일반논문-07-12-2-10

# MPEG-7 BiM 부호화기 및 복호화기의 구현

염 지 현[a], 김 혁 만[a]‡, 김 민 제[b], 이 한 규[b]

# Implementation of Encoder and Decoder for MPEG-7 BiM

Jihyeon Yeom[a], Hyeokman Kim[a]‡, Minje Kim[b], and Hankyu Lee[b]

요 약

본 논문은 MPEG-7에서 표준화한 BiM 부호화 방식을 이용하여, 특정 스키마 문서에 따라 작성된 XML 인스턴스 문서를 이진 형태로 부호화하고 또한 역으로 복호화 하는 소프트웨어 시스템의 구현에 관한 것이다. 본 논문에서는 BiM 부호화기 및 복호화기의 소프트웨어 구조를 클래스 계층구조로 설계하고, 설계한 BiM 부호화기 및 복호화기를 구현한다. 구현된 BiM 부호화기는 평균 90%에 해당하는 부호화 효율을 보였다. BiM 부호화기는 MPEG-7 스키마 문서 뿐만 아니라 XML Schema로 정의된 스키마 문서에 따르는 어떤 인스턴스 문서도 부호화할 수 있는 범용 소프트웨어로써, 디지털 방송을 포함한 XML 인스턴스 문서의 부호화가 필요한 많은 응용 분야에서 사용 될 수 있다.

Abstract

In the paper, we implemented a software system that encodes XML instance documents conforming to a schema document according to the MPEG-7 BiM compression method, and decodes the encoded documents vice versa. We designed software structures of BiM encoder and decoder as class hierarchies, and then implemented the structures. The implemented BiM encoder shows a compression ratio of 9.44% on the average. The BiM encoder is a general-purpose XML compressor that can encode any instance documents conforming to a schema document described in XML Schema language including the MPEG-7 schema. The BiM encoder thus can be used in many application fields including digital broadcasting environment, where encoding XML instance documents is needed.

Keywords : XML, MPEG-7, BiM, 부호화기, 복호화기

## Ⅰ. 서 론

최근 인터넷뿐만 아니라, 방송 환경에서도 데이터 교환 및 표현의 표준으로 XML(Extensible Markup Language)이 부각되고 있다. XML 인스턴스(instance) 문서는 텍스트 포맷으로써, 태그를 사용하여 데이터를 구별하여 표현한다. 따라서, 인스턴스 문서는 일반적인 텍스트 파일과 비교하면, 그 크기가 매우 크다. 인터넷 환경에서는 인스턴스 문서를 그대로 전송하더라도 그 크기가 문제가 되지 않을 수도 있지만, 디지털 방송 환경에서는 대역폭의 제약으로 텍스트 포맷의 인스턴스 문서를 그대로 전송하는 것은 비

a) 국민대학교 컴퓨터공학부
School of Computer Science, Kookmin University
b) 한국전자통신연구원 전파방송연구단 방송미디어연구그룹
Broadcasting Media Research Group, Radio & Broadcasting Research Division, ETRI
‡ 교신저자: 김혁만(hmkim@kookmin.ac.kr)

효율적이다. 따라서, 인스턴스 문서가 지니는 비정규적인 구조와 장황함 때문에, 인스턴스 문서의 텍스트 포맷을 바이너리 포맷으로 이진 부호화하는 방법이 필요하다. 대표적인 텍스트 파일 부호화 기법으로는 zlib를 이용한 gzip 부호화 방법이 사용되고 있다[1]. 하지만 이 방법은 XML 인스턴스 문서에 포함된 태그들도 일반 텍스트로 처리하기 때문에, XML 문서의 부호화 효율을 높이기 위해서는 XML 문서 부호화를 위한 별도의 방법이 필요하다. 이에 따라, XML 인스턴스 문서의 부호화 기법에 관한 여러 연구가 진행되어 왔다.

XML 문서 부호화 방법은 크게 스키마 문서를 이용하지 않는 방법과 이용하는 방법으로 나눌 수 있다. 전자의 경우에는 대표적인 방법이 XMill이다. XMill은 사전(dictionary) 부호화 기법을 통해 부호화하며 gzip 보다는 인스턴스 문서 부호화에 효율적이라고 알려져 있다[2]. 한편, 인스턴스 문서 부호화 시 스키마 문서를 이용하게 되면, 인스턴스 문서를 통해 시멘틱 구조를 정확하게 알 수 있으므로 더 효율적인 부호화가 가능하다. 뿐만 아니라 스키마 문서의 구조를 이용하여 인스턴스 문서를 부분적으로 부호화하거나, 부호화된 인스턴스 문서에 대한 질의 혹은 부분적 수정 또한 가능할 수 있다. 스키마 문서는 DTD 혹은 XML Schema로 기술하므로, 스키마 문서를 어떤 언어로 기술하느냐에 따라 다른 방법이 존재한다. DTD로 기술한 스키마 문서에 따라 작성된 인스턴스 문서를 부호화하는 대표적인 방법으로 XGrid가 있다[3]. XML Schema로 기술된 스키마 문서에 따라 작성된 인스턴스 문서를 부호화하는 대표적인 방법으로는 BiM이 있다. BiM은 MPEG 그룹에서 MPEG-7 국제 표준의 일부로 표준화한 부호화 방식으로, 원래는 XML Schema로 기술된 MPEG-7 스키마 문서에 따라 작성된 인스턴스 문서를 부호화하기 위한 것이었으나, 표준 제정 과정에서 MPEG-7 스키마 문서에 따르는 인스턴스 문서뿐만 아니라, XML Schema로 기술한 스키마 문서에 따라는 어떤 인스턴스 문서도 부호화할 수 있는 범용 부호화 기법으로 확장되었다[4].

본 논문에서는 XML 인스턴스 문서를 보다 효율적으로 부호화하는 기법으로 MPEG Forum에서 표준화한 MPEG-7 BiM을 구현하였다. 본 논문의 구성은 다음과 같다. 2장에서는 인스턴스 문서를 부호화하는 방법에 대해서 소개한다. 3장에서는 BiM 부호화의 원리에 대해서 서술하고, 4장에서는 BiM 부/복호화기의 구조를 설계한다. 5장에서는 설계한 구조를 어떻게 구현하였는지를 서술하며, 6장에서는 구현한 BiM 부호화기의 성능을 실험한다. 그리고 마지막으로 7장에서는 결론 및 추후 연구에 대해 서술한다.

## Ⅱ. XML 인스턴스 문서의 부호화 방법

일반적으로 인스턴스 문서의 구성을 보면, 태그와 같은 구조 정보와 순수 데이터의 비율은 1:4 정도로, 구조 정보가 인스턴스 문서의 크기에 대부분을 차지한다. 따라서, 인스턴스 문서를 부호화하고자 할 때, 태그를 얼마만큼 효과적으로 부호화할 수 있느냐가 관건이다.

인스턴스 문서를 부호화하는 방법은 태그를 부호화하는 알고리즘에 따라 gzip 부호화 기법, 사전 부호화 기법, 오토마타를 이용한 기법으로 나눌 수 있다.

### 1. gzip 부호화 기법

일반적인 텍스트 파일 부호화 기법으로 대표적인 것은 zlib을 이용한 gzip부호화 방법이다[1]. Zlib은 Jean-loup Gailly와 Mark Adler가 고안한 데이터 부호화 알고리즘으로, 디플레이트(deflate) 알고리즘을 사용한다. 디플레이트는 LZ77 알고리즘[5]과 허프만 코딩[6][7]의 혼합된 형태로 무손실 데이터 부호화 알고리즘이다. 즉, LZ77 알고리즘을 사용하여 중복된 데이터 스트링을 제거하고, 중복 제거된 데이터의 이진 코드 길이를 줄이기 위하여 허프만 코딩을 적용함으로써 데이터의 크기를 줄인다. 일반적으로 40-50% 이상의 부호화 효율을 보이는 gzip은 인스턴스 문서에 포함된 태그들도 일반 텍스트로 처리하기 때문에, 인스턴스 문서의 부호화 효율을 높이기 위해서는 별도의 방법이 필요하다.

## 2. 사전 부호화 기법

인스턴스 문서의 태그들은 많은 경우에 반복되는 패턴들로 이루어질 수 있다. 이 경우, 자주 발생하는 태그에 대한 사전 테이블을 구성하여, 태그를 사전 테이블에 부여된 특정 이진 코드로 변환함으로써 데이터의 크기를 줄일 수 있다. 즉, 사전 부호화 방법은 새로운 단어마다 다른 정수를 부여하여 데이터를 변환한다. 그러나, 사전 부호화에서 부여한 정수들 간의 크기는 본래 데이터의 크기와 다르므로 본래 데이터들로의 복원이 필요하다[3].

사전 부호화 기법을 사용하여 인스턴스 문서를 부호화하는 방법은 크게 스키마 문서를 이용하지 않는 방법과 이용하는 방법으로 나눌 수 있다. 전자의 경우의 대표적인 방법으로 XMill이 있으며, DTD로 기술한 스키마 문서에 따라 작성된 인스턴스 문서를 부호화하는 대표적인 방법으로 XGrind가 있다.

### 2.1 XMill

XMill은 스키마 문서를 이용하지 않고 인스턴스 문서를 부호화하는 대표적인 방법으로, 1999년에 Hartmut Liekfe와 Dan Suciu에 의해 개발되었고, AT&T 연구소에서 고안되었다[2].

XMill은 그림 1과 같이 순수 데이터를 구조 정보(structure)와 구분하고, 재 그룹화 전략(regrouping strategy)에 의해 의미상 관련 있는 데이터들을 컨테이너 별로 분류한 후, 각 컨테이너에 최적화된 데이터 부호화 방법을 적용하여 부호화한다.

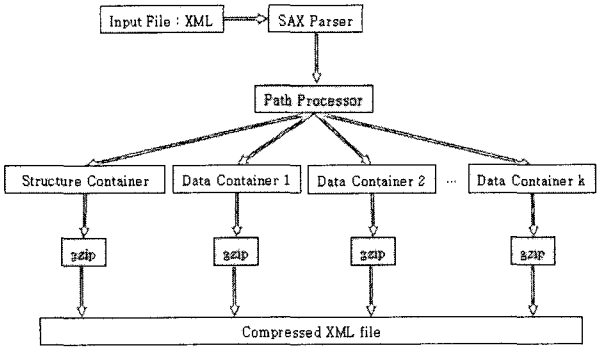



그림 1. XMill의 전체 구조
Fig. 1. Structure of XMill

일반적으로 XMill은 평균적으로 80% 이상의 부호화 효율을 보이며, gzip 보다는 인스턴스 문서 부호화에 효율적이라고 알려져 있다[2]. 그러나 XMill의 입력 인스턴스 문서의 크기가 20K 바이트보다 작을 경우, gzip 보다 뚜렷한 효과를 얻지 못하며, XMill은 부호화된 인스턴스 문서를 질의하거나 갱신할 수 없는 단점이 있다[9].

### 2.2 XGrind

XGrind는 DTD로 기술한 스키마 문서에 따라 작성된 인스턴스 문서를 부호화하는 대표적인 방법으로, 2002년에 Pankai M.Tolani와 Jayant R.Haritsa에 의해 개발되었다[3].

XGrind는 XMill과 유사하게 기본적으로 구조 정보와 데이터를 분리하여 부호화한다. 그림 2와 같이 DTD로 기술된 스키마를 통해 구성한 심볼 테이블(symbol table)과 인스턴스 문서 내의 엘리먼트와 애트리뷰트의 사용 빈도수를 나타낸 빈도 테이블(frequency table)을 기반하여 데이터 값은 허프만 코딩 기법으로, 태그는 사전 부호화 기법으로 부호화한다.

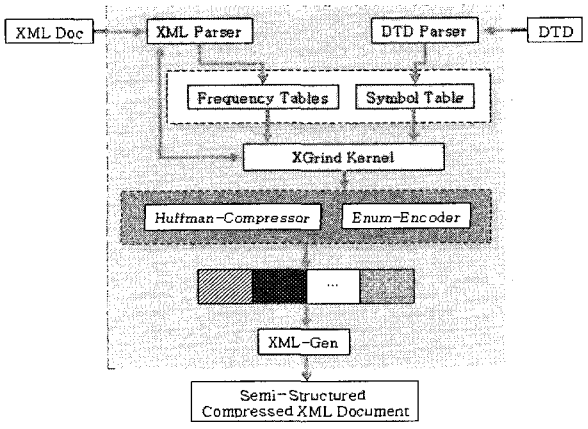



그림 2. XGrind의 전체 구조
Fig. 2. Structure of XGrind

XGrind는 부호화된 인스턴스 문서를 복호화 하지 않고 질의(query) 하는 것을 지원하며 본래의 인스턴스 문서의 구조를 얻을 수 있다. 따라서, XGrind는 이러한 특징으로

팜탑(palm-top) 컴퓨터와 같은 자원이 제한적인 처리 디바이스에서 유용한 애플리케이션이다.

### 2.3 오토마타를 이용한 기법

오토마타란 디지털 컴퓨터의 수학적인 모델인 오토마톤(automaton)의 복수형으로서, 자동 기계 장치란 뜻을 가진다. 이것은 입력 장치, 출력 장치, 저장 장치, 제어 장치를 가지고 있으므로 현대적인 디지털 컴퓨터가 작동하는 이론적인 메커니즘이라 볼 수 있다.

오토마타는 수학적 방법론에 바탕을 둔 디지털 컴퓨터의 추상적인 모델이므로 오토마타는 다음과 같은 필수적인 특징들을 가지고 있다. 첫째, 오토마타는 입력 데이터를 읽을 수 있는 기능을 가지고 있다. 입력 데이터는 입력 파일에 쓰여져 있는 알파벳상의 스트링으로 이루어져 있다. 둘째, 오토마타는 특정 형태의 출력 기능을 가지고 있다. 즉, '0'이나 '1'의 출력을 낼 수 있다. 셋째, 오토마타는 임시 저장 장치(storage device)를 가질 수 있으며, 넷째, 유한 개의 내부 상태(internal states)를 제어할 수 있는 제어 장치(control unit)을 가지고 있다. 이것의 제어에 따라 상태가 변화될 수 있다. 이와 같은 원리의 오토마타를 인스턴스 문서를 부호화하기 위한 방법으로 사용할 수 있다.

XML 스키마로 기술된 스키마 문서에 따라 오토마타를 구성하여, 생성된 오토마타를 사용하여 인스턴스 문서를 부호화하는 대표적인 방법으로 BiM이 있다. BiM에서는 태그들을 그림 3과 같이 스키마 문서를 이용하여 미리 구성한 오토마타를 이용해 부호화하며, 데이터는 zlilb로 부호화한다[4].

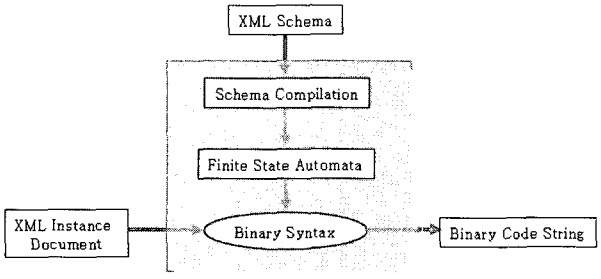



그림 3. BiM의 부호화 과정
Fig. 3. Encoding process of BiM

XML 스키마는 DTD보다 정교하게 스키마 문서의 구조를 기술할 수 있기 때문에, 최근에 많은 기업이나 국제 기구에서 그 사용이 확산되고 있다. 특히, 멀티미디어 메타데이터 표준인 MPEG-7/21/101과 맞춤형 방송 표준인 TV-Anytime 스키마 문서가 XML 스키마로 기술될 뿐만 아니라, 이들의 메타데이터 부호화 방법으로 BiM이 표준으로 채택되면서 그 중요성이 더욱 커지고 있다[8]. 따라서, 본 논문에서는 오토마타를 사용하여 인스턴스 문서를 부호화하는 MPEG-7 BiM의 구조를 설계하고 구현하였다.

## Ⅲ. BiM 부호화의 원리

BiM 부호화기는 스키마 문서의 정의에 따르는 오토마타를 사용하여 입력인 인스턴스 문서를 이진 스트링으로 부호화한다. 부호화하는 방법은 엘리먼트의 바인딩 타입(binding type)의 종류에 따라 다르다. 엘리먼트가 단순 타입(simple type)으로 정의된 경우에는 단순 타입의 종류에 따라 특정 부호화 코덱을 사용한다. 예를 들어 일반 텍스트는 zlib를 사용하고, 날짜와 시간에 해당하는 텍스트 스트링은 비트 스트링으로 변환하여 크기를 줄인다. 그러나 엘리먼트가 복합 타입(complex type)으로 정의된 경우는 FSA(Finite State Automata)를 사용하여 부호화한다. 스키마 문서에서 정의되는 대부분의 엘리먼트는 복합 타입으로 정의된다. 전자의 경우는 부호화 방법이 매우 잘 알려져 있으므로, 여기서는 FSA를 사용한 복합 타입의 부호화 원리에 대해서만 설명 하도록 한다.

FSA는 그림 4와 같이 상태(state)와 전이(transition)로 구성된다[4]. FSA의 상태는 입력 데이터를 내부 처리 규칙에 따라 결과 값을 도출 및 저장하고, 전이는 상태의 변화 및 그 전이를 이동하게 하기 위해 수행될 필요가 있을 조건 등을 설명한다. 상태에는 단순 상태(simple state)와 타입 상태(type state)가 있으며, 전이는 엘리먼트 전이(element transition), 루프 전이(loop transition), 코드 전이(code transition) 및 단순 전이(simple transition)로 나누어진다. 루프 전이에는 루프 시작 전이(loop start transition), 루프 연속 전이(loop continue transition), 루프 끝 전이(loop end

transition)가있다. 코드 전이에는 전환 전이(shunt transition)가 있으며, 코드 전이의 부여된 비트 값으로 부호화한다. 루프 전이는 반복 횟수를 정의할 수 있는 세 가지 XML 스키마 컴포넌트, 즉 엘리먼트, 지명 모델 그룹(named model group), 컴포지터(compositor) 등을 오토마타로 표현 할 때 사용한다. 이 때, 반복 횟수가 '0'이면 루프 전이 대신 전환 전이를 사용하여 나타낸다. 단순 상태와 단순 전이는 오토마타를 구성하기 위하여 사용하며, 이진 부호화 결과에는 영향을 미치지 않는다.

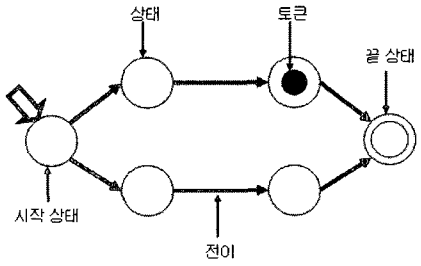



그림 4. Finite State Automata 구성
Fig. 4. Composition of the finite state automata)

임의의 엘리먼트에 대한 FSA는반복 횟수에 따라서, 표 1과 같이 구성된다. 표 1의 ①번 경우와 같이 최소 반복 횟수(minOccurs)가 1이고, 최대 반복 횟수(maxOccurs)가 1일 경우, FSA는 엘리먼트 전이가 시작 상태와 끝 상태를 연결하는 형태를 이룬다. 이때 끝 상태는 엘리먼트의 바인딩 타입의 타입 상태가 되며, 이러한 형태가 하나의 엘리먼트를 부호화하기 위한 FSA의 기본 구조이다. 표 1의 ②번 경우와 같이 최소 반복 횟수가 1이고, 최대 반복 횟수가 "unbounded"일 때, ①번의 기본 구조에 새로운 시작 상태와 끝 상태를 추가 한다. 그리고 코드 전이를 사용하여 새로운 시작 상태와 이전 시작 상태를 연결하고, 이전 끝 상태와 새로운 끝 상태를 루프 끝 전이를 사용하여 연결한다. 이때 코드 전이에는 이진 코드값 "1"을 할당한다. 또한, 이 엘리먼트가 무제한 반복될 수 있음을 표현하기 위해 이전 끝 상태와 이전 시작 상태를 루프 연속 전이를 이용하여 연결한다. 표 1의 ③번 경우와 같이 최소 반복 횟수가 0이고, 최대 반복 횟수가 1일 경우에는, ①의 기본 구조에 ②번과 비슷하게 새로운 시작 상태와 끝 상태를 추가 한다. 또한 최소 반복 횟수가 0임으로 엘리먼트가 생략될 수 있음을 표현하기 위해 새로운 시작 상태와 새로운 끝 상태를 코드 전이의 한 종류인 전환 전이를 사용하여 연결하고, 전환 전이에 이진 코드 값 "0"을 할당한다. 마지막으로 표 1의 ④번 경우와 같이 최소 반복 횟수가 0이고 최대 반복 횟수가 "unbounded"일 때는, 위의 ②, ③번 과정과 같이 새로운 시작 상태와 끝 상태를 추가 하고, 엘리먼트가 무제한 반복될 수 있음을 표현하기 위해 ②번과 같이 이전 끝 상태와 이전 시작 상태를 루프 전이로 연결한다. 그리고 ③과 같이 엘리먼트가 생략될 수 있음을 표현하기 위해 새로운 시작 상태

표 1. 반복 횟수에 따른 엘리먼트 FSA의 구성
Table 1. Composition of Element FSA according to occurrences

| 종류 | 반복 횟수 | | Element FSA |
|---|---|---|---|
| | 최소 | 최대 | |
| ① | 1 | 1 | |
| ② | 1 | unbounded | |
| ③ | 0 | 1 | |
| ④ | 0 | unbounded | |

* 기호 설명

단순 상태 코드 전이 루프 전이

타입 상태 엘리먼트 전이 단순 전이

와 끝 상태를 전환 전이를 사용하여 연결하고, 전환 전이에 이진 코드 값 "0"을 할당한다.

위와 같은 엘리먼트 FSA 구성 원리에 따라, 스키마 문서 전체에 대한 FSA는 스키마에서 엘리먼트 정의 시 중첩된 순서 및 나열 순서에 따라 차례대로 진행되어 해당 엘리먼트 FSA가 문서 전체에 대한 FSA에 하나씩 추가되며 구성된다. 예를 들어, 아래의 예제 스키마 문서는 한 개의 전역 엘리먼트 AElement와 바인딩 타입 AElementType을 정의한다. AElementType은 세 개의 자식 엘리먼트를 sequence 컴포지터를 사용하여 포함한다. 이와 같은 예제 스키마 문서를 토대로 문서의 루트인 AElement의 FSA를 구성하면 그림 5

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:bim="http://db.kookmin.ac.kr"
 targetNamespace="http://db.kookmin.ac.kr" elementFormDefault="qualified" attributeFormDefault="unqualified">
   <element name="AElement" type="bim:AElementType" />
   <complexType name="AElementType">
     <sequence>
       <element name="BElement" type="string" maxOccurs="unbounded"/>
       <element name="CElement" type="string" minOccurs="0"/>
       <element name="DElement" type="string"/>
     </sequence>
   </complexType>
</schema>
```

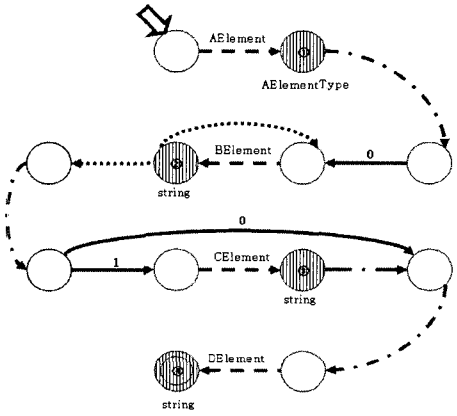



그림 5. 예제 스키마 문서의 FSA
Fig. 5. FSA of example schema document

```
<?xml version="1.0" encoding="UTF-8"?>
<AElement xmlns="http://db.kookmin.ac.kr" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://db.kookmin.ac.kr .₩SampleSchema.xsd">
   <BElement>1st BElement</BElement>
   <BElement>2nd BElement</BElement>
   <DElement>1st DElement</DElement>
</AElement>
```

표 2. 예제 인스턴스 문서의 부호화 결과
Table 2. Encoding result of example instance document

| 의미 | 코드 전이 | 반복 횟수 | 1 | s | t | 공백 | B |
|---|---|---|---|---|---|---|---|
| 이진 스트링 | 1 | 00010 | 00110001 | 01110011 | 01110100 | 00100000 | 01000010 |
| E | L | e | m | e | n | t | 2 |
| 01000101 | 01101100 | 01100101 | 01101101 | 01100101 | 01101110 | 01110100 | 00110010 |
| n | D | 공백 | B | E | I | e | M |
| 01101110 | 01100100 | 00100000 | 01000010 | 01000101 | 01101100 | 01100101 | 01101101 |
| e | N | t | 코드 전이 | 1 | s | t | 공백 |
| 01100101 | 01101110 | 01110100 | 0 | 00110001 | 01110011 | 01110100 | 00100000 |
| D | E | I | e | m | e | n | T |
| 01000100 | 01000101 | 01101100 | 01100101 | 01100101 | 01100101 | 01101110 | 01110100 |

와 같다. 굵은 화살표가 가리키는 것이 시작 상태이고, 두 겹의 원은 끝 상태를 나타낸다. 그림 5에서 알 수 있듯이 AElement는 최소 반복 횟수가 1이고, 최대 반복 횟수가 1이기 때문에, 표 1의 ①번처럼 구성되고, AElementType은 세 개 자식 엘리먼트를 포함하므로 AElementType에 대한FSA는 각각 세 개의 자식 엘리먼트 FSA를 연결하여 구성된다. 즉, BElement는 최소 반복 횟수가 1이고, 최대 반복 횟수가 "unbounded"이므로, 표 1의 ②번과 같이 구성되고, CElement는 최소 반복 횟수가 0이고, 최대 반복 횟수가 1이기 때문에, 표 1의 ③번처럼 나타낼 수 있다. 그리고, DElement는 반복 횟수의 기본 값인 최소 반복 횟수가 1, 최대 반복 횟수가 1이므로 표 1의 ①번과 같은 기본 구조로 구성된다. 이렇게 각각 구성된 세 개의 엘리먼트 FSA는 단순 전이로 연결되어 하나의 FSA를 형성한다. 그리고 AElement의 FSA와 AElementType에 대한 FSA를 단순 전이로 연결하여 예제 스키마 문서를 나타내는 완전한 FSA가 구성된다.

스키마 문서 전체에 대한 FSA를 이용하여 인스턴스 문서를 부호화하는 과정은 토큰(token)이 구성된 FSA의 시작 상태에서 시작하여 인스턴스 문서 내의 엘리먼트 순서대로 정의된 FSA의 각 상태들을 이동하면서 끝 상태에 도달할 때까지 수행된다. 토큰은 오토마타 이론에서는 문법적으로 의미를 갖는 최소의 단위를 의미하지만, MPEG-7에서는 현재 활성화된 상태를 표현하기 위한 표식으로 사용된다. 이 때, 토큰이 현재 상태에서 임의의 다른 상태로 이동하는 것을 "crossed"라 하고, 토큰이 임의의 상태로 "crossed"되면 해당 상태가 "activated"되었다고 한다. "crossed"와 "activated"를 반복하면서 점진적으로 부호화 과정을 수행한다. 이때 임의의 엘리먼트에 대한 부호화 결과는 세 부분으로 이루어 진다. 즉, 임의의 상태에서 바인딩 타입 상태까지의 경로 중 코드 전이에 할당된 비트 값, 엘리먼트가 인스턴스 문서에서 나타난 횟수, 그리고 엘리먼트의 실제 데이터를 이진 부호화 한 값들로 이루어진다.

예를 들어, FSA를 이용한 부호화 과정을 위의 예제 스키마 문서에 맞게 작성된 아래의 예제 인스턴스 문서를 사용하여 설명한다. 그림 5의 FSA에서 나타나듯이, 시작 상태에서 AElementType 타입 상태(①)로 토큰이 이동하면, AElementType 타입 상태는 "activated"되고, AElementType의 자식 엘리먼트의 부호화를 시작한다. 이 과정에서는 코드 전이가 발생하지 않으므로, 부호화된 값이 없다. AElementType 타입 상태에서 단순 전이, 코드 전이, 엘리먼트 전이를 "crossed"하여, BElement의 바인딩 타입 상태(②)로 토큰이 이동하게 된다. 이 때, 코드 전이의 비트 값인 "0"이 추가된다. 그리고, BElement는 인스턴스 문서에서 2번 나타나므로, 반복 횟수 "2"를vluimsbf5 형식으로 부호화 하여 "00010"으로 나타낸다. 그리고 실제 데이터인 "1st BElement", "2nd BElement"를 zlib을 사용하여 부호화 한다. 이와 같은 과정으로 BElement의 부호화 과정이 이루어진다. CElement는 스키마에서는 정의되었지만, 인스턴스 문서에서는 사용하지 않았다. 따라서, FSA의 토큰은 그림 5의 ②번 타입 상태에서 CElement의 바인딩 타입 상태(③)를 지나지 않고, 바로 DElement의 바인딩 타입 상태(④)로 이동하게 된다. 즉, 코드 전이의 한 종류인 전환 전이를 사용하여CElement 부호화 과정을 생략(skip)하고, 전환 전이의 비트 값인 "0"을 추가한다. DElement는 스키마의 정의에서 최소 및 최대 반복 횟수가 모두 1이므로, 인스턴스 문서에서 반듯이 1번 사용되어야 한다. 따라서, 반복 횟수를 부호화하여 나타내지 않고, DElement의 실제 데이터인 "1st DElement"만을 부호화한다. 인스턴스 문서의 부호화의 결과는 표 2를 통해 확인할 수 있다. 이후 그림 5 ④번 타입 상태가 끝 상태이므로 AElement의 전체 부호화 과정은 종료된다.

## Ⅳ. BiM 부/복호화기의 구조

### 1. BiM 부호화기의 구조

본 논문에서 설계한BiM 부호화기의 구조는 그림 6과 같다. 그림 6에서 BiM 부호화기는 크게 schema validator, schema manager, instance manager로 구성된다. Schema validator는 XML Schema로 문서의 구조를 표현한 스키마 문서와 XML로 표현되는 인스턴스 문서를 입력으로 하여, 인스턴스 문서의 유효 적합성을 점검한다. 스키마 문서는

단일의 문서로 구성될 수도 있으며, 임포트(include)/인클루드(include)/재정의(redefine) 메커니즘을 사용하여 여러 개의 스키마 문서가 복잡하게연결될 수도 있다[6]. 또한, 인스턴스 문서는 해당 스키마 문서에 기반한 완벽한 문서 일 수도 있고, 계속 생성되는 중간 단계의 문서일 수도 있다. 그러나, 어떠한 형태의 인스턴스 문서일 지라도 스키마 문서에 항상 유효(valid) 해야 한다. 만약, 스키마 문서의 정의가 올바르지 않거나, 인스턴스 문서가 스키마 문서에 유효하지 않은 경우에 부호화 작업을 수행하지 않는다.

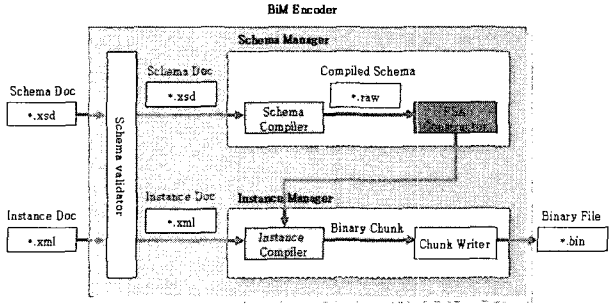



그림 6. BiM 부호화기의 구조
Fig. 6. Structure of BiM encoder

일반적으로 XML 스키마 문서 및 XML 인스턴스 문서 파싱을 위한 파서에는 DOM 파서와 SAX 파서가 있다. DOM 파서와 SAX 파서의 가장 큰 차이는 문서에 접근하는 방식이 다르다는 것이다. DOM 파서는 문서를 하나의 트리 구조로 구성하여 트리의 특정 노드를 접근하여 그 정의를 추출한다. 반면, SAX 파서는 문서를 하나의 긴 문자열로 취급하여 그 문자열을 앞에서부터 차례로 읽어가면서 정보를 추출한다. 이에 따라 DOM 파서는 문서를 파싱하기 위해 전체 문서를 메모리에 읽어 들임으로써 SAX 파서보다 메모리를 많이 소모하는 문제가 있다.

Schema manager는 스키마 문서 내의 정의된 순서대로 XML 스키마 컴포넌트를 파싱한다. 따라서, DOM 파서보다는 SAX 파서를 사용하는 것이 더욱 효율적일 수 있으므로 SAX 파서를 이용하여 스키마 문서를 파싱하도록 설계하였다. 파싱 과정을 통하여 스키마의 정의 내용을 내부 자료 구조로 재정의하고, 스키마 문서의 각 컴포넌트를 기호화하여 간소화된 스키마 문서인 raw 파일을 만든다. Schema manager는 생성된 raw 파일을 사용하여 FSA를 구성한다. BiM 부/복호화기의 가장 큰 특징 중 하나는 위에서 언급한 것과 같이 스키마 문서를 기반으로 인스턴스 문서를 부호화 및 복호화한다. 따라서 입력 스키마 문서를 구조화된 FSA로 생성, 관리, 사용하는 모듈은 BiM 부/복호화기의 핵심이다.

Instance manager는 인스턴스 문서를 schema manager에서 생성한 FSA를 사용하여 부호화하는 작업을 수행한다. 즉, 텍스트 형태의 인스턴스 문서를 이진 비트 스트링으로 변환하는 기능을 담당한다. 부호화된 이진 청크(binary chunk)는 chunk writer가 이진 파일 형태로 출력한다.

## 2. BiM 복호화기의 구조

BiM 복호화기의 구조는 그림 7과 같다. 그림 7의 BiM 복호화기는 크게 schema manager와 binary manager로 구성된다. BiM 복호화기는 스키마 문서 또는 raw 파일, 및 이진 부호화된 인스턴스 문서를 입력으로 한다. 그림 7의 schema manager는 그림 6의 BiM 부호화기에서 언급한 것과 동일하다. 단, 입력 파일로 BiM 부호화기에서 이미 생성한 raw 파일을 입력으로 받았을 경우에는 스키마 컴파일 작업은 수행하지 않고 바로 FSA를 생성한다. 결과적으로, 복호화 시 raw 파일이 주어질 경우에는 BiM 부호화기에서 이미 수행한 동일한 작업을 중복 수행을 하지 않아도 되는 이점이 생긴다.

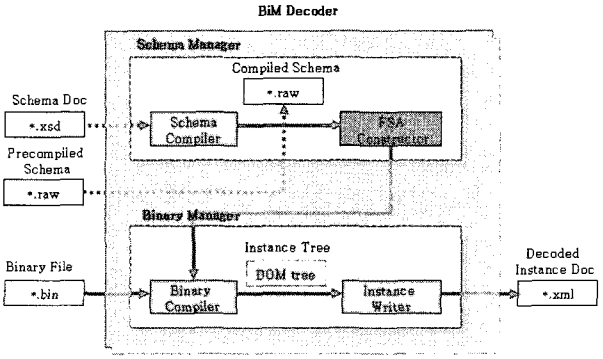



그림 7. BiM 복호화기의 구조
Fig. 7. Structure of BiM decoder

Binary manager는 schema manager에서 생성한 FSA를

사용하여 이진 부호화된 인스턴스 문서의 복호화 과정을 수행한다. 복호화의 결과로 메모리에 인스턴스 문서의 DOM tree 구조가 생성된다. 이 DOM tree를 instance writer를 통해서 원래의 인스턴스 문서로 출력한다.

## V. BiM 부/복호화기의 구현

여기서는 3장에서 설계한 BiM 부호화기 및 복호화기의 세부 구현 및 구현된 모듈간의 상호 동작에 대해 기술한다. 본 논문에서 구현한 BiM 부/복호화기는 Linux Fedora Core 4.0 (64 bit) OS 환경에서 C++로 구현하였다, 설계한 모듈 중 schema validator는 부호화 과정을 수행하기 위한 전처리 과정으로서, Apache사에서 개발한 Xerces 파서 (버전 2.7.0)[10]를 그대로 사용하였다. 따라서 여기서는 설계한 MPEG-7 BiM 부/복호화기 모듈 중 schema manager, instance manager, binary manager에 대하여만 기술한다. 이 세 부분은 총 350여 개 클래스에 10만여 줄의 코드 라인으로 구현하였다. 본 장에서는 구현한 BiM 부/복호화기의 자료 구조 및 동작을 구현한 클래스 계층구조(class hierarchy)를 중심으로 기술한다. 여기서 기술하는 클래스계층구조는 실제로 구현한 내용 중 중요 부분만을 요약한 것이며, 클래스들 간의 관계는 UML 다이어그램으로 표현하였다.

### 1. Schema manager의 구현

Schema manager는 BiM 부/복호화기의 공통 모듈로서, 스키마 문서를 파싱하여 문서의 내용을 내부 구조로 표현하고 raw 파일을 출력하는 schema compiler, 그리고 FSA를 생성하는 FSA constructor로 구성된다. 먼저, schema compiler가 파싱한 스키마 문서를 내부 구조로 표현하기 위해서 본 논문에서 설계한 클래스 계층구조는 그림 8과 같다.

스키마 문서에서 표현하는 요소 중 가장 기본은 엘리먼트와 타입이다. 엘리먼트는 오직 하나의 특정 타입에 바인딩(binding)된다. 타입에는 2장에서 설명하였듯이 크게 복합 타입과 단순 타입 두 종류가 있다[11]. 복합 타입은 일반적으로 엘리먼트와 애트리뷰트를 정의한다. 정의 하는 방법은 특정 타입 내에서 새로운 엘리먼트와 애트리뷰트를 정의하는 방법과, 상위 타입의 정의를 상속 받아 엘리먼트와 애트리뷰트를 확장하거나 제한하는 방법이다. 후자의 방법은 엘리먼트 요소를 확장 또는 제한 할 수 있는지의 여부에 따라 "simpleContent"와 "complexContent"로 나뉜다. 이러한 XML 스키마의 특징에 따라, 본 논문에서는 클래스 계층구조를 TypeDefinition를 최상위 클래스로 정의하고, ComplexTypeDefinition과 SimpleTypeDefinition을 TypeDefinition의 하위 클래스로 분류 정의한다. 그리고 ComplexTypeDefinition은 타입 정의 방법에 따라, com-

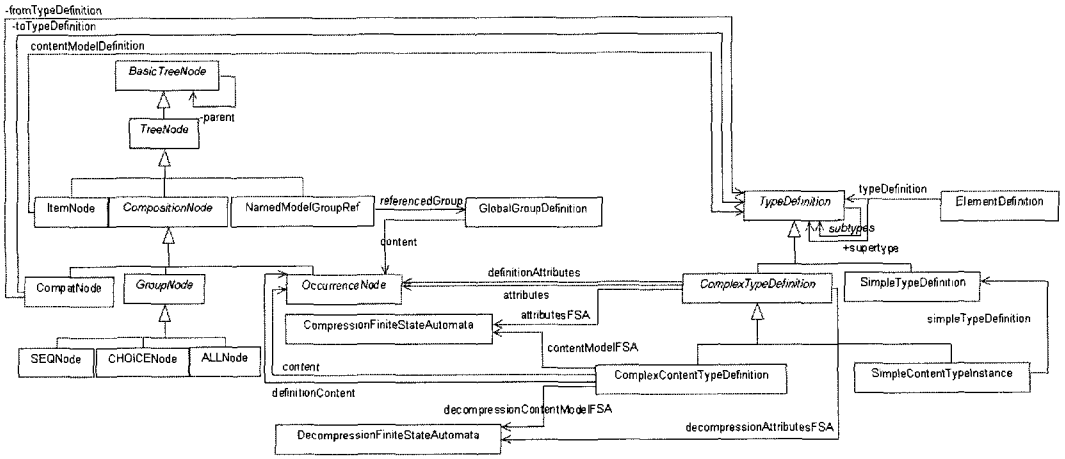



그림 8. 스키마 문서 표현을 위한 클래스 계층 구조
Fig. 8. Class hierarchy for representing schema document

plexContent 요소를 사용하여 확장하는 ComplexContent-TypeDefinition, simpleContent 기법을 사용하여 정의하는 SimpleContentTypeDefinition을 나누어 정의하도록 설계하였다. 이외의 XML 스키마 요소인, "sequence", "choice", "all"과 같은 컴포지터와 "group" 요소인 지명 모델 그룹을 위한 클래스도 정의하였다. 예를 들어, 그림 9의 클래스들 중 TypeDefinition 클래스와 그의 하위 클래스인 Complex-TypeDefinition 클래스의 정의는 다음과 같다.

TypeDefinition 클래스는 기본적으로 전역으로 정의된 타입인 경우에는 타입 이름을 설정하고, "isAnonymous" 변수를 false로 한다. 그 이외에 엘리먼트에 바인딩 되어 익명으로 정의된 타입인 경우에는"isAnonymous" 변수만true로 설정한다. "simpleContent" 또는 "complexContent" 요소를 사용하여 상속받은 복합 타입 이거나, "restriction", "list", "union" 요소를 사용하여 상속받은 단순 타입일 경우에는 realizeInheritance() 메소드를 사용하여 현재 타입을 기준으로 상위 타입과 하위 타입들 간의 계층 구조를 정의한다. 즉, 상위 타입의 이름으로 타입 정의를 추출하여 supertype 변수에 설정하고, addDirectSubtype() 메소드를 사용하여 supertype의 하위 타입에 현재 타입을 추가한다. ComplexTypeDefinition 클래스는 위에서 설명한 것과 같이, 복합 타입의 정의를 나타낸다. 만약, 복합 타입이 추상 타입으로 정의되어 있을 경우, 즉, "abstract" 속성이 "true" 일 경우에는, 인스턴스 문서에서 엘리먼트의 바인딩 타입으로 쓰일 수 없고, "xsiType" 속성을 사용하여 하위 타입으로 대체하여 사용해야 한다. 따라서, 본 클래스에서 이러한 경우에는 "isAbstract" 변수를 true로 설정한다. 그리고, "mixed" 속성을 사용할 경우, "mixed" 변수 값을 true로 설정한다. 이 후에, FSA constructor 모듈에서 generateFSA() 메소드를 사용하여, 현재 타입에서 정의한 애트리뷰트와 내용 모델의 FSA를 구성한다. 이를 위해 generateAttribute-FSA()와 generateContentFSA() 메소드를 호출한다. 이와 같이 schema manager가 내부적으로 사용하는 자료 구조를 갖게 함으로서, 후에 부호화 및 복호화 과정에서 스키마 문서의 정의 내용을 얻기 위해 다시 파싱하는 일을 거치지 않고, 효율적으로 스키마 문서의 구조를 접근 할 수 있도록 한다.

Schema compiler는 그림 8에서 정의한 자료 구조를 사용하여 입력 스키마 문서를 단순화 시킨 raw 파일을 생성한다. 아래는 간단한 예제 스키마 문서 및 이 문서에서 생성된 raw 파일을 나타낸다. 예제 스키마 문서는 "schema" 요소에서 2개의 네임스페이스(namespace)를 기술하고, 인스턴스 문서에 나타나는 엘리먼트는 네임스페이스로 한정(qualified)하고, 애트리뷰트는 한정하지 않도록(unqualified) 규정한다. 이와 같은 "schema" 요소의 네임스페이스와 속성(property)의 정의는 raw 파일에서 ①과 ②와 같이 각각 변환 된다. 즉, 네임스페이스를 "NAMESPACES" 키워드를 사용하여 알파벳 순서대로 네임스페이스 URI를 출력한다. 그리고 속성은 "PROPERTIES" 키워드를 나타내고 속성의 개수와 해당 값을 표시한다. 여기서, QE는 "element-FormDefault"가 "qualified"라는 것을 의미하고, UQA는 "attributeFormDefault"가 "unqualified"라는 것을 나타낸

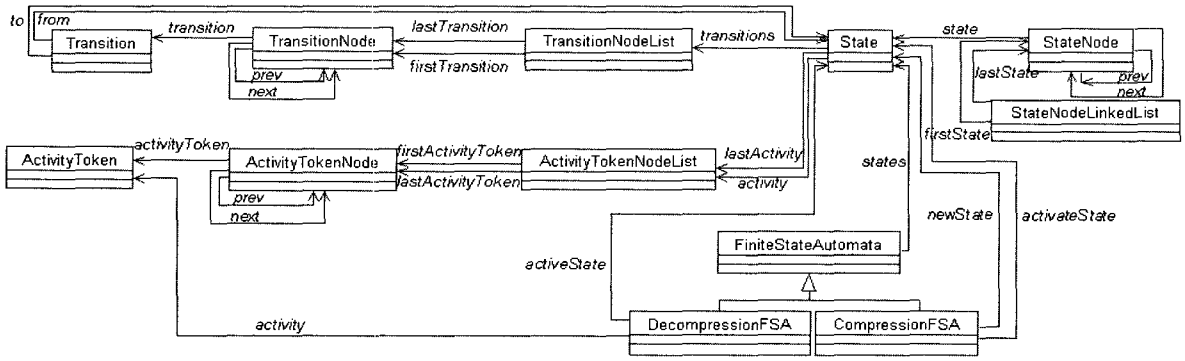



그림 9. Finite State Automata 구성을 위한 클래스 계층구조
Fig 9. Class hierarchy for construction of finite state automata

```
class TypeDefinition {
    const XMLCh *name;                                  /* attribute declaration */
    bool isAnonymous;
    set<TypeDefinition *> subtypes;
    TypeDefinition *supertype;
    const XMLCh* getName();                             /* method declaration */
    short getDerivationMethod();
    void setAnonymous(bool);
    void getAnonymous();
    bool hasSupertype();                                /* methods for processing supertype */
    void setSupertype(const XMLCh *, short);
    TypeDefinition *getSupertype();
    bool hasSubtypes(bool);                             /* methods for processing subtypes */
    void addDirectSubtype(TypeDefinition *);
    int getNumberOfSubtypes(bool);
    int getSubtypeIndex(const XMLCh *, bool);
    SubtypesIterator* getSubtypesIterator(bool);
    TypeDefinition* getSubtype(int, bool);
    void realizeInheritance(TypeDefinitions *);         /* method for type realization */
    . . .
}
class ComplexTypeDefinition : public TypeDefinition {
    bool isAbstract;                                    /* attribute declaration */
    bool isMixed;
    OccurrenceNode *attributes;
    OccurrenceNode *content;
    CompressionFSA *compressionAttributesFSA;
    DecompressionFSA *decompressionAttributesFSA;
    CompressionFSA *compressionContentFSA;
    DecompressionFSA *decompressionAttributesFSA;
    bool hasAttributes();                               /* method declaration */
    void setAbstract(bool);
    bool getAbstract();
    void generateFSA();
    void setMixed(bool);
    void getMixed();
    void generateFSA();
    FiniteStateAutomata* generateAttributesFSA();
    FiniteStateAutomata* generateContentModelFSA();
    void realize(TypeDefinitions *);
    . . .
}
```

다. "schema" 요소 내부에 정의된 XML 스키마 컴포넌트 요소들은 다음과 같은 간단한 규칙에 따라 출력된다. 엘리먼트는 "#namespace"nameOfElement", 복합 타입은 "{#namespace"nameOfComplexType}", 단순 타입은 "<#namespace"nameOfSimpleType>" 형식으로 표현한다. 예들 들어, 예제 스키마 문서의 "MPEG7Type"은 복합 타입으로 두 번째 네임스페이스인 "urn:mpeg:mpeg7:schema:2001"에서 정의한다. 따라서, 이 복합 타입은raw 파일에서 "{2"MPEG7Type}"으로 나타낸다. 또한, 복합 형식의 속성인 "abstract"와 "mixed"는 "ABSTRACT"와 "MIXED"와 같은 키워드를 사용하여 나타낸다. 컴포지터는 각각 "SEQ", "CHO", "ALL" 키워드로 나타내고, 반복 횟수는 "[minOccurs, maxOccurs]" 형식으로 표현함으로써 복잡한 스키마 문서를 단순한 형태로 변화시킨다. 이렇게 단순화시키는 효과는 메모리를 많이 사용하는 스키마 문서의 파싱 과정을 단순화한 파일을 읽는 과정으로 대치함으로써, 메모리 관리 측면에서 이득을 얻을 수 있다.

FSA constructor는 생성된 raw 파일을 사용하여 FSA를

생성한다. 본 논문에서는 2장에서 설명한 FSA의 구성 동작을 그림 9와 같은 클래스 계층구조로 구현하였다. FSA는 기능에 따라 부호화 FSA 및 복호화 FSA로 나누며, 그림 6에서는 각각 CompressionFSA 및 DecompressionFSA 클래스에 대응된다. 그리고 FSA의 구성요소인 상태와 전이는 각각 State와 Transition 클래스로 구현하며, 상태와 전이에 관련된 데이터들은 각각 연결 리스트(linked list) 구조로 표현하였다.

예를 들어, FiniteStateAutomata 클래스의 정의는 다음과 같다. FiniteStateAutomata 객체는 그림 8의 ComplexType-Definition 객체가 generateFSA() 메소드를 호출함으로써 생성된다. FiniteStateAutomata 클래스는 상태 객체를 생성하여 states 리스트에 상태를 추가하고, 상태와 상태를 전이로 연결 하거나 다른 임의의 오토마타의 상태들과 현재 상태들과 병합하는 과정을 수행한다. 이와 같이, 상태와 전이를 생성하고 병합하는 과정을 스키마 문서에서 정의된 스키마 요소들을 raw 파일에서 표현된 순서에 따라 반복적으로 수행하면서 FSA가 구성된다.

```
class FiniteStateAutomata {
StateLinkedList *states;
/* attribute declaration */
void addState(State *);
/* method declaration */
void merge(FiniteStateAutomata *);
void merge(State *, State *);
State* getFirstStartState();
State* getFirstFinalState();
StateLinkedList* getStartState();
StateLinkedList* getFinalState();
. . .
}
```

이러한 FSA 클래스 계층구조를 사용하여, 부호화 시에는 인스턴스 문서에 명시된 엘리먼트의 바인딩 타입의 타입 상태를 이동하면서 이동 경로 상에 있는 전이의 값을 이진 스트링으로 변환한다. 반대로 복호화 시에는 이진 스트링의 값으로 상태를 이동하며 해당 타입 상태의 정의를 도출하여 DOM tree를 재구성한다.

## 2. Instance manager와 binary manager의 구현

BiM 부/복호화기의 Instance manager와 binary manager는 그림 10에서 정의한 클래스 계층구조를 사용한다. Type-Encoder 클래스를 기반으로 타입의 특성에 따라 Simple-TypeInstance와 ComplexTypeInstance 클래스로 나누며, ComplexTypeInstance 클래스는 정의 방법에 따라 Simple-ContentTypeInstance와 ComplexContentTypeInstance 클래스로 세분화하여 정의한다.

예를 들어, 그림 7의 클래스들 중 TypeInstance와 Complex-TypeInstance 클래스의 정의는 아래와 같다.

Instance manager는 스키마 문서를 FSA를 사용하여 부호화하는 모듈로서, TypeInstance의 하위 클래스의 객체가

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="urn:mpeg:mpeg7:schema:2001"
 xmlns="http://www.w3.org/2001/XMLSchema xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001"
 elementFormDefault="qualified" attributeFormDefault="unqualified">
  <complexType name="Mpeg7Type" abstract="true">
    <sequence>
      <element name="DescriptionMetadata" type="mpeg7:DescriptionMetadataType" minOccurs="0"/>
    </sequence>
    <attributeGroup ref="mpeg7:timePropertyGrp"/>
    <attributeGroup ref="mpeg7:mediaTimePropertyGrp"/>
  </complexType>
</schema>
```

```
(NAMESPACES"http://www.w3.org/2001/XMLSchema" "urn:mpeg:mpeg7:schema:2001") --- ①
(PROPERTIES 2 QE UQA)---------- ②
{2"Mpeg7Type} (ABSTRACT) (ATTR 0"timeUnit[0,1]{2"durationType}
0"mediaTimeUnit[0,1]{2"mediaDurationType} 0"mediaTimeBase[0,1]{2"xPathRefType}
0"timeBase[0,1]{2"xPathRefType})
(SEQ[1,1] (SEQ[1,1] 2"DescriptionMetadata[0,1]{2"DescriptionMetadataType} ))|
...
```

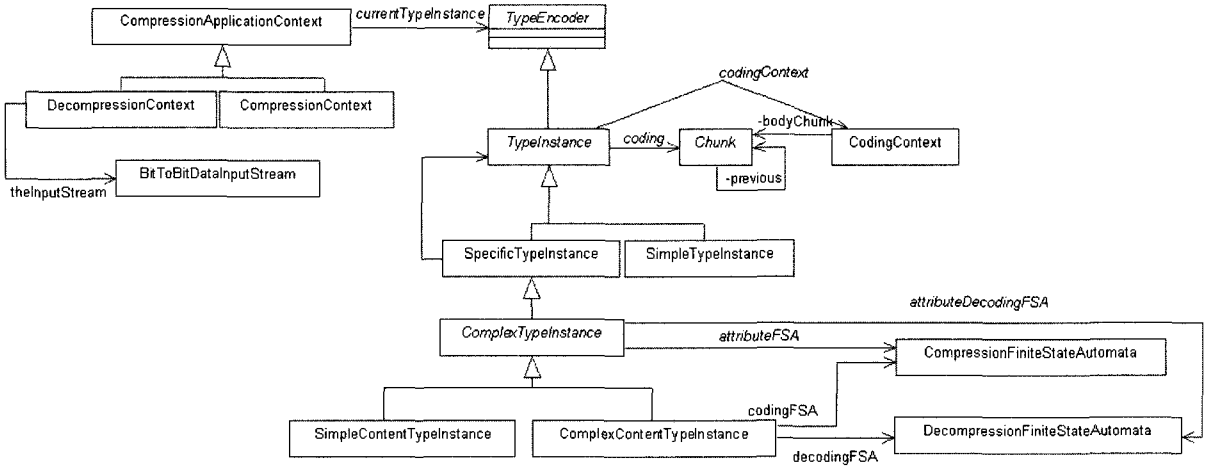



그림 10. Instance manager와 binary manager의 클래스 계층구조
Fig 10. Class hierarchy for instance manager and binary manager

startEncoding() 메소드를 호출함으로서 부호화 과정을 시작한다. 복합 타입의 경우에는 내용 모델과 애트리뷰트를 부호화하기 위해 각각 encodeContent()와 encodeAttribute() 메소드를 호출한다. 각 메소드는 FSA의 시작 상태에서 끝 상태로 이동하면서 타입 정의에 따라 TypeInstance 클래스의 encodeTypeInfo() 메소드를 수행하여, 부호화된 이진 스트링을 Chunk 객체로 저장한다. 이와 같은 반복적인 과정을 수행한 후에, FSA의 끝 상태에 도달하면 endEncoding() 메소드를 호출하고, chunk writer는 생성된 Chunk 객체들을 전달 받아 이진 파일 형태(*.bin)로 출력한다.

```
class TypeInstance : public TypeEncoder {
void encodeTypeInfo();
/* method declaration */
void decodeTypeInfo();
. . .
}
class ComplexTypeInstance : public TypeInstance {
void startEncoding();
/* methods for encoding */
TypeEncoder* encodeContent(const XMLCh *);
TypeEncoder* encodeAttribute(const XMLCh *);
void endEncoding();
void endContentEncoding();
void endAttributeEncoding();
void startDecoding(BitToBitDataInputStream *);
/* methods for decoding */
TypeEncoder* decode(BitToBitDataInputStream *);
void endDecoding();
. . .
}
```

Binary manger는 FSA를 사용하여 이진 스트링을 인스턴스 문서로 복호화하는 모듈이다. 이진 파일을 읽어 BitToBitDataInputStream 객체를 생성하고, TypeInstance 클래스의 하위 클래스의 객체가 startDecoding() 메소드를 호출하면서 복호화가 시작된다. 복합 타입의 경우에는 decode() 메소드를 호출하여 애트리뷰트와 내용 모델을 FSA를 사용하여 복호화한다. 즉, BitToBitDataInputStream 객체의 이진 스트링의 값에 따라 FSA의 시작 상태에서 끝 상태로 이동하면서, TypeInstance 클래스의 decodeTypeInfo() 메소드를 실행하여타입의 정의를 DOM tree 형태로 재구성한다. 동일한 반복 과정 후에, FSA의 끝 상태에 도달하여 endDecoding() 메소드가 호출되면, instance writer가 구조화된 DOM tree 구조를 인스턴스 문서(*.xml)로 출력한다.

## Ⅵ. 실험 및 결과

본 장에서는 NIST MPEG CVS Repository[12]에서 제공하는 예제 인스턴스 문서를 대상으로 BiM의 부호화 효율을 앞서 설명한 gzip, XMill, XGrind과 비교한다.

표 3. MPEG 인스턴스 문서의 통계적 정의
Table 3. Statistical definition of MPEG instance documents

| | | 예제 인스턴스 문서 명 | 문서 크기 (byte) | 최대 중첩 깊이 | 엘리먼트 개수 | 애트리뷰트 개수 | 실제 데이터 (%) |
|---|---|---|---|---|---|---|---|
| [MPEG-21] | 1 | 06-DIA-UED.xml | 6,474 | 6 | 75 | 79 | 6.00 |
| | 2 | 07-DIA-BSDLink.xml | 1,204 | 4 | 17 | 19 | 5.98 |
| | 3 | 08-DIA-Gbsd-01_₩(JP2₩).xml | 242,816 | 7 | 3,556 | 6,558 | 0.88 |
| | 4 | 09-DIA-AQoS.xml | 2,722 | 6 | 31 | 41 | 3.87 |
| | 5 | 10-DIA-UCD.xml | 2,137 | 7 | 25 | 33 | 0.44 |
| | 6 | 11-DIA-MDA.xml | 1,763 | 5 | 20 | 29 | 1.46 |
| | 7 | 13-DIA-DIAC.xml | 1,705 | 4 | 9 | 10 | 0.00 |
| [MPEG-7] | 8 | descriptionExample001.xml | 2,728 | 3 | 6 | 4 | 2.79 |
| | 9 | descriptionExample002.xml | 3,557 | 6 | 27 | 8 | 9.08 |
| | 10 | descriptionExample004.xml | 2,939 | 6 | 10 | 6 | 4.19 |
| | 11 | descriptionExample005.xml | 2,807 | 6 | 10 | 2 | 3.81 |
| | 12 | descriptionExample006.xml | 3,107 | 8 | 14 | 11 | 4.80 |
| | 13 | descriptionExample007.xml | 3,211 | 7 | 12 | 7 | 4.98 |
| | 14 | descriptionExample008.xml | 3,042 | 7 | 15 | 11 | 4.24 |
| | 15 | descriptionExample009.xml | 2,992 | 8 | 13 | 6 | 5.05 |

표 3은 예제 MPEG 인스턴스 문서[7]의 정의를 문서의 크기, 최대 중첩 깊이(nesting depth), 엘리먼트와 애트리뷰트의 개수 및 실제 데이터의 비율의 관점에서 나타낸 것이다. 문서의 크기는 바이트 단위로 나타내었고, 최대 중첩 깊이는 인스턴스 문서 내에서 엘리먼트 태그가 중첩된 정도를 의미한다. 실제 데이터의 비율은 인스턴스 문서 내에서 태그 및 공백 문자를 제외한 실제 데이터의 비율을 나타낸다.

사용한 예제는 각각 MPEG-7 스키마와 MPEG-21 스키마에 따라 정의된 인스턴스 문서 그룹으로 나누어 그 특징을 살펴볼 수 있다. MPEG-7 스키마 문서는 인클루드나 임포트를 사용하여 다른 스키마 문서를 포함하지는 않지만, 스키마 문서 자체가 500여 개의 타입 정의와 1000여 개의 엘리먼트로 구성된 거대한 구조이다. 한편 MPEG-21 스키마 문서들은 대부분 인클루드와 임포트 메커니즘을 사용하여 다른 스키마 문서를 포함하는 복잡한 구조로 되어 있다. 예를 들어, 09-DIA-AQoS.xml의 스키마 문서인 MPEG-21 AQoS.xsd의 경우에는 MPEG-7 스키마 문서를 포함한 10개의 다른 스키마 문서를 포함하고 있다.

표 4는 예제 MPEG 인스턴스 문서에 대한 부호화 실험 결과이다. 실험 기준인 부호화 효율과 부호화 효율 요소

표 4. BiM 부호화기를 사용한 결과
Table 4. Results of encoding instance files using BiM encoder)

| 예제 인스턴스 문서 명 | | $CR_{전체}$ | $CR_{압축}$ | $CR_{구조화}$ | $CR_{압축해제}$ | $CRF_{효율}$ | $CRF_{압축}$ | $CRF_{압축}$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 06-DIA-UED.xml | 0.85 | 0.79 | 0.56 | - | 1.08 | 1.53 | - |
| 2 | 07-DIA-BSDLink.xml | 0.76 | 0.61 | 0.56 | 0.67 | 1.24 | 1.36 | 1.13 |
| 3 | 08-DIA-Gbsd-01_₩(JP2₩).xml | 0.87 | 0.95 | 0.97 | - | 0.92 | 0.90 | - |
| 4 | 09-DIA-AQoS.xml | 0.89 | 0.71 | 0.68 | 0.77 | 1.25 | 1.30 | 1.15 |
| 5 | 10-DIA-UCD.xml | 0.87 | 0.71 | 0.69 | 0.75 | 1.22 | 1.26 | 1.16 |
| 6 | 11-DIA-MDA.xml | 0.77 | 0.71 | 0.67 | 0.73 | 1.08 | 1.16 | 1.06 |
| 7 | 13-DIA-DIAC.xml | 0.74 | 0.65 | 0.60 | 0.64 | 1.14 | 1.24 | 1.14 |
| 8 | descriptionExample001.xml | 0.98 | 0.60 | 0.56 | 0.94 | 1.63 | 1.75 | 1.04 |
| 9 | descriptionExample002.xml | 0.94 | 0.60 | 0.57 | 0.91 | 1.56 | 1.64 | 1.04 |
| 10 | descriptionExample004.xml | 0.98 | 0.61 | 0.58 | 0.94 | 1.61 | 1.70 | 1.05 |
| 11 | descriptionExample005.xml | 0.98 | 0.61 | 0.58 | 0.94 | 1.61 | 1.69 | 1.04 |
| 12 | descriptionExample006.xml | 0.98 | 0.64 | 0.60 | 0.92 | 1.54 | 1.62 | 1.07 |
| 13 | descriptionExample007.xml | 0.98 | 0.64 | 0.61 | 0.94 | 1.54 | 1.60 | 1.05 |
| 14 | descriptionExample008.xml | 0.98 | 0.63 | 0.60 | 0.92 | 1.55 | 1.64 | 1.06 |
| 15 | descriptionExample009.xml | 0.98 | 0.62 | 0.59 | 0.93 | 1.58 | 1.65 | 1.05 |
| 평 균 CR | | 0.90 | 0.67 | 0.63 | 0.86 | 1.37 | 1.47 | 1.08 |

(compression ration factor)의 정의는 다음과 같다. 부호화 효율 요소는 BiM의 부호화 효율과 gzip, XMill, XGrind의 부호화 효율을 비교하기 위한 것으로, 식의 'x'는 비교 대상인 gzip, XMill, XGrind를 의미한다. 에서 CRXGrind 중 '-'으로 표시한 것은 XGrind 애플리케이션의 오류로 해당 인스턴스 문서를 부호화하지 못한 것을 의미한다.

$$CR = 1 - \frac{\text{sizeof(compresed file)}}{\text{sizeof(original file)}}$$

$$CRF = \frac{CR_{BiM}}{CR_x}$$

그림 11은 각 부호화 기법의 실험 결과를 도식화하여 비교한 것이다. 인스턴스 문서 3을 제외한 모든 인스턴스 문서에 대한 BiM의 부호화 효율이 다른 부호화 기법보다 훨씬 높은 것을 알 수 있다. 인스턴스 문서 3은 엘리먼트 3,556개 중 4개를 제외하고 나머지 3,552개가 "gBSDUnit", "Parameter", "Value" 이름의 엘리먼트가 반복적으로 나타난다. Gzip의 경우에는 LZ77 알고리즘과 허프만 코딩을 사용하기 때문에, 동일한 엘리먼트는 반복적으로 사용하면 빈도수가 높아지기 때문에, 엘리먼트를 부호화하는 이진 코드의 길이를 줄일 수 있다. XMill은사전 부호화 기법을 사용하므로 인스턴스 문서 내에 사용하는 엘리먼트의 개수가 상대적으로 적어지므로 인스턴스 문서 3에 대한 부호화 효율이 BiM 보다 높게 나타날 수 있다.

또한, 인스턴스 문서를 두 부류로 분류하여 살펴보면, MPEG-7 스키마로 정의된 인스턴스 문서에 대한 BiM 의 부호화 효율이 다른 부호화기보다 월등하게 높은 것을 확인할 수 있다. MPEG-21 인스턴스 문서는 소수의 엘리먼트가 반복적으로 나타나는 반면, MPEG-7 인스턴스 문서는 중복되는 엘리먼트의 수가 적으므로 MPEG-7 스키마로 정의된 인스턴스 문서의 경우 다른 부호화기보다 BiM의 부호화 효율이 높게 나타난다.

그림 12는 각 부호화 기법의 효율이 인스턴스 문서 내의 엘리먼트 태그의 최대 중첩 깊이와의 관계를 나타낸 것이다. BiM의 경우 다른 부호화기에 비하여 인스턴스 문서 내의 엘리먼트 태그의 중첩 깊이가 깊을수록 부호화기의 효과를 더욱 얻을 수 있는 것을 알 수 있다.

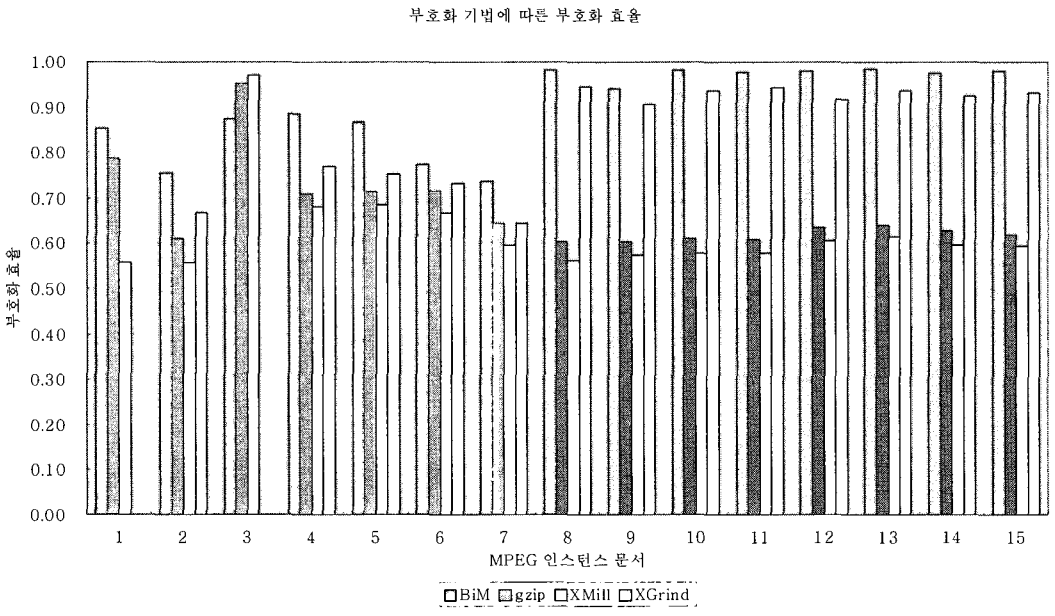



그림 11. 부호화 기법에 따른 부호화 효율
Fig 11. Compression ration according to encoding method

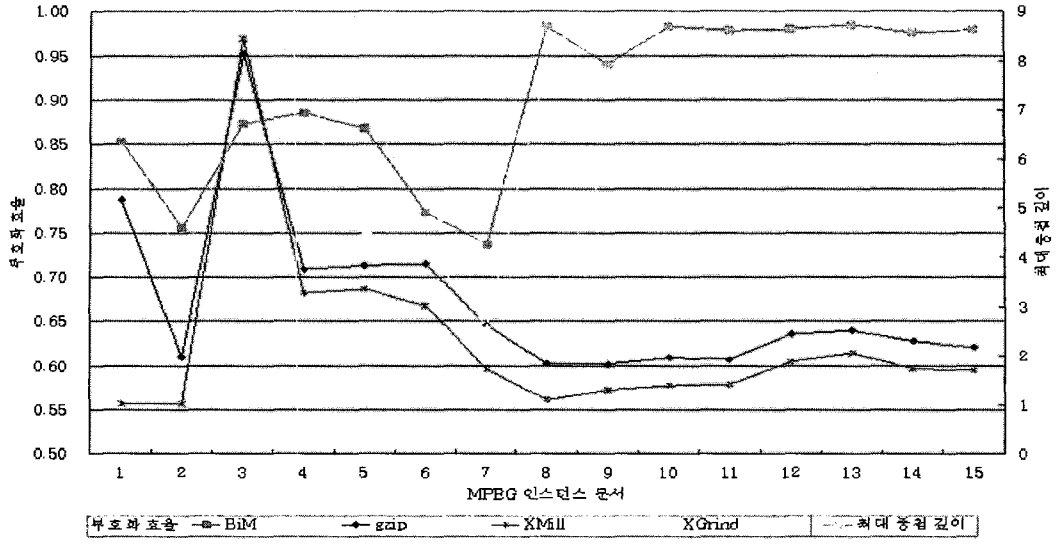



그림 12. 최대 중첩 깊이에 따른 부호화 효율의 변화
Fig 12. Compression ratio according to nesting depth

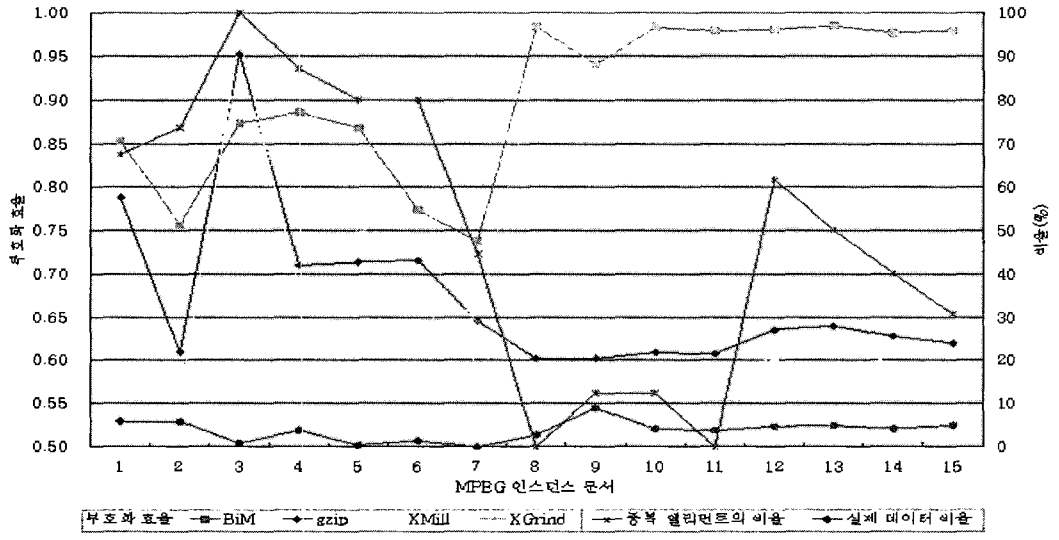



그림 13. 중복 엘리먼트와 실제 데이터의 비율에 따른 부호화 효율의 변화
Fig 13. Compression ratio compared with effective data ratio

그림 13은 중복 엘리먼트의 비율과 실제 데이터 비율에 따른 부호화 효율의 변화를 도식화 한 것이다. 각 부호화기를 비교해 본 결과, gzip과 XMill의 경우 중복 엘리먼트의 비율이 높을수록 대부분 부호화 효율이 높은 반면, BiM은 중복 엘리먼트의 비율의 영향을 적게 받는 것을 알 수 있다. 그리고 대부분 부호화기는 실제 데이터보다 엘리먼트 태그가 많은 경우에 부호화 효율이 높아지는 것을 할 수 있다. 또한, gzip이나 XMill의 경우는 엘리먼트 태그의 이진 코드의 길이가 전체 태그의 개수에 따라 정의되지만, BiM의 경우에는 엘리먼트의 바인딩 타입의 정의에 따라 자식 엘리

먼트의 개수로 이진 코드의 길이가 결정된다. 따라서, gzip과 XMill의 경우에는 중복되지 않은 엘리먼트의 개수가 많아질수록 부호화 효율은 떨어질 수 있지만, BiM의 경우에는 거의 영향을 받지 않는다.

결과적으로 BiM 부호화기를 사용하여 평균적으로 약 90%에 해당하는 부호화 효율을 얻을 수 있으며, 이것은 gzip보다 약 37%, XMill보다 약 47%, XGrind보다 약 8% 정도높은 효과를 얻은 것으로 확인하였다. 그리고 BiM 부호화기를 사용한 인스턴스 문서의 부호화 효율은 최대 중첩 깊이가 클수록, 인스턴스 내에서 실제 데이터의 비율이 적을수록 높은 것도 확인 할 수 있었다.

## Ⅶ. 결 론

본 논문은 MPEG-7 BiM 표준에 따라, 스키마 문서의 정의를 사용하여 인스턴스 문서를 이진 부호화하는 BiM 부호화기와 이진 파일을 스키마 문서를 정의를 기반으로 인스턴스 문서로 복호화하는 BiM 복호화기를 설계 및 구현하였다. 구현한 BiM 부호화기는 실험에 사용한 NIST MPEG CVS Repository 데이터 셋에 대하여 평균 9.44%에 해당하는 부호화 효율을 보였다. 구현한 BiM 부/복호화기는 MPEG 인스턴스 문서뿐만 아니라 XML Schema로 기술한 스키마 문서에 따르는 어떤 인스턴스 문서도 부호화할 수 있는 범용 XML 문서 부/복호화기이다. 따라서, XML Schema 및 XML이 사용되는 모든 응용 분야에서 활용될 수 있으며, 특히, 대역폭의 제약이 심한 디지털 방송 환경에서의 활용 가치가 매우 높을 것으로 예상된다. 현재 구현된 시스템은 아직 최적화 되어 있지 않다. 특히 복호화기는 매우 작은 메모리와 낮은 성능의 CPU가 장착된 휴대형 단말기에서 실행되는 경우가 많을 것이므로, 최적화에 특히 많은 노력이 필요하다. 앞으로는 이를 위한 연구가 필요할 것이다.

## 참 고 문 헌


[1] Jean-loup Gailly, Mark Alder, "Gzip", July 27th, 2003, Available at : http://www.gzip.org

[2] H. Liefke, D. Suciu, "XMill: An efficient compressor for XML data", Proc. of the 2000 ACM SIGMOD, pages 153-164, May 2000.

[3] P. M. Tolani and J. R. Haritsa, "XGRIND: A Query-friendly XML Compressor", Proc. of 18thInternational Conference on Database Engineering, pages 255-234, 2002.

[4] ISO/IEC JTC1/SC29/WG11 (MPEG), "Information Technology - Multimedia Content Description Interface - Part 1: Systems", International Standard 15938-1, ISO/IEC FDIS 15938-1:2001, Sep. 2001 (m7673).

[5] Jakob Ziv and Abraham Lempel, "A universal algorithm for sequential data compression", IEEE Transactions on Information Theory, 23(3), pp.337-343, 1977.

[6] D.Huffman, "A Method for Construction of Minimum-Redundancy Codes", In Proceedings of IRE, September 1952.

[7] R.Pajarola, "Fast Huffman Code Processing", UCI-ICS Technical Report No. 99-43, pp.1-6, 1999.

[8] TV-Anytime Forum, "TV-Anytime Phase 1, Part 3 : Metadata, Sub-part2 : System aspects in a uni-directional environment", ETSI Standard, ETSI TS 102 822-3-2, V.1.3.1 Jan. 2006.

[9] Smith S.Nair, "XML Compression Techniques : A Survey"

[10] Apache XML project, Xerces Java Parser 2.7.0 Release, 2005, Available at: http://xml.apache.org/xerces-c/.

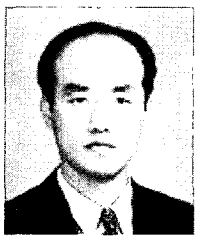
[11] H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, "XML Schema Part 1 : Structures Second Edition", Oct. 2004, Available at : http://www.w3.org/TR/xmlschema-1/

[12] ISO/IEC JTC1/SC29/WG11 (MPEG), NIST MPEG CVS Repository, Available at : http://mpeg.nist.gov/cvsweb/MPEG-7.

## 저 자 소 개

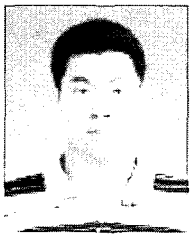

**염 지 현**

- 2005년 2월 : 국민대학교 컴퓨터학부 컴퓨터과학 전공 (학사)
- 2007년 2월 : 국민대학교 전산과학과 (석사)
- 2007년 3월 ~ 현재 : 국민대학교 컴퓨터공학과 박사과정
- 주관심분야 : TV-Anytime, MPEG-7, 디지털 방송, 맞춤형 방송, 메타데이터 압축



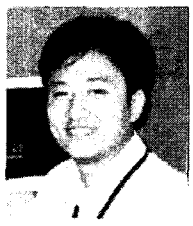
**김 혁 만**

- 1985년 2월 : 서울대학교 컴퓨터공학과 (공학사)
- 1987년 2월 : 서울대학교 컴퓨터공학과 (공학석사)
- 1996년 2월 : 서울대학교 컴퓨터공학과 (공학박사)
- 1996년 ~ 1999년 : 한국통신 멀티미디어연구소
- 1999년 ~ 현재 : 국민대 컴퓨터공학부 부교수
- 주관심분야 : XML, 메타데이터, 비디오 모델링 및 인덱싱



**김 민 제**

- 2004년 2월 : 아주대학교 정보 및 컴퓨터공학부 (학사)
- 2006년 2월 : 포항공과대학교 컴퓨터공학과 (석사)
- 2006년 2월 ~ 현재 : 한국전자통신연구원 방송미디어연구그룹 연구원
- 주관심분야 : 디지털방송, TV-Anytime, MPEG-7



**이 한 규**

- 1994년 2월 : 경북대학교 전자공학과 (학사)
- 1996년 2월 : 경북대학교 전자공학과 (석사)
- 1996년 2월 ~ 현저 : 한국전자통신연구원 방송미디어연구그룹 맞춤형방송연구팀 팀장
- 주관심분야 : 신호처리, 멀티미디어 지능형/양방향 시스템