measurements Note. W3C Working Draft, 25 July 2007.
- [17] SAX: The Simple API for XML. D. Megginson, et al. May 1998.  
<http://www.megginson.com/SAX/index.html>
- [18] OASIS. ebXML Message Service Specification 2.0. OASIS ebXML Messaging Services Technical Committee, 2002.
- [19] W3C. Standard Generalized Markup Language. <http://www.w3.org/MarkUp/SGML/>,1995
- [20] W3C. Web Services. <http://www.w3.org/2002/ws/>,2002
- [21] Apache. Apache XML Security.<http://santuario.apache.org/Java/index.html>, 2007
- [22] Wen-Hsiang Tsai. Formal Languages and Theory of Computation. Department of Computer Science/National Chiao TungUniversity.  
[http://www.cis.nctu.edu.tw/~whtsai/Course%20Teaching%20Affairs/Formal%20Languages/Chapter%200/Chapter\\_0\\_Introduction.ppt](http://www.cis.nctu.edu.tw/~whtsai/Course%20Teaching%20Affairs/Formal%20Languages/Chapter%200/Chapter_0_Introduction.ppt), pp 13-16