객체지향 속성 문법과 SML을 이용한 XML 컴파일러 생성기

최 종 명[†]·유 재 우[††]

요 약

XML은 데이터와 문서를 표현하기 위한 표준화된 메타언어로서 점차 많은 분야에서 사용되고 있지만, 각 분야에서 XML 문서를 올바르게 처리하기 위해서는 XML 컴파일러를 작성해야 한다. XML 컴파일러를 작성하는 것은 많은 시간과 노력을 필요로 하기 때문에 XML 컴파일러를 자동적으로 생성할 수 있는 방법이 필요해진다. 논문에서는 XML 문서를 의미에 맞게 처리할 수 있는 XML 컴파일러를 자동으로 생성할 수 있는 XCC라는 XML 컴파일러 생성기를 소개한다. XCC는 XML 문서의 DTD를 입력으로 받고, XML 원소(element)들 간의 관계를 이용해서 상속과 컴포지션 관계를 갖는 자바 클래스들을 생성한다. XCC는 또한 의미 규칙을 입력으로 받아서 XML 문서를 의미에 맞게 처리하기 위한 XML 컴파일러를 생성한다. XCC는 XML 컴파일러를 자동적으로 생성함으로써 XML 문서 처리를 위한 소프트웨어 개발에서 비용을 절감시킬 수 있다.

An XML Compiler Generator using Object Oriented Attribute Grammar and SML

Jong-Myung Choi[†] · Chae-Woo Yoo[††]

ABSTRACT

XML as a standard for representing data and document structure is widely used in every area, and we have to write XML compilers which process the XML documents according to a user's intention. Because it takes time and costs to write XML compilers by hand, we need some generators that automatically generate XML compilers. In this paper, we introduce an XML compiler generator named XCC. It reads DTD and semantic rules, and it generates XML compiler and Java classes which correspond to the elements defined in the DTD.

키워드 : XML 컴파일러 생성기(XML Compiler Generator), XML 컴파일러(XML Compiler), SML, 객체지향 속성 문법(Object-Oriented Attribute)

1. 서 론

XML(Extensible Markup Language)은 문서와 데이터를 표현하기 위한 표준화된 메타언어이다[1]. XML을 특정 영역에서 사용하기 위해서는 해당 문제를 잘 표현할 수 있는 문서의 구조와 의미 정보를 기술해야 한다. 문서의 구조는 일반적으로 DTD(Document Type Definition)를 이용해서 정의하며, 유효한(valid) XML 문서는 반드시 DTD에 맞게 작성되어야 한다. XML은 일종의 언어이기 때문에 기존 프로그래밍 언어와 여러 가지 공통점들을 가지고 있다. DTD는 프로그래밍에서 언어의 문법에 해당되고, DTD 문법에 맞게 작성된 XML 문서는 프로그램에 해당된다. <표 1>은 XML과 프로그래밍 언어 사이의 공통점들을 보여준다.

<표 1> XML과 프로그래밍 언어의 공통점

분야 / 특성	XML	프로그래밍 언어
문법 기술 방법	DTD 문법	BNF(Backus Naur Form)
언어의 문법 내용	DTD 문서	프로그래밍 언어의 문법
문법의 인스턴스	XML 문서	프로그램
문법 체크	XML 파서	구문 분석기

프로그래밍 언어에서 컴파일러는 입력 프로그램의 문법적인 사항을 분석하고, 이어서 의미 분석을 통해서 사용자의 의도에 맞는 타겟 코드를 생성한다. 유사하게 XML 문서도 사용자의 의도에 맞게 처리되기 위해서는 문법적인 사항만 체크하는 XML 파서 이외의 소프트웨어 모듈이 필요하다. 본 논문에서는 XML 문서를 사용자의 의도에 맞게

※ 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음.
† 준 회 원 : 국립목포대학교 컴퓨터공학 교수
†† 정 회 원 : 숭실대학교 컴퓨터학과 교수
논문접수 : 2003년 9월 25일, 심사완료 : 2004년 2월 24일

처리 혹은 변환하는 소프트웨어 모듈을 *XML 컴파일러*[1]라고 부르기로 한다.

XML 문서를 처리하기 위한 XML 컴파일러를 자동적으로 생성하기 위한 생성기에 관한 연구는 상대적으로 적었다. 프로그래밍 언어 분야에서 언어의 의미 분석을 위해서는 속성 문법(attribute grammar)을 사용하고, 컴파일러는 YACC[2] 혹은 ANTLR[3]과 같은 컴파일러 생성 도구를 통해서 자동적으로 생성된다. 이에 반해 XML 분야에서 XML 컴파일러는 프로그래머가 SAX(Simple API for XML)[4] 혹은 DOM(Document Object Model)[5]과 같은 API를 이용해서 직접 작성해야 하기 때문에 많은 비용과 노력이 소모되고, 개발 기간이 길어지며, 소프트웨어의 재사용성과 확장성이 떨어지는 문제점이 있었다. 특히 XML 문서는 사용자의 의도에 따라 다양한 방법으로 처리될 수 있기 때문에 하나의 XML 문서에 대해서도 여러 종류의 XML 컴파일러가 존재할 수 있다. 이러한 경우에 XML 컴파일러를 프로그래머가 API를 이용해서 개발하는 것은 비효율적이기 때문에 XML 컴파일러 생성기의 필요성은 더욱 커진다.

본 논문에서는 XML 문서의 의미 속성을 SML(Semantic Markup Language)을 이용해서 기술하고, 객체지향 속성 문법[6, 19]을 XML DTD에 적용해서 XML 컴파일러를 자동적으로 생성할 수 있는 방법과 이 방법을 구현한 컴파일러 생성기, XCC(XML Compiler Compiler)를 소개한다. 객체지향 속성 문법을 사용하는 경우에 DTD의 원소(element)는 클래스로 표현되고, XML 문서는 원소 클래스의 인스턴스를 노드로 갖는 XML 객체 트리로 표현된다. XCC는 XML의 DTD와 SML로 표현된 의미 정보를 읽고, DTD 원소들에 해당되는 자바 클래스와 XML 컴파일러를 생성한다. XML 컴파일러는 XML 문서를 읽고, XML 객체 트리로 표현하고, 트리를 순회하면서 구문 지향 변환(syntax-directed translation)[7] 방법으로 XML 문서를 처리한다. XCC를 이용하는 경우에 개발자는 XML 컴파일러를 개발하기 위해서 소모되는 시간과 노력을 줄일 수 있다.

본 논문은 2장에서 관련 연구들을 소개하고, 3장에서 객체지향 속성 문법과 SML의 기본적인 구조를 설명한다. 4장에서는 XCC의 구조와 동작 방법을 설명하고, 5장에서는 SML을 이용해서 의미 규칙들을 기술하는 방법을 소개한다. 마지막으로 6장에서는 결론 및 향후 연구를 밝힌다.

2. 관련 연구

프로그래밍 언어 분야에서 의미를 기술하기 위해서 속성 문법을 주로 사용하여 왔으며, 속성 문법에 관련된 많은 연구들이 수행되었다. 이러한 연구의 영향으로 SGML(Standard Generalized Markup Language)과 XML 문서를 처리하기 위해서 속성 문법을 사용하려는 시도가 있었다. 그러나 이러한 시도는 대부분 문서를 특정한 목적에 따라 처리하기 위한 방법으로 연구되었고, 일반적인 처리를 위한 연구는 많이 수행되지 않았다. 이러한 연구의 대표적인 것으로는 SIMON[8]과 DASTIR 시스템[9]이 있다. SIMON은 SGML, TeX, LATEX 등의 다른 포맷으로 작성된 문서들에 High Order 속성 문법을 적용해서, 하나의 포맷을 갖는 문서로 변환할 수 있는 시스템이다. DASTIR 시스템은 이종의 문서들을 속성 문법을 이용해서 일관된 형태로 표현하고, 정보 검색을 수행한다. SIMON과 DASTIR은 문서를 처리하기 위해서 속성 문법을 사용한다는 점에서 본 논문에서 소개하는 XCC와 공통점을 가지고 있지만, 시스템의 목적과 기능에서 많은 차이가 있다.

또 다른 형태의 기존 연구로는 YACC의 영향을 받아 문서 컴파일러를 자동적으로 생성하기 위한 방법에 관련된 것들이다. 대표적인 것으로는 Warmer[10]의 연구, FleXML[11] 시스템, Uljana[20]의 연구가 있다. Warmer는 SGML에서 YACC 스타일로 DTD에 C 코드를 추가하는 방법을 이용해서 SGML 문서를 처리하는 문서 처리기를 자동적으로 생성하는 연구를 수행하였다. Warmer는 DTD에서 원소 선언을 프로시저로 변환하고, SGML의 특정 문서 파서에 의미 정보를 추가하는 방법을 사용한다. Giuseppe[12]는 XML에서 속성 문법을 이용해서 의미를 기술하기 위한 방법으로 별도의 XML 파일에 의미 정보를 사용하는 방법을 사용한다.

FleXML은 DTD와 액션 파일을 입력으로 받아서 XML 문서를 처리할 수 있는 처리기를 자동으로 생성하는 생성기이다. FleXML은 Perl을 이용해서 작성되었으며, XML의 DTD와 액션 파일을 입력으로 받아 XML 컴파일러를 생성한다. FleXML을 통해서 생성된 처리기는 flex를 통해서 만들어진 스캐너를 포함하고 있으며, XML 태그의 시작과 끝의 이벤트를 이용해서 문서를 처리한다. Uljana[20]는 컴파일러에서 사용되던 LEX, YACC을 이용해서 XML 응용을 위한 변환기(translator)를 작성하는 방법을 보여줌으로써 기존 컴파일러 도구를 구조적인 문서에서도 적용할 수 있다는 것을 보여준다. 그러나 LEX와 YACC을 사용해서 XML 컴파일러를 작성하는 것은 상대적으로 복잡하고, 컴파일러에 대한 전반적인 지식을 가지고 있어야 하는 문제점을 가지고 있다.

Warmer와 Giuseppe의 연구, FleXML 시스템 및 Uljana의 연구는 모두 기존 연구에서 사용된 속성 문법은 Knuth[18]가 제안한 기본적인 형태의 속성 문법이기 때문에 객체지향 언어의 장점을 활용할 수 없다는 문제점을 가지고 있다. 이에 반해 본 논문에서 제안하는 XCC는 객체지향 속성 문법을 사용하기 때문에 확장성, 재사용성이 높다는 장점을 갖는다. 또한 기존 연구들은 자체적으로 개발한 XML 파서를 사용하고, 문서를 처리하는데 비해 XCC는 기존의 범용 XML 파서와 표준화된 API를 사용하는 XML 컴파일러를 생성하기 때문에 다른 소프트웨어와의 호환성도 높다.

<표 2>는 SGML 혹은 XML 컴파일러를 생성하는 생성기들의 유사성과 차이점을 보여준다. SIMON 시스템과 Warmer

1) XML 문서를 처리하는 소프트웨어 모듈은 XML 프로세서[12]라고도 하지만, 본 논문에서는 XML 표준의 XML 프로세서[1] 정의와 구별하기 위해서 XML 컴파일러라고 부르기로 한다.

의 연구는 SGML 문서를 위한 컴파일러 생성기인 반면에, FleXML과 XCC는 XML 문서를 위한 컴파일러 생성기이다. SIMON과 XCC는 각각 C++와 자바 언어를 지원함으로써 객체지향을 지원하는데 비해서 다른 시스템은 객체지향 프로그래밍을 지원하지 못한다. 또한 XCC는 XML 문서를 XML 객체 트리로 표현하고, 이 트리에 비지터 패턴(visitor pattern)[13]을 적용해서 속성을 평가하기 때문에 확장성과 재사용성이 높다.

〈표 2〉 XML 컴파일러 생성기 비교

특성 \ 시스템	SIMON	Warmer 연구	FleXML	XCC
목 적	문서 포맷 변환	범용 문서 처리	문서 포맷 변환	범용 문서 처리
문서 형태	SGML	SGML	XML	XML
의미 기술 방법	High Order 속성 문법	기본 속성 문법	기본 속성 문법	객체지향 속성 문법
사용되는 파서	고유 파서	고유 파서	고유 파서	범용 파서
객체지향 지원 여부	지 원	지원안함	지원안함	지원함
확 장 성	낮 음	낮 음	낮 음	높 음
재사용성	낮 음	낮 음	낮 음	높 음
지원 언어	C++	C	C	자 바

3. 객체지향 속성 문법과 SML

3.1 객체지향 속성 문법의 적용

XML의 DTD는 확장된 문맥 자유 문법(ECFG, Extended Context Free Grammar)을 사용하고, ECFG는 객체지향 속성 문법이 적용될 수 있는 제한된 CFG(Restricted CFG)로 변환될 수 있다. 따라서 XML 문서의 DTD는 적절한 형태로 변환하는 경우에 객체지향 속성 문법을 적용할 수 있다. DTD에 객체지향 속성 문법을 적용하는 경우에 원소는 클래스로 표현된다. 일반 DTD는 앞으로 설명할 방법에 따라서 객체지향 속성 문법을 적용할 수 있는 제한된 형태의 DTD로 변환될 수 있다.

DTD의 구성 요소들의 집합은 **[정의 1]**과 같이 표현될 수 있다. 즉, DTD에서 모든 원소들은 집합 E로 표현하고, 원소 A의 속성들의 집합은 Att(A)로 표현한다.

[정의 1] DTD 구성원의 집합
E는 DTD에서 선언된 원소들의 집합이고, Att(A)는 원소 A의 속성들의 집합이다. ε는 빈 문자열이다. □

XML에서 원소의 자식 원소들은 CHOICE(|) 혹은 SEQ(,) 중에서 하나로 구성되어 있고, 원소가 자식 원소들을 CHOICE 관계를 이용해서 기술하고, 원소가 속성을 갖지 않는 경우에 이 원소를 추상 원소(abstract element)라고 한다.

[정의 2] 추상 원소
원소 A는 다음과 같이 정의될 때 A를 추상 원소라고 한다.

$$<!\text{ELEMENT A } (B_1O_1 \mid B_2O_2 \mid \cdots \mid B_nO_n)>$$

단, $1 \le i \le n$, $O_i \in \{\varepsilon, \text{'?'}\}$, $Att(A) = \varnothing$이다. □

객체지향 속성 문법을 이용하면, 추상 원소와 자식 원소들은 상속 관계를 이용해서 표현한다. 즉, [정의 2]와 같은 형태로 원소 A가 정의되는 경우에 클래스 A는 B_i 클래스의 부모 클래스로 표현된다.

자식 원소들을 SEQ를 이용해서 기술하는 경우에 이 원소를 구조적 원소(structured element)라고 한다.

[정의 3] 구조적 원소

$$<!\text{ELEMENT A } (B_1O_1, B_2O_2, \cdots, B_nO_n)>$$

단, $1 \le i \le n$, $O_i \in \{\varepsilon, \text{'?'}, \text{'*'}, \text{'+'}\}$이다. □

객체지향 속성 문법을 사용하는 경우에 구조적 원소는 원소 내용으로 표현된 자식 원소들을 클래스 관계에서 컴포지션(composition)으로 표현한다. 즉, [정의 3]에서 원소 A는 B_i 원소 들을 멤버 필드로 갖는 클래스로 표현된다.

[정리 1] 혼합 내용(mixed contents)
XML에서 혼합 내용은 CHOICE를 사용하지만, 자식 원소와 PCDATA 내용이 반복적으로 나타날 수 있기 때문에 구조적 원소로 취급한다. □

추상 원소와 구조적 원소를 제외한 원소들은 복합 원소라고 하고, 복합 원소는 새로운 원소를 도입해서 추상 원소와 구조적 원소로 변환해야 한다.

[정의 4] 복합 원소
XML 원소에서 추상 원소와 구조적 원소를 제외한 모든 원소들을 복합 원소라고 한다. □

객체지향 속성 문법은 결합과 상속을 동시에 지원하지 못하기 때문에 XML에서 복합 원소는 새로운 원소를 도입해서 추상 원소와 구조적 원소의 형태로 변경하여야 한다. 예를 들어, <!ELEMENT A((B1, B2) | D | F)>는 다음과 같이 변경될 수 있다.

$$<!\text{ELEMENT A(B | D | F) }>$$
$$<!\text{ELEMENT B } (B_1, B_2)>$$

XML에서 원소의 속성은 원소의 메타 정보들을 표현하기 위해서 주로 사용된다. 속성은 자식 원소로 변경해서 기술할 수 있으며, 어떤 정보를 속성으로 기술할 것인지 혹은 자식 원소로 기술할 것인지는 DTD 설계자의 취향에 따라 결정될 수 있다. 따라서 객체지향 속성 문법을 사용하는 경우에 원소의 속성은 원소의 자식 원소와 동일하게 취급할 수 있다.

[정리 2] 속 성
자식 원소가 CHOICE로 연결되는 경우에 원소의 속성은

원소의 자식 원소로 취급하고, 자식 원소들과 SEQ 관계를 갖는다. □

<!ELEMENT X ($B_1 | B_2 | \cdots | B_n$) > 이고, Att(X) = {a_1, a_2, ···, a_m}인 경우에 원소 X는 다음과 같은 복합 원소로 변경할 수 있다.

<!ELEMENT X ((a_1, a_2, ···, a_m), ($B_1 | B_2 | \cdots | B_n$)) >

객체지향 속성 문법을 DTD에 적용하기 위해서는 원소들을 제한된 DTD 형태로 변경하여야 한다.

[정의 5] 제한된 DTD
DTD에 선언된 모든 원소가 추상 원소 혹은 구조 원소이고, 다음 조건을 만족할 때 제한된 DTD라고 한다.
$\forall A \in E, |Parent(A)| \leq 1$, 단 Parent(A)는 원소 A의 부모 원소들의 집합이다. □

3.2 SML을 이용한 의미 기술

본 논문에서는 Giuseppe[12]의 접근 방법에 따라 XML 문서의 의미 규칙을 별도의 SML이라는 XML 파일에 기술하는 방법을 사용한다. 이것은 하나의 DTD에 다양한 처리 규칙들을 각 SML 파일에 기술할 수 있다는 장점을 가지고 있다. SML 문서에는 XCC가 XML 컴파일러를 자동으로 생성할 수 있도록 XML 원소와 속성의 의미와 이것들이 어떻게 처리되어야 하는지에 대한 정보를 기술한다. XCC에서 의미 규칙은 XML 문서 형태로 기술할 수 있으며, 의미 규칙을 정의하는 XML 문서의 구조는 <표 3>과 같은 형태로 구성된다. 문서의 가장 상위에는 semantics 원소가 있고, semantics는 code, rules이라는 자식 원소를 포함한다.

〈표 3〉 의미 규칙을 기술하기 위한 문서의 DTD

```
<!ELEMENT semantics (code?, rules) >
<!ELEMENT code      (import, var, function) >
<!ELEMENT import    (#PCDATA) >
<!ELEMENT var       (#PCDATA) >
<!ELEMENT function  (#PCDATA) >
<!ELEMENT rules     (tag) + >
<!ELEMENT tag       (node | rule | method | exe) + >
<!ATTLIST tag
          name      NMTOKEN #REQUIRED
          parent    NMTOKEN #IMPLIED
          interface NMTOKENS #IMPLIED
          userMode  (yes | no) "no" >
<!ELEMENT node      (attribute) + >
<!ELEMENT attribute EMPTY >
<!ATTLIST attribute
          name      NMTOKEN #REQUIRED
          type      NMTOKEN #REQUIRED >
<!ELEMENT rule      (#PCDATA) >
<!ATTLIST rule
          name      NMTOKEN #REQUIRED >
<!ELEMENT method    (head, body) >
<!ELEMENT head      (#PCDATA) >
<!ELEMENT body      (#PCDATA) >
<!ELEMENT exe       (#PCDATA) >
```

code 원소에는 생성되는 XML 컴파일러에서 필요로 하는 내용들을 기술한다. code 원소의 내용은 생성되는 XML 컴파일러의 소스에 그대로 복사된다. code 원소는 XML 컴파일러에 전달되는 내용의 형태에 따라 세 종류의 자식 원소들을 갖는다. 첫째로 import 원소는 생성되는 XML 컴파일러가 소속되는 패키지에 관련된 정보를 기술하거나, 혹은 필요로 하는 라이브러리를 임포트하기 위한 내용들을 기술한다. 예를 들어, import 원소에는 다음과 같이 자바의 import 문장 혹은 패키지 문장을 기술할 수 있다.

```
< import >
    import java.lang.* ;
</ import >
```

var 원소는 XML 컴파일러에서 필요로 하는 멤버 필드들을 기술하기 위해서 사용된다. var 원소에서 기술된 변수들은 function, rule, exe 원소에서 사용될 수 있다. function 원소는 XML 컴파일러에서 필요로 하는 메소드들을 정의하기 위해서 사용된다.

rules 원소에는 문서에서 필요로 하는 의미 속성들과 의미 속성들을 계산하기 위한 규칙들을 기술한다. rules는 여러 개의 tag 원소들로 구성되고, tag에는 의미 속성이나 속성을 평가하기 위한 속성 평가 규칙들을 기술한다. tag는 의미 속성이 적용되기 위한 XML 문서의 원소 이름을 기술하기 위해서 name 속성을 갖는다. parent와 interface 속성은 XML 원소를 클래스로 변환하는 경우에 클래스의 부모 클래스와 구현하는 인터페이스를 기술하기 위해서 사용한다. userMode 속성은 XML 원소의 클래스가 DTD를 통해서 자동적으로 생길 것인지 아니면, 사용자가 정의한 내용만 갖는 형태로 만들어질 것인지 여부를 기술한다. userMode 속성 값이 "no"인 경우에는 DTD를 통해서 자동적으로 클래스가 생성된다.

tag 원소는 node, rule, method라는 자식 원소들로 구성된다. node 원소는 의미 속성을 기술하기 위해서 사용되고, 여러 개의 attribute 원소들로 구성된다. attribute의 name 속성은 의미 속성의 이름을 기술하고, type은 의미 속성의 자료형을 기술한다. rule 원소는 의미 속성의 값을 계산하기 위한 내용을 포함한다. rule은 계산할 의미 속성의 이름을 name 속성을 이용해서 기술하고, 의미 속성을 계산하기 위한 내용은 PCDATA 내용에 포함된다. tag 원소에서 node 자식 원소는 한번만 나타날 수 있고, rule은 여러 번 사용될 수 있다. method 원소는 XML 원소 클래스에 포함되어야 하는 자바 메소드를 기술하기 위해서 사용된다. method 원소는 메소드 이름을 기술하기 위한 head 자식 원소와 body라는 메소드 내용을 갖는 원소로 구성된다. 다음 예는 exp라는 원소에 적용될 의미 속성은 value이고, 이 속성의 타입은 Number라는 것을 의미한다.

```
< tag name = 'exp' >
    < node >
        < attribute name = 'value' type = 'Number' / >
    < / node >
< / tag >
```

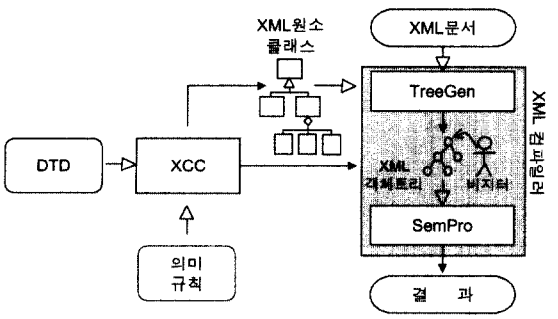
의미 규칙은 tag의 자식 원소인 rule 원소에 기술한다. rule 원소의 name 속성은 처리해야 할 의미 속성의 이름을 의미한다. 의미 규칙의 내용에서 $문자와 같이 사용되는 숫자는 XML 트리에서 현재 노드의 몇 번째 자식 노드임을 의미한다. 예를 들어, $1.value는 첫 번째 자식 노드의 value 속성 값을 의미한다. #TEXT는 원소의 PCDATA 타입의 텍스트를 의미한다. 다음 예의 의미 규칙은 add 원소에 적용되고, 의미 속성은 value이며, value 속성의 값은 두개의 자식 원소의 value 값을 더해서 계산된다는 것을 보여준다.

```
< tag name = 'add' >
    < rule name = 'value' >
        $$ = NumberUtil.add($1.value, $2.value) ;
    < / rule >
< / tag >
```

XCC는 문자열과 숫자를 처리하기 위한 기본적인 유틸리티 클래스 라이브러리를 제공한다. NumberUtil 클래스는 다양한 형태의 숫자 관련 클래스들을 처리할 수 있는 정적 메소드들을 제공한다.

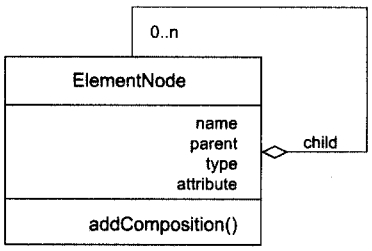
4. XCC의 구성

XCC는 XML 응용의 DTD와 SML로 작성된 의미 규칙을 입력으로 읽고, 원소에 해당되는 자바 클래스와 XML 컴파일러를 생성한다. 생성된 XML 컴파일러는 XML 문서를 입력으로 받고, XML 문서를 처리해서 결과를 만들어낸다. XCC와 XML 컴파일러는 (그림 1)과 같은 형태로 작업을 수행한다. 그림에서 화살표 머리가 흰 것은 처리 과정의 입력으로 사용된다는 의미이고, 검은색은 처리의 결과라는 의미이다.



(그림 1) XCC와 XML 컴파일러의 동작 방법

XCC는 DTD를 읽고, 원소들을 (알고리즘 1)에 따라서 XML 원소들 간의 관계를 형성하고, 원소에 관한 정보는 ElementNode라는 클래스를 통해서 관리한다. ElementNode는 (그림 2)와 같은 정보들을 가지고 있다. parent는 원소의 부모 클래스를 표현하기 위해서 사용된다. type 멤버 필드는 원소의 구조가 어떤 형태인지를 알려준다. child 멤버 필드는 원소의 자식 원소들에 대한 정보를 가지고 있고, attribute는 원소의 속성들을 표현한다. addComposition(ElementNode e) 메소드는 원소 e를 결합 관계로 묶는 것을 의미한다.



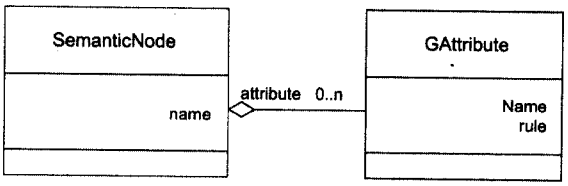
(그림 2) ElementNode 클래스

(알고리즘 1)은 DTD에서 문서 타입 원소를 시작으로 문서 전체 구조를 클래스의 상속과 결합 관계를 표현하는 그래프를 생성하게 된다.

```
매개 변수 : ElementNode e
begin
  if (e.attribute.length == 0) then
      if (e.type == 추상원소) then
          foreach c in e.child
              c.parent = e
              클래스_관계_만들기(c)
          end-foreach
      else if (e.type == 구조원소) then
          foreach c in e.child
              e.addComposition(c)
              클래스_관계_만들기(c)
          end-foreach
      else
          error("에러 : 복합 원소를 추상 원소와 구조 원소로 변
              경해야함.")
      end-if
  else
      error("에러 : 속성을 원소로 변경해야함.")
  end-if
end
```

(알고리즘 1) 클래스_관계_만들기

XCC는 XML 파서를 이용해서 SML로 작성된 의미 정보를 읽고, 각 원소별로 속성과 의미 정보를 추출해서 해시 테이블로 관리한다. 이때 의미 정보는 (그림 3)의 Semantic Node 클래스로 표현된다. SemanticNode는 의미 속성을 위한 GAttribute 클래스로 구성되어 있으며, GAttribute는 의미 속성의 이름(name 멤버 필드)과 의미 속성을 평가하기 위한 코드(rule 멤버 필드)를 갖는다.



(그림 3) SemanticNode 클래스

XCC는 (알고리즘 1)에서 만들어진 클래스 그래프와 SemanticNode로 구성된 해시 테이블을 이용해서 (알고리즘 2)에 따라서 자바 클래스 코드를 생성한다.

```
매개변수 : ElementNode e
매개변수 : SemanticNodeHash s
매개변수 : 원소_그래프 G
begin
  if (e.parent != NULL) then
      클래스 이름이 e.name이고, 부모 클래스 이름이 e.parent인
      클래스를 작성한다.
  else
      클래스 이름이 e.name인 클래스를 작성한다.
  end-if
  SemanticNode node = s.find(e.name)
  foreach a in node.attribute
      클래스의 멤버 필드로 a.name을 추가한다.
      클래스의 메소드로 a.rule의 내용을 추가한다.
  end-foreach
  if (e.child != NULL) then
      foreach c in e.child
          클래스의 멤버 필드로 c를 추가한다.
          클래스_생성하기(c)
      end-foreach
  end-if
  foreach c in G
      if (e.name == c.parent) then
          클래스_생성하기(c)
      end-if
  end-foreach
end
```

(알고리즘 2) 클래스_생성하기

XCC에 의해서 생성된 XML 컴파일러는 XML 문서를 입력으로 받아들여서 TreeGen 단계를 거쳐서 XML 객체 트리를 생성한다. XML 객체 트리의 각 노드들은 XML 원소 클래스의 객체들로 구성된다. XML 객체 트리는 SemPro 단계의 입력으로 사용되고, SemPro 단계에서는 XML 객체 트리를 순회하면서 XML 문서를 의미에 맞게 처리한다. XML 객체 트리의 순회는 비지터 패턴[13]을 따르며, 비지터는 의미 규칙에서 기술된 내용들을 포함한다.

5. XCC에서 의미 정보 기술 예

5.1 XML 원소 클래스 생성

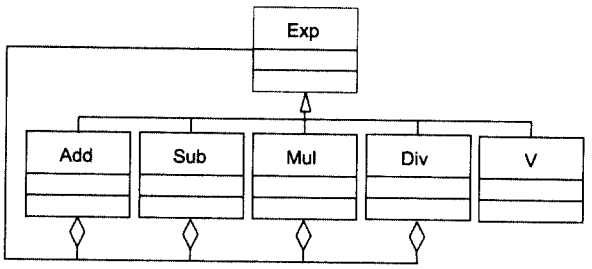
XML 원소가 자바 클래스로 매핑되는 것을 좀더 구체적으로 알아보기 위해서 수식을 표현하는 XML 문서를 예로 들어 보자. 프로그램적인 내용을 XML 문서로 표현하는 것이 이상할지 모르겠지만, XML을 이용해서 프로그램을 표현하는 것에 관한 연구는 점차 많아지고 있다[14, 15]. 프로그램을 XML로 표현하는 주된 이유는 XML에는 표준화된 파서와 각종 라이브러리들이 풍부하게 제공된다는 장점때문이다.

XML에서 사칙 연산을 표현하기 위한 수식의 문법은 <표 4>의 DTD와 같이 기술할 수 있다. exp 원소는 add, sub, mu, div, v 자식 원소들 중에서 하나를 사용할 수 있고, 덧셈 연산을 위한 add 원소는 두 개의 exp 원소로 구성된다. v 원소는 숫자 상수를 표현하기 위해서 사용된다.

〈표 4〉 expression.dtd

```
<!ELEMENT exp (add | sub | mul | div | v)>
<!ELEMENT add (exp, exp)>
<!ELEMENT sub (exp, exp)>
<!ELEMENT mul (exp, exp)>
<!ELEMENT div (exp, exp)>
<!ELEMENT v (#PCDATA)>
```

객체지향 속성 문법에 따르면 expression.dtd의 각 원소들은 자바 클래스로 표현되며, 클래스들은 (그림 4)와 같은 관계를 갖는다. Add, Sub, Mul, Div, V 클래스들은 모두 Exp 클래스로부터 상속받으며, V를 제외한 다른 클래스들은 모두 Exp를 멤버 필드로 갖는다.



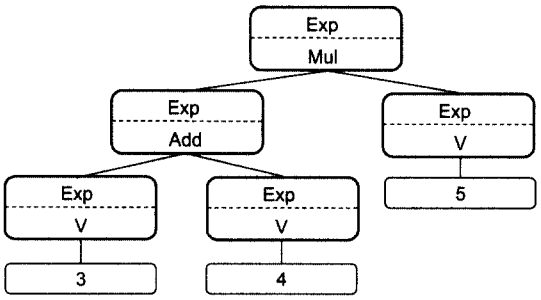
(그림 4) expression.dtd의 객체지향 표현

수식 DTD가 주어지면, 다양한 형태의 4칙 연산을 XML을 이용해서 표현할 수 있다. 예를 들어, (3+4)×5를 위한 XML 문서는 <표 5>와 같이 작성할 수 있다.

〈표 5〉 exp.xml

```
<?xml version = "1.0" encoding = "euc-kr"?>
<!DOCTYPE exp SYSTEM "expression.dtd">
<exp>
  <mul>
    <exp>
      <add>
        <exp><v>3</v></exp>
        <exp><v>4</v></exp>
      </add>
    </exp>
    <exp><v>5</v></exp>
  </mul>
</expr>
```

<표 5>와 같은 XML 문서는 XML 파서를 이용하는 경우에 DOM 트리로 표현된다. DOM 트리는 프로그래밍 언어에서 파스 트리와 유사하다. DOM 트리는 너무 많은 정보를 가지고 있고, 너무 크기 때문에 XML을 표현하는 내용을 좀더 작은 형태의 트리로 표현해야 할 필요가 있다. XCC에서는 객체지향 속성 문법의 규칙을 따라서 XML 문서를 XML 객체 트리를 생성하도록 한다. <표 4>의 내용을 XML 객체 트리로 표현하는 경우에 (그림 5)와 같은 형태로 표현된다. 트리의 각 노드는 객체를 의미하고, Mul, Add, V는 모두 Exp 클래스로부터 상속받는다. 따라서 Mul, Add, V 클래스의 각 객체들은 내부적으로 Exp 클래스의 내용들을 포함하고 있다.



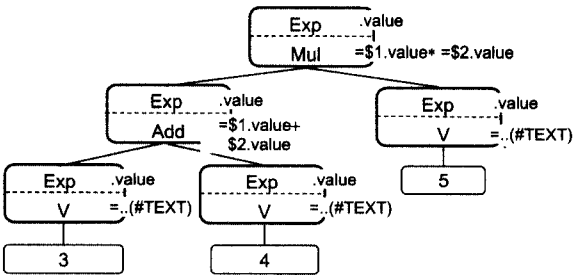
(그림 5) XML 객체 트리

XCC는 기본적으로 XML 문서의 DTD를 이용해서 (그림 5)와 같은 XML 객체 트리를 생성할 수 있는 Exp, Add, Mul, V와 같은 클래스들을 자동적으로 생성한다. 그러나 XML 객체 트리만으로는 실질적인 XML 처리를 수행할 수 없다. 따라서 XML 문서를 의미에 맞게 처리하기 위해서는 의미 정보들을 좀더 구체적으로 기술해야 한다. <표 3>과 <표 4>의 사칙 연산 수식을 위한 의미 XML 문서는 <표 6>과 같이 작성할 수 있다.

〈표 6〉 expression.dtd를 위한 의미 규칙

```
<code>
    import java.lang.* ; …
<rules>
    <tag name = 'exp'><node>
            <attribute name = 'value' type = 'Number'/>
…
    <tag name = 'add'>
            <rule name = 'value'>
            $$ = NumberUtil.add($1.value, $2.value) ;
…
    <tag name = 'v'>
            <rule name = 'value'>
            $$ = NumberUtil.s2n ( #TEXT) ;
…
```

<표 6>에서 exp는 value라는 의미 속성을 가지고 있으며, add는 value 속성의 값을 계산하기 위해서 2개의 자식 원소의 value 속성 값을 더한다. v의 value 속성은 v 원소의 PCDATA 내용을 숫자로 변환해서 계산한다. XML 객체 트리에 <표 6>에서 지정한 의미 정보를 추가하는 경우에 (그림 5)는 (그림 6)과 같은 형태로 표현될 수 있다. XML 객체 트리의 각 노드에는 의미 속성과 의미 속성을 평가하기 위한 규칙들을 포함하고 있다.



(그림 6) 속성 정보를 갖는 XML 객체 트리

<표 4>의 DTD 정보와 <표 6>의 의미 정보를 이용하는 경우에 문법 클래스 생성기는 add 원소를 위해 <표 7>과 같은 Add 클래스를 생성한다. Add 클래스는 Exp 클래스로부터 상속받고, Exp 클래스 타입의 _exp0와 _exp1이라는 2개의 컴포넌트 클래스를 갖는다. 또한 의미 속성을 평가하기 위해서 evaluate() 메소드를 갖는다.

〈표 7〉 생성된 Add 클래스

```
public class Add extends Exp implements IVisitable {
        protected Exp_exp0 ;
        protected Exp_exp1 ;
        protected IVisitor visitor ;
…
        public Object evaluate(IVisitor visitor) {
            this.visitor = visitor ;
            return this.visitor.visit(this) ;
…
```

XML 문서의 의미를 처리하기 위한 의미 처리기는 비지터 패턴 형태로 구현된다. 수식의 XML 문서를 처리하기 위한 의미 처리기는 SemVisitor 클래스이고, SemVisitor 클래스는 <표 8>과 같은 형태로 생성된다. SemVisitor의 visit() 메소드는 XML 객체 트리에서 각 노드의 타입에 따라 적절한 메소드를 호출하게 된다. 예를 들어, add 원소가 사용된 경우에는 visitAdd() 메소드를 호출한다. visitAdd() 메소드는 자신의 컴포넌트 클래스로 갖는 _exp0와 _exp1의 value 의미 속성 값을 얻고, 2개의 값을 더해서 리턴한다.

〈표 8〉 생성된 SemVisitor 클래스

```
public class SemVisitor implements IVisitor {
    protected Object visitAdd(Add e ) {
        Object o1 = e.get_exp0() ;
        Object oa1 = visit((IVisitable)o1) ;
        Attribute _a1 = (Attribute) oa1 ;
…
        value = NumberUtil.add(_a1.value, _a2.value) ;
        return new Attribute(value) ;
```

```
...
    public Object visit (IVisitable e) {
        if (e instanceof Add) {
            return visitAdd ((Add) e) ;
        } else if (e instanceof Sub) {
...
```

XCC는 <표 9>와 같은 XML 컴파일러를 자동적으로 생성한다. XML 컴파일러는 SAX 파서를 생성해서 XML 문서의 유효성을 파악하고, SAX의 컨텐트 핸들러 클래스를 이용해서 XML 문서를 처리하면서 XML 객체 트리를 생성한다. 생성된 XML 객체 트리는 의미 처리기인 SemVisitor 클래스의 accept() 메소드의 매개 변수로 전달된다. SemVisitor는 트리를 순회하면서 XML 문서의 의미를 처리하고, 결과를 Object 타입으로 리턴한다.

〈표 9〉 생성된 XML 컴파일러 클래스

```
public class DocProcessor {
    public static void main(String args[]) {
        try {
            TreeGen ha = new TreeGen();
            XMLReader parser = new SAXParser();
...
            parser.setContentHandler(ha);
            parser.parse(args[0]);
            IVisitable root = ha.getRoot();
            SemVisitor visitor = new SemVisitor();
            Object o = root.accept(visitor);
...
```

XCC는 XML 컴파일러와 함께 SAX의 컨텐트 핸들러 클래스를 생성한다. <표 10>은 생성된 컨텐트 핸들러 클래스의 예이다. 컨텐트 핸들러는 SAX 파서가 XML 문서를 처리하면서 발생하는 이벤트에 따라 XML 객체 트리의 노드를 위한 객체들을 생성하고, 트리를 생성한다.

〈표 10〉 SAX 컨텐트 핸들러 클래스

```
...
public class TreeGen extends DefaultHandler {
...
    public void startElement (String uri,…) throws SAXException {
        try {
            if(localName.equals ("add")) {
                _o = Class.forName ("Add").newInstance() ;
                stack.push (_o) ;
            }
...
```

5.2 사용자가 제공하는 메소드

XCC에 의해서 자동적으로 생성되는 XML 원소 클래스는 기본적인 정보들을 가지고 있지만, 좀더 효율적인 코드를 작성해야 하거나 혹은 좀더 추가적인 기능을 필요로 하는 경우가 있을 수 있다. XCC는 사용자가 지정한 형태로 원소 클래스를 생성하고, 새로운 메소드를 추가할 수 있는 기능을 제공한다. tag 원소의 userMode 속성 값을 "yes"로 선언하는 경우에 XCC는 사용자가 정의한 의미 속성들만 클래스의 멤버 필드로 갖는다. 또한 원소 클래스에 메소드를 추가하고자 하는 경우에는 method 원소를 이용해서 기술할 수 있다.

userMode와 method 원소를 사용하는 예를 알아보기 위해서 <표 11>과 같은 도형 데이터를 갖는 XML 문서를 가정해보자. XML 문서는 그래픽 정보를 표현하기 위해서도 사용된다[16, 17]. <표 11>의 DTD에서 figures 원소는 내부에 oval과 rectangle 원소들을 포함할 수 있다. oval 원소는 타원을 표현하기 위한 것으로 좌표 값을 위한 x, y 속성과 폭과 높이를 위한 width와 height 속성을 갖는다. rectangle은 사각형을 위한 원소로 x, y, width, height 속성을 포함한다.

〈표 11〉 figures.dtd

```
<!ELEMENT figures (figure) *>
<!ELEMENT figure(oval | rect)>
<!ELEMENT oval EMPTY
<!ATTLIST  oval
            x NMTOKEN #REQUIRED
            y NMTOKEN #REQUIRED
            r NMTOKEN #REQUIRED>
<!ELEMENT rect EMPTY>
<!ATTLIST  oval
            x NMTOKEN #REQUIRED
            y NMTOKEN #REQUIRED
            width NMTOKEN #REQUIRED
            height NMTOKEN #REQUIRED>
```

<표 12>의 XML 문서는 도형들의 정보를 갖고 있는 XML 문서이다. 이 XML 문서는 원과 사각형에 관한 정보를 가지고 있다.

〈표 12〉 figures.xml

```
<figures>
    <figure>
        <oval x = "50" y = "60" r = "10"/ >
    </ figure>
</figures>
```

figures.xml를 위한 XML 컴파일러는 문서의 유효성을 체크하고 유효한 문서로 파싱되는 경우에 문서의 내용에 따라 원과 사각형을 화면에 렌더링해 주어야 한다. 이러한 경우에도 XML 내용의 의미를 렌더링 엔진에 올바르게 전달할 수 있는 방법이 필요하다.

<표 11>과 <표 12>의 그래픽 정보를 렌더링하는 XML 컴파일러를 위한 의미 정보는 <표 13>과 같이 작성할 수 있다. 렌더링 엔진(예 : DrawingTool)과 그래픽 컴포넌트들(예 : Oval)은 미리 정의되어 있다고 가정한다. figures.dtd의 의미 정보를 보면 figures 원소가 나타나는 경우에 DrawingTool이라는 렌더링 엔진을 생성하고, oval 혹은 rectan-

gle 원소가 나타날 때마다 해당 그래픽 컴포넌트들을 생성해서 랜더링 엔진에 등록한다. exe 원소에서 @ 문자는 XML 속성을 의미한다. 즉, @x는 XML 문서에서 x 속성의 값을 의미한다.

〈표 13〉 figures.dtd를 위한 의미 정보

```
<tag name = 'figure' userMode = "yes" interface = "Drawable">
    <node>
        <attribute name = 'x' type = 'int'/> ···
<tag name = 'oval' parent = "Figure" userMode = "yes">
    <node><attribute name = 'r' type = 'int'/></ node>
    <rule name = 'x'>
    $$ = NumberUtil.String2int(@x) ;
    </rule>
...
    <method>
        <head> public void paint(Graphics g) </head>
        <body>
        g.drawOval (x, y, 2*r, 2*r) ;
...
```

figure 태그를 위한 원소 클래스는 Drawable 인터페이스를 구현한다. oval를 위한 원소 태그는 Figure 클래스로부터 상속받으며, method 원소를 이용해서 paint라는 메소드를 정의한다. paint 메소드에서는 속성 값들을 이용해서 원을 그리게 된다. <표 14>는 XCC를 통해서 생성된 Oval 클래스의 코드를 보여준다.

〈표 14〉 생성된 Oval 클래스

```
public class Oval extends Figure {
...
    public void paint(Graphics g) {
        g.drawOval(x, y, 2 * r, 2 * r) ;
    }
}
```

6. 결 론

XML은 데이터와 문서를 표현하기 위한 표준으로서 점차 많은 분야에서 사용되고 있다. XML이 특정 분야에서 사용되기 위해서는 해당 분야에 맞는 문서 구조를 DTD를 이용해서 정의해야 한다. XML 문서의 문법적인 사항은 XML 파서를 통해서 체크되지만, 문서의 의미적인 내용을 처리하기 위해서는 개발자가 직접 SAX 혹은 DOM과 같은 API를 이용해서 XML 컴파일러를 작성해야 한다. XML 컴파일러를 작성하는 것은 상당히 번거롭고, 시간과 노력을 소모하기 때문에 보다 효과적으로 XML 컴파일러를 생성할 수 있는 방법이 필요하다.

본 논문에서는 XML 컴파일러를 자동적으로 생성할 수 있는 XCC라는 생성기를 소개하였다. XCC는 문서 구조를 표현하는 DTD와 문서를 처리하기 위한 의미 규칙을 입력으로 받고, XML 원소 클래스들과 XML 컴파일러를 자동으로 생성한다. XML 원소 클래스들은 DTD에 적용된 객체지향 속성 문법에 따라 만들어진다. XML 컴파일러는 XML 문서의 내용에 따라 XML 객체 트리를 구성하게 된다. SML 문서에 기술된 의미 규칙에 따라 생성된 비지터는 XML 객체 트리를 순회하면서 XML 문서를 사용자의 의도에 맞게 처리한다. XCC는 XML 문서를 처리하기 위한 XML 컴파일러를 자동적으로 생성하기 때문에 프로그래머의 노력과 비용을 상당히 줄여줄 수 있을 것이다.

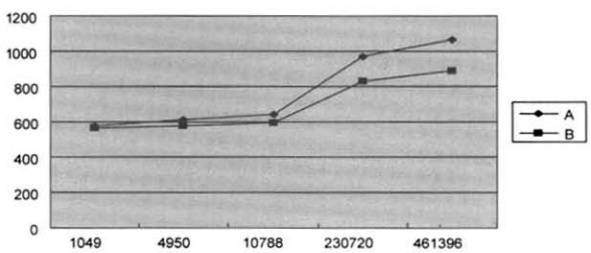
실험 결과 XCC에 의해서 생성된 XML 컴파일러는 사용자가 직접 작성한 컴파일러에 비해서 약간 느린 것으로 나타났다. <표 15>는 <표 4>의 expression.dtd에 적합한 XML 문서를 XML 컴파일러를 통해서 결과가 출력되는 시간을 측정한 것이다(파일 크기 단위 : 바이트, 시간 단위 : 1/1000초). 테스트는 1.41GHz의 인텔 펜티엄 4 CPU와 512MB의 메모리에서 J2SDK 1.5와 아파치 Xerces2 Java Parser 버전 2.6을 사용하여 수행하였다. 각 XML 컴파일러의 실행 시간은 5번씩 실행하면서 측정 값 중에서 최소 시간을 측정하는 방법을 사용하였다. <표 15>에서 A는 XCC를 통해서 자동적으로 생성된 XML 컴파일러이고, B는 사용자가 작성한 XML 컴파일러이다.

〈표 15〉 XML 컴파일러의 처리 속도

파일 크기 / 시스템	1,049	4,950	10,788	230,720	461,396
A	578	610	640	969	1063
B	562	578	594	828	890

(그림 7)은 XML 컴파일러 A와 B의 성능을 그래프 형태로 비교한 것이다. 처리해야 할 XML 파일이 커질 수록 처리 시간의 차이는 점차 커지는 양상을 띠고 있다.

XCC를 통해서 생성된 XML 컴파일러의 속도가 느린 것은 이 컴파일러가 확장성과 유연성을 위해서 비지터 패턴을 사용하기 때문인 것으로 판단된다. 이에 반해 사용자가 직접 작성한 XML 컴파일러는 해당 문제만 해결할 수 있도록 고정되어 있기 때문에 속도면에서는 우수하지만, 확장성과 유연성이 떨어지는 문제점이 있다. 앞으로 XML 문서가 커지고, 많은 문서를 처리해야 할 필요성이 증가하기 때문에 XML 컴파일러의 최적화 필요성은 점차 커질 것으로 판단된다. 따라서 XML 컴파일러의 최적화에 관련된 내용은 향후에 좀더 연구를 수행해야 할 것이다.



(그림 7) XML 컴파일러의 성능 비교

XCC는 현재 DTD를 이용하는 XML 문서를 위한 XML 컴파일러를 생성한다. 그러나 점차 DTD 대신에 XML Schema의 사용이 점차 증가하기 때문에 XCC에서 XML Schema를 지원할 수 있는 방법이 필요하다. 또한 XCC와 SML을 이용해서 XML 컴파일러를 생성하는 것이 기존 방법에 비해 간단하지만, 아직 일반 사용자들이 사용하기에는 어려움이 있을 수 있다. 따라서 일반 사용자들도 쉽게 사용할 수 있는 SML을 위한 그래픽 편집기가 필요하다. 이러한 문제점들은 향후 연구에서 해결될 것이다.

참 고 문 헌

[1] Extensible Markup Language(XML) 1.0, available at http://www.w3.org/TR/REC-xml.

[2] Stephen C. Johnson, "Yacc : Yet Another Compiler-Compiler," *Computing Science Technical Report 32*, AT&T Bell Labs, Murray Hill(NJ), 1975.

[3] ANTLR Translator Generator, available at http://www.antlr.org/.

[4] The Simple API for XML, http://www.saxproject.org/.

[5] Document Object Model (DOM), http://www.w3.org/DOM/.

[6] Kai Koskimies, "Object-Orientation in Attribute Grammars," In *Attribute Grammars, Applications and Systems*, LNCS 545, Springer-Verlag, pp.297-329, 1991.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[8] An Feng and Toshiro Wakayama, "SIMON : A grammar-based transformation system for structured documents," In *Proc. of Electronic Publishing-Origination, Dissemination and Design*, Vol.6, No.4, pp. 361-372, 1993.

[9] Alda Lopes Gacarski, "Using Attribute Grammars to Uniformly Represent Structured Documents-Application to Information Retrieval," In *Proc. of 3rd DELOS Network of Excellence Workshop on Interoperability and Mediation in Heterogeneous Digital Libraries*, 2001, available at http://www.ercim.org/publication/ws-proceedings/DelNoe03/.

[10] Jos Warmer and Hans Van Vliet, "Processing SGML Documents," In *Electronic Publishing*, Vol.4, No.1, pp. 3-26, 1991.

[11] FleXML, available at http://flexml.sourceforge.net/.

[12] Giuseppe Psaila and Stefano Grespi-Reghizzi, "Adding Semantics to XML," In *Proc. of Attribute Grammars and their Applications*, pp.113-132, 1999.

[13] Erich Gamma, et al, *Design Patterns*, Addison-Wesley Pub., 1995.

[14] Greg K. Badros, "JavaML : A Markup Language for Java Source Code," In *WWW Conf.*, available at http://www.cs.washington.edu/homes/gjb/JavaML/, 2000.

[15] "Bean Markup Language," IBM, available at http:// www.alphaworks.ibm.com/formula/bml, 1999.

[16] Vector Markup Language (VML), http://www.w3.org/TR/NOTE-VML.

[17] Scalable Vector Graphics (SVG), http://www.w3.org/Graphics/SVG/.

[18] Donald E. Knuth, "Semantics of context-free languages," In *Math. Systems Theory*, Vol.2, No.2, pp.127-145, 1968.

[19] Jukka Paakki, "Attribute Grammar Paradigms-A High-level Methodology in Language Implementation," In *ACM Computing Surveys*, Vol.27, No.2, pp.197-255, 1995.

[20] Uljana Timoshkina, Yury Bogoyavlenskiy, and Martti Penttonen, *Structured Documents Processing Using Lex and Yacc*, Technical Report, Univ. of Kuopio, Finland, 2001, available at http://www.cs.uku.fi/research/publications/reports/.

최 종 명

e-mail : jmchoi@mokpo.ac.kr

1992년 숭실대학교 전자계산학과, 학사

1996년 숭실대학교 전자계산학과, 석사

2003년 숭실대학교 컴퓨터학과, 박사

2004년~현재, 목포대학교 컴퓨터공학 전임강사

관심분야 : 시각 프로그래밍, 멀티패러다임 시스템, XML, 유비쿼터스 컴퓨팅

유 재 우

e-mail : cwyoo@ssu.ac.kr

1976년 학사, 숭실대학교 전자계산학과

1985년 박사, 한국과학기술원 전산학과

1986년~1987년, 1996년~1997년 코넬대학교, 피츠버그대학교 객원교수

1999년~2000년 한국정보과학회 프로그래밍언어 연구회 위원장

1983년~현재 숭실대학교 컴퓨터학부 교수

관심분야 : 프로그래밍언어, 컴파일러, 인간과 컴퓨터 상호작용