SGML 문서의 관리를 위한 객체지향 데이타베이스 설계

한 에 노[†] · 박 인 호[††] · 강 현 석[†††] · 김 완 석[††††]

요 약

SGML로 기술된 전자문서는 객체지향 데이타베이스로 관리하는 것이 효과적이다. 이를 위해서는 SGML 문서들을 위한 데이타베이스 설계가 선행되어야 한다.

본 논문은 전자문서의 논리적 구조를 기술하는 SGML DTD 정보를 관리하기 위한 객체지향 메타 스키마를 제안하고, 이 메타 스키마로 부터 동적으로 생성된 응용 데이타베이스 스키마를 통해 전자문서를 객체지향 데이타베이스에서 관리하는 방법을 제안한다.

An Object-Oriented Database Design for Managing SGML Documents

Eh-No Han[†] · In-Ho Park[††] · Hyun-Syug Kang[†††] · Wan-Syug Kim[††††]

ABSTRACT

Electronic documents described in SGML DTD are effective to manage with the object-oriented database. To do that, first of all, a database design for SGML documents must be preceeded.

In this paper, we suggest an object-oriented meta schema for managing SGML DTDs, which describes logical structures of electronic documents. We also propose a method to manage electronic documents in object-oriented databases through the application database schema which is dynamically created from the meta schema.

1. 서 론

최근 각종 문서, 책, 잡지 등이 점차 전자문서화되면서 이들 전자문서들을 이기종 시스템들에서 처리하고 교환할 수 있게 하는 것이 중요하게 되었다. 이를 위해서 전자문서는 그것의 내용외에도 내용의 배치를 위한 레이아웃(layout), 글자체 등의 정보를 필요로 한다. 이런 추가적인 정보를 문서에 삽입하는 것을 마크업(markup)이라 하며 메타 언어로 SGML(Standard Generalized Markup Language)이 제안되었다[1]. 이는 원래 출판 분야에서 사용할 목적으로 제안되었으나 최근 그 응용 분야가 하이퍼미디어 시스템 등으로 넓어지고 있다[2].

이러한 SGML로 기술된 전자문서들을 통합 데이타베이스로 관리하면 많은 이득을 얻을 수 있다. 즉, 질의 처리 등 데이타베이스 시스템이 제공하는 특성들을 제공받을 수 있다[3]. 그러나 이러한 전자문서는 다양한 미디어와 복잡한 구조를 갖기 때문에 기존의 관계형 데이타베이스 보다는 객체지향 데이타베이스[4]로 관리하는 것이 효과적이다.

SGML DTD로 기술된 전자문서를 객체지향 데이

※본 연구는 1996년도 정보통신부 대학 기초 연구와 1996년도 한국전자통신연구소 위탁과제 지원사업에 의해이루어졌습니다.

† 정 회 원:신한정보 시스템(주) 연구원
†† 정 회 원:경상대학교 전자계산학과 박사과정
††† 종신회원:경상대학교 컴퓨터과학과 교수
†††† 정 회 원:한국전자통신연구원 연구원
논문접수:1996년 6월 5일, 심사완료: 1997년 2월 18일

타베이스에서 관리할 수 있기 위해서는 우선 적절한 객체지향 데이타베이스의 설계가 이루어져야 한다. 우리는 이를 위해 DTD들을 통합적으로 다룰 수 있도록 SGML DTD 메타 스키마를 설계하고 이 메타 스키마에 인스탄스로 저장된 DTD에 대한 정보를 이용하여 각 응용 스키마를 동적으로 생성시키는 방법을 제안한다.

논문의 구성은 다음과 같다. 2장에서는 SGML의 특성과 SGML을 이용해 전자문서를 기술하는 방법을 알아 본다. 3장에서는 SGML DTD들을 통합적으로 관리할 수 있도록 하는 SGML DTD 메타 스키마를 제안한다. 4장에서는 SGML DTD 메타 스키마에 저장된 DTD에 대해 응용 데이타베이스 스키마를 자동적으로 생성시키는 방법을 알아 본다. 5장에서는 관련 연구를 알아 본 후, 6장에서 결론 및 향후 과제에 대해 논한다.

2. SGML과 DTD의 기술 방법

2.1 SGML의 구성요소

SGML은 이기종 시스템들 간의 전자문서 교환을 위해 마크업 언어 구문을 정의하는 메타 언어(meta language)이다[1]. 이 SGML을 이용하여 문서를 작성하기 위해서는, 우선 문서를 분석해서 구조를 인식한 후, 마크업을 이용하여 정의하고, 이 마크업에 맞춰서 실제 문서를 기술하는 과정이 필요하다.

SGML 문서는 크게 SGML 선언부(Declaration), 문서 형 정의부(Document Type Definition, DTD), SGML 문서실례부(SGML Document Instance)로 이루어진다.

SGML 선언부는 문서가 한 시스템에서 다른 시스템으로 전송되었을 경우 시스템간에 사용하는 문자 코드가 같도록 하고, 특별한 역할을 하는 코드에 대해 해석이 같도록하기 위해서 사용된다. 이를 위해 선언부에서는 SGML 문서에서 이용하는 문자 집합, 문서에서 특수한 기능을 하는 문자에 대한 설명, 그리고 문서에서 사용되는 여러 기능등을 기술한다. 본 논문에서는 표준으로 참조 구체적 구문(reference concreate syntax) ISO 8879를 가정한다.

문서 형 정의부는 문서의 전체적인 논리적 구조를 정의한다. 문서 형 정의는 SGML 선언부에서 정의된 구문(syntax)에 맞춰 작성해야만 한다. 여기서는 엘리먼트(ELEMENT), 엔티티(ENTITY), 애트리뷰트리스트(ATTLIST)의 세가지 기본 구성요소를 이용하여 문서의 논리적 구조를 정의한다. (그림 2.1)은 Memo 문서를 위한 문서 형 정의부의 한 예이다.

SGML 문서실례부 는 정의된 문서 형 정의부에 맞추어 마크업을 하면서 문서 내용을 삽입하는 부분이다. 사용자가 입력을 쉽게 하기 위해서 엔티티와 생략 태그가 이용된다. (그림 2.2)는 (그림 2.1)에서 정의한 DTD에 따라서 작성된 SGML 문서의 한 예를 보인 것이다.

```
<!-- DTD for simple office memoranda                                    -->

<!ENTITY % doctype   "Memo" --  document type generic identifier     -->
<!ENTITY % details   "Q|Pref"                                          >

<!--        ELEMENTS   MIN    CONTENT (EXCEPTIONS)                     -->
<!ELEMENT   Memo       - -    ((To & From), Body, Close?)               >
<!ELEMENT   (To|From)  - O    (#PCDATA)                                 >
<!ELEMENT   Body       - O    (P*)                                      >
<!ELEMENT   P          - O    (#PCDATA|%details)*                       >
<!ELEMENT   Q          - -    (#PCDATA)                                 >
<!ELEMENT   Pref       - O    EMPTY                                     >
<!ELEMENT   Close      - O    (#PCDATA)                                 >

<!--        ELEMENTS   NAME     VALUE              DEFAULT           -->
<!ATTLIST   Memo       STATUS   (confiden|public)  public             >
<!ATTLIST   P          id       ID                 #IMPLIED          >
<!ATTLIST   Pref       refid    IDREF              #REQUIRED         >
```

(그림 2.1) 문서 형 정의부(DTD)의 예(Memo DTD)[2]
(Fig. 2.1) An example of DTD(Memo DTD)

```
<!DOCTYPE Memo SYSTEM "C:\MYDIR\MEMO.DTD">
<Memo>
<To>Comrade Napoleon</To>
<From>Snowball</From>
<Body>
<P>In Animal Farm, George Orwell says: <Q>...the pigs had to
expend enormous labour every day upon mysterious things called
files, reports, minutes and memoranda. These were large sheets
of paper which had to be closely covered with writing, and as
soon as they were so covered, they were burnt in the furnace...
</Q>. Do you think SGML would have helped the pigs?
</P>
</Body>
<Close>Comrade Snowball</Close>
</Memo>
```

(그림 2.2) SGML 문서실례부의 예(Memo 문서)[2]
(Fig. 2.2) An example of SGML document instance(Memo document)

2.2 DTD 기술 방법

DTD의 주요 구성요소에는 엘리먼트(ELEMENT), 엔티티(ENTITY), 애트리뷰트리스트(ATTLIST)가 있으며, 그외 시스템에 의존하는 처리를 위한 처리 명령어(processing instruction) 선언, 그래픽 데이타 등과 같은 비 SGML(Non-SGML) 데이타를 위한 노테이션(notation) 선언, 마크업을 보다 쉽게 하기 위한 단축 참조표(short reference map) 선언 등이 있다. 본 논문에서는 엘리먼트, 엔티티, 애트리뷰트리스트등의 세가지 주요 구성요소만을 상세히 다룬다.

2.2.1 엘리먼트

엘리먼트는 문서의 구조에 대한 논리적인 요소이며, 엘리먼트의 이름은 SGML 문서에 마크업할 때 태그 역할을 한다. 예를 들어, (그림 2.1)의 엘리먼트 'Memo'에 대한 선언을 보면, 한 엘리먼트의 구성은 엘리먼트 이름, 생략 규칙, 그리고 그 엘리먼트의 내용 모델로 이루어진다.

내용 모델은 엘리먼트들(예 : To, From, Body, Close)이 여러가지 연결자들(',' , '&' , '|')과 발생 지시자들('?', '+', '*')로 구성된 형태를 취한다.

연결자(connector)는 엘리먼트들 사이의 관계를 나타내며 세가지 종류가 있다. SEQUENCE 연결자(,)는 각 하위 엘리먼트들이 순서대로 나타나야 함을 의미한다. 예를 들어, 〈!ELEMENT Memo--(To, Body)〉에서 To 엘리먼트 다음에 반드시 Body 엘리먼트가 와야 함을 의미한다. AND 연결자(&)는 순서를 갖지 않지만 반드시 문서상에서 요소들이 모두 있어야 함을 의미한다. 예를 들어, 〈!ELEMENT Memo--(To & From)〉에서 To 엘리먼트와 From 엘리먼트가 반드시 와야하는데 어떤게 먼저라는 순서가 없음을 의미한다. OR 연결자(|)는 표시된 엘리먼트 혹은 모델 그룹중에 하나만이 발생할 수 있음을 의미한다. 예를 들어, 〈!ELEMENT P-O (Q|Pref)〉에서 Q 엘리먼트 또는 Pref 엘리먼트가 올 수 있음을 의미한다. 본 논문의 메타스키마 기술에서는 ','를 ORDERING으로, '&'를 ANDRING으로, '|'를 ORING으로 표기한다.

발생 지시자(occurrence indicator)는 엘리먼트 또는 내용 그룹(content group)이 문서상에서 몇 번이나 발생할 수 있는지를 나타내며 역시 세가지 종류가 있다. 선택적 발생 지시자(?)는 엘리먼트 혹은 모델 그룹이 기껏해야 한번 발생하거나 발생하지 않음을 의미한다. 반복 발생 지시자(+)는 엘리먼트가 한번 이상 나타남을 의미한다. 선택적 반복 발생 지시자(*)는 엘리먼트가 0번 이상 발생함을 의미한다.

선언된 내용은 SGML 파서(parser)에 의해서 특별한 방법으로 처리되어야 할 필요가 있는 엘리먼트들을 포함하는 경우에 사용된다. 가능한 데이타 형의 종류는 CDATA, RCDATA, #PCDATA, EMPTY, ANY 등이 있다.

2.2.2 엔티티

SGML에서 모든 형태의 데이타를 기호 이름으로 정의할 수 있도록 지원하는 기능이다. 엔티티 참조시에, SGML 파서는 엔티티로 정의된 기호 이름을 해당 데이타로 대치할 수도 있다. 엔티티는 긴 문자열을 간단히 표시하거나, 자판에서 제공되지 않는 문자를 사용할 때 또는 외부 문서를 포함시키거나, 이미지 등과 같은 비 SGML 데이타를 위하여 사용된다.

이러한 엔티티는 크게 일반 엔티티와 파라메타 엔티티로 나뉘어지며, 이들은 각각 내부 엔티티와 외부 엔티티로 다시 나뉜다.

2.2.3 애트리뷰트리스트

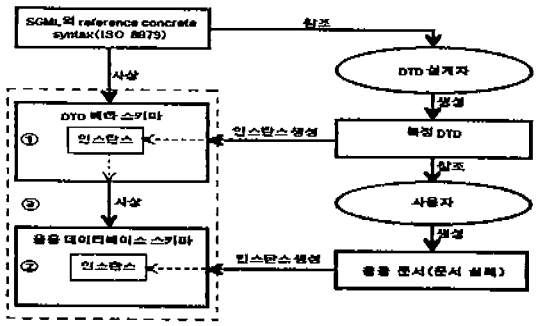
각 엘리먼트는 하나 이상의 애트리뷰트를 가질 수 있다. 각 애트리뷰트는 엘리먼트에 부가적인 정보를 제공한다. 엘리먼트에 정보를 추가하는 애트리뷰트는 해당 엘리먼트의 애트리뷰트명과 값, 그리고 디폴트 값 등을 정의하여 나타낸다. 예를 들면, 〈!ATTLIST Memo status (confiden|public) public〉에서 엘리먼트 Memo는 애트리뷰트 status를 가지며, 그 값은 confiden 혹은 public이고, 디폴트 값은 public임을 나타낸다. 문서실례에서 애트리뷰트는 시작 태그에 포함된다. 예를 들어, 〈Memo status=confiden〉는 Memo 엘리먼트의 애트리뷰트 status의 값이 'confiden'임을 나타낸다.

3. SGML DTD 메타 스키마 설계

이 장에서는 SGML DTD를 객체지향 데이타베이스에서 관리할 수 있도록 SGML DTD 메타 스키마를 설계하고 이를 이용하여 SGML DTD 정보를 관리하는 방법을 알아 본다. (그림 3.1)은 이 논문에서 제안하는 것을 그림으로 표현한 것이다. ①부분은 이 장에서 다루는 부분이고, ②부분은 다음 장에서 다룬다. ①과 ②를 합한 ⓐ부분이 실제로 이 논문에서 제안하는 부분이다.

3.1 SGML DTD의 메타 스키마 설계

(그림 3.2)는 SGML DTD를 객체지향 데이타베이스로 사상하기 위해 설계한 메타 스키마를 OMT 객체도[5]를 이용하여 표현한 것이다. 즉, 클래스는 클래스 명과 속성을 갖는 네모로 나타내고 클래스들 간의



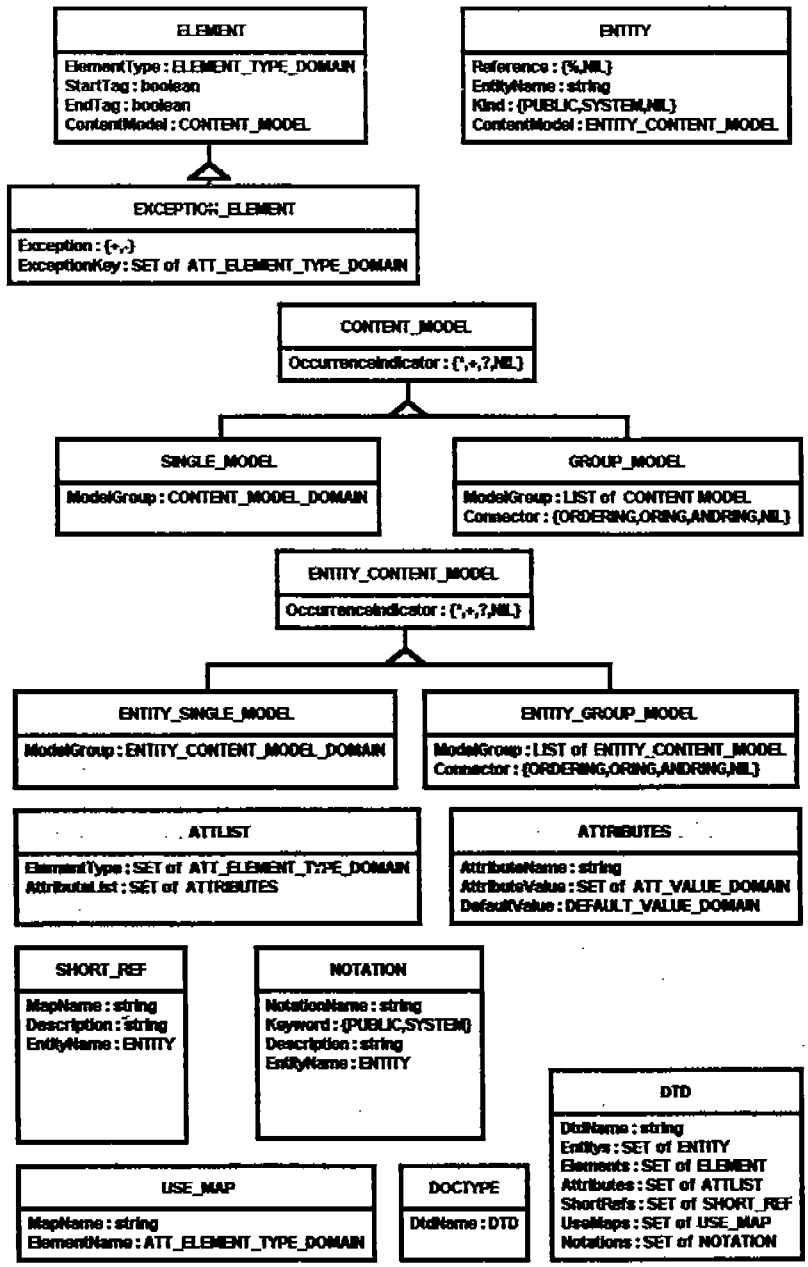
(그림 3.1) SGML DTD를 위한 메타 스키마 및 응용 스키마 사상 과정

(Fig. 3.1) Meta schema and application schema mapping system for SGML DTD

상속 관계는 작은 세모로 표현하였다.

SGML DTD에서 키워드(〈!DOCTYPE, 〈!ENTITY, 〈!ELEMENT, 〈!NOTATION, 〈!ATTLIST, 〈!SHORTREF, 〈!USEMAP중 하나)로 시작하는 구성요소들을 각각 묶어 7개의 기본 클래스들(DOCTYPE, ENTITY, ELEMENT, NOTATION, ATTLIST, SHORTREF, USEMAP)로 정의하고, 키워드와 함께 기술된 구성요소들의 구문(syntax)에 해당하는 파라미터(parameter)들은 각 클래스의 속성들로 표현한다. 예를 들어, 클래스 ELEMENT는 엘리먼트 이름부를 ElementType으로, 시작태그를 StartTag로, 종료태그를 EndTag로, 내용모델을 ContentModel로 하는 속성을 가진다.

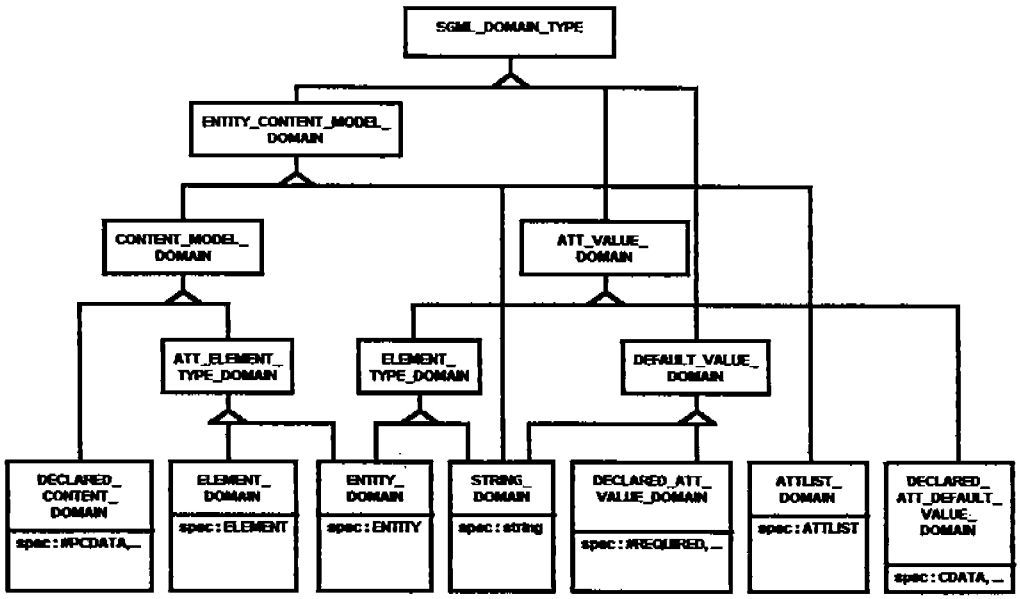
이 클래스들중 클래스 CONTENT_MODEL은 엘리먼트의 정의 내용인 내용 모델을 지원하기 위한 것으로, 한개의 구성요소로 이루어진 내용 모델을 저장하는 클래스 SINGLE_MODEL과 여러개의 다양한 형태들이 그룹으로 묶여져 있는 내용 모델을 저장하는 클래스 GROUP_MODEL을 하위클래스(subclass)로 갖는다. 클래스 ENTITY_CONTENT_MODEL은 엔티티의 정의 내용인 대치 영역을 지원하기 위한 것으로, 한 개의 구성요소로 이루어진 대치 영역을 위한 클래스 ENTITY_SINGLE_MODEL과 여러개의 다양한 형태들이 그룹으로 묶여진 경우를 위한 클래스 ENTITY_GROUP_MODEL을 하위클래스로 갖는다. 클래스 ATTLIST의 속성 AttributeList의 값으로 한 개의 클래스에 여러개의 속성명이 존재하는 경우가 있는데 이를 지원하기 위해 ATTRIBUTES라는 클래스를 정의하였다. 그리고 포함자(INCLUDE)



(그림 3.2) SGML DTD 메타 스키마를 위한 객체도
(Fig. 3.2) Object diagram for SGML DTD meta schema

를 파라미터로 가지는 엘리먼트를 위해 클래스 EXCEPTION_ELEMENT를 클래스 ELEMENT의 하위클래스로 둔다. 각 클래스의 용법은 3.2절에서 다루어진다.

SGML DTD를 위한 메타 스키마에서, 클래스의 속성 도메인에 나타날 수 있는 타입들이 무척 다양하고 복잡하여 DTD의 구성요소로서 나타나는 클래스 계층 구조와는 별도로 도메인 타입 계층 구조를 설계한



(그림 3.3) SGML DTD를 위한 속성 도메인 클래스 타입 계층
(Fig. 3.3) Domain class hierarchy for SGML DTD

다. 이 구조에서 단말 클래스의 속성은 (그림 3.2)에서 정의된 해당 클래스의 객체식별자만을 가진다(STRING_DOMAIN, DECLARED_COTENT_DOMAIN, DECLARED_ATT_DEFAULT_DOMAIN은 실제 데이타). (그림 3.2)에서 정의된 해당 클래스 역시 도메인 타입 계층 구조에서 정의된 클래스의 객체식별자를 가지며 최종적으로 DECLARED_COTENT_DOMAIN, DECLARED_ATT_DEFAULT_DOMAIN, STRING_DOMAIN 클래스에서 실제 데이타를 가진다. (그림 3.3)은 SGML DTD를 위한 속성 도메인 클래스 타입 계층이다.

3.2 SGML DTD의 메타 스키마 사상

이 절에서는 SGML DTD를 앞 절의 SGML DTD 메타 스키마에 인스탄스로 사상시키는 과정을 기술한다.

앞으로 발생되는 각 인스탄스는 설명의 편의상 해당 클래스명을 소문자로 표시한 것에 발생하는 순서대로 첨자를 붙여서 나타내며, 객체식별자는 그 인스탄스에 첨자 'Oid'를 붙여서 나타낸다. 그리고 인스탄스의 내용은 해당 클래스의 속성에 따라 값을 표현하여 소괄호로 묶어 객체식별자 뒤에 붙인다. 한편 속성의 값이 여러개의 리스트로 구성되는 경우는 중괄호([])를 사용하여 나타낸다. 그리고 이러한 과정을 지원하기 위해 임시 클래스 UNDEFINED_ELEMENT, UNDEFINED_ENTITY, 그리고 UNDEFINED_EXCEPTION을 생성시켜 이용한다. 이들은 설계된 DTD가 메타 스키마에 사상이 완료되면 자동적으로 소멸된다.

SGML DTD의 메타 스키마로의 사상은 크게 6단계로 이루어진다.

[단계 1] SGML DTD의 엔티티들을 데이타베이스 스키마에 사상

DTD에 나타난 각 엔티티는 클래스 ENTITY의 한 인스탄스로 사상되며 엔티티를 위한 객체식별자와 엔티티 객체 속성값들을 가진다. ENTITY 클래스의 정의는 아래와 같다.

```
ENTITY(Reference:{%, NIL}, EntityName:string,
Kind:{PUBLIC, SYSTEM, NIL},
ContentModel:ENTITY_CONTENT_MODEL);
```

이 클래스 정의에 따라 실제 발생되는 객체의 예 entity; 는 아래와 같다.

entity$_i$:(Reference:%, EntityName:"details", Kind:NIL, ContentModel:entity_group_modeliOid)

앞에서 언급한 바와 같이 인스탄스는 그것의 클래스명을 소문자로 표기하여 발생한 순서대로 첨자를 붙여서 나타낸다. 예를 들어, 클래스 ENTITY의 인스탄스는 entity$_i$와 같이 표기되며, 첨자 i는 발생된 순서를 나타낸다. 그리고 ':'는 객체식별자가 가리키는 객체를 나타내고 있다. 이후 객체식별자를 이용하여 객체를 언급할 경우, 그 객체식별자가 가리키는 객체를 뜻한다. 각 속성값이 결정되는 과정은 다음과 같다.

Reference는 엔티티 참조 형태를 구분하여 참조 형태가 파라메타 엔티티인 경우는 '%', 일반 엔티티인 경우는 'NIL'을 속성값으로 한다. EntityName은 문자열("")을 사용하여 기술한다. Kind는 어느 종류의 엔티티를 참조했는가를 구분하기 위한 것으로, 엔티티 참조가 공용으로 선언된 외부 엔티티를 참조할 경우에는 'PUBLIC', 특정 시스템에서 내부적으로 참조할 경우에는 'SYSTEM', 해당 문서에서 새롭게 정의하여 사용할 경우에는 'NIL'을 속성값으로 한다. ContentModel은 엔티티가 파라메타 엔티티인 경우, 엔티티의 대치 영역이 여러개의 구성요소들로 이루어져 있으면 그들을 하나로 묶어 클래스 ENTITY_GROUP_MODEL의 인스탄스로 발생시키고, 한 개의 구성요소로 이루어져 있으면 클래스 ENTITY_SINGLE_MODEL의 한 인스탄스로 생성시킨 후 ContentModel의 속성값으로 그에 해당되는 객체식별자를 기술한다. ENTITY_GROUP_MODEL과 ENTITY_SINGLE_MODEL의 클래스 정의는 다음과 같다.

```
ENTITY_GROUP_MODEL(OccurrenceIndicator:
    {*, +, ?, NIL},
    ModelGroup:LIST of ENTITY_CONTENT_MODEL,
    Connector:{ORDERING, ORING, ANDRING, NIL});
ENTITY_SINGLE_MODEL(OccurrenceIndicator:
    {*, +, ?, NIL},
    ModelGroup:ENTITY_CONTENT_MODEL_DOMAIN);
```

여기에 따라 발생되는 entity_single_model$_i$의 ModelGroup이 문자열이거나 이미 정의된 키워드(#PCDATA, EMPTY, CDATA 등) 형태인 경우 그 문자열이나 키워드를 직접 기술한다. 그러나 아직 생성되지 않은 엘리먼트나 엔티터 객체를 그 속성값으로 할 경우, 그 엘리먼트나 엔티티 객체가 생성될 때 그 객체를 속성값으로 하는 entity_single_model$_i$과 연결되어야 한다. 그래서 두 임시 클래스 UNDEFINED_ELEMENT(Object:Oid, ElementName:string)와 UNDEFINED_ENTITY(Object:Oid, EntityName:string)에 생성되지 않은 엘리먼트나 엔티티 객체 정보를 저장한다. 이때 ModelGroup의 속성값이 엘리먼트 객체이면 클래스 UNDEFINED_ELEMENT 클래스에 객체식별자 entity_single_model$_i$을 Object 속성값으로 등록하고 엘리먼트 이름을 속성 ElementName에 등록한다. ModelGroup의 속성값이 엔티티 객체이면 임시 클래스 UNDEFINED_ENTITY에 객체식별자 entity_single_model$_i$과 엔티티 이름을 각각 등록한다. 이때 한번 등록된 엘리먼트나 엔티티일지라도 객체 entity_single_model$_i$의 속성 OccurrenceIndicator 값이 다르면 계속 등록한다.

예를 들어, 〈ENTITY % details "Q|Pref"〉일 경우,

entity$_1$:(%, "details", NIL, entity_group_model$_{1Oid}$)
entity_group_model$_1$:(NIL, [entity_single_model$_{1Oid}$, entity_single_model$_{2Oid}$], ORING)
entity_single_model$_1$:(NIL, NIL)
entity_single_model$_2$:(NIL, NIL)
undefined_element$_1$:(entity_single_model$_{1Oid}$, "Q")
undefined_element$_2$:(entity_single_model$_{2Oid}$, "Pref")

가 된다. 여기서 'NIL'은 아직 해당 객체가 생성되지 않았음을 의미한다.

[단계 2] SGML DTD의 엘리먼트들을 메타 스키마에 사상

엘리먼트의 경우 엘리먼트가 예외키를 가지면 클래스 EXCEPTION_ELEMENT의 인스탄스가 되고 그렇지 않으면 클래스 ELEMENT의 인스탄스로 사상된다. 엘리먼트 클래스는 다음과 같은 속성과 속성형을 가진다.

```
ELEMENT(ElementType:ELEMENT_TYPE_DOMAIN,
    StartTag:boolean,
```

EndTag:boolean, ContentModel:CONTENT_MODEL);
EXCEPTION_ELEMENT(ElementType:ELEMENT_TYPE_DOMAIN, StartTag:boolean, EndTag:boolean, ContentModel:CONTENT_MODEL, Exception:{+, −}, ExceptionKey:SET of ATT_ELEMENT_TYPE_DOMAIN);

이에 따라 발생되는 객체 element$_i$의 속성값을 결정하는 과정은 아래와 같다.

ElementType의 속성값은 엘리먼트명이 되며, 이미 생성된 엔티티가 올때는 엔티티 객체식별자를 그 속성값으로 한다. StartTag, EndTag의 속성값에는 태그의 생략이 가능한 경우는 'T', 생략이 불가능할 경우는 'F'로 한다. 속성 ContentModel에는 내용 모델의 값이 한 개의 구성요소로 이루어져 있으면 클래스 SINGLE_MODEL의 인스탄스를 하나 생성시키고 그에 대한 객체식별자를 속성값으로 갖는다. 내용 모델의 구성요소가 여러개로 이루어져 있으면 클래스 GROUP_MODEL의 한 인스탄스를 발생시키고 그에 대한 객체식별자를 속성값으로 갖는다. 클래스 SINGLE_MODEL과 GROUP_MODEL의 인스탄스 객체가 갖는 속성과 속성형은 다음과 같다.

GROUP_MODEL(OccurrenceIndicator:{*, +, ?, NIL}, ModelGroup:LIST of CONTENT_MODEL, Connector:{ORDERING, ORING, ANDRING, NIL});
SINGLE_MODEL(OccurrenceIndicator:{*, +, ?, NIL}, ModelGroup:CONTENT_MODEL_DOMAIN);

이에 따라 발생되는 객체 single_model$_i$의 ModelGroup이 이미 정의된 키워드(#PCDATA, CDATA, EMPTY 등) 형태인 경우 그 키워드를 직접 기술한다. 만약 아직 생성되지 않은 엘리먼트나 엔티티 객체를 그 속성값으로 할 경우에는, ModelGroup의 속성값이 엘리먼트 객체이면 임시 클래스 UNDEFINED_ELEMENT에 객체식별자 single_model$_i$와 엘리먼트명을 각각 등록하고, ModelGroup의 속성값이 엔티티 객체이면 UNDEFINED_ENTITY 클래스에 객체식별자 single_model$_i$와 엔티티명을 각각 등록한다. 이때 한번 등록된 엘리먼트나 엔티티일지라도 객체 single_model$_i$의 속성 OccurrenceIndicator값이 다르면 계속 등록한다. Exception은 예외키가 있을 경우 include이면 '+'를 exclude이면 '−'를 속성값으로 갖는다. ExceptionKey는 예외키로 사용되는 해당 엘리먼트에 대한 객체식별자를 속성값으로 갖고 아직 정의되지 않은 엘리먼트인 경우에는 임시 클래스 UNDEFINED_EXCEPTION(Object:Oid, ElementName:SET of string)에 객체식별자 exception_element$_i$와 예외키 엘리먼트들을 등록한다.

예를 들어, <!ELEMENT P-O (#PCDATA|%details)*>일 경우,

element$_1$:('P', F, T, group_model$_{1Oid}$)
group_model$_1$:(*, [single_model$_{1Oid}$, single_model$_{2Oid}$], ORING)
single_model$_1$:(NIL, #PCDATA)
single_model$_2$:(NIL, NIL)
undefined_entity$_1$:(single_model$_{2Oid}$, "details")

가 된다.

[단계 3] 정의되지 않은 엘리먼트, 엔티티, 예외 정보를 찾아 사상

임시 클래스 UNDEFINED_ELEMENT의 속성 ElementName의 값과 클래스 ELEMENT의 속성 ElementType의 값에 해당하는 객체의 값을 비교하여 element$_i$를 얻은 다음 Object의 값에 해당하는 객체를 찾는다. 그런 후 속성 Object의 값에 해당하는 객체의 속성 ModelGroup 값으로 객체식별자 element$_i$를 사상시킨다. 임시 클래스 UNDEFINED_ENTITY와 UNDEFINED_EXCEPTION에서도 같은 방법으로 사상한다. 그런 다음 두 클래스에서 그 정보를 삭제한다. 예를 들면, undefined_entity$_1$:(single_model$_{2Oid}$, "details"), single_model$_2$:(NIL, NIL), entity$_1$:(%, "details", NIL, entity_group_model$_{1Oid}$)일 경우,

single_model$_2$:(NIL, entity$_{1Oid}$)
entity$_1$:(%, "details", NIL, entity_group_model$_{1Oid}$)

가 된다.

[단계 4] 엘리먼트들과 엔티티에 대한 애트리뷰트리스트를 메타 스키마에 사상

애트리뷰트리스트는 클래스 ATTLIST의 인스탄스들로 발생되며 다음과 같은 속성들을 가진 형태로 기술된다.

ATTLIST(ElementType:ATT_ELEMENT_TYPE_DOMAIN, AttributeList:SET of ATTRIBUTES);

이에 따라 발생되는 객체 $attlist_i$의 ElementType은 애트리뷰트리스트가 속한 엘리먼트나 엔티티의 객체식별자를 속성값으로 갖는다. AttributeList는 ElementType에 나타난 엘리먼트나 엔티티의 애트리뷰트들을 클래스 ATTRIBUTES의 인스탄스로 발생시키고 그 인스탄스 객체식별자들의 집합을 속성값으로 갖는다. 이때 발생되는 클래스 ATTRIBUTES의 인스탄스 객체들이 가지는 속성은 다음과 같다.

ATTRIBUTES(AttributeName:string,
AttributeValue:SET of ATT_VALUE_DOMAIN,
DefaultValue:DEFAULT_VALUE_DOMAIN);

[단계 5] 기타 NOTATION, SHORTREF, USEMAP 등을 메타 스키마에 사상

지금까지 사상되지 않은 NOTATION, SHORTREF, USEMAP 등을 각각 메타 스키마의 해당 클래스에 사상시킨다.(본 논문에서는 생략한다.)

[단계 6] 발생된 모든 엘리먼트와 엔티티, 애트리뷰트리스트들의 인스탄스들을 포함하는 DTD를 DTD 클래스에 사상

DTD 문서를 위한 객체가 가지는 속성은 다음과 같다.

DTD(DtdName:string, Entitys:SET of ENTITY,
Elements:SET of ELEMENT, Attributes:SET of ATTLIST,
ShortRefs:SET of SHORT_REF,
UseMaps:SET of USE_MAP,
Notations:SET of NOTATION);

속성 DtdName에는 DTD의 이름을 갖게하고 이제까지 발생된 모든 엔티티 인스탄스 객체식별자들과 모든 엘리먼트 인스탄스 객체식별자들, 참조, 사상, 노테이션식별자들 그리고 애트리뷰트리스트 인스탄스 객체식별자들이 속성 Entitys, Elements, Attributes, ShortRefs, UseMaps, Notations의 속성값이 된다. (ShortRefs, UseMaps, Notations 등은 본 논문에서 생략한다.)

(그림 3.4)는 SGML DTD 메타 스키마에 저장된 Memo DTD((그림 2.1))를 위한 인스탄스 객체들이다.

```
doctype1 : (dtd1);
entity1 : (%, "details", NIL, entity_group_model10id);
entity2 : (%, "doctype", NIL, entity_single_model10id);
entity_group_model1 : (NIL, [entity_single_model20id, entity_single_model30id],
ORING);
entity_single_model1 : (NIL, element10id);
entity_single_model2 : (NIL, element60id);
entity_single_model3 : (NIL, element70id);

element1 : ('Memo', F, F, group_model10id);
group_model1 : (NIL, [group_model20id, single_model10id,
            single_model20id], ORDERING);
group_model2 : (NIL, [single_model30id, single_model40id], ANDRING);
single_model1 : (NIL, element40id);
single_model2 : (?,   element60id);
single_model3 : (NIL, element20id);
single_model4 : (NIL, element30id);
element2 : ('To', F, T, single_model50id);
```

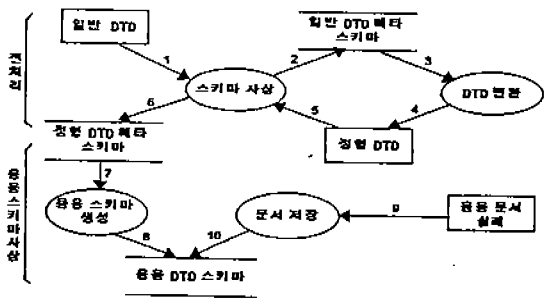
```
single_model5 : (NIL, #PCDATA);
element3 : ('From', F, T, single_model50id);
element4 : ('Body', F, T, single_model60id);
single_model6 : (*, element50id);
element5 : ('P', F, T, group_model30id);
group_model3 : (*, [single_model50id, single_model70id], ORING);
single_model7 : (NIL, entity10id);
element6 : ('Q', F, F, single_model50id);
element7 : ('Pref', F, T, single_model80id);
single_model8 : (NIL, EMPTY);
element8 : ('Close', F, T, single_model50id);

attlist1 : (element10id, {attributes10id});
attlist2 : (element50id, {attributes20id});
attlist3 : (element70id, {attributes30id});
attributes1 : ('STATUS', {confiden|public}, public);
attributes2 : ('id', {ID}, #IMPLIED);
attributes3 : ('refid', {IDREF}, #REQUIRED);

dtd1 : ('Memo', {entity10id, entity20id}, {element10id, element20id, element30id,
            element40id, element50id, element60id, element70id, element80id},
            {attlist10id, attlist20id, attlist30id})
```

(그림 3.4) 메타 스키마에 Memo DTD를 사상시킨 결과
(Fig. 3.4) Meta schema mapping result of Memo DTD

4. DTD 응용 데이타베이스 스키마 사상

이 장은 SGML DTD 메타 스키마에 인스탄스로 저장된 DTD에서 응용 데이타베이스 스키마를 자동적으로 생성시키는 방법을 기술한다. 모든 SGML DTD들은 메타 스키마의 객체 인스탄스들로 저장되며, 이들을 이용하여 SGML 문서 인스탄스가 저장될 수 있는 응용 데이타베이스 스키마가 생성된다. 이 장에서 다루어질 과정들을 그림으로 표현하면 (그림 4.1)과 같다. 그림에서 보는 것과 같이 특정 DTD를 메타 스키마에 사상하고, 이를 기초로 응용 스키마를 생성하고, 이 응용 스키마에 응용 문서를 저장하기 위해서는 DTD 정보가 1에서 6까지의 전처리 작업을 포함하여 10까지의 경로를 거치게 된다.

우리의 SGML 문서 데이타베이스는 파라미터 엔티티(parameter entity) 참조가 나타나는 부분은 파라미터 엔티티로 정의된 각각의 객체식별자들을 분해하여 대치한 정형(canonical) DTD의 형태로 관리된다. 그래서 모든 DTD는 정형 DTD로 번역된 다음(전처리) 응용 데이타베이스 스키마를 생성하게 된다. 여기서 일반 DTD 메타 스키마는 일반 DTD를 추출해 내기 위해 필요하다. 따라서 이를 포함한 응용 데이타베이스 스키마 생성은 크게 전처리 단계와 응용 스키마 사상 단계로 이루어 진다.



(그림 4.1) SGML DTD의 메타 스키마 사상 및 응용 스키마 사상
(Fig. 4.1) SGML DTD meta schma mapping and application schema mapping

[전처리 단계] 일반 DTD(엔티티가 포함된 DTD)를 정형 DTD로 번역한다.

정형 DTD는 엘리먼트와 애트리뷰트리스트에 엔티티 참조가 나타나지 않는 형태를 말한다. 그리고 엘리먼트 정의에서 엘리먼트 이름부에 OR(|)로 묶여져 있는 요소들 각각은 분해되어 같은 내용 모델을 가지는 형태이다. 즉, 정형 DTD에서는 엘리먼트 정의의 머리부에 항상 하나의 엘리먼트만 올 수 있고, 애트리뷰트리스트의 엘리먼트명에도 하나의 엘리먼트만 올 수 있다. 또한 엘리먼트와 애트리뷰트리스트에 나타나는 엔티티 참조는 그 엔티티로 정의된 객체식별자들로 대치되어진 형태이다.

일반 DTD를 정형 DTD로 변환하기 위해서는 3장에서 처리하여 일반 DTD 메타 스키마에 사상한 것 중, 변환하고자 하는 DTD를 추출한 후 엔티티 참조를 처리하고 엘리먼트 이름부와 애트리뷰트리스트의 엘리먼트명에 OR(|)로 묶여져 있는 요소들도 처리하여 정형 DTD로 변환한다. 그런후 다시 3장에서 제안한 방법으로 정형 DTD 메타 스키마에 사상한다.

[응용 스키마 사상 단계] 정형 SGML DTD 메타 스키마에서 응용 DTD 메타 스키마를 생성한다.

우선 클래스 DTD의 인스탄스들 중 속성 DtdName의 값과 DTD명이 같은 것을 찾은 다음 각 엘리먼트 및 해당 속성들을 하나씩 분석하여 처리한다. 정형 DTD 클래스의 속성 Elements의 값인 엘리먼트 인스탄스 객체들의 속성들을 차례로 분석하여 클래스와 속성을 생성한다. 생성되는 클래스는 임시 클래스인 UNDEFINED_CLASS에 저장한다. 클래스를 만들 때는 이미 그 이름으로 존재하는 클래스가 있는지를 임시 클래스 UNDEFINED_CLASS에서 조사한 후, 있으면 그 클래스명과 클래스 ELEMENT의 속성 ElementType이 같은 것을 찾는다. 그리고 그 클래스 ELEMENT의 속성 ContentModel을 계속 조사하여 속성을 발생시킨다. 가능한 속성이 모두 발생되면 임시 클래스 UNDEFINED_CLASS에서 그 클래스명을 삭제한다. 이는 크게 다섯 과정으로 이루어 진다.

(1) 엘리먼트의 속성 ElementType의 값을 이름으로하는 클래스를 생성한다.

예를 들어, $element_1$:('Memo', F, F, $group\_model_{10oid}$)일 경우, Memo()와 $undefined\_class_1$(Memo)가 생성된다.

(2) 엘리먼트의 속성 ModelGroup의 값이 SINGLE_MODEL 객체인지 GROUP_MODEL 객체인지 검사한다.

(2.1) ModelGroup의 값이 SINGLE_MODEL 객체인 경우, 클래스 SINGLE_MODEL의 객체 $single_model_i$의 ModelGroup의 값을 검사한다.

(2.1.1) ModelGroup의 값이 엘리먼트 객체인 경우, 해당 엘리먼트 객체를 찾아 ElementType의 값을 이름으로하는 속성을 만들고, 같은 이름으로 새로운 클래스를 생성한다. 본 논문에서는 속성명을 클래스명에 첨자 'A'를 붙여서 표기한다. 생성된 속성의 속성값은 새롭게 만들어진 클래스의 객체식별자가 된다. 본 논문에서는 속성값을 새롭게 생성된 클래스명에 첨자 'Oid'를 붙여서 표기한다. 그리고 속성 OccurrenceIndicator의 값을 검사하여 '+'나 '*'이면 만들어진 속성값은 생성된 엘리먼트 객체들의 리스트들로 이루어진다.

예를 들어, $element_4$:('Body', F, T, $single\_model_{6oid}$), $single\_model_6$:(*, $element_{5oid}$), $element_5$:('P', F, T, $group\_model_{3oid}$)일 경우, $single_model_6$의 ModelGroup 값이 $element_5$이므로 $element_5$의 ElementType의 값을 이름으로 하는 클래스 P()를 새로 생성시키고, 클래스 Body에 P_A라는 속성을 생성시킨 후 새롭게 생성된 P 클래스의 객체식별자를 속성 P_A의 값으로 가진다. 그리고 클래스의 OccurrenceIndicator 값이 '*'이므로 Body 클래스에서 속성 P_A의 값은 P의 리스트가 된다. 따라서 최종적으로 Body(P_A : LIST of P_{oid}), P()가 생성된다.

(2.1.2) ModelGroup의 값이 지정된 키워드(#PCDATA, EMPTY 등)인 경우,

속성을 발생시키고 있는 클래스에 $PCDATA_A$라는 이름의 속성을 만들고, 역시 PCDATA라는 이름으로 새로운 클래스를 생성한다. 생성된 속성 $PCDATA_A$의 값은 새롭게 만들어진 클래스 PCDATA의 객체식별자가 되며, 클래스 PCDATA에는 도메인(domain) 값을 문자열로 하는 ModelGroup이라는 속성을 발생시킨다.

예를 들어, $element_2$:('To', F, T, single_

$model_{5Oid}$), $single\_model_5$:(NIL, #PCDATA)일 경우, To($PCDATA_A$:$PCDATA_{Oid}$), PCDATA(ModelGroup:string)가 생성된다.

(2.2) ModelGroup의 값이 GROUP_MODEL인 경우, $group_model_i$의 Connector의 값을 검사한다.

(2.2.1) Connector의 값이 ORING인 경우,

새로 생성시킨 클래스에 $group_model_i$이 발생되는 순서(n)에 따라 번호를 붙인 TMPn이라는 속성을 발생시킨다. 그리고 그 클래스의 클래스명과 문자열 'CLASS'를 연결한 명칭을 갖는 가상 클래스(virtual class)를 ORING 순서에 따라 번호를 붙여 생성한다. 생성된 가상 클래스의 객체식별자를 속성 TMPn의 속성값으로 발생시킨다.

속성 OccurrenceIndicator의 값을 검사해서 '+'나 '*'이면 속성 TMP의 값은 생성된 가상 클래스의 리스트(LIST of)가 된다.

속성 ModelGroup의 값이 SINGLE_MODEL의 객체일 경우에는 그 $single_model_i$의 ModelGroup 값을 이름으로 하는 클래스를 UNDEFINED_CLASS에서 찾는다. 이때 찾지 못하면 새로 생성시킨다. 이 클래스는 생성된 가상 클래스의 서브클래스가 된다. $single_model_i$의 ModelGroup 값을 검사하여 그 값이 지정된 키워드(#PCDATA, CDATA, EMPTY 등)인 경우에는 가상 클래스명과 'PCDATA'를 합친 명칭으로 새로운 클래스를 생성한다. 이때 새로 생성된 클래스는 가상 클래스의 서브클래스가 된다. 새로 생성된 클래스에는 $PCDATA_A$라는 이름의 속성을 만들고, 속성 $PCDATA_A$의 값은 클래스 PCDATA의 객체식별자가 된다. 그리고 $group_model_i$의 ModelGroup 값중 그 값이 $group_model_i$이고 Connector의 값이 ORING이면 다시 (2.2.1)을 수행하고, 그렇지 않으면 (2.2.2)를 수행한다.

예를 들어,

$element_5$:('P', F, T, $group\_model_{3Oid}$)

$group\_model_3$:(*, [$single\_model_{7Oid}$, $single\_model_{12Oid}$], ORING)

$single\_model_7$:(NIL, $element_{6Oid}$)

$single\_model_{12}$:(NIL, $element_{12Oid}$)

$element_6$:('Q', F, F, $single\_model_{5Oid}$)

$element_{12}$:('Pref', F, T, $single\_model_{9Oid}$)

$single\_model_5$:(NIL, #PCDATA)

$single\_model_9$:(NIL, EMPTY)일 경우,

P(TMP1:LIST of P_CLASS_{Oid})

P_CLASS()

Q($PCDATA_A$:$PCDATA_{Oid}$) isA P_CLASS

Pref() isA P_CLASS 가 생성된다.

(2.2.2) Connector의 값이 ORING이 아닌 경우,

속성을 발생시키고 있는 클래스에 $group_model_i$이 발생되는 순서에 따라 번호를 붙인 TMPn이라는 속성을 생성한다. 그리고 그 클래스의 클래스명과 문자열 'TMPn'를 연결한 명칭을 갖는 클래스를 생성한다. 그리고 속성 TMPn는 새롭게 생성된 이 클래스의 객체식별자를 값으로 가진다.

객체 $group_model_i$의 속성 OccurrenceIndicator의 값을 검사해서 '+'나 '*'이면 생성된 속성의 값은 생성된 TMP 클래스의 객체 리스트들이 된다. 그리고 생성된 클래스 TMP의 속성을 발생시킨다. 이때 객체 $group_model_i$의 속성 ModelGroup의 값이 $single_model_i$일 경우에는 (2.1)을 수행한다. 그렇지 않고 그 값이 $group_model_i$이면서 Connector의 값이 ORING일 경우에는 (2.2.1)을 수행하고 그렇지 않으면 (2.2.2)를 수행한다.

예를 들어, $element_1$:('Sample', F, F, $group\_model_{1Oid}$), $group\_model_1$:(*, [$single\_model_{1Oid}$, $single\_model_{2Oid}$], ORDERING)일 경우 Sample(TMP1:LIST of $Sample\_TMP_{1Oid}$), Sample_TMP1()가 생성된다.

(3) 엘리먼트가 EXCEPTION_ELEMENT의 객체인가를 검사한다.

(3.1) 속성 Exception을 검사한 후 생성된 클래스에 exceptionA이라는 이름으로 속성을 만들고 그 속성값은 '+' 또는 '−'로 한다.

(3.2) 속성 $exception_key_A$를 생성된 클래스에 추가하고 이 속성값은 ElementType의 값으로 한다. 그리고 이 속성값을 이름으로 하는 클래스를 생성한다. 이때 이미 클래스가 만들어져 있을 경우는 새로 생성하지 않는다.

예를 들어, $exception\_element_1$:('General', F, T, $single\_model_{5Oid}$, '+', 'Ix'), $single\_model_5$:(NIL, #PC-

DATA)일 경우, General($PCDATA_A$:$PCDATA_{Oid}$, exception$_A$:'+', exception_key$_A$:ix_{Oid})가 생성된다.

(4) 엘리먼트 객체를 메타 클래스 ATTRIBUTES에서 찾아서 가지고 있으면 'ATT_'와 속성 AttributeName의 값을 연결한 것을 명칭으로 하여 새로운 속성을 생성한다. 그리고 속성 AttributeValue의 값을 생성된 속성의 값으로 발생시킨다.

(5) UNDEFINED_CLASS에 임시로 저장된 클래스

```
Memo(TMP1: Memo_TMP1Oid, ATT_STATUS: {'public', 'confiden'}),
Memo_TMP1(TMP1: Memo_TMP1_TMP1Oid, BodyA: BodyOid, CloseA: CloseOid),
Memo_TMP1_TMP1(ToA: Toid, FromA: FromOid)
To(PCDATAA: PCDATAOid),
PCDATA(ModelGroup: string),
From(PCDATAA: PCDATAOid),
Body(PA: LIST of POid),
P(TMP1: LIST of P_CLASSOid, ATT_id: string),
P_CLASS(),              /* type = {P_CLASS_PCDATA, Q, Pref} */
P_CLASS_PCDATA(PCDATAA: PCDATAOid) isA P_CLASS,
Q(PCDATAA: PCDATAOid) isA P_CLASS,
Pref(ATT_refid: integer) isA P_CLASS,
Close(PCDATAA: PCDATAOid)
```

(그림 4.2) Memo 응용 문서를 위한 응용 데이타베이스 스키마
(Fig. 4.2) Application schema for Memo document instance

```
Memo1(Memo_TMP110id, 'public')
Memo_TMP11(Memo_TMP1_TMP110id, Body10id, Close10id)
Memo_TMP1_TMP11(To10id, From10id)
To1(PCDATA10id)
PCDATA1("Comrade Napoleon")
From1(PCDATA20id)
PCDATA2("Snowball")
Body1(P10id)
P1([P_CLASS_PCDATA10id, Q10id, P_CLASS_PCDATA20id], NULL)
P_CLASS_PCDATA1(PCDATA30id)
PCDATA3("In Animal Farm, George Orwell says :")
Q1(PCDATA40id)
PCDATA4("... the pigs had to expend enormous labour every day upon
  mysterious things called files, reports, minutes and memoranda. These
  were large sheets of paper which had to be closely covered with writing,
  and as soon as they were so covered, they were burnt in the furnace...")
P_CLASS_PCDATA2(PCDATA50id)
PCDATA5("Do you think SGML would have helped the pigs?")
Close1(PCDATA60id)
PCDATA6("Comrade Snowball")
```

(그림 4.3) Memo 응용 문서 실례의 저장 결과
(Fig. 4.3) Result of Memo document instance

를 읽어서 (2)부터 차례로 수행한다.

이러한 생성 절차를 거쳐 (그림 3.4)의 SGML DTD 메타 스키마에 인스탄스로 저장된 Memo DTD를 Memo 응용 데이타베이스 스키마로 생성한 결과는 (그림 4.2)와 같다.

그리고 이렇게 생성된 응용 데이타베이스 스키마에는 응용 문서 실례들이 저장된다. (그림 2.2)의 Memo 문서를 발생시킨 결과는 (그림 4.3)과 같다.

5. 관련 연구

SGML 문서를 위한 다양한 데이타베이스 모델들이 [6]에서 제시되었다. 여기서 제시된 모델들은 스키마에서 DTD를 직접 표현하거나, DTD를 위한 스키마를 외부적으로 정의한 것들이다. 우리도 후자의 접근을 따르지만 객체지향 데이타베이스에서 다룬다. 객체 지향 DBMS를 사용할 때에는 집성화(aggregation), 일반화(generalization), 동적 스키마 변경(dynamic schema modification) 등의 특성을 제공 받을 수 있다.

[7]의 작업은 구조화된 문서를 관계형 DBMS에서 SGML로 다루기 위해 SQL을 확장한 후자의 예이다.

[8]과 [9]는 하이퍼미디어의 관리에 관계형 데이타베이스가 문제가 있음을 보이고 구조화된 멀티미디어 문서의 검색과 저장에 객체지향 데이타베이스를 사용하는 예를 보였다. 그리고 [10]은 SGML 문서의 질의 처리에 대해 중점적으로 연구하였다. 본 논문은 이들에 비해 주로 SGML 문서를 위한 객체 지향 모델링에 중점을 두었으며, 앞으로 이를 기초로 다양한 질의 처리에 대해 연구해야 한다.

[11]은 SGML 문서의 효과적인 저장을 위해 OBMS를 설계하였는데, 특히 동적인 DTD를 메타 클래스로 구현하는 방법을 보였다. 이 OBMS의 응용인 D-STREAT에서는 DTD의 특징들을 두개의 클래스 계층으로 나누어서 다룬다. 여기에는 TERMINAL과 NONTERMINAL이라는 두개의 메타 클래스가 있다. TERMINAL 클래스는 문서 계층에서 종말 노드에 관련되고 NONTERMINAL은 구조화된 엘리먼트에 관련된다. 이 방법의 단점은 엘리먼트가 의미를 가지지 않는다는 것이다. [12]에서는 HyTime을 다루기 위해 D-STREAT의 확장을 제시하였다. 우리는 이 방법과 같이 동적으로 DTD 응용 스키마를 생성하지만 두 개의 메타 클래스 방법이 아닌 DTD 메타 스키마를 통해 이루어진다. 그리고 이 방법에서는 고려되지 않은 것으로 엘리먼트가 의미를 갖고 엘리먼트의 형 검사를 수행함으로써 중복을 최소화하도록 하였다.

6. 결론 및 향후 계획

본 논문에서는 SGML로 기술된 전자문서들을 객체지향 데이타베이스에서 효과적으로 관리할 수 있도록 SGML DTD(ISO 8879)의 구성 성분들을 고려하여 SGML DTD 메타 스키마를 설계하였다. 그리고 이 메타 스키마에 인스탄스로 저장된 DTD를 역시 객체지향 데이타베이스에서 관리할 수 있도록 응용 데이타베이스 스키마를 동적으로 생성시키는 절차(알고리즘)를 고안하였다. 이는 ISO 8879에 기반한 SGML DTD로 기술된 모든 전자문서들을 DTD 자체 정보 뿐만 아니라 DTD에 의한 SGML 문서 정보를 통합 데이타베이스로 관리할 수 있게 해준다. 따라서 유사한 DTD들 사이에 정보를 공유할 수 있게 해주며, 특히 새로운 DTD의 설계에 기존 DTD 정보를 활용할 수 있게 해준다.

앞으로 이 데이타베이스 설계를 바탕으로 SGML DTD로 기술된 전자문서를 쉽게 객체지향 데이타베이스로 관리할 수 있도록 지원하는 도구가 개발되어야 한다. 이는 사용자에게 시각적 설계를 지원할 수 있어야 할 것이며, SGML 파서가 통합 데이타베이스와 연결될 수 있도록 확장해야 할 것이다.

한편 최근 시간 관계성을 지원하는 HyTime이 제안되었는데[13], 하이퍼미디어 응용을 위해서는 이의 지원이 반드시 요구된다. 따라서 본 논문에서 설계된 데이타베이스가 HyTime도 지원할 수 있도록 확장되어야 할 것이다.

참 고 문 헌

[1] ISO, *International Standard ISO/IEC8879: Information Processiong-Text and Office Information Systems-Standard Generalized Markup Language (ISO 8879)*, Geneva/NewYork, 1986.

[2] E. Herwijnen, *Practical SGML*, 1st Edition, Klu-

wer Academic Publishers, 1990.

[3] E. Herwijnen, *Practical SGML*, 2nd Edition, Kluwer Academic Publishers, 1994.

[4] W. Kim, "Object-Oriented Databases: Definition and Research Directions," IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 1, pp. 327-341, Sept. 1990.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[6] R. Davis, T. Moore, and J. Zobel. "Database Systems for Structured Documents," In *Int'l. Symp. Advanced Database Tech. and their Integration (ADTI '94)*, 1994.

[7] G. Blake, M. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. Tompa, "Text/Relational Database Management Systems: Harmonizing AQL and SGML," In *Proc. First Int'l, Conf. Appl. of Database*, pp. 267-280, June 1994.

[8] B. Thuraisingham, "On Developing Multimedia Database Management Systems Using the Object-Oriented Approach," Multimedia Review, Vol. 3, No. 2, pp. 11-19, Summer 1992.

[9] V. Balasubramaniam, "State of the Art Review on Hypermedia Issues and Applications," Internal document, Graduate School of Management, Rutgers University, Newark, New Jersey, 1993.

[10] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, "From Structured Documents to Novel Query Facilities," *In Proc. ACM SIGMODE Intl. Conf. Management of Data*, pp. 313-324, May 1994.

[11] K. Böhm and K. Aberer, "Storing HyTime Documents in An Object-Oriented Database," In *Proc. of CIKM '94*, pp. 26-33, 1994.

[12] K. Bhm, A. M ller, and E. Neuhold, "Structured Document Handling-A Case for Integrating Databases and Information Retrieval," In *Proc. of CIKM '94*, pp. 147-154, 1994.

[13] ISO, *Hypermedia/Time-based Structuring Language: HyTime(ISO 10744)*, 1992.

한 에 노

1995년 2월 경상대학교 수학과 졸업(학사)
1997년 2월 경상대학교 전자계산학과 공학석사
1996년 12월~현재 신한정보 시스템 개발팀
관심분야: 객체지향 데이터베이스, Client/Sever 기술

박 인 호

1990년 2월 경상대학교 전산통계학과 졸업(학사)
1997년 2월 경상대학교 전자계산학과 공학석사
1997년 3월~현재 경상대학교 전자계산학과 박사과정 재학중
관심분야: 객체지향 데이터베이스, 멀티미디어

강 현 석

1981년 2월 동국대학교 전자계산학과 졸업(학사)
1983년 2월 서울대학교 계산통계학과 이학석사 (전산학)
1989년 2월 서울대학교 계산통계학과 이학박사 (전산학)
1981년~1984년 한국전자통신연구원 연구원
1984년~1992년 전북대학교 전자계산학과 부교수
1993년~현재 경상대학교 컴퓨터과학과 교수
관심분야: 객체지향 데이타베이스, 컴퓨터 그래픽스, 멀티미디어

김 완 석

1982년 2월 영남대학교 물리학과 졸업(학사)
1992년 2월 계명대학교 전자계산학과 석사
1988년~현재 한국전자통신연구원 데이타베이스연구실 선임연구원
관심분야: 실시간 운영체제, 데이타베이스, 웹 게이트웨이