私立東海大學資訊工程與科學研究所

碩士論文

指導教授：周忠信

XML 文件的內容同等轉換

Content Equivalent Transformation in XML Document

研究生：黃冠綸

中華民國九十三年六月二十八日

摘要

近年來，XML 技術被大量應用於軟體系統發展上。許多 XML 相關的研究課題不斷的被提出，而其中有關 XML 文件間相似或相同的判斷問題，對於 XML 技術應用於內容管理或資訊安全等領域上乃至爲重要。此類問題的關鍵基礎，在於先能分辨何謂二份 XML 文件相等。過去的研究重點較偏向於 XML 文件 DOM tree 的結構間是否相似，至於 XML 文件的內容與內容間關聯則鮮少提及。本論文旨在提出能夠說明 XML 文件內容與內容間關聯相等的新概念，我們將之稱爲「XML 文件內容同等」—意即即使兩份 XML 文件的樹狀結構不同，但其內容與內容之間的關聯性卻相同。爲說明 XML 文件間之內容同等確實存在，我們特別介紹三個 XML 文件的轉換機制，它們分別爲更名轉換、合併轉換以及分割轉換。透過運用這三個轉換，我們可以維持 XML 文件的內容同等。本論文最後透過一個範例，來展現三個內容同等轉換的效果。

關鍵字：XML、內容同等、內容同等轉換、轉換資訊

Abstract

Recently, XML technology has been widely adopted by the modern software and XML related issues have become major research directions in many fields. Among these issues, how to detect the similarity between two XML documents is very important for content management and information security control. The key point to detect the similarity among XML documents is to define the equivalence concept of XML documents first. Previous solution was based on the comparison among the DOM tree structures of the XML documents. However, the content and the content relationship of two different DOM tree structure XML documents might be the same. We say that the documents with the above property are *content equivalent*. Based on this new content equivalent definition, we further propose three content equivalent transformations for XML documents. These three transformations are *Rename* transformation, *Merge* transformation and *Split* transformation.

Keywords：XML、content equivalent、content equivalent transformation、transformation information

目次

圖目錄

第一章 緒論

1.1 研究動機與目的

近年來，隨著 XML 技術被大量應用於網路服務、軟體系統發展、電子商務以及內容管理等領域上[6,7,12,13,14]，許多與 XML 相關的研究課題乃不斷被提出與應用。而其中有關 XML 文件間相似或相同的判斷問題，對於 XML 技術應用於內容管理或資訊安全等領域上乃至爲重要。舉例來說，一些至關重要的 XML 文件，很可能被修改後經由電子郵件而洩漏出去。此時判斷機密文件與電子郵件傳送出去的文件間是否相似，可作爲決定放行與否的準則。此類判斷二份 XML 文件是否相等、相似或差異的課題，統稱爲「XML 相似度」計算問題[4,8,9,11]。探討此類問題的關鍵基礎，首先在於定義二份 XML 文件間相等的意義。目前常見做法是將 XML 文件轉換成其對應之文件樹狀結構(*DOM tree*)[5]，再以此樹狀結構分辨兩份 XML 文件間相同與否。譬如在許多文獻中乃利用 tree to tree editing[10]的延伸做法[1,2,3,4,8,9.11]，透過對樹狀節點的新增、修改與刪除，將一顆 DOM tree 轉換成同構(*isomorphism*)的 DOM tree 來定義相等。然而利用此類技術來判斷 XML 文件是否相等卻有其盲點。現實上可存在某些 XML 文件，雖然 DOM tree 結構不同，但是其內容與內容之間的對應關聯，卻可以利用例如像 XML 的屬性(attributes)等方式來記錄輔助資訊，進而使得兩份 XML 文件保持相同。

有鑑於此，本論文目的乃在於提出一種新的 XML 文件相等概念。這個概念希望能夠分辨，儘管 XML 文件之間的樹狀結構不同，但是其內容以及內容間關聯性卻可能相同，而此種相等關係我們稱之爲「內容同等」(*content equivalent*)。爲了能夠說明 XML 文件內容同等的概念，我們先定義 XML 文件的「內容同等轉換」(*content equivalent transformation*)機制。這些機制可以改變 XML 文件的樹狀結構，但是透過保留文件內容與內容之間的關聯性資訊，進而確保轉換後的 XML 文件內容與內容之間的關聯不變。

1.2 論文組織架構

本論文第二章將介紹可擴展標記語言、文件型態定義與文件物件模型等

XML 相關技術，並介紹本論文中所使用之 XML 元件樹。在第三章中，說明 XML 文件同等轉換之概念，介紹更名、合併與分割三個轉換操作，並舉例及證明它們符合文件同等轉換之定義，最後說明混用轉換操作時的限制。第四章爲實作三個轉換操作，並作出一個文件同等轉換之編輯器。再以編輯器實際轉換一 XML 範例作爲說明。第五章爲本論文結論，同時探討文件同等轉換的優缺點，最後則介紹未來研究方向。

第二章 XML 相關背景知識

2.1 可擴展標示語言 (XML)

「可擴展標示語言」(XML)[7]是用來標示具有結構性電子文件資料的一種語言。透過 XML，電子文件具有結構性，不僅可讓軟體解析文件，人類也可以透過文件中的標示來瞭解文件資料的涵義。XML 的格式類似網際網路上「超鏈結標示語言」(*HTML*) ，但是 XML 可以自行定義標籤及文件結構。XML 是由「全球資訊網協會」(*World Wide Web Consortium*，*W3C*)所制定，在 1998 年 2 月成爲正式的標準規範。

XML 是「標準通用標記語言」(*SGML*)的子集合，而非一種 SGML 的應用。XML 繼承 SGML 的可擴展性，但 XML 大量簡化了 SGML 複雜的語法規定，所以 XML 是以 SGML 的格式精簡後所制定。主要目的在於彌補現今 HTML 的不足，以及讓 SGML 能像現有的 HTML 一樣，容易地在網路上使用並擴充其網路應用。

XML 與 HTML 設計的目標並不一樣：HTML 是爲網頁設計而制定，著重在資料的展現；XML 目標在於提供一個具有彈性與擴展性的標示語言，使其能廣泛應用在網際網路上，並著重於如何將文件結構化以方便資料的管理與交換。目前 W3C 將以新一代的 XHTML 來取代 HTML，XHTML 是 HTML 以 XML 規範爲基礎重新制定出的網頁語言新標準。

XML 不只是一個標示語言，也是一個用來定義其他語言的「元語言」(*meta-language*)。元語言可以用來定義及產生另一種新的語言。如 HTML 是採用 SGML 規範所制定出來的一種語言，所以 SGML 即可稱爲元語言。目前有許多的語言是基於 XML 規範所定義。比如說「資源描述架構」(*RDF*) 是用來描述資源資訊的一種語言以及商業上使用的 *ebXML* [6]等。此外尚有許多他種 XML 定義出的語言在網際網路上被廣泛應用，相關資料可在參考 W3C 網站。

爲何XML能被應用於各個領域？我們以下列九點來說明其特性與優點：

1. XML 具有自我描述(*self-describing*)能力：圖 2.1 是一份簡單的 XML 文件範例，由文件中的標籤(*tag*)及屬性(*attribute*)與實際的文字內容(*text*)，可以輕易瞭解此份文件所紀錄之訊息與資料。此範例記載一本書的資訊，包括書名、作者、分類與簡單的介紹。與資料庫不同，從資料庫的內文資料無法知道此項資料記錄的資訊意義。

```
<?xml version=”1.0”encoding=”UTF-8”>
<LIBRARY>
    <BOOK CLASS="JAVA">
     <TITLE> Java Servlet </TITLE>
     <AUTHOR> Jason Hunter </AUTHOR>
     <INTRO>
         This is a java servlet handbook.
     </INTRO>
    </BOOK>
</LIBRARY>
```

圖 2.1. 簡單的 XML 文件範例

2. XML 是一個標準：雖然 XML 是由 W3C 所訂制，但實際上是由許多軟體市場領導者的專家參與合作而成。因此 XML 獲得許多軟體大廠的支持，廣泛被應用在各領域。
3. 國際化：XML 支援多種文件編碼，包括 Unicode 字集。因此 XML 檔中可以支援與使用所有語言。
4. 可攜性：由於 XML 具有極佳的自我描述能力，XML 文件內容乃可以被容易地被轉換成另一種文件，因此 XML 成爲資料交換的一項主流技術。
5. 簡單：XML 以文字來呈現，該有的資訊皆包含在文件內，不需再取得其他資訊即可明白其內容。撰寫 XML 甚至比撰寫 HTML 還要容易，因爲 XML 本身只需記載資料以及資料之結構，而 HTML 除了前二者外還必須兼顧到資料內容的展現方式。
6. 可擴展性： XML 是一種元語言，並沒有固定的標籤，當需要新的標籤時，只需新增定義即可。
7. 內容(*Content*)、結構(*Structure*)與呈現方式(*View*)分離：XML 文件的標示可用來描述內容與結構，但並不用來定義資料的展現方法。因此 XML 文件內容的呈現，端看內容提供者之意願。一份 XML 文件內容，可以容易的以不同形式來呈現給不同使用者。

8. XML 可包含驗證文件內容與結構的文法：XML 文件可以包含定義內容與結構的文法，稱爲「文件型態定義」(*Document Type Definition，DTD*)。文件型態定義可用來描述文件結構、標籤、屬性以及文件內其他內容。當處理重要而複雜的資料時，透過文件型態定義可以驗證使用者撰寫的 XML 文件是否正確。
9. 搜尋更爲容易：XML 文件的內容與結構非常容易解析。因此在一份 XML 文件內，可以輕易找到對應之內容。同時透過 DTD 搭配，我們可以輕易剖析與操作整份文件，根據 DTD 製訂對應搜尋方式，進而能夠更迅速找到所需內容。

接下來，我們介紹 XML 文件與其結構概念。首先，參考圖 2.2，它是一份完整的 XML 文件範例。在本論文後面，我們將持續使用這個範例，作爲對 XML 文件詳細說明之共用範例。

```
<?xml version = "1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="book.css"?>
<!DOCTYPE LIBRARY [
<!ELEMENT LIBRARY (BOOK)*>
<!ELEMENT BOOK (AUTHOR,TITLE,INTRODUCTION)>
<!ELEMENT AUTHOR ((LAST_NAME,FIRST_NAME)|#PCDATA)>
<!ELEMENT LAST_NAME (#PCDATA)>
<!ELEMENT FIRST_NAME (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT INTRO (#PCDATA)>
<!ATTLIST BOOK JAVA CDATA #REQUIRD>
]>
<LIBRARY>
    <BOOK CLASS="JAVA">
     <TITLE>Java Servlet </TITLE>
     <AUTHOR>Jason Hunter</AUTHOR>
     <INTRO>This is a java servlet handbook.</INTRO>
    </BOOK>
</LIBRARY>
```

圖 2.2. XML 共用範例

- XML 序文 (*XML Prolog*)

XML 序文放置在一份 XML 文件的最前端。雖然一份 XML 文件不一定要有序文部分，但是一般建議至少在 XML 序文部分加上版本宣告。一份 XML 序文主要分爲三個部分：

1. XML 宣告 (*XML Declaration*)：圖 2.3 爲 XML 宣告。XML 宣告放在 XML 文件的第一行，請參照圖 2.1 中的位置。它包含三種屬性：

A. Version：版本。

B. Encoding：編碼。

C. Standalone：是否有參考外部實體。

```
<?xml version = "1.0" encoding="UTF-8"?>
```

圖 2.3. XML 宣告

2. 處理指令 (*Processing Instructions*)： XML 文件中，並沒有說明要如何去處理或呈現 XML 文件內容。不過使用者可以透過「處理指令」來描述 XML 要用何種方式處理或呈現。在 XML 文件規格書中並沒有詳細定義有關於「處理指令」的部分。圖 2.4 爲處理指令之範例。

```
<?xml-stylesheet type="text/css" href="book.css"?>
```

圖 2.4. 處理指令

3. 文件型態定義 (DTD)：DTD 就是一份 XML 文件的文法。關於 DTD 在後面有詳細的說明與解釋。圖 2.5 爲 DTD 之範例。

```
<!DOCTYPE LIBRARY [
<!ELEMENT LIBRARY (BOOK)*>
<!ELEMENT BOOK (AUTHOR,TITLE,INTRODUCTION)>
.........
<!ELEMENT INTRO (#PCDATA)>
<!ATTLIST BOOK JAVA CDATA #REQUIRD>
]>
```

圖 2.5. 文件型態定義

- 元素(*Element*)

XML 文件的結構是由許多元素所構成。而元素又是由起始標籤(*Start Tag*)與結束標籤(*End Tag*)所組成。圖 2.6 中由「<」開始與「>」結束就是起始標籤，由「</」開始與「>」結束的就是結束標籤。二者合稱爲一個元素。因此在圖 2.6

```
<LIBRARY>
   <BOOK CLASS="JAVA">
    <TITLE>Java Servlet </TITLE>
    <AUTHOR>Jason Hunter</AUTHOR>
    <INTRO>
        This is a java servlet handbook.
    </INTRO>
   </BOOK>
</LIBRARY>
```

圖 2.6. 元素

中共有 LIBRARY、BOOK、TITLE、AUTHOR 以及 INTRO 等五個元素。元素當中有二種特殊的型態元素。第一種是空元素。第二種是根元素。

- 空元素 (*Empty Element*)：本身只有一個標籤，這標籤同時包含起始與結束標籤的功能。一般而言，空元素通常都會包含屬性在其內部。圖 2.7 是一個空元素範例。

```
<CLASS NAME="JAVA" />
```

圖 2.7. 空元素

- 根元素 (*Root Element*)：在每一份 XML 文件當中，一定會有一個元素包含其他所有的元素。這一個元素就稱爲根元素。如圖 2.8 中，元素 LIBRARY 就是這一份 XML 的根元素。

```
<LIBRARY>
   <BOOK CLASS="JAVA">
    <TITLE>Java Servlet </TITLE>
    .........
   </BOOK>
</LIBRARY>
```

圖 2.8. 根元素

- 屬性 (*Attribute*)

屬性是用來在元素標籤中，加入「名-值對」(name-value pair)來紀載額外資訊。如圖 2.9，BOOK 這一個元素，加入 CLASS 屬性來代表其分類。

```
<BOOK CLASS="JAVA">
```

圖 2.9. 屬性

此外撰寫一份 XML 文件時，一個元素的屬性與子元素是允許互換的，但是如果全部改用屬性來表示，可能會造成文件難以閱讀，因此屬性的使用以四至五個爲上限。圖 2.10 爲全部使用屬性的 XML 文件，非常難以閱讀。而圖 2.11 爲

```
<BOOK CLASS=”JAVA” TITLE=”Servlet” NUM=”032498312”
AUTHOR=” Jason Hunter” >
```

圖 2.10. 全部使用屬性的 XML 文件

```
<BOOK CLASS=”JAVA”>
    <NUM>032498312</NUM>
    <TITLE>Java Servlet </TITLE>
    <AUTHOR>Jason Hunter</AUTHOR>
</BOOK>
```

圖 2.11. 屬性與子元素混用的 XML 文件

屬性與子元素混用的 XML 文件，這樣閱讀上就較爲方便。

- CDATA 區塊 (*CDATA Sections*)

在一份 XML 文件當中，特殊的字元，如「<」，使用上是透過已經定義好的字串「<」來置換它。但是當 XML 文件必須包含一大段含有特殊字元的文字資料時，使用這種置換的方式，反而顯得複雜與沒有效率。因此 XML 規範中提出 CDATA 區塊來處理這個問題。CDATA 區塊由「<![CDATA」開始，「]]>」結束。在 CDATA 區塊當中的特殊字元，完全被視爲是一般的文字，所以不需要以事先定義好的字串進行取代。如圖 2.12 與圖 2.13 的範例。圖 2.12 爲不使用 CDATA 區塊的狀況，包含很多的取代字串，非常難閱讀與理解。而圖 2.13 中使用 CDATA 區塊後，可以清楚的分辨出內容是表達何種意義。這是 CDATA 區塊強大的地方。但是，CDATA 區塊不能像元素一樣可以巢狀套疊，也就是一個 CDATA 區塊內不能包含另一個 CDATA 區塊。

```
<TEXT>
&lt; HTML &gt;
This is a &quot; CDATA Sections &quot; example.
&lt;/HTML &gt;
</TEXT>
```

圖 2.12.不使用 CDATA 區塊的 XML 文件

```
<TEXT>
<![CDATA
<HTML>  This is a “CDATA Sections“ example. </HTML>
]]>
</TEXT>
```

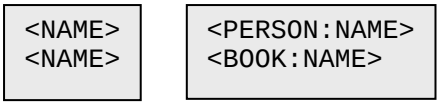
圖 2.13. 使用 CDATA 區塊的 XML 文件

- XML 名稱領域 (*XML Namespace*)

一份 XML 文件可以讓使用者透過 DTD 定義自己的標籤與屬性，但是隨著 XML 文件數量越來越多。就產生了一些問題，其中之一就是「所定義的標籤與屬性名稱有可能重複」。當在同一份 XML 文件中，用到來自不同 DTD 但是名稱卻相同的標籤或屬性時，會產生衝突。爲了解決這個問題，W3C 提出了 XML 名稱領域來解決標籤或屬性命名重複的問題。透過再標籤與屬性名稱前面加上一個新的命名與冒號，來解決相同標籤與屬性名稱所造成的混淆。

如圖 2.14 之範例，假設在同一份 XML 文件中用到來自不同二個 DTD 裡面所定義的標籤。在沒有使用名稱領域以前，如圖 2.14(a)，是無法區分出標籤來

自哪一個 DTD，代表何種意義。若加上名稱領域，如圖 2.14(b)，就可以輕易的將它們區隔開來。



(a)無名稱領域　(b)有名稱領域

圖 2.14. 名稱領域之範例

以上爲 XML 文件的內容與結構基本介紹，實際上 XML 文件的內容與結構有更複雜的規定，不足的地方請參考 XML 規格書 [7]。

最後，來看一下「格式正確的 XML 文件」(*Well-Formed XML Documents*)與「有效的 XML 文件」(*Valid XML Documents*)爲何。

- 格式正確的 XML 文件

一份 XML 如果符合 XML 語法規範，就稱爲「格式正確的 XML 文件」，它並不一定需要符合 DTD 定義。其中需要遵守的 XML 語法規範包括：

  - 元素標籤成對出現，一個元素有開始標籤，則必須有相對應的結束標籤，空元素除外。
  - 英文大小寫有區別。
  - XML 宣告須出現在文件之首。
  - 文件中只有一個根元素，在它之內包含了所有其他的元素。
  - 元素所構成的結構可以成巢狀，但不可交錯。
  - 特殊字元的使用必須以預設好的實體取代，如「&」用「&」取代。

這是大部分的基本規則，而更詳細的規範必須參考 XML 規格書 [7]。一般而言，在使用的 XML 文件最少會是一份格式正確的 XML 文件，因爲在處理文件的時候，非格式正確的 XML 文件，不易處理並容易產生錯誤。因此「格式正確」是成爲一份 XML 文件最基本的條件。

- 有效的 XML 文件

「有效的 XML 文件」是指 XML 文件中標籤與屬性的用法符合某一套 DTD 所規定的規則。一份有效的 XML 文件必定是一份「格式正確的 XML 文件」。它是由格式正確的 XML 文件加上 DTD 的宣告而成。DTD 的宣告可以在 XML 文件中直接宣告或是引用外部獨立 DTD 檔案。

2.2 文件型態定義 (DTD)

「文件型態定義」(DTD)就是 XML 的文法。DTD 是用來指定一份 XML 文件的結構與語法，而不是用來指定內容。舉個例子來說，DTD 就像一份公文的寫作規範，哪一部份要填寫什麼，都做了嚴格的規定，但是內容部分就必須自己填寫，依照這個規範寫出來的公文，才是有效力的。而依照一份 DTD 所寫出來的「格式正確 XML 文件」，則稱爲一份「有效的 XML 文件」。圖 2.2 是一份完整 XML 文件，而在序文部分就包含了一份 DTD。這份 XML 文件的內容主體就是依照 DTD 所產生的。

接下來將說明 DTD 內容是如何撰寫與產生。我們將其分爲幾個部分來說明：

- 文件型態宣告 (*Document Type Declarations*)

DTD 是用來規範 XML 內的結構與語法。而「文件型態宣告」則是用來宣告這些定義。在宣告的內部直接定義 DTD，或是引用外部的 DTD 來使用。基本上，文件型態宣告方式非常簡單，使用<!DOCTYPE>來建立即可。根據 XML 的規格書，分爲二大類：

1. 引用外部已定義好的 DTD：有二種方式，如圖 2.15 所示。其中 root_element_name 代表根元素的名稱。system_identifier 與 public_identifier 都是放置 URL 指向外部的 DTD。

```
<!DOCTYPE root_element_name SYSTEM system_identifier >
<!DOCTYPE root_element_name PUBLIC public_identifier system_identifier >
```

圖 2.15. 引用外部 DTD 的方式

2. 直接定義於文件宣告之內：如圖 2.16 所示。其中 root_element_name 是根元素的名稱。而後面就是直接擺放 DTD 實際內容，在圖 2.2 的範例中，「文件型態宣告」就是採取這一種模式。

```
<!DOCTYPE  root_element_name  [DTD]>
```

圖 2.16. 直接定義 DTD 的方式

- 元素宣告 (*Element Declarations*)

XML 文件中最基本的單位就是元素，所以在 DTD 中我們最常宣告的也是元

素的語法。元素的宣告方式如圖 2.17 所示。

```
<!ELEMENT Name Model>
```

圖 2.17. 元素宣告的方式

其中，ELEMENT 指出這一行是要宣告元素。Name 指定此元素的名稱。Model 則是指出此元素的型態。而元素的型態分爲下列五種：

- 任意 (*ANY*)
  - 代表可以包含任何種類的內容，一般而言是不會用 ANY 的。
- 空元素 (*EMPTY*)
  - 宣告成爲一個空元素，在前一節中有說明空元素爲何。
- 解析字元資料 (*#PCDATA*)
  - 解析字元資料(*Parsed Character Data*)爲 XML 中儲存文字內容的宣告。在 DTD 中，無法宣告資料型態，所以所有的資料內容在 XML 中都是純文字資料(*plain text*)。
- 子元素串列 (*Sub Element List*)
  - 宣告此 Element 的 Sub Element 爲何，有許多種變化，後面會另行說明。
- 混合內容 (*Mixed Content*)
  - 把#PCDATA 與 Sub Element 混合使用時就成爲 Mixed Content。

當宣告一個元素，包含許多其他的元素所形成的子元素串列時，有下列幾種作法：

- 一個或多個子元素 (*One or more*)
  - 在一個元素下包含一個或多個子元素，用 e+來表示，其中 e 爲元素名稱。例如：

    <!ELEMENT LIBRARY (BOOK)+>

    代表在元素 LIBRARY 之下可以包含一個以上的元素 BOOK。
- 零個或多個子元素 (*Zero or more*)
  - 在一個元素下包含零個或多個子元素，用 e*來表示。例如：

    <!ELEMENT LIBRARY (BOOK)*>

代表在元素 LIBRARY 之下可以包含零個以上的元素 BOOK。

- 零個或一個子元素 (*Zero or one*)
  - 在一個元素下包含零個或一個子元素，用 e?來表示，例如：

```
<!ELEMENT LIBRARY (BOOK)?>
```

代表在元素 LIBRARY 之下可以選擇包含零個或一個的元素 BOOK。

- 序列 (*Sequences*)
  - 用來指定元素包含子元素的順序。用逗號依序隔開，如 e1,e2 代表元素 e1 先出現然後元素 e2 再出現。例如：

```
<!ELEMENT BOOK (AUTHOR,TITLE,INTRO)>
```

代表在元素 BOOK 之下的子元素出現的順序爲 AUTHOR，TITLE，INTRO 三者。

- 選擇 (*Choice*)
  - 用來選擇子元素的種類。用 | 依序隔開。如 e1|e2 代表，可以選擇元素 e1 或是元素 e2，這個例子與副序列範例一起討論。
- 副序列 (*Subsequences*)
  - 用來建立一個副序列，副序列的作用在增加 DTD 的變化性。一個副序列將用小括號將其包起來，例如：

```
<!ELEMENT AUTHOR ( (LAST_NAME,FIRST_NAME) | #PCDATA )>
```

在此例當中，使用 LAST_NAME 與 FISRT_NAME 二者建立了一個副序列。元素 AUTHOR 可以使用解析字元資料直接填入作者的全名，或使用副序列，將全名拆成 LAST_NAME 與 FIRST_NAME 二子元素再填寫。在此可以看到使用副序列使可以使 DTD 的宣告方式有更多的變化。

- 實體 (*Entities*)

一份 DTD 除了宣告元素外，還宣告了另外二個部分，實體與屬性。實體是 XML 文件中非常好用的單元。.它的主體主要是文字資料。它的概念非常的簡單。假設在 DTD 中做了實體宣告(*Entities Declaration*)後，就可以在文件中使用實體

參考(*Entities Reference*)來引用實體。而 XML 處理器在處理這些實體參考時,會自動將實體的文字內容給代換進去。來看一個簡單的範例。如圖 2.18，在 DTD 中宣告了實體，而在 XML 文件中透過實體參考的方式來使用它，在最後處理 XML 文件時，它就會被代換掉。

```
<?xml version = "1.0" standalone="yes"?>
<!DOCTYPE LIBRARY[
<!ELEMENT LIBRARY (CLASS)*>
….
<!ENTITY DATE “May 2, 2003">
]>
```

(a)DTD 檔案

```
<LIBRARY>
 <CLASS>
 …
<DATE>&DATE;</DATE>
 <CLASS>
…
</LIBRARY>
```

(b)XML 文件

```
<LIBRARY>
 <CLASS>
 …
<DATE>May 2, 2003</DATE>
 <CLASS>
…
</LIBRARY>
```

(c) 處理時 XML 實際的內容

圖 2.18. 實體的概念

以上的範例，已經很清楚的說明了實體的概念與其中一種用途。接下來要說明，實體在 XML 規範中的分類。實體可以用三種方式來分類。

依照實體參考引用的地方來區分：

- 一般實體 (*General entity*)
  - 一般實體應用在 XML 文件內容或內部 DTD 內容。圖的範例就是一般實體的應用。
- 參數實體 (*Parameter entity*)
  - 參數實體應用在外部 DTD 內容。

而參考實體的方式，依照二種不同的實體也分爲二種參考的方式：

- 一般實體參考 (*General entity references*)
  - 使用 and 符號(&)開始，分號(;)結束。
- 參數實體參考 (*Parameter entity references*)
  - 使用百分比符號(%)開始，分號(;)結束。

依照實體的來源也分爲二大類：

- 內部實體 (*Internal entity*)
  - 在 XML 文件內被定義並使用。
- 外部實體 (*External entity*)
  - 在 XML 文件之外被定義，通常是透過一個 URI 來指向這個被參考到的外部資源。

最後，依照實體是否能被 XML 文件剖析器(*XML Parser*)所解析來區分：

- 解析實體 (*Parsed Entity*)
  - 解析實體可以直接由 XML 文件剖析器處理，一般是指文字資料。
- 未解析實體 (*Unparsed Entity*)
  - 未解析實體所包含的資料可以是文字資料或是其他的資源。如圖檔、聲音、影像等非文字資料。每一個未解析實體都有一個相關聯的標記法(*Notation*)。

爲了說明標記法，我們必須詳細說明未解析實體的組成。未解析實體用於非文字資料。假設有一張 JPEG 圖片，要宣告成未解析實體。則必須宣告它在 XML 中的名稱、資源所在位置與資源種類。如圖 2.19 所示。

```
<!ENTITY  the_car  SYSTEM  “http://www.car.com/car.jpg” NDATA  jpeg>
```

圖 2.19. 標記法的使用

其中 the_car 是實體的名稱。http://www.car.com/car.jpg 是所代表資源的位置。而 NDATA jpeg 則是指出這項資源的種類。XML 當中並沒有 jpeg 這種型態的資料。事實上，它必須透過標記法宣告(*Notation declaration*)後才能使用它。宣告的方式如圖 2.20。

```
<!NOATION  jpeg  SYSTEM  “image/jpeg”>
```

圖 2.20. 標記法的宣告

在圖 2.20 當中，宣告 jpeg 代表在網際網路上常見的 MIME 媒體型態(*MIME media type*)中的 JPEG 圖檔格式。經此宣告，就可以使用 jpeg 來定義資源的種類了。

以上的部分是實體的簡單介紹，而實體在 XML 文件當中還有許多變化與應用，實際上是很複雜的，但在本論文中將不多做討論。

- 屬性 (*Attribute*)

屬性是在 XML 文件中除了元素之外最常用的單元。當然在使用屬性之前，必須要在 DTD 中先宣告。屬性的宣告的方式如圖 2.21 所示。

```
<!ATTLIST ELEMENT_NAME
        NAME TYPE VALUE
        NAME TYPE VALUE
        .........
        NAME TYPE VALUE>
```

圖 2.21 屬性的宣告

其中 ELEMENT_NAME 是包含此屬性的元素名稱，NAME 是屬性名稱，TYPE 是屬性的型態，VALUE 是屬性的預設值。其中 XML 的屬性型態可區分爲三大類，十小類，如下：

- 字串型態 (*String type*)
  - CDATA：在屬性中只有字元資料可被使用。
- 標記型態 (*Tokenized type*)
  - ID：屬性值應是唯一的。如果一個文件包含的 ID 屬性有相同的屬性值，則處理器應該會產生錯誤。
  - IDREF：其值應參照到文件中別地方宣告的 ID。如果屬性沒有符合參照到的 ID 值，則處理器應該會產生錯誤。
  - IDREFS：和 IDREF 相同，但是允許多個數值，可由空白鍵分隔開來。
  - ENTITY：屬性值必須參照到在 DTD 中宣告的實體。
  - ENTITIES：和 ENTITY 相同，但是允許多個數值，可由空白鍵分隔開來。
  - NMTOKEN：屬性值是字元名稱的組合，這些字元應爲字母、數字、虛線、冒號或底線。
  - NMTOKENS：和 NMTOKEN 相同，但是允許多個數值，可由空白鍵分隔開來。
- 列舉型態 (*Enumerated type*)
  - NOTATION：屬性值必須參照到 DTD 中其他地方宣告的 NOTATION。宣告也可以是 NOTATION 列表，而這個值必須是

NOTATION 列表中的一個，每個 NOTATION 必須在 DTD 中都有它自己的宣告。 NOTATION 可以宣告爲非 XML 資料的格式。

- Enumeration：屬性值必須符合列舉值之一。舉例來說 ：

  <!ATTLIST MyAttribute (Type1 | Type2）>

  其中 MyAttribute 可選擇 Type1 或 Type2 其中之一。

而屬性的預設值分爲四種：

- VALUE：給予一個預設值
- IMPLIED：沒有預設值 屬性可以省略
- REQUIRED：沒有預設值 屬性不可以省略
- FIXED VALUE：給予一個固定的預設值,這個值不會被改變

以上爲 DTD 內容與宣告方式的一些簡單的介紹，透過這些就可以定義出一份完整的 DTD 檔案以供 XML 件所使用。

2.3 文件物件模型 (DOM)

文件物件模型(DOM) [10]是一套把HTML文件或XML文件轉換成某種特殊的資料結構，進而進行資料存取或操作的應用程式介面(*API*)。DOM是由W3C所定義出來的規格。目前正式的版本已經到DOM Object Model Level 2，而新的版本仍然在修訂當中。而DOM本身是爲了程式設計者所開發制定，透過DOM，設計者可以建立一個文件的資料結構，在這一個資料結構上對此文件作新增修改或是刪除的動作。

在DOM規格書內，並沒有詳細指出文件必須實作成何種特定的資料結構。但以目前的XML文件來說，文件當中的元素與元素之間都會存有父子(*parent-child*)的關係或是兄弟(*sibling*)的關係，因此我們可以把XML文件看成是一個擁有階層式結構的文件。而展現一個階層式文件最好的資料結構就是樹狀結構。因此，以目前來講，常用的DOM API都是採取樹狀結構來實作DOM的規格。而實作出來的樹狀結構我們稱爲Document tree或是DOM tree。

DOM 規格的版本已經到第三版，W3C 把 DOM 的版本命名方式稱作爲 Level，越高的 Level 修訂舊的 Level 中的不足，加上更強大的功能。在最早的

Level 1 版本當中，只著重在描述整個物件結構與管理文件內容的議題上。Level 2 時，提供了一整組新的規格書，包括 Core (Level 1 的修定與延伸)、Views、Events、Style、Traversal and Range。而目前最新仍在修訂中的 Level 3 版本，提供了 Load and Save、XPath、Validation 等新規格，並更新的 Level 2 中的 Core 與 Event 部分。在本論文當中，會使用到的部分主要爲 Core 的部分，我們將對 Core 部分作進一步的介紹。

DOM Core 部分無疑的是 DOM 規格內最基本的部分。Core 被分爲二個部分，基礎(*Fundamental*)部分與延伸(*Extended*)部分。在基礎部分提供了許多共用介面(*Interface*)給予 HTML 文件與 XML 文件使用。而延伸部分則是因爲 XML 比 HTML 多出許多規定，因此在此部分提供處理 XML 文件一些額外的介面。

在 DOM Core 的部分當中，最重要的就是 Node 介面。大部分的介面都是衍生於 Node 介面。Node 介面內有節點型態(nodeType)，不同的節點型態經由繼承後由不同的介面所定義。而 Node 介面定義了許多通用的屬性與方法來存取整個文件與內容。例如：

- nodeType：得知介面是何種型態的節點。
- parentNode：節點的父節點。
- childNode：節點的子節點。
- attributes：節點的屬性。
- nodeName：節點的名稱。
- nodeValue：節點的值。

以上只是Node介面一些簡單的內容與方法，其他的請參考 [5]。基本上介面內容與方法都是爲了操作整份文件而產生的。而繼承Node介面的新介面，如Element、Attr與Text等，因種類不同，都有它們本身特有的屬性與方法。圖2.22是一張DOM Core部分中，介面的繼承圖。很清楚可以看到除了NodeList、NamedNodeMap、DOMImplementation與DOMException之外，其他型態的介面都是繼承於Node介面。

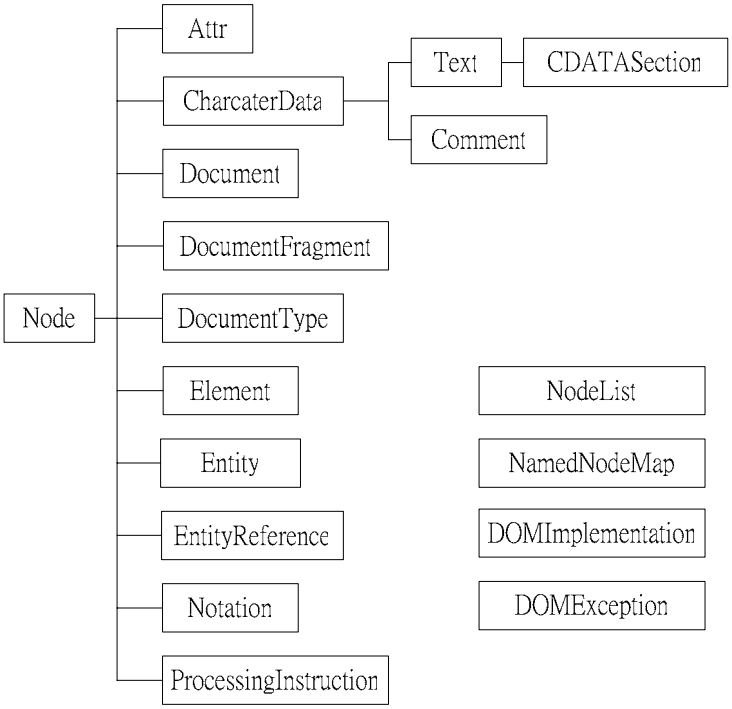


圖 2.22. Node 繼承關係圖

但在 DOM tree 當中，組成的單位爲節點，而這些不同的節點，是由 DOM 規格書中不同的介面所定義，因此可以用「節點」來稱呼這些介面。

接下來，我們將說明上述節點內容與其代表 XML 文件當中的哪一部份。分爲基礎與延伸二部分來說明。

- 基礎部分(Fundamental)：共有 12 類，屬於基礎部分。
  1. DOMException：在DOM當中，操作可能會在某些特殊狀況下造成例外(*Exception*)。如，在找尋內容時找不到，或是在操作NodeList時，index超過了上限。DOM在發生這一些狀況的時候，會傳回一些特別的錯誤值來代表操作當中所遇到的錯誤。當然，再不同實作方式下的DOM，會有其自己特別的錯誤回傳值。在規格書內，只是定義了一些通用的錯誤回傳值。
  2. DOMImplementation：DOMImplementation提供了方法給使用者去判斷文件是否有符合某種特徵，如：這份文件有沒有符合HTML或XML的特徵。
  3. Document：Document代表了整份的XML文件。它會顯現在整個DOM

tree最上端，所有其他的節點都必須包含在Document之內。而Document內也包含了其他節點的產生方法。

4. DocumentFragment：DocumentFragment是目前DOM tree的一部分。在規格書中，可以被稱爲lightweight或minimal的Document。DocumentFragment是經常用到的，在DOM tree的操作過程中，常有需要對tree的部分做剪下(*cut*)、貼上(*paste*)或移動(*move*)的動作，在過程中，此時被從原來tree所分離的部分就必須在Document tree之外暫存起來，這暫存的部份就是DocumentFragment。當然，Document本身也可以擔任這個任務，但是Document本身有太多新增的屬性與方法在此處並用不上，若使用了Document反而是種浪費。因此，才另外產生DocumentFragment。
5. Node：Node是最基本的節點。在整個DOM tree當中，大部分的節點都是由Node的繼承者。因此Node當中定義了一切最基本的屬性與方法，以供不同種類的節點所繼承。雖然不同節點繼承了Node中的方法與屬性，但是並不是所有的節點都會擁有子節點。比如說，繼承Node的Text節點，就不會有子節點。同樣的，Node節點的許多屬性，在不同的形態的繼承者當中，也不一定會擁有，如Element節點中的nodeValue與Comment節點中的attribute，當透過Node當中的方法取得它們的時候，都會回傳null的狀態。
6. NodeList：NodeList提供了一個處理節點的「即時有序集合」(*live ordered collection*)概念。但是並沒有定義或規範要如何去時作出這一個集合。而live的概念，代表著在DOM tree中任何節點的改變，都會立即反映到NodeList之中。而NodeList常用在取得Element節點之下的Text節點或子Element節點，因爲這些節點是有順序的。
7. NamedNodeMap：NamedNodeMap與NodeList類似，不過在功能上有些差異性，NamedNodeMap並不是繼承於NodeList。NamedNodeMap同樣的提供了一個處理節點的集合，不過他是透過名稱來存取裡面的節點。NamedNodeMap是不帶有順序(*unordered*)，因此適合用在處理Attr節點的時候。
8. CharacterData：CharacterData提供了許多屬性與方法來處理DOM之內有

關於文字資料的部分。在DOM當中，並沒有直接去使用到CharacterData。而是透過繼承他的介面去使用CharacterData本身的屬性與方法來處理文字資料。繼承於CharacterData的節點有Text、Comment與CDATASection三種節點，在後面我們會說明他們的內容。

9. Element：Element是在DOM tree當中最常見的一種節點了。他代表著一份XML文件中的元素。
10. Attr：Attr代表Element節點之下的一個屬性。Attr繼承了Node節點的屬性與方法。在DOM當中，Attr並不被直接視爲一個Element節點的子節點，而是被視爲Element節點的一項屬性(property)。因爲Attr單獨出現並沒有任何意義，所以通常Attr是伴隨Element節點而出現。但在概念上，我們仍可以把Att想成一個Element的子節點。Attr雖然與其他種節點同樣繼承於Node節點但是他卻和其他節點非常的不同
11. .Text：Text繼承於CharacterData節點。它代表著XML中的文字資料(*character data*)。如果在元素內容裡面並沒有任何的標示(markup)，則它就成爲一個單一的Text節點，變成一個Element Node的子節點。如果在元素內容當中包含標示，則這內容將會解析爲新的元素與這個新元素的內容。
12. Comment：Comment與Text同樣繼承於CharacterData。而他代表著XML註解(Comment)的內容，也就是出現在‘<!--' 與'-->’之間的文字資料。

- 延伸部分(Extended)：共有 6 類。
  1. CDATASection：CDATASection用來展現XML文件中的CDATA Sections。而CDATASection透過繼承Text節點而繼承了CharacterData節點的屬性與方法。
  2. DocumentType：DocumentType提供了一個介面來取得在DTD當中所宣告的實體(entities)與標記法(notations)。而一份XML文件最多只會有一個DTD，所以DocumentType最多也只會有一個，它是透過Document而取得。
  3. Notation：Notation代表在DTD中所宣告的標記法。在DOM tree當中，Notation只能存在於在DocumentType之下。它由DocumentType之內的方

法所產生。但是他並不是DocumentType的子節點，這一點跟Attr相類似，它們都是節點之下的屬性，而不是節點之下的子節點。

4. Entity：Entity代表在DTD中所宣告的實體。與Notation相同的，在Document tree當中，Entity也只能存在於DocumentType之下。同樣是DocumentType的一項屬性，而不是DocumentType的子節點。
5. EntityReference：EntityReference是用來代表在文件中用來指向實體的實體參考。
6. ProcessingInstruction：ProcessingInstruction是用來代表XML文件中的處理指令。

以上爲 DOM 的一些簡單介紹。

2.4 XML 元件樹 (XML Component Tree，XCT)

本論文中爲方便本研究的進行，我們介紹新的資料表示方式以作爲基礎。此種資料表示方式從 XML DOM tree 修改而來，稱之爲「XML 元件樹」(*XML Component Tree*)。此節中我們說明 XML 元件樹是如何由 DOM tree 轉換而來。

在 2.3 節中，本論文所介紹 DOM 的 18 種基本型態節點中，其中只有三種節點在本論文當中被考慮，它們分別是 Element 節點、Attr 節點與 Text 節點，這三個節點的變動會影響到「內容」以及「內容與內容之間的關係」。至於其他節點則與內容無直接關係或是不考慮，其原因分述如下：

- DocumentType、ProcessingInstruction、Notation 與 Entity 等四個節點是用於 DTD 部分，不影響 XML 文件實際內容。
- NodeList 與 NamedNodeMap 爲存放與操作節點的容器，與 XML 文件內容無關。
- Document 用來代表整份 XML 文件，不是 XML 文件內容。
- DocumentFragment 用於操作過程中暫存部分的 DOM tree，與 XML 文件內容無關。
- DOMException 和 DOMImplementation 與 XML 文件內容無關。
- Node 爲大部分節點之祖先，不直接用在 DOM tree 當中，與 XML 文件內容無關。

- CharacterData 爲中介繼承介面，不直接用在 DOM tree 當中，與 XML 文件實際內容無關。
- Comment 爲註解，並非 XML 文件中的真正內容，因此本論文不考慮。
- CDATASection 用於紀錄標示(markup)內容，與非標示內容之 Text 類似，因此本論文僅考慮 Text。
- EntityReference 爲實體參考，本論文暫不考慮其轉換問題。

因此，去除無關的節點，將 DOM tree 進行簡化僅由 Element、Attr 以及 Text 等三種節點所構成。圖 2.23 爲簡化後的 DOM tree 範例。其中，Text 節點爲 Element 節點的子節點，在 DOM tree 中，以實線來表示之。而 Attr 節點是 Element 節點的屬性(property)，以虛線來表示它們的關係。

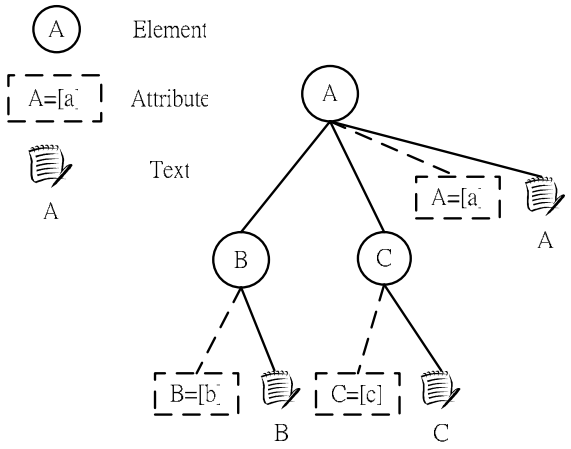
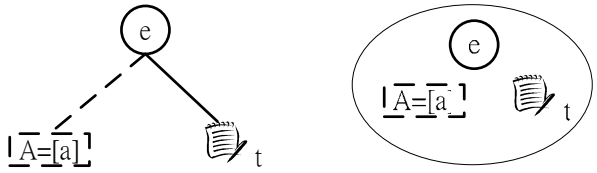


圖 2.23. 簡化後的 DOM tree 範例圖

XML 元件樹是經由簡化的 DOM tree 轉變而來。其中基本組成單位爲「XML 元件」(*XML component*)，簡稱爲 *component*。一個 XML component 是以一個 DOM tree 中的 Element 節點與其相對應之 Attr 節點及 Text 節點所形成。圖 2.24 (a)與(b)分別說明原本在 DOM tree 中的節點表示方式以及其對應之 component。

在本論文中，component 另以 $c=(e,A,t)$ 表示之。其中：

- c 代表 component；



(a) DOM tree 中的節點　(b) XML component

圖 2.24. 節點與 Component

- *e* 代表 *c* 的元素名稱；
- *A* 代表 *c* 的屬性，其中 $A=[a_1,a_2,\ldots,a_i,\ldots,a_n]$，$a_i$ 爲一個「名值對」(name-value pair)，$a_i=(n_i, v_i)$，n_i 爲屬性名稱，而 v_i 則爲其屬性值；
- *t* 代表 *c* 的文字內容(text)。

接下來，定義一個通用串列結構，這個串列結構在定義 component 的基本函數時會用到：

- $S=[s_1,s_2,\ldots,s_i,\ldots,s_n]$，S 爲一個串列(list)，由 s_i $(1 \le i \le n)$個物件所組成。
- $num(S)=n$，代表 list S 內的元素個數。

緊接著我們介紹在本論文中，使用到的 component 之相關基本函數：

- $Element(c)=e$ ，component *c* 的 Element name。
- $Attributes(c)=A=[a_1,a_2,\ldots,a_i,\ldots,a_n]$ ，component *c* 中所有屬性所構成的 list。
- $Attribute\text{-}Names(c)=[n_1,n_2,\ldots,n_i,\ldots,n_n]$ ，component *c* 中所有屬性名稱所構成的 list。
- $Attribute\text{-}Values(c)=[v_1,v_2,\ldots,v_i,\ldots,v_n]$ ，component *c* 中所有屬性值所構成的 list。
- $Attribute(c,i)=a_i$ ，component *c* 中之第 *i* 個屬性。
- $Attribute\text{-}Name(c,i)= n_i$ ，component *c* 中之第 *i* 個屬性名稱。
- $Attribute\text{-}Value(c,i)= vi$ ，component *c* 中之第 *i* 個屬性值。
- $Text(c)=t$ ，component *c* 中的文字內容。
- $size(Text(c))= size\ (t)$ ，代表 component *c* 中的文字內容大小。

將 XML 文件的 DOM tree 中的節點全部轉換成 components，即可得到其對應之 XML 元件樹，在後文當中我們以 T_X 表示之。

一個 T_X 中，位於同一個 component 下的所有 child components，其順序具有關係，更動順序可能會造成內容結構的破壞。

一個 T_X 中，位於同一個 component 之下的 child component 之間，含有「順序關係」。因 T_X 是由 XML DOM Tree 轉變而來。在 DOM tree 當中，Element 節點是有順序的，而 component 是以 Element 節點爲主體，合併屬於此 element 的 attribute 與 text 而成。所以在 T_X 中的 component 同樣是有順序的。

Component 的順序關係到內容的結構，更動順序可能會造成內容結構的破壞，因此 component 的順序是重要而不可任意更動的。例如：

- 假設在 T_X 當中，有三個 components 位於同一 component 之下。其中，第一位的 component 其 Text 爲「Sam」，第二爲「is」，第三爲「boy」。三個 component 的內容依序構成一個有意義句子「Sam is boy」。但若是互換第二與第三個 component 的順序則變爲「Sam boy is」，造成內容的無意義。因此 component 的順序關係更動會破壞內容結構的正確性。

當然也有內容順序不重要的 XML 文件，但在本論文當中，由 XML 文件所轉出來的 T_X 皆會被看成「有序樹狀結構」。

2.5 總結

在本章中，我們介紹可延伸性標記語言(XML)、文件型態定義(DTD)與文件物件模型(DOM)的基本概念與定義。實際上，上列三者所包含的內容非常廣泛，但在本論文中不需更深入的內容。若需要相關資料，請參考 [5] [7]。

此外，我們說明本論文需要使用的資料表示方式「XML 元件樹」以及組成單元 XML component 其結構與對應之可用函數，這些基礎將供下一章使用。

第三章 XML 文件內容同等轉換

在討論內容同等轉換前，首先參考圖 3.1，我們定義串列 $S(T_X)$如下：

$$S(T_X) = ([r, [S(T_r)]])+。$$

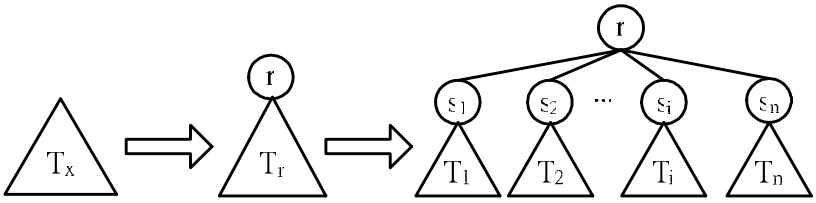


圖 3.1 T_X 結構串列示意圖

而 $S(T_X)$可以被進一步展開如下：

$S(T_X)=[r, [[s_1, [S(T_1)]], [s_2, [S(T_2)]], \ldots, [s_i, [S(T_i)]], \ldots, [s_n, [S(T_n)]]]]$。
這一個串列稱爲「結構串列」(*Structural List*)，$S(T_X)$用來展現 T_X 的實體結構。

針對 $S(T_X)$，我們來做進一步的探討。

- $S(T_r)$可以透過展開式被遞迴展開，用 T_r 之中第一層 components 以及以 child components 爲根節點(root)的子樹(subtree)所取代。
- 被取代後的 child component 順序，依照順序由左至右出現在結構串列中。
- 在結構串列中，樹狀結構的每一層之間用 [ ] 來隔開，以建立出層次。
- 展開時，child component 與子樹會成對出現，若 component 爲 leaf component，則沒有子樹項目。

明白了 $S(T_X)$後，後面要介紹「內容串列」(*Content List*)。首先，我們先來解釋「XML 元件內容」(*XML component content*)。

假設 component c 是 T_X 的一個節點。其中 *Attribute-Value(c)*與 *Text(c)*二者合稱爲「XML 元件內容」(*XML component content*)或簡稱爲「元件內容」(*component content*)。在本論文中，使用「$C(c)$」表示之。

若將 $S(T_X)$中之 component 改以「元件內容」來取代，則我們可以得到一個新的串列如下，稱之爲「內容串列」(*Content List*)，用 $L(T_X)$表示之。$L(T_X)$ 用來展現 T_X 的內容結構。

$L(T_X)=[C(r), [L(T_r)]]$
$=[C(r), [[C(s_1), [L(T_1)]],[C(s_2), [L(T_2)]], \ldots, [C(s_i), [L(T_i)]], \ldots, [C(s_n), [L(T_n)]]]]$

Definition：假設 T_X 可以被轉換成另一個 T_X'，其中 $S(T_X)$可能不再等於 $S(T_X')$，但 $L(T_X)$卻相等於 $L(T_X')$。此時，由 T_X 轉變至 T_X'的轉換，我們將之稱爲「**內容同等轉換**」(*content equivalent transformation*)或簡稱「轉換」(transformation)。而兩份 XML 文件之間若存在有對應的內容同等轉換，則我們說此兩份 XML 文件「**內容同等**」(*content equivalent*)。

本章後三節將介紹三個內容同等轉換，它們分別是：「更名」(*rename*) 轉換、「合併」(*merge*) 轉換以及「分割」(*split*)轉換。透過運用這三個轉換，一份 XML 文件可以被轉換成另一份內容同等的 XML 文件。

3.1 更名轉換

首先我們介紹更名轉換(Rename Transformation)，在本論文當中更名轉換分爲二類，其定義分別如下：

- 元素更名轉換(*Element Rename Transformation*) ：對於 T_X 中任意 component c 更改其 *Element*(*c*)。
- 屬性更名轉換(*Attribute Rename Transformation*)：對於 T_X 中任意 component c 中第 *i* 個屬性更改其 *Attribute-name*(*c*,*i*)，其中$1 \le i \le num(Attribute(c))$。

依據此定義，更名轉換演算法說明如下。我們將元素更名與屬性更名合併在同一個演算法當中，透過判斷來決定將使用哪一種更名轉換。完整之演算法列於圖 3.2。

1. 輸入 T_X。
2. 由 T_X 中選擇欲進行更名轉換之 component c。
3. 判斷爲元素更名或屬性更名轉換。
4. 若爲屬性更名轉換，將新的名稱指定給 *Element*(*c*)。
5. 若爲屬性更名轉換，則於 component c 中選擇欲更改名稱之第 i 個屬性，將新的名稱指定給 *Attribute-Name*(*c*,*i*)。

```
Input Tx
select component c from Tx
// element Rename or attribute rename
boolean  element_rename_flag = checkRenameType() ;
if(element_rename_flag == true){
    Element(c)=new_Element_Name;
}
else{
    select the i-th attribute form c
    Attribute-Name(c,i)=new_Attribute_Name;
}
```

圖 3.2. 更名轉換演算法

下面是更名轉換的範例。首先，我們建立此 XML 文件的 T_X。接著選定 T_X 要更名之部分，在範例中我們選擇 component TITLE 去更改它的元素名稱，將它從「TITLE」更改成「NAME」。圖 3.3 是更名前與更名後的 XML 文件。圖 3.4 則是此二份文件的 T_X 示意圖。

```
<LIBRARY>
  <CLASS>
    <BOOK BID="JAVA001">
      <TITLE>Java Servlet </TITLE>
      <AUTHOR>Jason Hunter</AUTHOR>
    </BOOK>
</CLASS>
</LIBRARY>
```

(a) 更名前

```
<LIBRARY>
  <CLASS>
    <BOOK BID="JAVA001">
      <NAME>Java Servlet </NAME>
      <AUTHOR>Jason Hunter</AUTHOR>
    </BOOK>
</CLASS>
</LIBRARY>
```

(b) 更名後

圖 3.3. 更名轉換的 XML 文件

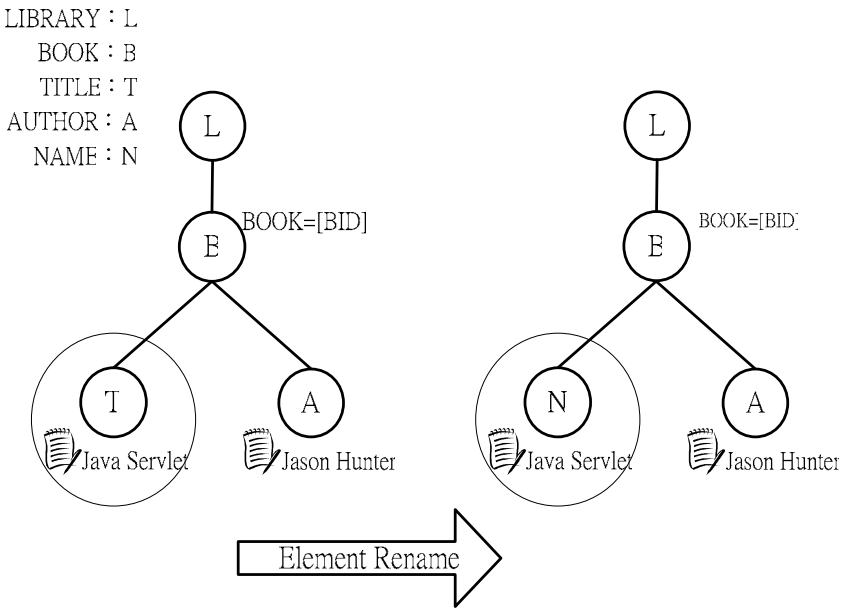


圖 3.4. 更名轉換的 T_X 示意圖

Lemma 1：更名轉換(Rename Transformation)是一個內容同等轉換。

證明：

參考圖 3.5，轉換前：

$S(T_X) = [S(T), [[s_1, [S(T_1)]], ..., [s_i, [S(T_i)]], \ldots, [s_n, [S(T_n)]]]]$，而

$L(T_X) = [L(T), [[C(s_1), [L(T_1)]], \cdots, [C(s_i), [L(T_i)]], ..., [C(S_n), [L(T_n)]]]]$。

假設對 s_i 進行元素更名轉換，改變 s_i 成新的 s_i'。更名過程並不影響到 Text 與 Attribute-Value 的值，所以不影響到 $C(s_i)$，因此 $C(s_i')=C(s_i)$。

轉換後：

$S(T_X) = [S(T), [[s_1, [S(T_1)]], ..., [s_i',[S(T_i)]], ..., [s_n,[S(T_n)]]]]$，而

$L(T_X) = [L(T), [[C(s_1), [L(T_1)]], ..., [C(s_i'), [L(T_i)]], ..., [C(S_n), [L(T_n)]]]]$。

雖然 $S(T_X') \neq S(T_X)$ 但是 $C(s_i')=C(s_i)$ 所以 $L(T_X') = L(T_X)$。

因此元素更名轉換確實爲內容同等轉換。同理可證屬性更名轉換亦爲內容同等轉換。

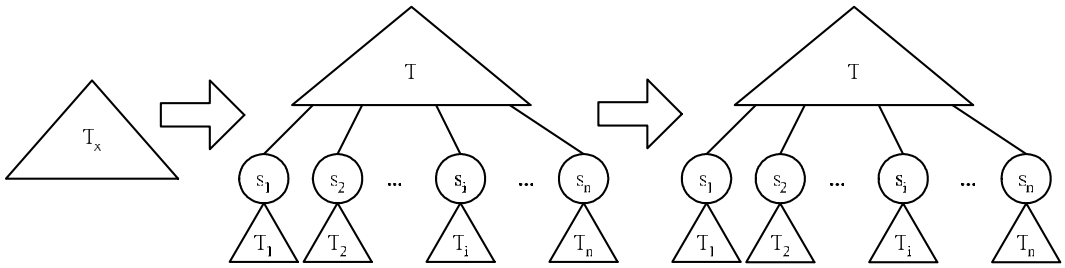


圖 3.5.更名轉換證明示意圖

3.2 合併轉換 (Merge Transformation)

在說明合併轉換(Merge Transformation)與分割轉換(Split Transformation)前，我們先介紹「轉換資訊」(*transformation information*)，用 *I* 來表示之。轉換資訊會由參與合併轉換或分割轉換的各個 component 所產生。假設參與的 component 有 $c_1, c_2, \ldots, c_n$，則產生相對應的轉換資訊有 $Ic_1, Ic_2, \ldots, Ic_n$，轉換資訊會串接在各 component 所有的屬性名稱之後。而未經轉換的 component 不會擁有轉換資訊。轉換資訊 I 會紀錄執行轉換過程中改變的資訊，而如何去紀錄這些資訊，將在說明合併轉換或分割轉換時再介紹之。

轉換資訊 I 分由五個部分所構成，每一個部分間使用「.」來連接，*I* 本身則構成一個字串。使用 $x_1.x_2.x_3.x_4.x_5$ 來表示。

- x_1 爲執行分割轉換的次數。每執行一次，x_1 值加一。
- x_2 爲執行合併轉換的次數。每執行一次，x_2 值加一。

- x_3 爲 component text 的長度。紀錄合併轉換前 component text 的長度。
- x_4 爲合併轉換前，被合併的 child component 之所在順序。
- x_5 爲合併轉換後，原來 component text 在新 component text 中的順序。
- x_1 至 x_5 所有欄位預設值爲 0。

其中，x_4 與 x_5 的「順序」是不相同的。x_4 的順序是 component 之間的順序關係。如：component c 之下含有 child component s_1, s_2 與 s_3。其順序分別爲 1, 2 與 3。所以，若 s_2 進行合併轉換，則其 x_4 將等於 2。x_5 的順序則是 component text 在合併後於新的 component text 中之順序。假設將 s_1, s_3 與 c 合併，新合併後的 text 順序爲 s_1 的 text, s_3 的 text 與 c 的 text。則 s_3 在合併後，因爲 s_3 的 text 排在第二位，其 x_5 等於 2。有關轉換資訊中 x_5 的計算方式，在後面說明合併轉換部分時，會有更詳細的說明。

後續章節中，爲方便演算法說明及正確性證明，我們先定義一些運算。假設 component c 進行轉換後，產生之轉換資訊爲 *Ic*，,則：

- *Attribute*(*c*) ⊕ *Ic* 表示：
  - 將 *Ic* 串接到 c 所有的屬性名稱之後。*Attribute*(*c*) ⊕ *Ic* = [a_1', a_2', …, a_i', …, a_n']，其中 a_i' =(n_i ⊕ *Ic* , v_i)，而 n_i ⊕ I_c 表示字串間串接 (concatenation)。
- *c* ⊕ *Ic* 表示：
  - 代表經過 *Attribute*(*c*) ⊕ *Ic* 修改後的新 component。

令 c_1 與 c_2 爲二個 components，則：

- *Attribute*(c_1) ⊕ *Attribute*(c_2) 表示:
  - 將 c_1 與 c_2 的 attribute list 合併。若 *Attribute*(c_1)=[a_{11}, a_{12}, …, a_{1n}] 與 *Attribute*(c_2) =[a_{21}, a_{22}, …, a_{2m}]，則 *Attribute*(c_1) ⊕ *Attribute*(c_2) = [a_{11}, a_{12}, …, a_{1n}, a_{21}, a_{22}, …, a_{2m}]。
- *Text*(c_1) ⊕ *Text*(c_2) 表示：
  - 將 c_1 與 c_2 的 text 合併。其中 *Text*(c_1)在前，*Text*(c_2)在後。二者之間以一個空白(*white space*)字元隔開。
- *C*(c_1) ⊕ *C*(c_2) 表示：
  - 將 c_1 與 c_2 的 component content 合併，爲執行 *Attributes*(c_1) ⊕

Attribute(c_2)*與 Text*(*c1*) ⊕ *Text*(*c2*)二項運算的結果。

接下來我們開始介紹合併轉換，其中合併轉換僅能合併 leaf components 至其 parent component。本論文中，合併轉換定義如下：

- 考慮在 T_X 中之任意 component c ，將其下的 leaf child component $c_1, c_2, \ldots, c_n$ 中第 i 個 component ($0 \le i \le n$)之 component content 合併至 component c 內。

依據合併轉換的定義，其演算法步驟如下，完整演算法則列於圖 3.6。

1. 輸入 *Tx*。
2. 由 T_x 中找到 component c。
3. 從 component c 的 child component 當中選擇要合併之 components 並將其放入 S。
4. 計算所有進行轉變之 components 的轉換資訊。
5. 進行 component text 的合併。
6. 進行 component attributes 的合併。
7. 將 S 中所有的 components 從 T_X 中移除。

```
Input Tx;
select the component c from Tx;
select the target components form the child components of c , and put
them into S;
computeTransformationInformation(c,S);
mergeText(c,S);
mergeAttribute(c,S);
remove S from Tx
```

圖 3.6.合併轉換的演算法

其中，合併其他 components 的稱爲「合併者」，如演算法中 component c。被合併的 components 則稱爲「被合併者」，如演算法中串列 S 裡面所包含的 components 。接下來我們再說明演算法中的 computeTransformationInformation(c,S)、mergeText(c,S)與 mergeAttribute(c,S)等三個函數之做法。

在轉換中，所有進行轉換的 components 都需要計算其轉換資訊。computeTransformationInformation(c,S)就是負責執行這項工作。其計算方式，依被合併者 components 的狀態不同而有所差異，分爲下列三種：

Case 1：所有被合併者 components 中，沒有任一個 component 曾爲合併者。

Case 2：所有被合併者 components 中，所有 components 皆曾爲合併者。

Case 3：所有被合併者 components 中，部分的 components 曾爲合併者。

上述三種情形之轉換資訊計算演算法列於圖 3.7 中。

```
Input c , S;
switch(checkMergeStatus(S)) { //Case1, 2 ,3 的判斷
case 1:                              //在 S 中所有被合併者，沒有任一曾爲合併者
    for(i=0;i<num(S);i++){
        si = get component i in S;
        Isi = create Transformation Information for si;
        Isi.x2=1;                        //執行第一次合併
        Isi.x3=size(Text(si));           //計算 text 長度
        Isi.x4=orderInTheChildComponent(si);
                                         //計算出 si 在原來 child component 中排第幾位
        Isi.x5= i+1 ;                    //si 在 S 中的順序，由 1 開始
        Do Attribute(si) ⊕ Isi ;         //將轉換資訊串接至 si 屬性名稱之後
    }
    Ic = create Transformation Information for c;
    Ic.x2=0;                             //合併其他 component 不增加
    Ic.x3=size(Text(c));                 //計算 text 長度
    Ic.x4=0;                             //合併者此項爲 0
    Ic.x5= num(S)+1;                         //放在所有被合併者之後爲 num(S)+1
    Do Attribute(c) ⊕ Ic ;                       //將轉換資訊串接至 c 屬性名稱之後
    break;
case 2:                              //在 S 中所有 component 皆經過合併
    for(i=0;i<num(S);i++){
        si = get component i in S;
        Isi[ ]= get all Transformation Information from si;
                 //合併過的 component 裡面包含多個轉換資訊須對它們進行更改的動作
        for(j=0;j<Isi.length;j++){
            Isi[j].x2= Isi.x2 +1;        //進行合併，x2 值加一
            if (Isi[j].x4 == 0)
                Isi.x4= orderInTheChildComponent(si);
            Isi[j].x5= forntComponentMerge(Isi) + Isi[j].x5; //重新計算 X5
        }
    }
    Ic = createTransformation Information for c;
    Ic.x2=0;  Ic.x3=size(Text(c));Ic.x4=0;
    Ic.x5= childComponentMerge(c) +1 ;
    Do Attribute(c) ⊕ Ic ;
    break;
case 3:                              //部分 component 經過合併，部分未經過合併
    for(i=0;i<num(S);i++){
        si = get component i in S;
        if(checkMerge(si)==true){
                        // checkMerge (si) 判斷 component si 是否曾經過合併
            Isi[ ] = get all Transformation Information from si;
            for(j=0;j<Isi.length;j++){
                Isi[j].x2 = Isi.x2 +1 ;
                if (Isi[j].x4 == 0)
                    Isi.x4= orderInTheChildComponent(si);
                Isi[j].x5= forntComponentMerge(Isi) + Isi[j].x5 ;
        }
```

圖 3.7. Transformation Information 計算演算法

```
        else{
             Isi = create Transformation Information for si;
             Isi.x2=1;
             Isi.x3=size(Text(si));
             Isi.x4=orderInTheChildComponent(si);
             Isi.x5= forntComponentMerge(Isi) + 1;
             Do Attribute(si) ⊕ Isi ;
        }
    }
    Ic = createTransformation Information for c;
    Ic.x2=0;
    Ic.x3=size(Text(c))
    Ic.x4=0;
    Ic.x5= childComponentMerge (c) +1 ;
    Do Attribute(c) ⊕ Ic ;
    break;
}//switch
```

圖 3.7. Transformation Information 計算演算法 續

演算法當中，使用到二個特殊函數，以下以範例來解釋，請參考圖 3.8：

- frontComponentMerge(component s_3)：
  - 代表 component s_3 的順序值，此順序值含出現在 component s_3 前的其他所有 components 及其被合併者個數。例如圖 3.8 中，實線圈代表此次之被合併者，而虛線圈代表被合併者的上一次被合併者。因此從圖中我們可以發現，s_1 合併過 4 個 components(包含自己本身)，s_2 合併了 3 個 components，則 frontComponentMerge(s_3)就等於 7。

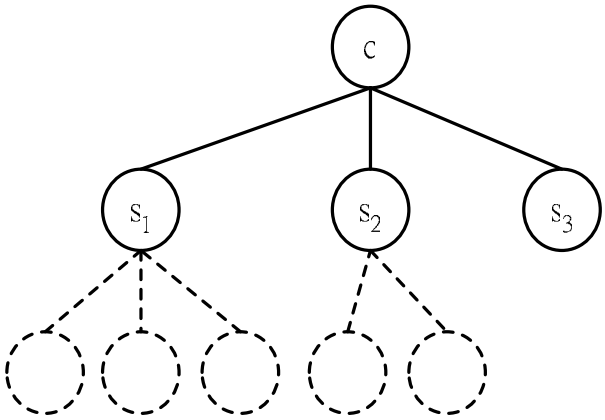


圖 3.8. frontComponentMerge 與 childComponentMerge 說明圖

- childComponentMerge(component c)：
  - 將最右邊的被合併者之順序值加一，即是 component c 的順序值。圖 3.8 中，s_1 合併過 4 個 components，s_2 合併 3 個 components，s_3 沒有合併過任何 component，則 childComponentMerge(c)等於 8。

計算完轉換資訊後，再來進行 text 合併。首先，將被合併者(在 S 當中的

component)的 text 依序作字串串接。當被合併者的 text 串接完成後，再串接上合併者(component c)的 text，最後的結果作爲合併者的新 text。圖 3.9 爲完整演算法。

```
Input c,S;
String tempText="";
for(i=0;i<num(S);i++){
    si = get component i in S;
    tempText = tempText ⊕ Text(si) ;    //text串接
}
Text(c) = tempText ⊕ Text(c) ;    //最後合併c的text，結果放回合併者c
```

圖 3.9 Merge Text 演算法

text 合併完後，則進行 attribute 合併。attribute 合併與 text 合併過程類似。其中，attribute 的合併並不需要依照順序合併，只需要將所有的 attribute 合併在一起即可。圖 3.10 爲完整演算法。

```
Input c,S;
AttributeList attributeList;
for(i=0;i<num(S);i++){
    si = get component i in S;
    attributeList = attributeList ⊕ Attribute(si) ;
}
Attribute(c) = attributeList ⊕ Attribute(c) ;
```

圖 3.10. Merge Attribute 演算法

以上爲合併轉換的完整演算法說明。接下來，用一個範例來展現演算法的效果。

下面是這一個合併轉換的範例。在這一個範例中，我們將圖 3.11 的 T_X 進行二次合併轉換，最後得到圖 3.13 的 T_X。我們使用這個範例來說明合併轉換過程，

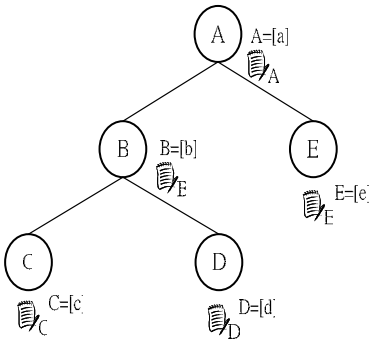


圖 3.11. 合併轉換前的 T_X 示意圖

以及過程中如何改變轉換資訊。在圖 3.11 中，圓形內的 A、B、C、D 與 E 爲 component 的元素名稱(element name)。而「A=[a]」代表 component 的屬性(attribute)，在範例中爲方便說明，假設每個 component 只有一個屬性。而文件的圖形則代表 component 的文字內容(text)，在本範例中假設 text 的長度爲 500 個字。

首先假設合併者爲 B，被合併者爲 C 與 D。接下來，計算 B、C 與 D 的轉換資訊。因爲 C 與 D 未合併過，所以我們得到轉換資訊爲 I_B=0.0.500.0.3、I_C=0.1.500.1.1 與 I_D=0.1.500.2.2，並串接到各自的屬性名稱後。接下來，進行 text 合併，先將 component B 的 child component C 與 D 之 text 依序作文字串接，串接完再串接上 B 的 text。最後，將 component B、C 與 D 的屬性部分合併。並將合併完畢的 text 與屬性放至 component B 內，結果如圖 3.12 所示。

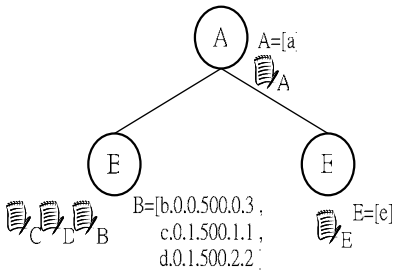


圖 3.12. 第一次合併轉換後的 T_X 示意圖

假設我們再對圖 3.12 中的 B 與 C 合併至 A 中。首先計算 A、B 與 C 的轉換資訊。其中 B 是合併過的 component，C 是沒有合併過的 component，所以得到轉換資訊 I_B=0.1.500.1.3、I_C=0.2.500.1.1、I_D=0.2.500.2.2、I_A=0.0.500.0.5 與 I_E=0.1.500.2.4，並將新產生的 I_A 與 I_E 串接到各自的屬性名稱後。接下來，進行 text 的合併。先將 component B 中合併過的 text 放在最前面，後面串接上 component E 的 text，再串接上 component A 的 text。同樣的，將 component A、B 與 E 的屬性部分合併，其最終合併結果顯示於圖 3.13。

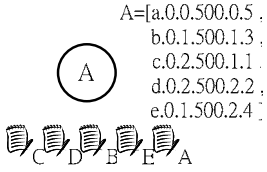


圖 3.13. 第二次合併轉換後的 T_X 示意圖

後面我們將證明合併轉換是一個同等轉換，但在證明前，必須先說明如何透過轉換資訊(transformation information)的運作，取回原來的 component content 及 component content 所在的位置。此步驟我們稱之爲 restore，由取回屬性(Retrieval Attribute)、取回內容(Retrieval Text)與取回內容位置(Retrirval Location)三者所構成。

首先，我們來討論取回屬性。因爲在轉換時，所有的 component 中屬性名稱之後皆會串接上轉換資訊，而來自同一個 component 的屬性名稱後，會串接上相同的轉換資訊。因此，只要透過找尋相同的轉換資訊，就能找到來自相同 component 的屬性。圖 3.14 是 Retrieval Attribute 的完整演算法。

```
Input c;
if(checkMerge(c)==true)                    //判斷c有沒合併其他component
{
     transformationInformationList[]
          = get different Transformation Information from Attribute(c);
     Sort the transformationInformationList[] by Transformation Information
     x5-value ;
     New an AttributeList[][];
     for(i=0;i<=num(Attributes(c));i++){
          for(j=0;j<=transformationInformationList.length;j++){
              //比對轉換資訊是否相同
               boolean same=compare(transformationInformationList[j] ,
                      Transformation Information in the Attribute (c,i));
               //若相同則爲來自同依個component的屬性 將其放入同一類
               if(same==true){
                    Add Attribute(c,i) to AttributeList [j][]
                    break;
               }
          }
     }
     return AttributeList[][];       //傳回分好類的屬性串列
}// if
```

圖 3.14. Retrieval Attribute 演算法

而在合併過的 component 當中取回合併前各個 component 的 text，只需透過轉換資訊的計算即可取得。首先，仍然從合併過的 component 中，取出不同的轉換資訊。然後，對取出的轉換資訊以 x_5 的值作排序。透過轉換資訊的順序以及 x_3 部分所記載的長度。我們可以由合併過的 component text 中，將原來所有的 component text 分離開來。完整的演算法列於圖 3.15。

利用 Retrieval Attribute 與 Retrieval Text 二個演算法，可以順利取回原來的 component content。但是我們不知道 component content 是由哪一個 component 合併過來的。因此發展 Retrieval Location 演算法來還原 component content 的原始位

置。

```
Input c
if(checkMerge(c)==true)  //判斷c有沒合併其他component
{
     transformationInformationList[]
          = get different Transformation Information from Attribute(c);
     Sort the transformationInformationList[] by Transformation Information
     x5-value ;
     New a string list TextList[];
     int base=0;
     for(i=0;i<= transformationInformationList.length;i++){
          TransformationInformation ti = transformationInformationList[i];
          int length= ti.x3
          Add Text(c).subString(base,base+length) to TextList[i]
          base=base+length+2;    //加上一個white space
     }
     return TextList[];         //傳回分好類的text串列
}// if
```

圖 3.15. Retrieval Text 演算法

首先，產生一個資料結構，用來儲存還原位置後的結果。這個物件含二個項目。第一是儲存 component 本身的轉換資訊。第二則是一個串列，儲存此 component 的 child components。以下爲此資料結構：

component (TransformationInformation ti, List componentList[])

接下來是演算法的計算過程，我們先輸入一個要還原位置的component。由其中取出不同的轉換資訊，並以x_5的值對轉換資訊作排序。之後將轉換資訊以reLocation演算法運算，reLocation演算法會利用上面的資料結構產生出一個以輸入component爲根節點(root)的樹狀結構。這一個樹狀結構可以展現合併前各個component content之間的位置關係。圖3.16爲其Retrieval Location演算法。

```
Input c;
transformationInformationList[] =
          get different Transformation Information from Attribute(c);
Sort   transformationInformationList []   by Transformation Information x5-value ;
New a startComponent;
reLocation(transformationInformationList [],startComponent);
return startComponent;
```

圖 3.16. Retrieval Location 演算法

reLocation演算法是透過遞迴的方式，一層層將component以轉換資訊所紀錄的資訊還原回來。我們以圖3.17來說明演算法的概念。在圖中，假設component *A* 合併了component B、C、D、E、F與G。則透過合併轉換的演算法所計算出來的 I_A至I_G，其x_2與x_5值如圖上所標示。x_2是由合併的次數所產生，而x_5是由合併的順

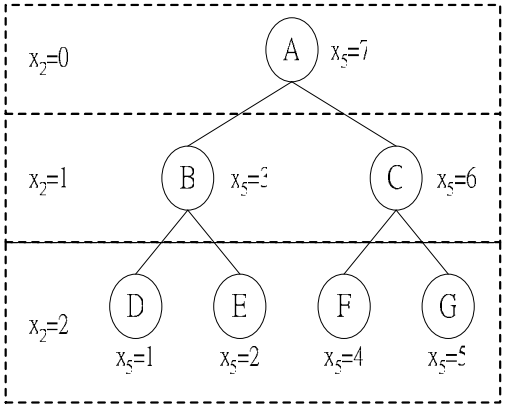


圖 3.17. 合併概念圖

序所產生。由此圖我們可以得知，若有c_1與c_2二個components，其中c_1的x_2比c_2少一，而c_1的x_5比c_2的sibling component當中x_5值最大者多一，則c_1爲c_2的parent component。透過這個關係，可以發展一個演算法來解析這個關係，重建出合併前樹狀結構。圖3.18爲其reLocation演算法。

```
reLocation(transformationInformationList [] , node){
  New a newComponent;
  New a inputList[ ][ ];
  int inputListCount=0;
  int inputListSubCount=0;
  for(i=0;i<= transformationInformationList.length ;i++){
      TransformationInformation ti =
  transformationInformationList[i];
      if(ti.x2 > 1){
          ti.x2= ti.x2 -1;
          ti.x5= inputListSubCount +1;
          inputList[inputListCount][ inputListSubCount]=ti;
          inputListSubCount++;
      }
      else if (ti.x2=1){
          ti.x2= ti.x2 -1;
          ti.x5= inputListSubCount +1;
          inputList[inputListCount][ inputListSubCount]=ti;
          inputListCount++;
          inputListSubCount=0;
      }
      else if (ti.x2=0){
          newComponent.ti= ti;
          Add newComponent to stratComponent.componentList[];
      }
  }
  for(i=0; i<=inputList.length ; i++){
    reLocation(inputList[i][] , newNode);
  }
}//function
```

圖 3.18. reLocation 演算法

當整個 restore 步驟執行完畢後，我們已經重建合併前以合併點爲 root 的子元件樹(sub component tree)。

而在下面的証明當中，*restore*(*L*(T_X))代表對整個 T_X 中所有的 component 作還原。而 *L*(T_X)可以遞迴展開，因此可以變爲對每個 component 作 restore。而還原 component 時，若 component 未經過合併，則 restore 並無效用。假設有一 component *c* 並未合併過，則 *restore*(*C*(*c*))= *C*(*c*)。若合併過，則進行 Retrieval Text、Retrieval Atribute 與 Retrieval Location 三步驟，取回其內的 component content 與位置。

Lemma 2：合併轉換是一個內容同等轉換。

證明：

參考圖 3.19，假設將 T_X 中任一個 component *Y* 下的第 *i* 個 leaf child component s_i 合併至 *Y*, 其中 s_i 爲 leaf component.

轉換前：

$S(T_X)=[S(T), [Y, [s_1, \ldots, s_{i-1}, s_i, s_{i+1}, ..., s_n]]]$，而

$L(T_X)= [L(T), [C(Y), [C(s_1), ..., C(s_{i-1}), C(s_i), C(s_{i+1}), ..., C(s_n)]]]$。

轉換後：

執行合併轉換後 s_i 產生一個 Is_i，*Y* 產生一個 I_Y，來紀錄其轉換資訊。進行合併後，s_i 移除，*Y* 變爲 *Y*’。其中：

$Attribute(Y') = (Attribute(Y) \oplus I_Y) \oplus (Attribute(s_i) \oplus Is_i)$，

$Attribute\text{-}Value(Y')=Attribute\text{-}Value(Y) \oplus Attribute\text{-}Value(s_i)$，

$Text(Y')= Text(s_i) \oplus Text(Y)$　所以 $C(Y')= C(Y \oplus I_Y) \oplus C(s_i \oplus Is_i)$，

$S(T_X')=[S(T), [Y', [s_1, ..., s_{i-1}, s_{i+1}, ..., s_n]]]$，而 $L(T_X')= [L(T), [C(Y'), [C(s_1), ..., C(s_{i-1}), C(s_{i+1}), ..., C(s_n)]]]$。

雖然，$S(T_X') \neq S(T_X)$，但是由於：

$restore(L(T_X'))= restore([L(T), [C(Y'), [C(s_1), ...,C(s_{i-1}), C(s_{i+1}), ..., C(s_n)]]])$

$=[restore(L(T)), [restore(C(Y')), [restore(C(s_1)), ..., restore(C(s_{i-1})), restore(C(s_{i+1})), ..., restore(C(s_n))]]]$

$= [L(T), [restore(C(Y')), [C(s_1), ...,C(s_{i-1}), C(s_{i+1}), ..., C(s_n)]]]$

$= [L(T), [restore(C(Y \oplus I_Y) \oplus C(s_i \oplus Is_i)), [C(s_1), ...,C(s_{i-1}), C(s_{i+1}), ..., C(s_n)]]]$

$= [L(T), [C(Y), [C(s_1), ...,C(s_{i-1}), C(s_i), C(s_{i+1}), ..., C(s_n)]]]$

$= L(T_X)$　。

所以合併轉換爲內容同等轉換。

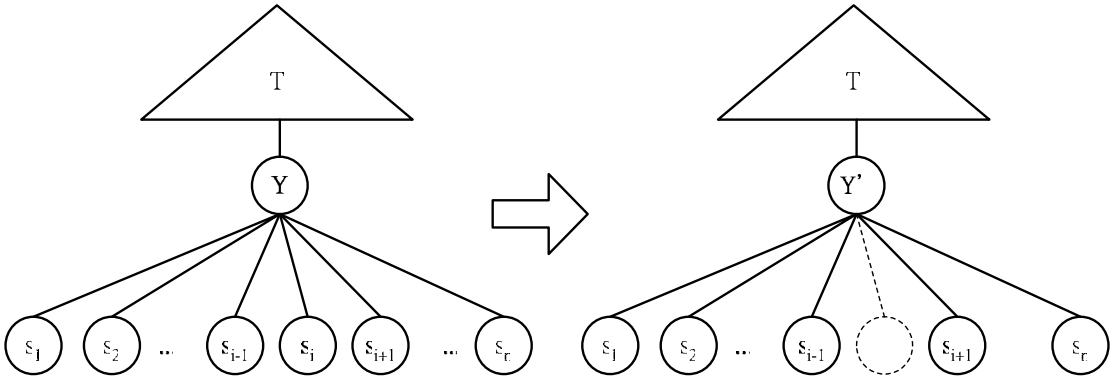


圖 3.19. 合併轉換證明示意圖

3.3 分割轉換 (Split Transformation)

本論文中，分割轉換的定義如下：

- 考慮在 T_X 中任意 component *c* ，將其 component content 分割成 *i* 個 components 的 component content。

依據上面的定義，發展分割轉換演算法。其步驟如下。完整演算法列於圖 3.20。

1. 輸入 *Tx*。
2. 由 T_x 中尋找欲作分割轉換的 component *c* 。
3. 選定 text 分割位置，將其分割成 n 個部分。
4. 產生 n 個新的 components，其內包含分割後的 text。
5. 計算進行轉換所需之 component 轉換資訊。
6. 將新產生的 components 加入 component c 的原有 child components 後。

```
Input Tx
select a component c in Tx
textList[] = chooseSplitLocation(c)
component list S=splitComponent(TextList[],c)
computeSplitTransformationInformation(c,S)
add S to become component c's child component
```

圖 3.20. 分割轉換演算法

割轉換演算法中的 chooseSplitLocation(c)、splitComponent(TextList[])與 computeSplitTransformationInformation(c,S)三個函數分述如下。

在分割轉換中，從 Text(c)選擇要分割的位置，並將位置紀錄下來。若切開位置數爲 n，則 text 將被切爲 n+1 份。其中前 n 份分別變爲新產生之 child

component 的 text。而最後一份(第 n+1 份)則留於原來 component 內。若切開位置選定在 text 之後的空白，則第 n+1 份爲空白。Text 最前端之處不能進行切割(位置爲 0 的地方)。切割之位置必需有所限制，將來還原時才不會造成錯誤，本論文中所定義的切割位置必須發生於 text 中「空白」(white space)之處。圖 3.21 是 chooseSplitLocation 的演算法。

```
Input c;
Choose n split location in text of component c ( 0<= location <=Text(c) ,
location is no duplicate ) and get a locationList [];
New a String List textList[];
int base=0;  int next=0;  int i;
for(i=0; i< locationList.length;i++ ){
    next= locationList[i];
    textList[i]=Text(c).subString(base, next-1);
    // Text(c) is a string
    base=next+1;;
}
textList [i+1]= Text(c).subString(base, Text(c).size());
return textList[ ];
```

圖 3.21. chooseSplitLocation 演算法

接下來的步驟則爲產生新的 components，圖 3.22 爲 splitComponent 演算法。我們將分割好的 text 放入新的 component 當中，並且將原來 component 中的屬性複製一份至新的 components 中。

```
Input textList[ ] ,c;
New a component list S[ ];
for (int i=0; i < textList.length -1 ; i++  ){
    New a component s;
    Text(s)= textList[i];
    Attributes(s)= Attributes(c);
    S[i]=s;
}
Text(c)= textList[textList.length];
return S[];
```

圖 3.22. splitComponent 演算法

再來就是計算新產生 components 的轉換資訊，若原本的 component 已經分割過，則將複製到新 components 當中的轉換資訊的分割次數加一。原本的 component 並未分割過，則對原本的 component 與新產生的 component 產生新的轉換資訊。圖 3.23 爲 computeSplitTransformationInformation 演算法。

以上爲分割轉換的過程，接下來使用範例來說明分割轉換的效果。在此範例

```
Input c,S
//判斷 c 是否包含 Transformation Information
boolean check = checkTransformationInforamtionStatus(c);
if(check==true){
    for (int i=0;i< S.length;i++){
        Isi =get Transformation Information from si ;
        Isi.x1= Isi.x1+1;
    }
}
else{
    Ic= create new Transformation Information ;
    for (int i=0;i< S.length;i++){
        Isi = create new Transformation Information ;
        Isi.x1= 1;
    }
}
```

圖 3.23. computeSplitTransformationInformation 演算法

中，我們將圖 3.24 中 T_X 的 component A 進行二次分割轉換，最後得到圖 3.27 的 T_X。

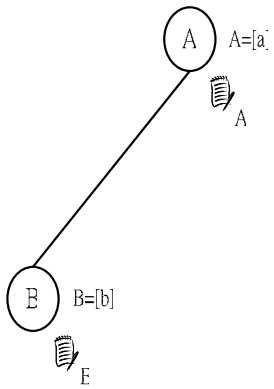


圖 3.24. 分割轉換前的 T_X 示意圖

在圖 3.24 中，假設每個 component 只擁有一個屬性。而文件的圖形則代表 component 的 text，在範例中假設 text 的長度爲 2000 個字。

對 component A 進行分割轉換。第一先挑選分割 text 的位置。在此假設挑選二個分割點，分別第一個在第 1001 個字的地方，第二則挑選在最後的位置。如圖 3.25。則整個 Text(A)將被分割成 3 份，第一份包含 1000 字，第二份包含 999 字，但第三份沒有包含任何內容。

再來產生新的 components，因爲分割成三份，所以產生二個新的 components，其中 component 的元素名稱並不重要，因此我們先以 A1 與 A2 來代表它們。產生 A1 與 A2 的過程中，將第一份 text 給予 A1，第二份給予 A2，

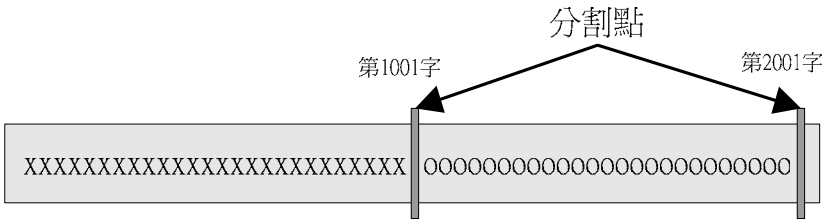


圖 3.25. 分割 text 示意圖

而第三份留給 component A，並將 component A 的屬性複製給 A1 與 A2。最後，我們計算分割轉換的 component 之轉換資訊。因爲是第一次進行轉換，所以所有的 component 都沒有轉換資訊，其中 A1 與 A2 是經過一次分割轉換後所產生的 component 因此，合併次數 x_1 的值爲 1。圖 3.26 爲經過一次分割轉換的結果。

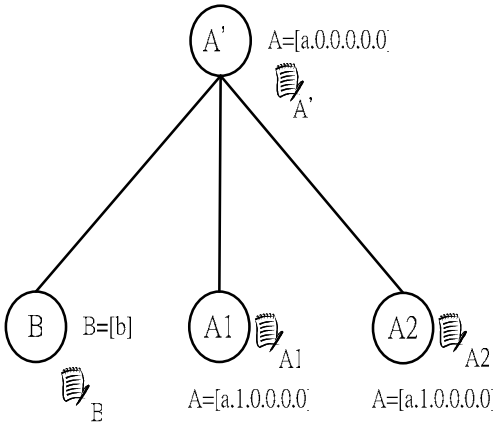


圖 3.26. 第一次分割轉換後的 T_X 示意圖

接下來進行第二次的分割轉換，此次我們對 A1 進行分割，A1 的 text 爲 1000 字，同樣我們選擇 2 個分割點，將其分成 3 份。第一份長度 500 字，第二份爲 499 字，第三份則爲 0 字。接著產生新 components A11 與 A12。將分割好的 text 放入其內。複製 A1 的所有屬性給 A11 與 A12。最後計算轉換資訊，其中因爲 A1 曾經過轉換。因此將 A11 與 A12 的轉換資訊中合併次數 x_1 再加一。圖 3.27 就是最後的結果。請注意在範例當中，我們假設分割點都符合規定，也就是分割點必須爲「空白」。

在證明分割轉換是一個同等轉換前，同樣的先說明如何透過轉換資訊，取回原來的 component content 及其所在位置。至於取回分割轉換前的 component content 位置，並不困難。由範例圖 3.27 中可看到，所有的分割轉換起源，都是來自同一個 component，所以只需要找到分割轉換的源頭，我們稱之爲 source

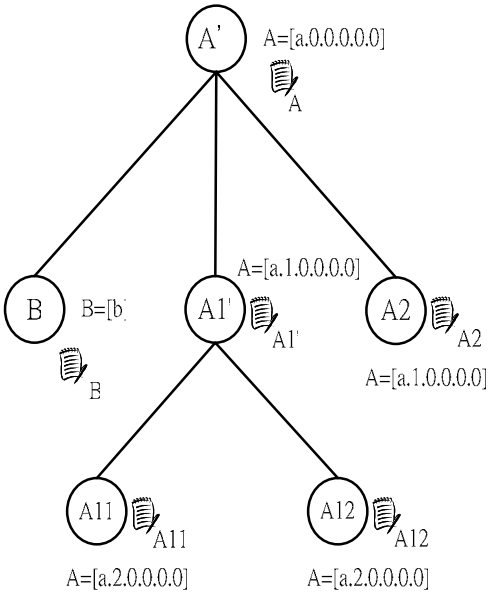


圖 3.27. 第二次分割轉換後的 T_X 示意圖

component，向下找出所有分割產生的 components，就可以得到以 source component 爲 root 產生一個樹狀結構，稱之爲 T_S。而 T_S 透過 restoreSplit 演算法，可以合併成爲一個 component，。在還原過程中，因當初在分割轉換時，分割的位置點有限制爲「空白」字元，所以在還原時將補上「空白」字元。圖 3.28 爲 restoreSplit 的演算法，其中 combine 演算法，請參考圖 3.29。

```
Input a split component c;
Find the split source component sourceComponent;
Find all the split component and create Ts rooted at sourceComponent;
Component c = combine(sourceComponent);
```

圖 3.28. restoreSplit 演算法

```
combine(Component c){
  ComponentList S[]=getChildComponents of c in Ts
  for(int i=0;i<S.length;i++){
      Component si=S.getComponent(i);
      if(checkChildComponent (si)==true){
        combine(si);
    }
  }
  Component tempC=new Component();
  for(int i=0;i<S.length;i++){
    Component si=S.getComponent(i);
    Text(tempC)= Text(tempC) ⊕ Text(si)
  }
  Text(tempC)= Text(lastC) ⊕ Text(c);
  DELETE all components in S;
  return tempC;
}
```

圖 3.29 combine 演算法

當一個 component *c* 分割而成的子樹，透過 restoreSplit 的演算法將子樹組合成原本的 component *c*，此時，我們已經取回原本的 component content。而 *restore*Split($L(T_X)$)代表它對整個 T_X 中所有的分割產生之 components 作分割還原。而 $L(T_X)$可以遞迴展開，因此可以變爲對每個 component 作 *restore*Split。若 component 未經過分割，則 restoreSplit 並無效果。如 component *c* 並未合併過，則 *restoreSplit*($C(c)$)= $C(c)$。

Lemma 3：分割轉換是一個內容同等轉換。

證明：

參考圖 3.30，假設將 T_X 中 component *Y*，分割成 m 個新 components。

轉換前：

$S(T_X)$=[$S(T)$, [*Y*, [[s_1, [$S(T_1)$]], ..., [s_n, [$S(T_n)$]]]]]，而

$L(T_X)$= [$L(T)$, [$C(Y)$, [[$C(s_1)$, [$L(T_1)$]], …, [$C(S_n)$, [$L(T_n)$]]]]]。

轉換後：

執行分割轉換，*Y* 分割成 m 個新的 component $c_1,c_2,...,c_m$，並產生轉換資訊 Ic_1, Ic_2, …, Ic_m 以及 I_Y。而 *Y* 經分割變爲 *Y'*。

其中：

Attribute(*Y*) = *Attribute*(*Y'*) = *Attribute*(c_1) = … = *Attribute*(c_m)，而

Text(*Y*)= *Text*(c_1) ⊕ *Text*(c_2) ⊕ …⊕ *Text*(c_m) ⊕ *Text* (*Y'*)。

所以 $C(Y)$= $C(c_1 \oplus Ic_1) \oplus C(c_2 \oplus Ic_2) \oplus \dots \oplus C(c_m \oplus Ic_m) \oplus C(Y' \oplus I_Y)$ ，

$S(T_X)$=[$S(T)$, [*Y*, [[s_1, [$S(T_1)$]], ..., [s_n, [$S(T_n)$]], c_1, c_2, …,c_m]]，而

$L(T_X)$= [$L(T)$, [$C(Y)$, [[$C(s_1)$, [$L(T_1)$]], …, [$C(S_n)$, [$L(T_n)$]], $C(c_1)$, $C(c_2)$, …, $C(c_m)$]]]。

雖然：

$S(T_X') \neq S(T_X)$，但是：

restoreSplit($L(T_X')$)=restoreSplit([$L(T)$, [$C(Y)$, [[$C(s_1)$, [$L(T_1)$]], …, [$C(S_n)$, [$L(T_n)$]], $C(c_1)$, $C(c_2)$, …, $C(c_m)$]]])

=[restoreSplit($L(T)$), [restoreSplit($C(Y)$), [[restoreSplit($C(s_1)$), [restoreSplit($L(T_1)$)]], …, [restoreSplit($C(S_n)$), [restoreSplit($L(T_n)$)]], restoreSplit($C(c_1)$), restoreSplit($C(c_2)$), …, restoreSplit($C(c_m)$)]]]

=[$L(T)$, [restoreSplit($C(Y')$), [[$C(s_1)$, [$L(T_1)$]], …, [$C(S_n)$, [$L(T_n)$]], restoreSplit($C(c_1)$), restoreSplit($C(c_2)$), …, restoreSplit($C(c_m)$)]]]

=[$L(T)$, [restoreSplit($C(Y' \oplus I_Y)$), [[$C(s_1)$, [$L(T_1)$]], …, [$C(S_n)$, [$L(T_n)$]], restoreSplit($C(c_1 \oplus Ic_1)$), restoreSplit($C(c_2 \oplus Ic_2)$), …, restoreSplit($C(c_m \oplus Ic_m)$)]]]

=[$L(T)$, [$C(Y)$, [[$C(s_1)$, [$L(T_1)$]], …, [$C(S_n)$, [$L(T_n)$]]]]]

= $L(T_X)$。

所以分割轉換爲內容同等轉換。

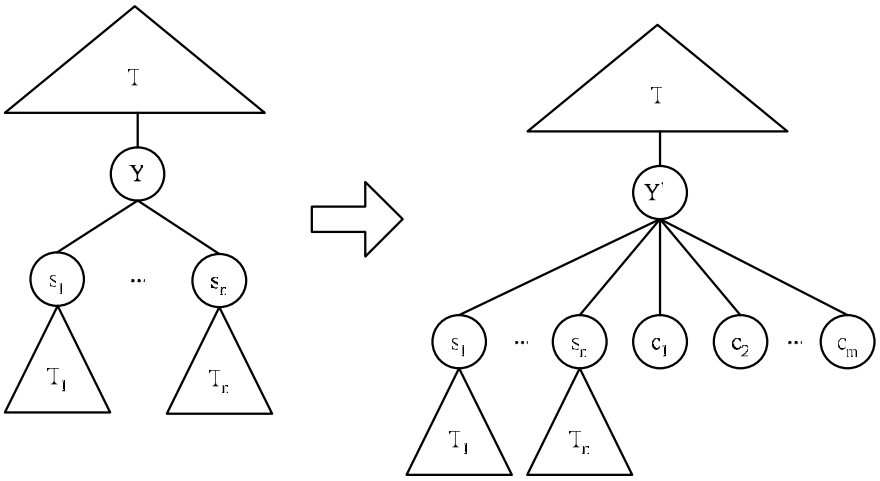


圖 3.30 分割轉換證明示意圖

3.4內容同等轉換的順序限制

介紹完三個同等轉換後，我們必須說明三個轉換在混用時的限制，因爲要讓混用各個轉換不會破壞到轉換資訊，我們必須對轉換操作的順序做一些規定。其順序如下：

1. 屬性更名轉換
2. 合併轉換
3. 分割轉換

而其中元素更名轉換可以於任意時間點執行。我們舉個例子來說明，假設我們對一個 T_X 進行二次屬性更名轉換、一次元素更名轉換、三次合併轉換與二次分割轉換。則在順序上，我們必須先做完所有的屬性更名轉換，再進行合併轉換，最後執行分割轉換。而元素更名轉換可以隨時進行。

3.5 總結

在本章中，我們介紹三個內容同等轉換，並且詳細說明它們的操作過程以及對應之演算法。同時，我們也證明它們符合內容同等轉換的定義。在下一章中，我們要透過一個完整範例來展現本章節的三個轉換的效果。

第四章 XML 文件內容同等轉換範例

在這一章中，將使用一個完整的範例，來說明第三章所介紹的內容同等轉換演算法。透過此範例，我們可以瞭解 XML 文件內容同等轉換的效果。

4.1 XML 內容同等轉換架構

依據第三章所述，混用三種同等轉換時會有順序的限制，因此我們定義同等轉換運作的流程於圖 4.1。首先，由一份 XML 文件產生出 DOM tree 結構。經由

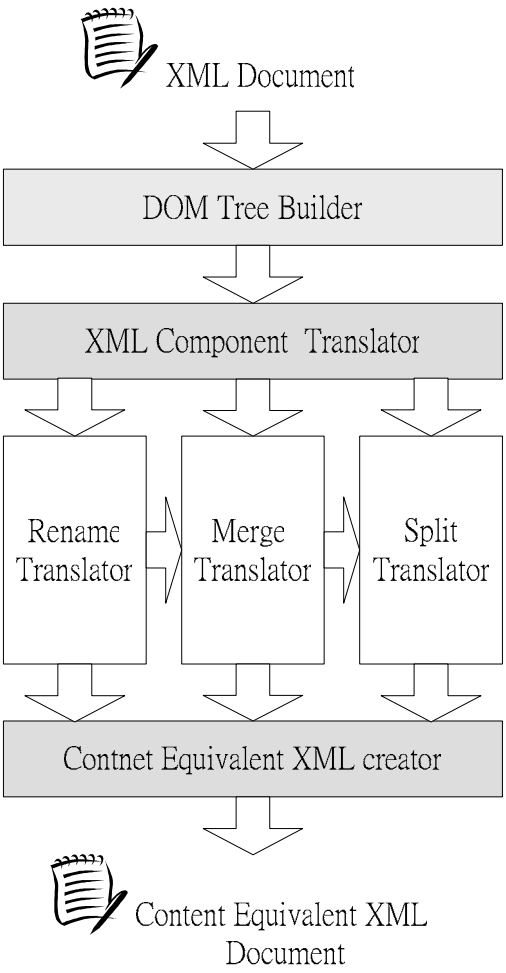


圖 4.1. XML 文件內容同等轉換架構與流程

XML Component Translator 將 DOM tree 轉換成 XML 元件樹。得到 XML 元件樹後，就可以執行同等轉換。圖 4.1 中，更名(Rename)、合併(Merge)與分割(Split)三個同等轉換，皆可單獨執行，並產生最終的內容同等(content equivalent)之 XML

文件。如若需要混合執行時，則必須依照先更名、再合併，最後再分割。當執行完分割後，即可進行輸出 XML 文件。過程中，同一種轉換必須一次進行完畢。例如我們要進行 3 次的合併轉換，則這三次的轉換必須一起做完，才能輸出或進行分割轉換。

在介紹範例前，我們先說明轉換資訊串接至屬性名稱後的方法。轉換資訊爲 x1.x2.x3.x4.x5 的一個字串，而在串接至屬性名稱後時，我們必須在他的前後加上一個旗標(flag)「-1111111-」，以減低與使用者命名的重複機率。如原本爲「class=java」，假設轉換資訊爲 0.1.500.1.2。則屬性名稱加上轉換資訊後，變爲「class-1111111-0.1.500.1.2-1111111-=java」。

4.2 轉換範例

在本節當中，我們將一份 XML 文件，依第三章所述的方式進行 XML 文件內容同等轉換。將其轉換成二份實體結構上不同，但是內容同等的另一份 XML 文件。首先我們先介紹範例所使用的文件型態定義(DTD)，請參考圖 4.2。

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT LIBRARY (BOOK*)>
<!ELEMENT BOOK (TITLE, VERSION, AUTHOR+, RELATEDBOOKS*, INTRODUCTION?)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT VERSION (#PCDATA)>
<!ELEMENT AUTHOR (FIRSTNAME, LASTNAME)>
<!ELEMENT FIRSTNAME (#PCDATA)>
<!ELEMENT MIDDLENAME (#PCDATA)>
<!ELEMENT LASTNAME (#PCDATA)>
<!ELEMENT RELATEDBOOKS (RELATEDBOOK*)>
<!ELEMENT RELATEDBOOK (R_TITLE, R_VERSION)>
<!ELEMENT R_TITLE (#PCDATA)>
<!ELEMENT R_VERSION (#PCDATA)>
<!ELEMENT INTRODUCTION (PARAGRAPH*)>
<!ELEMENT PARAGRAPH (SENTENCE*)>
<!ELEMENT SENTENCE (#PCDATA)>
<!ATTLIST BOOK CLASS CDATA #REQUIRED>
<!ATTLIST RELATEDBOOK CLASS CDATA #REQUIRED>
```

圖 4.2. 範例文件的 DTD

我們所選用的範例很簡單，是一個圖書館藏書資料所使用的 DTD。裡面所記錄的只有書籍，當中項目包括書籍名稱、書籍版本、作者、相關書籍以及此書籍的簡單簡介。而依據此 DTD，我們設計一份 XML 文件範例，此文件將作爲我

們進行文件同等轉換的來源檔，此份文件列於圖 4.3。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LIBRARY SYSTEM "example.dtd">
<LIBRARY>
    <BOOK CLASS="Java">
        <TITLE>Java Servlet Programming</TITLE>
        <VERSION>2nd Edition</VERSION>
        <AUTHOR>
            <FIRSTNAME>Jason</FIRSTNAME>
            <LASTNAME>Hunter</LASTNAME>
        </AUTHOR>
        <RELATEDBOOKS>
            <RELATEDBOOK CLASS="Java">
                <R_TITLE>JavaServer Pages</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
            <RELATEDBOOK CLASS="Web">
                <R_TITLE>Web Design in a Nutshell</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
        </RELATEDBOOKS>
        <INTRODUCTION>
            <PARAGRAPH>
                 <SENTENCE>Java Servlet Programming covers everything
                 Java developers need to know to write effective
                 servlets.
                 </SENTENCE>
                 <SENTENCE>The second edition has been completely
                 updated to cover the new features of Version 2.2 of
                 the Java Servlet API.
                 </SENTENCE>
            </PARAGRAPH>
        </INTRODUCTION>
    </BOOK>
    <BOOK CLASS="Java">
        <TITLE>Java and  XML</TITLE>
        <VERSION>2nd Edition</VERSION>
        <AUTHOR>
            <FIRSTNAME>Brett</FIRSTNAME>
            <LASTNAME>McLaughlin</LASTNAME>
        </AUTHOR>
        <RELATEDBOOKS>
            <RELATEDBOOK CLASS="Java">
                <R_TITLE>Java Servlet Programming</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
            <RELATEDBOOK CLASS="Java">
                <R_TITLE>JavaServer Pages</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
        </RELATEDBOOKS>
```

圖 4.3. XML 文件原始檔案

```
        <INTRODUCTION>
            <PARAGRAPH>
                <SENTENCE>This second edition of Java and  XML adds
                chapters on Advanced SAX and Advanced DOM, new
                chapters on SOAP and data binding, and new examples
                throughout.
                </SENTENCE>
                <SENTENCE>A concise chapter on XML basics introduces
                concepts, and the rest of the book focuses on using
                XML from your Java applications.
                </SENTENCE>
            </PARAGRAPH>
        </INTRODUCTION>
    </BOOK>
</LIBRARY>
```

圖 4.3. XML 文件原始檔案，續

依照此份 XML 文件，我們可以產生此份文件的 T_X，圖 4.4 即爲此 T_X 產生出來的示意圖。圖中我們只將 XML component 繪出，即可觀察其結構。其中我們在第三章中所提之實體結構 $S(T_X)$與內容結構 $L(T_X)$，它們在原始檔案當中是相等的。

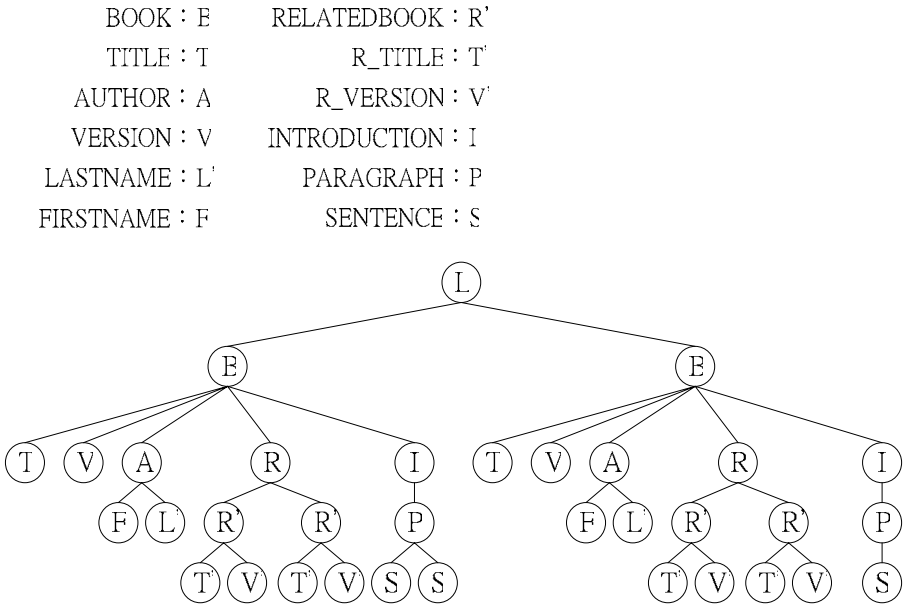


圖 4.4.原始檔案的 T_X 示意圖

接下來，將此份 XML 文件進行文件同等轉換，依序進行更名轉換、合併轉換、分割轉換等三步驟。

其中對 component TITLE 與 component BOOK 中的屬性 CLASS 進行更名轉

換，對 component AUTHOR 其下的 FIRSTNAME、LASTNAME 進行合併，最後對元素 SENTENCE 進行分割動作，我們對此三步驟逐一進行更深入的解釋。首先是更名轉換，截取圖 4.3 的部份 XML 文件來做說明，如圖 4.5 所示。其中我們更改 component TITLE 的元素名稱(element name)，依照更名轉換的演算法，首先找出 TITLE 所在，並將之換爲新的元素名稱 NAME，則圖 4.5 內容轉變爲圖 4.6 之內容。

```
<BOOK CLASS="Java">
    <TITLE>Java Servlet Programming</TITLE>
    <VERSION>2nd Edition</VERSION>
    .........
```

圖 4.5. 更名轉換前的部份 XML 文件

```
<BOOK CLASS="Java">
    <NAME>Java Servlet Programming</NAME>
    <VERSION>2nd Edition</VERSION>
    .........
```

圖 4.6. 元素更名轉換後的部份 XML 文件

接下來，對圖 4.6 中 component BOOK 的屬性 CLASS 進行更名，假設更名爲 TYPE。則內容即變爲圖 4.7 之內容。在更名轉換的過程中，並沒有更動到 component content，因此我們並沒有紀錄任何額外資訊來維持 component content 的一致性。

```
<BOOK TYPE="Java">
    <NAME>Java Servlet Programming</NAME>
    <VERSION>2nd Edition</VERSION>
    .........
```

圖 4.7. 屬性更名轉換後的部份 XML 文件

更名轉換完畢後，接著進行合併轉換，我們將 component AUTHOR 其下的 FIRSTNAME、LASTNAME 進行合併。在合併過程中，我們必須計算轉換資訊。其中，我們注意到，在所選擇進行轉換的 components 上，皆沒有屬性項目，因此不能對屬性作增加轉換資訊的動作。此時我們對沒有包含屬性的 component 創造一個空的屬性「_default=””」，用來與轉換資訊進行串接。轉換前內容顯示於圖 4.8。轉換後產生轉換資訊，並將轉換資訊附加在 component AUTHOR 當中。其中 FIRSTNAME 的轉換資訊爲 0.1.5.1.1，LASTNAME 的轉換資訊爲 0.1.6.2.2。

```
<AUTHOR>
    <FIRSTNAME>Jason</FIRSTNAME>
    <LASTNAME>Hunter</LASTNAME>
</AUTHOR>
```

圖 4.8. 合併轉換前的部份 XML 文件

將 FIRSTNAME 與 LASTNAME 之 text 合併，去掉 component FIRSTNAME 與 LASTNAME。最後我們得到圖 4.9 中之內容。

```
<AUTHOR _default-1111111-0.1.5.1.1-1111111=""
        _default-1111111-0.1.6.2.2-1111111="" >
    Jason Hunter
</AUTHOR>
```

圖 4.9. 合併轉換後的部份 XML 文件

最後，我們對 component SENTENCE 進行分割。將 SENTENCE 中的 text 依照空白的切割開來，並且不留下 text 於 component SENTENCE 當中，因 SENTENCE 並無屬性，因此我們同樣產生一預設屬性。其中分割出來的 component 我們給予名稱爲 WORD。圖 4.10 爲分割前、圖 4.11 則爲分割後之內。

```
<SENTENCE>
Java Servlet Programming covers everything Java developers need to know
to write effective servlets.
</SENTENCE>
```

圖 4.10. 分割轉換前的部份 XML 文件

```
<SENTENCE  _default-1111111-0.0.0.0.0-1111111="">
<WORD _default-1111111-1.0.0.0.0-1111111="">Java</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">Servlet</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">Programming</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">covers</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">everything</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">Java</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">developers</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">need</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">to</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">know</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">to</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">write</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">effective</WORD>
<WORD _default-1111111-1.0.0.0.0-1111111="">servlets.</WORD>
</SENTENCE>
```

圖 4.11. 分割轉換後的部份 XML 文件

經過上述的三項轉換，最後我們將原始的 XML 文件轉換成爲圖 4.12 中的新 XML 文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<LIBRARY>
    <BOOK TYPE="Java">
        <NAME>Java Servlet Programming</NAME>
            <VERSION>2nd Edition</VERSION>
            <AUTHOR _default-1111111-0.1.5.1.1-1111111=""
            _default-1111111-0.1.6.2.2-1111111=""
            _default-1111111-0.0.0.0.3-1111111="">
                Jason Hunter
        </AUTHOR>
        <RELATEDBOOKS>
            <RELATEDBOOK CLASS="Java">
                <R_TITLE>JavaServer Pages</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
            <RELATEDBOOK CLASS="Web">
                <R_TITLE>Web Design in a Nutshell</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
        </RELATEDBOOKS>
        <INTRODUCTION>
        <PARAGRAPH>
        <SENTENCE _default-1111111-0.0.0.0.0-1111111="">
        <WORD _default-1111111-1.0.0.0.0-1111111="">Java</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Servlet</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Programming</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">covers</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">everything</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Java</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">developers</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">need</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">to</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">know</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">to</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">write</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">effective</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">servlets.</WORD>
        </SENTENCE>
        <SENTENCE _default-1111111-0.0.0.0.0-1111111="">
        <WORD _default-1111111-1.0.0.0.0-1111111="">The</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">second</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">edition</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">has</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">been</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">completely</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">updated</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">to</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">cover</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">the</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">new</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">features</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">of</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Version</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">2.2</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">of</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">the</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Java</WORD>
```

圖 4.12. 同等轉換後所得新 XML 文件之一

```
        <WORD _default-1111111-1.0.0.0.0-1111111="">Servlet</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">API.</WORD>
        </SENTENCE>
        </PARAGRAPH>
        </INTRODUCTION>
    </BOOK>
    <BOOK TYPE="Java">
        <NAME>Java and  XML</NAME>
        <VERSION>2nd Edition</VERSION>
        <AUTHOR _default-1111111-0.1.5.1.1-1111111=""
        _default-1111111-0.1.10.2.2-1111111=""
        _default-1111111-0.0.0.0.3-1111111="">
            Brett McLaughlin
        </AUTHOR>
        <RELATEDBOOKS>
            <RELATEDBOOK CLASS="Java">
                <R_TITLE>Java Servlet Programming</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
            <RELATEDBOOK CLASS="Java">
                <R_TITLE>JavaServer Pages</R_TITLE>
                <R_VERSION>2nd Edition</R_VERSION>
            </RELATEDBOOK>
        </RELATEDBOOKS>
        <INTRODUCTION>
        <PARAGRAPH _default-1111111-0.0.0.0.0-1111111="">
        <SENTENCE>
        <WORD _default-1111111-1.0.0.0.0-1111111="">This</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">second</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">edition</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">of</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Java</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">and</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">XML</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">adds</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">chapters</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">on</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Advanced</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">SAX</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">and</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">Advanced</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">DOM,</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">new</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">chapters</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">on</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">SOAP</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">and</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">data</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">binding,</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">and</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">new</WORD>
        <WORD _default-1111111-1.0.0.0.0-1111111="">examples</WORD>
        <WORD
        _default-1111111-1.0.0.0.0-1111111="">throughout.</WORD>
        </SENTENCE>
        </PARAGRAPH>
        </INTRODUCTION>
    </BOOK>
</LIBRARY>
```

圖 4.12. 同等轉換後所得新 XML 文件之一，續

而此份新 XML 文件建立出來的 T_X 顯示於圖 4.13。

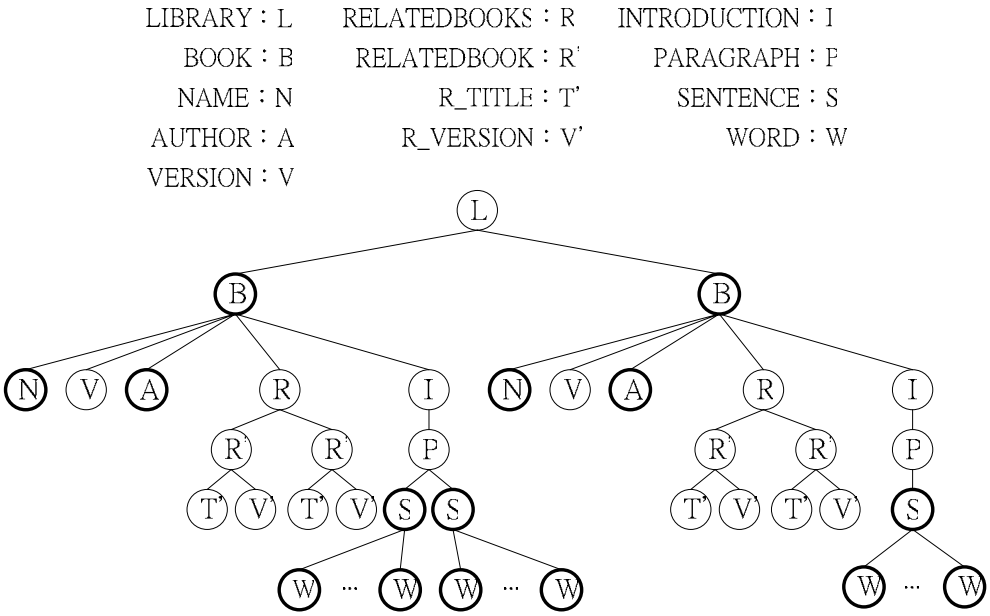


圖 4.13. 新 XML 文件的 T_X 示意圖

同樣的，我們可以透過另一組同等轉換過程得到另一個新的 XML 文件。此次轉換當中，對 component TITLE 進行更名轉換，使其變爲 NAME。對 component PARAGRAPH 其下的 component SENTENCE 與 component RELATEDBOOK 下的 R_TITLE 及 R_VERSION 個別進行合併轉換。最後再對合併過後的 component PARAGRAPH 進行分割轉換，直接將其分割成 compoennt WORD。詳細轉換過程與上述範例類似，在此不在詳述。圖 4.14 爲第二次同等轉換過程後所產生的 XML 文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<LIBRARY>
    <BOOK CLASS="Java">
        <NAME>Java Servlet Programming</NAME>
        <VERSION>2nd Edition</VERSION>
        <AUTHOR>
            <FIRSTNAME>Jason</FIRSTNAME>
            <LASTNAME>Hunter</LASTNAME>
        </AUTHOR>
        <RELATEDBOOKS>
            <RELATEDBOOK _default-1111111-0.1.16.1.1-1111111=""
            _default-1111111-0.1.11.2.2-1111111=""
            CLASS-1111111-0.0.0.0.3-1111111="Java">
                JavaServer Pages 2nd Edition
            </RELATEDBOOK>
```

圖 4.14 同等轉換後所得新 XML 文件之二

```
            <RELATEDBOOK _default-1111111-0.1.24.1.1-1111111=""
            _default-1111111-0.1.11.2.2-1111111=""
            CLASS-1111111-0.0.0.0.3-1111111="Web">
                Web Design in a Nutshell 2nd Edition
            </RELATEDBOOK>
        </RELATEDBOOKS>
        <INTRODUCTION>
        <PARAGRAPH _default-1111111-0.1.100.1.1-1111111=""
        _default-1111111-0.1.112.2.2-1111111=""
        _default-1111111-0.0.0.0.3-1111111="">
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Java</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Servlet</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Programming</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">covers</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">everything</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Java</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">developers</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">need</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">to</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">know</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">to</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">write</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">effective</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">servlets.</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">The</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">second</WORD>
```

圖 4.14 同等轉換後所得新 XML 文件之二，續

```
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">edition</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">has</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">been</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">completely</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">updated</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">to</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">cover</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">the</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">new</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">features</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">of</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Version</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">2.2</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">of</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">the</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Java</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">Servlet</WORD>
        <WORD _default-1111111-1.1.100.1.1-1111111=""
        _default-1111111-1.1.112.2.2-1111111=""
        _default-1111111-1.0.0.0.3-1111111="">API.</WORD>
        </PARAGRAPH>
        </INTRODUCTION>
    </BOOK>
    <BOOK CLASS="Java">
```

圖 4.14 同等轉換後所得新 XML 文件之二，續

```
        <NAME>Java and  XML</NAME>
        <VERSION>2nd Edition</VERSION>
        <AUTHOR>
            <FIRSTNAME>Brett</FIRSTNAME>
            <LASTNAME>McLaughlin</LASTNAME>
        </AUTHOR>
        <RELATEDBOOKS>
            <RELATEDBOOK _default-1111111-0.1.24.1.1-1111111=""
            _default-1111111-0.1.11.2.2-1111111=""
            CLASS-1111111-0.0.0.0.3-1111111="Java">
                Java Servlet Programming 2nd Edition
            </RELATEDBOOK>
            <RELATEDBOOK _default-1111111-0.1.16.1.1-1111111=""
            _default-1111111-0.1.11.2.2-1111111=""
            CLASS-1111111-0.0.0.0.3-1111111="Java">
                JavaServer Pages 2nd Edition
            </RELATEDBOOK>
        </RELATEDBOOKS>
        <INTRODUCTION>
        <PARAGRAPH _default-1111111-0.1.151.1.1-1111111=""
        _default-1111111-0.0.0.0.2-1111111="">
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">This</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">second</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">edition</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">of</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">Java</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">and</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">XML</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">adds</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">chapters</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">on</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">Advanced</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">SAX</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">and</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">Advanced</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">DOM,</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">new</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">chapters</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">on</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">SOAP</WORD>
```

圖 4.14 同等轉換後所得新 XML 文件之二，續

```
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">and</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">data</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">binding,</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">and</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">new</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">examples</WORD>
        <WORD _default-1111111-1.1.151.1.1-1111111=""
        _default-1111111-1.0.0.0.2-1111111="">throughout.</WORD>
        </PARAGRAPH>
        </INTRODUCTION>
    </BOOK>
</LIBRARY>
```

圖 4.14 同等轉換後所得新 XML 文件之二，續

由第二份新 XML 文件所建立出來的 T_X，顯示意於圖 4.15 中。

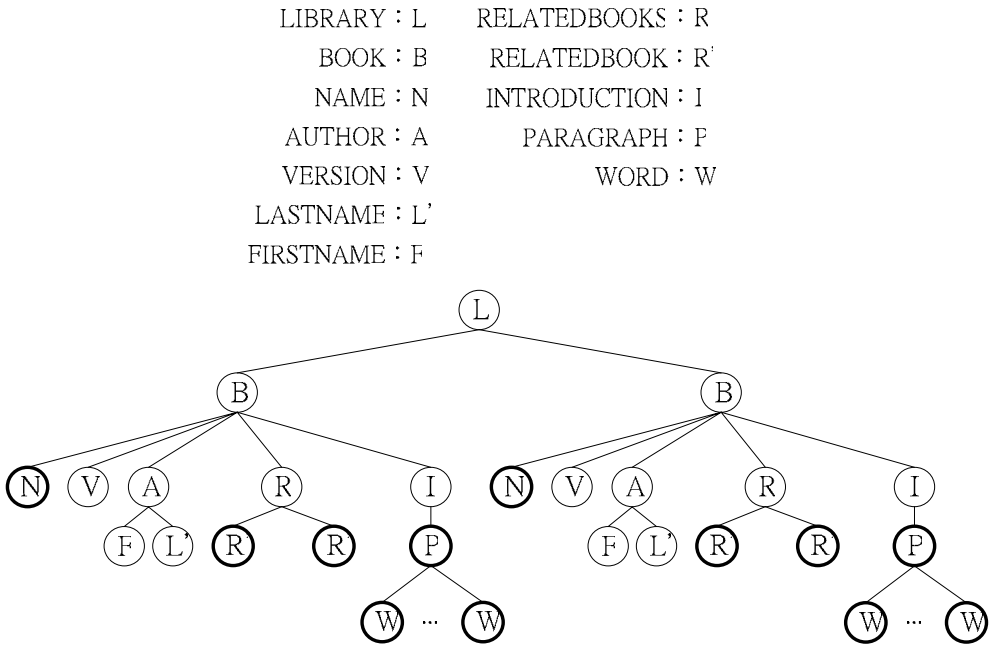


圖 4.15. 第二份新 XML 文件的 T_X 示意圖

在得到二個新 T_X 後，來做個比較。透過圖 4.13、圖 4.15 與原來的圖 4.4 相比較，發現經過同等轉換後所得到的 XML 文件實體結構，明顯與原來不同。但實際上，這些 XML 文件皆可以經由轉換資訊來計算取回原本之內容結構(圖 4.4)，因而保持它們之間的內容同等。

4.3 總結

在本章中，我們以一份 XML 文件作爲範例，經由多次轉換產出不同結構的 XML 文件。雖然這些 XML 文件結構不一，但其內容則仍能維持同等關係。

第五章 結論及未來研究方向

5.1 結論

本論文針對 XML 文件相等的概念作出不同的詮釋，在過去的許多研究 XML 相等與相似的報告中，大部分的學者專家針對 XML 文件的相等，偏重於實體結構上的相等，而非內容的相等。本論文提出一個內容同等(content equivalent)的概念，透過特定的轉換方式，將一份 XML 文件經過內容同等轉換(content equivalent transformation)得到另一份 XML 文件。雖然它們在實體結構上不相同，但是在內容結構上，則是代表相同的文件內容。本論文中介紹三種同等轉換，它們分別是：更名轉換、合併轉換與分割轉換。我們同時發展此三種轉換的對應實作方法，並證明這些轉換有達到內容同等轉換的定義。本論文最後並以一個範例來展現這些轉換機制所達到的效果。

5.2 未來研究方向

以下是本論文未來可行研究方向的探討：

1. 同等轉換方式的增加：在本論文中，只提出更名、合併與分割三種內容同等轉換。但對一份 XML 文件之同等轉換，仍有不足之處。因此未來研究可以增加新的轉換方式，以應用到更廣的領域。

2. 轉換限制的去除：本論文中，同等轉換上有使用順序的限制，必須先更名、再合併，最後分割。目前無法去除這樣的轉換限制，這是另一項不足之處。所以去除轉換使用順序，是未來研究的可行方向之一。

3. 轉換資訊(Transformation Information)的處理：以目前狀況來看，在不增加新檔案的前提下(以 XSLT 同樣可以進行內容同等轉換，但是就會多出一個 XSL 文件)，將轉換資訊串接在屬性後，是不錯的選擇。但是必須對進行轉換的 component 中每一個屬性增加轉換資訊。當沒有屬性時，則必須產生一個空的屬性。這些動作對轉換上是一個很大的負擔。因此，在未來的研究中，如何減低增加轉換資訊所構成的負擔，是重要的研究課題之一。

4. XML 文件相似度計算的應用：目前大部分的研究文獻中，XML 文件相似度

計算仍以透過利用 tree to tree editing 的延伸應用，來計算二份 XML 文件間相似度或找出二份 XML 文件的不同處。本論文成果應可用於分析二份 XML 文件的相似度計算。

有了上述改進與應用，相信在未來的研究中，XML 文件內容同等的概念，將能被應用到更廣的領域裡。

參考文獻

[1] S. Chawathe, " Comparing hierarchical data in extended memory.", In Proceedings of 25th VLDB, pp.90-101, 1999

[2] S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom, " Change detection in hierarchically structured information.", In Proceedings of ACM SIGMOD, pp.493-504, 1996

[3] S. Chawathe, and H. Garcia-Molina, " Meaningful change detection in structured data.", In Proceedings of ACM SIGMOD, pp.26-37, 1997

[4] G. Cobena, S. Abiteboul, and Marian, A. (2002), "Detecting Change in XML Documents.", 18th International Conference on Data Engineering, pp.41-52, 2002

[5] "Document Object Model (DOM) Level 1 Specification (Second Edition)", World Wide Web Consortium, http://www.w3c.org/TR/2000/WD-DOM-Level-1-20000929/

[6] "ebXML", Organization for the Advancement of Structured Information Standards(OASIS), http://www.ebxml.org/

[7] "Extensible Markup Language (XML) 1.0 (Second Edition)", World Wide Web Consortium, http://www.w3.org/TR/REC-xml

[8] S. Flesca, G. Manco, E. Masciarim, L. Pontieri and A. Pugliese, "Detecting Structural Similarities between XML Documents.", Fifth International Workshop on the Web and Databases, 2002

[9] A. Nierman and H.V. Jagadish, "Evaluating Structural Similarity in XML Documents.", Fifth International Workshop on the Web and Databases, 2002

[10] M. Selkow, "The Tree-To-Tree Editing Problem.", Information Processing Letters, Volume.6, No.6, pp.184-186, 1977

[11] Y. Wang, , D.J. DeWitt and J. Cai, "X-Diff: An Effective Change Detection Algorithm for XML Document.", 19th International Conference on Data Engineering, pp.519-530, 2003

[12] "Web Services Activity", World Wide Web Consortium, http://www.w3.org/2002/ws/

[13] "XML Metadata Interchange", Object Management Group, http://www.omg.org/technology/documents/formal/xmi.htm

[14]"XSL Transformations (XSLT) Version 1.0", World Wide Web Consortium, http://www.w3.org/TR/xslt