속성 문법과 XMLSchema를 이용한 XML 컴파일러 생성기

(An XML Compiler Generator Using Attribute Grammar and XMLSchema)

최 종 명 † 박 호 병 ††
(Jong-Myung Choi) (Ho-Byung Park)

요 약 XML 문서를 위한 컴파일러를 개발하기 위해서는 많은 노력을 필요로 하기 때문에 XML 컴파일러를 자동적으로 생성할 수 있는 방법에 대한 연구의 필요성이 증가하고 있다. XMLSchema가 표준으로 지정된 이후에 많이 사용되고 있지만, XMLSchema를 사용하는 XML 문서를 위한 XML 컴파일러 생성기에 관한 연구는 현재까지 거의 이루어지지 않았다. 본 논문에서는 속성 문법을 사용해서 XMLSchema를 사용하는 XML 문서를 위한 XML 컴파일러를 자동적으로 생성할 수 있는 방법을 소개한다. XML 컴파일러 생성기는 XMLSchema의 데이타 타입 정보와 별도로 제공되는 의미 정보를 이용해서 의미 클래스와 XML 컴파일러를 생성한다. 생성된 XML 컴파일러는 XML 문서를 파싱해서 의미 클래스의 인스턴스로 구성된 트리로 변환하고, 트리를 순회하면서 XML 문서를 사용자의 의도에 맞게 처리한다.

키워드 : XML 컴파일러 생성기, XMLSchema, 속성 문법

Abstract As XML is widely used across the computer related fields, and it costs expensive for its compiler, the study on the automatic generation of the compiler is becoming important. In addition, though the XMLSchema became a standard, there have been few works on the automatic compiler generation for XML applications based on the XMLSchema. In this paper, we introduce a method that we can automatically generate a compiler for an XML application based on the XMLSchema. Our XML compiler generator uses data type information in XMLSchema document and semantic information in another file and produces semantic classes and a compiler for the XML application. The compiler parses an XML document, builds a tree in which each node is an instance of semantic class, and processes the document through the traversal of the tree.

Key words : XML Compiler Generator, XMLSchema, Attribute Grammar

1. 서 론

XML은 표준 마크업 언어로서 데이타는 물론 선언적 언어를 표현하기 위해서도 사용되고 있다[1]. 이처럼 XML 문서가 이러한 언어의 프로그램으로 사용되는 경우에 프로그래밍 언어 분야에서 컴파일러 혹은 인터프리터처럼 XML 문서를 위한 처리기 혹은 컴파일러가 존재해야 한다. 그러나 현재 XML 파서는 XML 문서의 유효성만 체크할 수 있고, 문서를 의미에 맞게 처리할 수 있는 기능은 제공하지 않기 때문에 사용자는 직접 XML 컴파일러를 작성해야 하는 어려움이 있다. 현재까지 DTD를 사용하는 XML 문서의 경우에 XML 컴파일러를 자동적으로 생성하기 위한 연구들이[2-6] 수행됐지만, XMLSchema[7]를 사용하는 경우에는 이러한 연구가 거의 이루어지지 않았다. XMLSchema가 W3C의 표준으로 제정된 이후로 많은 XML 응용이 XMLSchema를 이용하기 때문에 XMLSchema용 XML 컴파일러를 자동적으로 생성할 수 있는 방법의 필요성은 더욱 커지고 있다. 이러한 요구를 만족시키기 위해 본 논문에서는 XMLSchema용 XML 컴파일러를 자동적으로 생성할 수 있는 방법을 소개한다.

DTD와 XMLSchema는 XML 문서 구조를 정의하는 스키마 언어라는 점에서 유사하지만, 세부적으로는 많은 차이점을 가지고 있다. DTD와 XMLSchema의 차이점

· 본 논문은 2004학년도 목포대학교 학술연구비 지원에 의하여 연구되었음
† 정 회 원 : 목포대학교 컴퓨터공학 교수
choijm@dreamwiz.com
†† 정 회 원 : 고등기술연구원 연구원
hobyung@iae.re.kr
논문접수 : 2006년 1월 31일
심사완료 : 2006년 8월 7일

은 여러 논문[8,9]에서 발표되었는데, 대표적인 차이점으로는 XMLSchema는 DTD에 비해서 구체적인 자료형들을 기술할 수 있고, 재사용성과 확장성이 뛰어난 장점을 가지고 있다[9]. 또 다른 차이점으로는 DTD와 XMLSchema의 문법 형태와 표현 범위가 다르다는 점이다[8]. 이러한 차이점 때문에 기존의 DTD용 XML 컴파일러 생성기에 관한 연구는 XMLSchema에 직접 적용할 수 없고, XMLSchema를 위한 연구가 별도로 수행되어야 한다.

본 논문에서는 XMLSchema에 객체지향 속성 문법을 적용해서 XML 컴파일러를 자동적으로 생성할 수 있는 방법과 이것을 구현한 XsCC(XMLSchema Compiler-Compiler)라는 컴파일러 생성기를 소개한다. 객체지향 속성 문법을 사용하는 경우에 CFG(Context Free Grammar)의 넌터미널을 클래스로 표현해야하지만, XMLSchema에 적용하기 위해서는 XML 문서의 구조를 결정하는 데이타 타입을 클래스로 표현해야 한다. 따라서 본 논문에서는 XMLSchema의 데이타 타입을 클래스로 표현하고, 이 클래스로부터 상속받는 클래스(의미 클래스)에 의미 정보를 추가하는 방법을 사용한다. 이때 의미 정보는 SML(Semantic Markup Language)이라는 별도의 XML 파일을 이용해서 기술한다. 생성되는 XML 컴파일러는 기존 범용 XML 파서를 이용해서 XML 문서를 파싱하고, SAX 핸들러와 트리 빌더를 이용해서 XML 문서를 의미 클래스의 인스턴스 트리로 변환하고, 트리를 순회하면서 사용자의 의도에 맞게 XML 문서를 처리한다.

XsCC가 생성하는 XML 컴파일러의 인스턴스인 XC는 XML 문서를 파싱하고, 원하는 결과를 생성할 수 있는 응용프로그램으로서 다음과 같은 영역에 적용해서 사용할 수 있다.

- XML 문서를 의미에 맞게 인터프리트
- XML 문서를 의미에 맞게 타겟 코드로 변환
- XML 문서의 변환
- XML 문서의 정보 추출 및 변경
- XML 바인딩
- 기타 XML 문서를 처리할 수 있는 작업

본 논문은 기존 연구들에 비해 세 가지 측면에서 다르다. 첫째로 XMLSchema를 위한 XML 컴파일러 생성기에 대한 연구라는 점이다. 현재까지 XMLSchema를 지원하는 XML 컴파일러 생성기에 대한 연구가 거의 없었기 때문에 본 논문은 기존에는 없었던 새로운 형태의 연구라고 볼 수 있다. 둘째로 객체지향 프로그래밍을 지원한다는 점이다. 기존 XML 컴파일러 생성기들이 객체지향 프로그래밍을 지원하지 않은 것에 비해 본 논문에서는 객체지향 속성 문법을 적용하였기 때문에 객체지향 프로그래밍을 지원한다. 셋째로 본 논문의 XML 컴파일러 생성기는XMLSchema의 상속 기능을 지원할 수 있다는 점이다.[1] 따라서 본 논문에서 소개하는 XML 컴파일러 생성기를 이용하는 경우에 문법 확장 및 상속의 개념에 적용할 수 있다.

본 논문은 총 5개의 장으로 구성되어 있다. 2장에서는 XML 문서를 처리하는 방법에 관한 관련 연구를 소개한다. 3장에서는 XMLSchema를 위한 XML 컴파일러를 생성하는 방법을 소개하고, 4장에서는 XsCC 시스템의 구성에 대해서 기술한다. 5장에서는 결론 및 향후 연구 과제를 밝힌다.

2. 관련 연구

응용프로그램에서 XML 문서를 처리하기 위해서 가장 널리 사용되는 것은 SAX[10] 혹은 DOM[11] API를 사용하는 것이다. 그러나 이러한 API를 사용하는 것은 많은 노력을 필요로 하기 때문에 다른 여러 가지 방법들이 연구되었다. 이러한 연구들은 크게 2가지 형태로 구분할 수 있다. 첫째는 XML 문서를 처리하기 위한 특수한 언어를 사용하는 방법이다. 이러한 언어의 가장 대표적인 것으로는 XSLT[12]가 있다. XSLT는 규칙 기반으로 XML 문서를 변환하기 위한 규칙들을 지정한다. XSLT는 비교적 쉽게 사용할 수 있다는 장점을 가지고 있다. 그러나 XSLT는 XSLT 프로세서에 의해서 처리되어야 하기 때문에 성능 면에서 떨어지고, XSLT는 범용 프로그래밍 언어가 아니기 때문에 사용자의 의도가 복잡한 경우에 문제를 해결할 수 없다는 문제점이 있다.

XML 문서를 처리하는 두 번째 방법은 XML 문서를 위한 처리기 혹은 컴파일러를 자동적으로 생성하기 위한 방법에 관한 연구이다. 이러한 연구는 SGML(Standard Generalized Markup Language) 시절부터 연구되어 온 바 있는데, 초기에는 SGML 문서를 다른 형태의 문서로 변환하기 위한 목적으로 시도되었다. 이러한 연구의 대표적인 것으로는 SIMON[13]과 DASTIR 시스템[14]이 있다. SIMON은 SGML, TeX, LATEX 등의 다른 포맷으로 작성된 문서들에 High Order 속성 문법을 적용하여, 다른 형태를 갖는 문서로 변환할 수 있는 시스템이다. DASTIR 시스템은 이종의 문서들을 속성 문법을 이용해서 일관된 형태로 표현하고, 이것을 이용해서 정보 검색을 수행한다. SIMON과 DASTIR은 문서를 처리하기 위해서 속성 문법을 사용한다는 특징이 있다.

컴파일러를 자동적으로 생성하는 연구는 초반 이후에는 점차 일반적인 처리를 수행하기 위해서 YACC에서

1) XMLSchema의 상속에 관한 내용은 본 논문에서 다루기는 복잡하므로 본 논문에서는 다루지 않는다.

속성 문법을 기술하는 방법과 유사한 방법을 사용하는 형태로 발전하였다. 이러한 연구의 대표적인 것으로는 Warmer[15]의 연구, FleXML[4] 시스템, Uljana[6], XCC[2,3]의 연구가 있다. Warmer는 SGML에서 YACC 스타일로 DTD에 C 코드를 추가하는 방법을 이용해서 SGML 문서 처리기를 자동적으로 생성하는 연구를 수행하였다. Warmer는 DTD에서 원소 선언을 프로시저로 변환하고, SGML의 특정 문서 파서에 의미 정보를 추가하는 방법을 사용한다. Giuseppe[5]는 XML에서 속성 문법을 이용해서 의미를 기술하기 위한 방법으로 별도의 XML 파일에 의미 정보를 사용하는 방법을 사용한다. XCC[2,3]의 연구의 경우에 DTD를 사용하는 XML 문서를 객체지향 속성 문법을 사용해서 XML 컴파일러를 생성한다. XCC는 XML 문서를 처리하기 위해서 별도의 속성 문법을 기술하기 위한 파일을 사용하고, 객체 트리를 이용해서 XML 문서를 표현한다는 특징이 있다. XCC의 단점은 XMLSchema를 지원하지 못한다는 점과 객체지향 속성 문법을 적용하기 위해서 변경해야 한다는 점이다. 즉, DTD에 객체지향 속성 문법을 적용하기 위해서는 제한된 CFG(Restricted Context Free Grammar)[19]로 변환해야 하는 단점이 있다.

이러한 연구들은 모두 XML 컴파일러를 사용하기 위해서 DTD를 사용하였다는 특징이 있다. 그러나 본 논문에서 소개하는 XML 컴파일러 생성기는 기존 시스템이 지원하지 않는 XMLSchema를 지원한다는 점에서 가장 큰 차이점을 갖고 있다.

3. XMLSchema를 위한 XML 컴파일러 생성 방법

3.1 로직 중심의 XML 문서

XML은 다양한 분야에서 다양한 목적으로 사용되고 있다. XML이 사용되는 패턴은 Eric Johnson[1]에 의해서 연구되었다. XML의 사용 패턴은 데이타 중심 사용 패턴과 로직 중심 사용 패턴으로 분류할 수 있는데, 데이타 중심 사용 패턴은 XML 문서에 응용프로그램에서 필요로 하는 데이타를 저장하는 방식이고, 로직 중심 패턴은 XML 문서에 응용프로그램의 로직을 표현하는 방식이다.

데이타 중심 패턴과 로직 중심 패턴은 응용프로그램에서 XML 문서를 처리하는 방식에서 차이점을 보인다. 응용프로그램이 XML 문서에 대해서 수행할 수 있는 연산은 다음 정의와 같이 세 가지로 분류할 수 있다.

정의 1. XML 정보 연산

응용프로그램이 XML 문서의 정보 내용에 적용할 수 있는 연산을 XML 정보 연산이라고 한다. XML 정보 연산은 get, set, interpret로 구분할 수 있다. get은 응용프로그램에서 XML 정보를 얻기 위한 연산이다. 이에 반해 set 연산은 응용프로그램에서 XML 정보를 변경하는 연산이다. interpret는 XML 정보를 해석하는 연산이다. □

XML 정보 연산 측면에서 볼 때 데이타 중심 사용 패턴은 get과 set 연산을 사용해서 작업을 수행한다. 따라서 원하는 데이타에 접근하기 위한 XPath[16], DOM, JAXB[17] 등의 기술을 이용할 수 있다. 그러나 로직 중심 사용 패턴의 경우에는 get과 interpret 연산을 사용한다. get은 XPath, DOM, JAXB 등의 기술을 이용할 수 있지만, interpret 연산을 위해서 제공되는 특별한 기술은 현재까지는 없다. interpret 연산은 XML 문서에서 사용자에게는 묵시적으로 알고 있는 도메인 정보를 명시적으로 기술해서 컴퓨터로 작업을 수행할 수 있도록 하는 것을 목적으로 한다. 따라서 interpret 연산을 수행하기 위해서는 도메인 정보를 명시적으로 기술할 필요가 있고, 기술된 정보를 가공해서 사용자가 원하는 작업을 수행해야 한다. 따라서 interpret 연산을 위한 특별한 기술이 없는 현재에는 interpret 연산을 위한 프로그램을 작성해야 한다.

interpret 연산을 수행하는 프로그램을 자동적으로 생성하기 위해서는 프로그래밍 언어의 컴파일러 기술을 활용할 수 있다. 컴파일러 기술을 이용해서 interpret 연산을 위한 코드를 자동적으로 생성하기 위해서는 XML 문서의 스키마에 의미 정보를 기술해야 한다. 즉, 프로그래밍 언어에서 컴파일러를 생성하기 위해서 속성 문법을 사용하는 것과 유사한 방법으로 XML 문서의 스키마에 의미 정보를 기술할 수 있어야 한다.

XML 문서가 로직 중심으로 사용되는 예를 들어 보기 위해서 사칙 연산을 표현하는 XML 문서가 있다고 가정해보자. XML에서 사칙 연산을 표현하기 위한 수식의 문법은 표 1의 XMLSchema와 같이 기술할 수 있다. exp 원소는 add, sub, mul, div, v 자식 원소들 중에서 하나를 사용할 수 있고, 덧셈 연산을 위한 add 원소는 두 개의 exp 원소로 구성된다. v 원소는 정수 상수를 표현하기 위해서 사용된다.

사칙연산을 위한 exp.xsd가 정의되면, 다양한 형태의 사칙연산을 XML을 이용해서 표현할 수 있다. 예를 들어, (3 + 4) * 5를 표현하기 위한 XML 문서 exp.xml은 표 2와 같이 작성할 수 있다.

컴파일러가 프로그램을 파싱해서 파스 트리로 표현하는 것과 유사하게 XML 파서는 XML 문서를 DOM 트리로 표현할 수 있다. DOM 트리는 XML 문서의 원소, 속성, 텍스트 등을 노드로 표현한 트리이다. DOM 트리에서 원소는 Element라는 클래스로 표현되고, 속성은 Attr 클래스, 텍스트는 TEXT 클래스로 표현된다. exp.

표 1 exp.xsd

```
<xs:schema ...>
  <xs:complexType name="expType">
    <xs:choice>
      <xs:element name="add" type="addType"/>
      <xs:element name="sub" type="subType"/>
      <xs:element name="mul" type="mulType"/>
      <xs:element name="div" type="divType"/>
      <xs:element name="v" type="xs:int"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="addType">
    <xs:sequence>
      <xs:element name="exp" type="expType"/>
      <xs:element name="exp" type="expType"/>
    </xs:sequence>
  </xs:complexType>
...
    <xs:element name="exp" type="expType"/>
</xs:schema>
```

표 2 exp.xml

```
<exp>
  <mul>
    <exp>
      <add>
        <exp><v>3</v></exp>
        <exp><v>4</v></exp>
      </add>
    </exp>
    <exp><v>5</v></exp>
  </mul>
</exp>
```

xml 문서는 XML 파서를 통해서 DOM 트리로 표현되는 경우에 그림 1과 같은 형태로 표현된다.

DOM 트리는 XML 문서에 대한 정보를 트리로 표현한 것이기 때문에 많은 정보들이 포함되어 있다. 그러나 DOM 트리는 XML 문서를 가장 일반적인 방법으로 표현한 트리이기 때문에 사용하기 불편하다는 문제점을 가지고 있다. 또한 트리에 표현된 정보들이 적절한 데이타 타입을 갖지 않는 것도 사용하기 어려운 점이다.

컴파일러에서 파스 트리는 파싱 과정의 결과로 나타나지만, 의미 분석과 코드 생성을 위해서 사용되기에는 부적합하기 때문에 파스 트리는 AST(Abstract Syntax Tree)로 변환해서 사용하게 된다. XML 문서에서도 DOM 트리는 많은 정보를 가지고 있지만, 문서를 효과적으로 처리하기 위해서는 보다 간략한 형태로 표현하여야 한다.

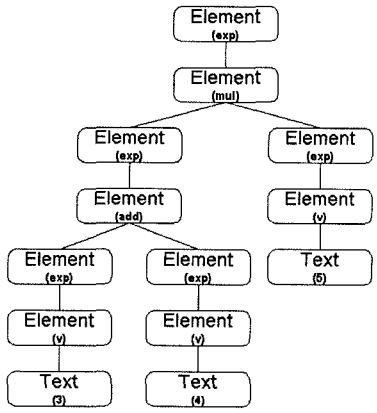

그림 1 exp.xml의 DOM 트리

3.2 XMLSchema의 타입 클래스

프로그래밍 언어에서 객체지향 방법으로 컴파일러를 작성하기 위해서 넌터미널 혹은 생성 규칙을 클래스로 표현하는 방법을 사용한다[18,19]. XMLSchema에서도 이 방법을 적용할 수 있는데, XMLSchema에서는 XML 문서의 기본 구조를 데이타 타입을 이용해서 기술하기 때문에 XMLSchema의 데이타 타입을 클래스로 표현해야 한다. 즉, XMLSchema의 데이타 타입은 XML 원소의 구조를 결정하기 때문에 Context Free Grammar (CFG)의 넌터미널에 해당된다고 할 수 있다. XMLSchema의 데이타 타입을 클래스로 표현하기 위해서 정의 2와 같은 타입 클래스를 사용할 수 있다.

정의 2. 타입 클래스

타입 클래스는 XMLSchema의 simpleType, complexType을 클래스로 표현한 것이다. 이때 XMLSchema의 데이타 타입에 소속된 element와 attribute는 타입 클래스의 멤버 필드로 표현된다. □

타입 클래스는 XML 컴파일러에 의해서 사용되기 위해서 의미 정보를 처리할 수 있는 메소드를 가져야 한다. 이러한 기능을 추가하기 위해서 타입 클래스는 evaluate() 메소드를 가지고 있는 ISemantics 인터페이스를 구현해야한다. 표 3은 ISemantics 인터페이스의 모습을 보여준다. ISemantics의 evaluate() 메소드는 제네릭 프로그래밍 방법에 따라 타입 파라미터 S와 I를 사용한다. I는 상속된 속성을 표현하기 위한 타입 파라미터이고, S는 합성된 속성을 표현하기 위한 타입 파라미터다.

표 3 ISemantics 인터페이스

```
public interface ISemantics<S> {
    public <I> S evaluate(I param);
}
```

타입 클래스를 사용하는 경우에 exp.xsd는 그림 2와 같은 클래스 관계로 표현할 수 있다. 그림 2에서는 ISemantics 인터페이스에 관한 내용은 생략하였다. 즉, ExpType 클래스의 인스턴스는 AddType, SubType, MulType, DivType 클래스의 인스턴스 중 하나 혹은 정수 값 하나만 가질 수 있다. 반면에 AddType, SubType, MulType, DivType는 ExpType의 인스턴스를 2개씩 멤버 필드로 가질 수 있다. 그림 2에서 아크는 컴포지션에서 배타적으로 하나만 가질 수 있다는 의미이다.

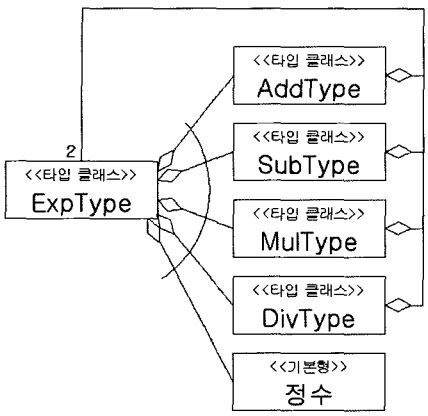

그림 2 exp.xsd의 타입 클래스

그림 2의 타입 클래스 관계는 2가지 문제를 가지고 있다. 첫째는 멤버 필드를 배타적으로 값을 갖도록 하는 것은 UML로 표기하기 어렵다는 점이다. 둘째는 정수라는 기본형을 다른 클래스 타입과 동등한 형태로 표현하기 어렵다는 점이다.

첫 번째 문제를 보면, ExpType 클래스가 5개의 서로 다른 타입을 갖는 멤버 필드들 중에서 한 개의 값만 갖도록 하는 것은 UML로 표기하기 어려울 뿐만 아니라, 프로그램적으로도 매우 복잡한 코드를 요구한다. 따라서 UML로 표현할 수 있고, 프로그램적으로도 쉽게 작성할 수 있는 방법을 생각해야 한다. 여러 가지 해결책들 중에서 공통 인터페이스를 사용하는 방법은 가장 간단하게 이 문제를 해결할 수 있다. 즉, AddType, SubType, MulType, DivType에 공통된 인터페이스(예: IExpType)를 작성하는 방법으로 해결할 수 있다. 따라서 AddType, SubType, MulType, DivType는 IExpType 인터페이스를 구현하고, ExpType는 IExpType 인터페이스 타입을 멤버 필드로 갖도록 한다. 이때 IExpType와 같은 인터페이스를 초이스 인터페이스라고 하자.

정의 3. 초이스 인터페이스

XMLSchema의 타입 A가 CHOICE로 자식 원소를 가질 때 타입 A를 위한 인터페이스를 정의하고, 이것을 초이스 인터페이스라고 한다. A의 자식 원소들은 A의 초이스 인터페이스를 구현해야 하고, A의 타입 클래스는 A의 초이스 인터페이스를 멤버 필드로 가져야 한다. □

두 번째 문제는 기본형은 ExpType와 컴포지션 관계로 표현할 수 없다는 것인데, 이것은 기본형도 초이스 인터페이스를 사용하는 경우에는 클래스로 표현하는 방법을 사용하면 쉽게 해결할 수 있다. 이때 기본형을 클래스로 표현한 것을 기본형 클래스라고 하고, 기본형 클래스는 초이스 인터페이스를 구현하여야 하며, 기본형 클래스 이름은 원소 이름으로 지정한다.

정의 4. 기본형 클래스

기본형 클래스는 XMLSchema의 기본 자료형을 초이스 인터페이스를 구현한 클래스로 표현한 것이다. □

IExpType라는 초이스 인터페이스와 기본형 클래스를 추가해서 그림 2의 클래스 관계를 그림 3과 같은 클래스 관계로 변경할 수 있다.

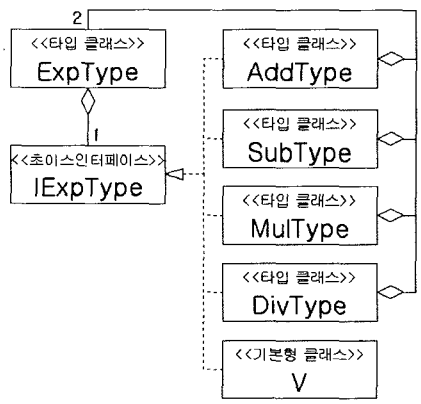

그림 3 exp.xsd의 타입 클래스 II

XML 컴파일러 생성기는 XMLSchema 문서를 읽고, 그림 3과 같은 타입 클래스를 생성한다. 이때 XML 컴파일러 생성기는 "타입_클래스_만들기" 알고리즘을 통해서 타입 클래스를 생성할 수 있다. 이 알고리즘은 2단계로 구성되어 있는데, 1단계에서는 XMLSchema 문서를 읽고, MetaClass 클래스의 인스턴스 그래프를 생성한다. 2단계에서는 1단계에서 생성한 그래프를 순회하면서, 타입 클래스들을 생성한다.

MetaClass는 XMLSchema의 내용을 표현하기 위해서 사용되는 클래스로서 데이타 타입인지 혹은 원소 선언인지 여부를 알 수 있는 kind, 이름을 표현하기 위한 name 멤버 필드를 가지고 있다. MetaClass의 인스턴스는 다른 MetaClass 인스턴스와 관계를 형성하는데, 이러한 관계는 Relation이라는 클래스로 표현된다. XML-

Schema에서 관계는 원소와 데이타 타입을 연결하는 TypeRelation, 데이타 타입과 자식 원소들의 관계를 표현하기 위한 CompRelation이 있다. CompRelation은 콘텐츠 모델을 표현하기 위한 model, 자식 원소들을 표현하기 위한 children 멤버 필드를 가지고 있다. 그림 4는 MetaClass와 Relation의 관계를 보여준다.

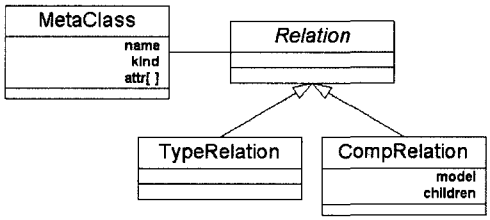

그림 4 MetaClass 클래스 관계

"타입_클래스_만들기" 알고리즘의 1단계에서는 XML-Schema 문서의 DOM 트리를 매개 변수로 받고, MetaClass 클래스의 인스턴스 그래프를 생성한다. 알고리즘에서 typeList, unsolved, elementList는 전역 변수이다.

알고리즘 1. 타입_클래스_만들기 - 1단계
매개변수: DOM 트리
리턴타입: MetaClass 그래프
main() {
  typeList = new List()
  XMLSchema의 기본형들을 typeList에 등록한다.
  unsolved = new List()
  elementList = new List()
  visit (DOM의 루트, schema)
  while (unsolved가 빌 때까지) {
  unsolved에 있는 내용들을 찾아서 TypeRelation 관계를 만들어 준다.
  }
}

visit (node, parentType) {
nodeType = 현재 태그 이름
switch (nodeType) {
  case complexType || simpleType :
    MetaClass 인스턴스를 생성하고, kind 값을 complexType 혹은 simpleType 으로 지정한다.
    typeList에 타입 이름을 등록한다.
    parentType이 element인 경우에는 부모를 찾아서 TypeRelation으로 연결한다.
  case element :
    elementList에 element에 해당되는 것이 없으면, MetaClass 인스턴스를 생성하고, kind 값을 element로 지정한다.
    parentType이 CHOICE, SEQUENCE, ALL 중에 하나이면, DOM 트리의 조상 중에서 첫 번째로 만나는 complexType에 element를 CompRelation으로 연결한다.
    typeList에서 타입을 찾아서 TypeRelation으로 연결한다. 만약 typeList에 존재하지 않으며, unsolved에 등록한다.
  case attribute || attributeGroup :
    조상 중에서 첫 번째로 만나는 complexType의 MetaClass 인스턴스에 이름과 타입정보를 등록한다.
  default : skip
}
foreach child in (node의 자식원소들) {
  visit(child, nodeType)
}
}

XML 컴파일러 생성기에 의해서 1단계가 완료되면, 그림 5와 같은 MetaClass 인스턴스 그래프가 생성된다.

"타입_클래스_만들기" 알고리즘의 2단계에서는 그래프를 너비 우선 탐색 방식을 통해서 순회하면서 타입 클래스를 생성한다.

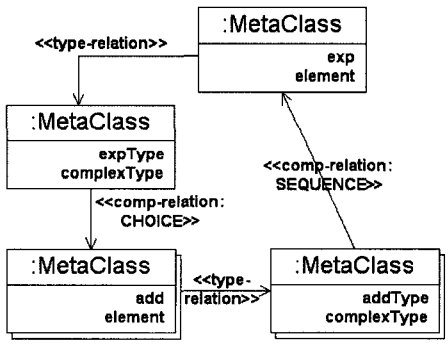

그림 5 MetaClass 인스턴스 그래프

알고리즘 2. 타입_클래스_만들기 - 2단계
매개변수: MetaClass 인스턴스 그래프
결과 : 타입 클래스 생성
while(그래프를 너비 우선순위로 순회) {
  switch(MetaClass의 kind)
    case complexType:
      ISemantics 인터페이스를 구현하는 추상 클래스를 정의한다.

```
            MetaClass에 속성 정보가 있으면, 속성을
            멤버 필드로 갖도록 한다.
            CompRelation으로 연결된 MetaClass들을
            멤버 필드로 갖도록 한다.
            CompRelation이 CHOICE 형태인 경우에
            초이스 인터페이스를 생성한다.
        case simpleType:
            simpleType의 베이스 타입을 멤버 필드로
            갖고, ISemantics 인터페이스를 구현하는
            추상 클래스를 정의한다.
        case element:
            element와 연결된 complexType이 Comp-
            Relation의 CHOICE 형태인 경우에 ISe-
            mantics와 초이스 인터페이스를 구현하는
            추상 클래스를 정의한다.
        default : skip
    }
}
```

3.3 SML을 이용한 의미 기술

XML 문서의 타입 클래스와 초이스 인터페이스를 사용하면, XML 문서를 트리로 표현할 수 있다. 그러나 이 XML 문서 트리는 XML 문서를 어떻게 처리해야 하는지에 대한 별도의 정보는 가지고 있지 않다. 따라서 타입 클래스에 의미 정보와 의미 정보를 평가하기 위한 메소드를 추가한 클래스가 필요한데, 이것을 의미 클래스라고 하자.

정의 5. 의미 클래스

XML 문서를 사용자의 의도에 맞게 처리하기 위해서 타입 클래스에 의미 정보와 의미 정보를 평가하기 위한 메소드를 추가한 것을 의미 클래스라고 한다. □

의미 클래스는 XMLSchema에서 정의한 데이타에 관련된 정보, 의미 정보, 의미를 평가하기 위한 메소드들로 구성되어 있다. 그림 6은 의미 클래스의 구성을 보여준다. XMLSchema의 데이타 타입에 관련된 정보는 노드의 전체적인 구조와 정보들이 포함되어 있다. 의미 속성은 XML 문서를 처리하기 위해서 필요로 하는 정보들을 말하며, XMLSchema가 아닌 별도의 파일에서 제공하는 정보이다. 의미 정보들은 XML 문서를 처리하기 위해서 필요로 하는 정보들을 가지고 있다. 평가 메소드는 XMLSchema 정보와 의미 속성을 이용해서 의미 정보의 값을 결정하기 위해서 사용되는 메소드들이다.

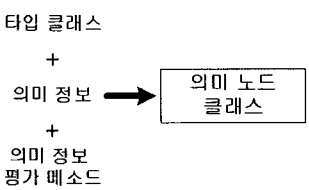

그림 6 의미 클래스 구성

XMLSchema에서 의미 클래스를 생성하기 위해서는 의미 정보와 의미 정보를 평가할 수 있는 메소드를 기술할 수 있어야 한다. 본 논문에서는 XML 문서의 의미 정보를 별도의 SML이라는 XML 파일에 기술하는 방법을 사용한다. SML 문서에는 XML 컴파일러 생성기가 XML 컴파일러를 자동으로 생성할 수 있도록 XML 원소 및 속성의 의미와 이것들이 어떻게 처리해야 하는지에 대한 정보를 가지고 있다. SML이 XML 컴파일러에 영향을 미치는 곳은 크게 3곳이다.

- XML 컴파일러의 컨텍스트 - XML 컴파일러에서 컨텍스트는 심볼 테이블과 같이 XML 문서를 처리함에 있어서 정보를 보관 및 관리하기 위해서 사용되는 부분이다. 이러한 컨텍스트 역시 SML 문서에 의해서 결정된다.
- XML 의미 클래스 - XML 문서를 사용자의 의도대로 처리하기 위해서 타입 클래스에 의미 정보를 추가할 수 있어야 하고, 이때 SML은 타입 클래스에 의미 정보를 제공하는 역할을 한다.
- XML 문서 트리의 visitor - XML 컴파일러에서 XML 의미 클래스의 인스턴스로 구성된 트리를 순회하는 visitor는 XML 문서를 해석하거나 타겟 코드를 생성하는 역할을 한다. 따라서 SML의 정보는 visitor의 행동에 영향을 미친다.

SML이 XML 컴파일러에 영향을 미치는 부분을 효과적으로 지원하도록 구성되어야 한다. SML은 LL(k) 파서를 생성하는 컴파일러 생성기로서 널리 사용되고 있는 ANTLR[20]의 문법 규칙과 유사한 방식으로 헤더 섹션, 규칙 섹션으로 구성된다. XML 컴파일러의 컨텍스트는 컴파일러 전반에 영향을 미치기 때문에 헤더 섹션에 기술해야 한다. 의미 클래스는 XMLSchema의 데이타 타입에 따라 달라지기 때문에 규칙 섹션에 기술해야 한다. visitor에 관한 내용도 규칙 섹션에 기술한다. SML 문서의 개략적인 구조는 그림 7과 같이 표현할 수 있다.

SML 문서는 표 4와 같은 형태로 구성된다. SML의 가장 상위 원소는 semantics이며, semantics는 header, code, rules라는 자식 원소를 갖는다.

header 원소는 속성 문법에서 필요로 하는 정보들을 가지고 있다. 우선 상속되는 속성(inherited attribute)의 데이타 타입을 지정하기 위해서 inherited 원소를 사용하고, 합성되는 속성(synthesized attribute)의 데이타 타입을 기술하기 위해서 synthesized 원소를 사용한다. extends 원소는 XMLSchema를 상속을 통해서 확장하는 경우에 SML 문서를 확장하기 위해서 사용된다.

표 4 의미 규칙을 기술하기 위한 문서의 XMLSchema

```
<xs:element name="semantics">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="header" type="headerType"/>
         <xs:element name="code" type="codeType" minOccurs="0"/>
         <xs:element name="rules" type="rulesType"/>
...
<xs:complexType name="headerType">
   <xs:sequence>
      <xs:element name="extends" type="xs:string" minOccurs="0"/>
      <xs:element name="inherited" type="xs:string"/>
      <xs:element name="synthesized" type="xs:string"/>
...
   <xs:complexType name="ruleType">
      <xs:sequence>
         <xs:element name="action" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="match" type="xs:string"/>
...
```

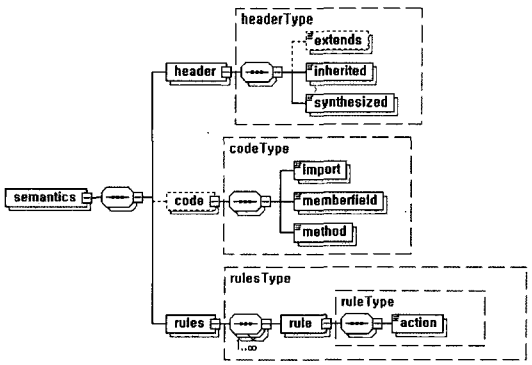

그림 7 sml.xsd 구조

code 원소는 XML 컴파일러에서 필요로 하는 내용들을 기술하며, 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. memberfield 원소는 XML 컴파일러에서 필요로 하는 멤버 필드들을 기술하기 위해서 사용된다. 이러한 멤버 필드는 의미 정보를 표현하기 위해서 사용된다. memberfield 원소에서 기술된 변수들은 method, rules 원소에서 사용될 수 있다. method 원소는 XML 컴파일러에서 필요로 하는 메소드들을 정의하기 위해서 사용된다.

rules 원소에는 문서에서 필요로 하는 의미 속성들을 평가하기 위한 규칙들을 기술한다. rules는 여러 개의 rule 원소들로 구성되고, rule 원소는 XMLSchema의 원소를 지칭하기 위한 target 속성을 갖는다. target 속성에는 원소의 이름 혹은 XPath를 지정할 수 있다. rule은 action이라는 자식 원소를 가지고 있다. action은 XML 문서를 처리하기 위한 기능을 정의하기 위해서 사용된다. action 원소에 정의된 내용들은 XML 컴파일러에서 함수로 표현된다.

action 원소에 포함된 내용 중에서 $는 자식 원소가 하나인 경우 자식 원소의 evaluate() 메소드의 리턴 값을 의미하고, 자식이 여러 개 있는 경우에는 자식 원소 이름 앞에 $를 붙이고, 인덱스와 함께 표기한다. 또한 $$는 원소의 내용을 의미한다.

SML에서 기술한 의미 정보와 의미 평가 규칙들은 의미 클래스를 생성하기 위해서 사용된다. 의미 클래스는 타입 클래스로부터 상속받고, ISemantics 인터페이스의 evaluate() 메소드를 구체적으로 구현한다.

XML 컴파일러 생성기는 SML 문서를 읽어서 의미 정보를 평가하는 규칙을 바탕으로 의미 클래스의 메소드를 정의한다. "의미_클래스_생성" 알고리즘은 XML 컴파일러 생성기가 타입 클래스와 SML 파일로부터 의미 클래스를 생성하는 방법을 보여준다.

표 5 exp.sml

```
<semantics>
  <header>
    <inherited>Object</inherited>
    <synthesized>Double</synthesized>
  </header>
  <rules>
    <rule match="expType">
      <action>
        return $ ;
...
    <rule match="addType">
      <action>
        return $exp[1] + $exp[2] ;
...
    <rule match="V">
      <action>
        return $$ ;
...
```

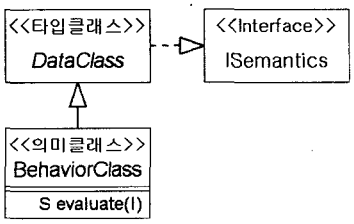

그림 8 의미 클래스의 상속 관계

알고리즘 3. 의미_클래스_생성
매개 변수 : 타입클래스 list[], SML문서 s
유틸리티 함수
search(s, n) - SML 문서 s에서 rule 원소의 target 속성 값이 n에 해당되는 rule 원소를 리턴한다.

```
foreach ( t in list ) {
    의미 클래스 st는 타입 클래스 t로부터 상속받고,
    evaluate() 메소드를 재정의한다.
    rule = search(s, name(t))
    rule의 action 원소의 내용에서 특수한 의미(예: $,
    $$ 등)들을 프로그램 코드로 변환해서 의미 클래
    스 st의 evaluate() 메소드로 복사한다.
}
```

XML 컴파일러 생성기가 의미 클래스를 생성하면, XML 컴파일러는 XML 문서를 의미 클래스의 인스턴스로 구성된 트리로 표현할 수 있다. 이러한 트리를 XML 의미 객체 트리라고 하자.

정의 6. XML 의미 객체 트리
XML 의미 객체 트리는 XML 문서의 인스턴스를 표현하는 트리이고, 다음과 같이 정의된다.

1. 루트 노드는 XML 문서의 문서 타입(document type)을 표현하는 의미 클래스의 인스턴스이다.
2. XMLSchema의 타입 A가 CHOICE로 자식 원소들을 갖는 경우에 타입 A의 의미 클래스는 A의 초이스 인터페이스를 멤버 필드로 갖고, 타입 A의 자식 원소들은 A의 초이스 인터페이스를 구현하는 클래스의 인스턴스이다.
3. 노드 X가 $b_1, b_2, .., b_n$인 자식 노드를 갖고, $b_1, ..., b_n$이 각각 $B_1, B_2, ..., B_n$ 클래스의 인스턴스일 때 노드 X는 $B_1, B_2, ..., B_n$ 타입의 원소를 SEQUENCE 혹은 ALL로 자식 원소를 갖는 원소이다.
4. 단말 노드 t는 XMLSchema의 원소 A의 타입의 XMLSchema의 기본형인 원소이다.

표 2의 exp.xml 문서는 XML 의미 객체 트리를 이용해서 표현하는 경우에 그림 9와 같은 형태의 트리로 나타낼 수 있다.

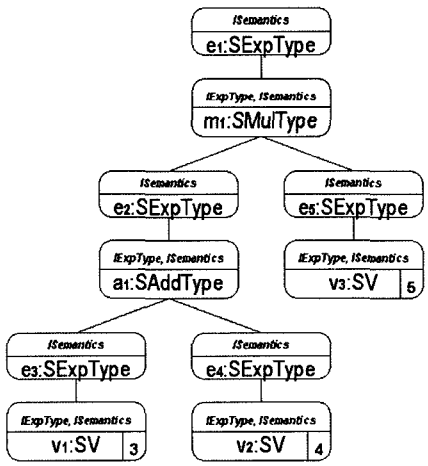

그림 9 exp.xml 문서의 XML 의미 객체 트리

4. XsCC 시스템 구성

본 논문에서 구현한 XML 컴파일러 생성기인 XsCC는 XMLSchema와 SML을 입력으로 읽고, XML 컴파일러를 생성한다. XsCC은 현재 0.7 버전으로 XMLSchema의 대부분의 기능을 지원하지만, 본 논문의 내용을 우선적으로 테스트하기 위해서 상속, 임포트, 인클루드, ref 등의 기능은 지원하지 않고 있다. XsCC의 구현 언어는 자바이고, J2SE 5.0을 사용하여 개발되었으며, XML 파서로는 JDK 1.5에 포함된 크림슨 파서를 사용한다.

XsCC가 생성하는 XML 컴파일러는 일반 프로그래밍 언어의 컴파일러와는 달리 기존 범용 XML 파서를 이용해서 XML 문서의 정적인 문법 체크를 수행한다. 정적 문법 체크가 올바른 경우에 의미 분석과 코드 생성 과정을 거치는데, 의미 분석부터는 XsCC가 생성한 코드가 사용된다. XML 컴파일러는 그림 10과 같은 형태로 구성되고, 회색으로 채워진 XML Parser는 기존 범용 파서를 의미한다.

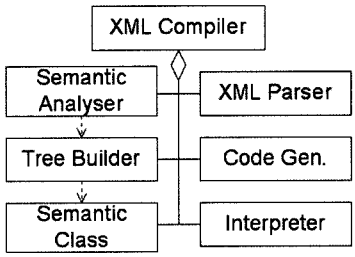

그림 10 XML 컴파일러 구성

XML 컴파일러는 범용 파서를 이용해서 XML 문서를 파싱하면서, XML 문서를 의미 객체 트리 형태로 변환한다. 의미 객체 트리의 각 노드들은 의미 클래스의 인스턴스들이기 때문에 XML 문서에 대한 타입 정보와 함께 의미 정보들도 모두 포함되어 있다. 비지터(visitor)는 비지터 패턴에 따라서 의미 객체 트리를 순회하면서 의미 분석을 수행하고, 코드를 생성하거나 혹은 XML 문서를 해석한다.

XML 컴파일러 생성기인 XsCC는 XMLSchema와 의미 정보를 가지고 있는 SML을 이용해서 XML 컴파일러를 생성하는데, XsCC는 그림 11과 같은 구조로 구성되어 있다. 그림에서 회색으로 채워진 것들은 XsCC에서 자동적으로 생성된 클래스들을 의미한다.

XsCC는 앞에서 설명한 알고리즘들을 이용해서 타입 클래스와 의미 클래스를 생성한다. 표 5는 XsCC에서 생성한 타입 클래스의 예이다. 표 6의 AddType 클래스는 exp.xsd의 addType 타입을 타입 클래스로 표현한 것이다. 타입 클래스에서는 기본적으로 자식 원소를 멤버 필드로 갖고, setter와 getter 메소드들을 갖는다. AddType 클래스는 ISemantics 인터페이스를 구현하지만, evaluate() 메소드는 구현하지 않기 때문에 추상 클래스로 남는다.

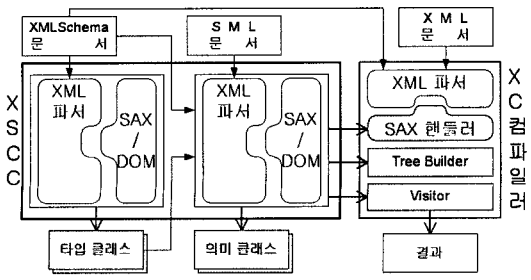

그림 11 XsCC와 XML 컴파일러 구조

표 6 생성된 타입 클래스 코드

```
public abstract class AddType implements
IExpType, ISemantics {
    protected ExpType exp1, exp2;
    public ExpType getExp1() { return this.exp1; }
    public void setExp1(ExpType exp1) {
        this.exp1 = exp1; }
    ...
}
```

표 7은 XsCC에서 생성한 의미 클래스의 예이다. 의미 클래스는 타입 클래스로부터 상속받고, ISemantics 인터페이스의 evaluate() 메소드를 구현한다. evaluate() 메소드는 SML 파일의 내용에 따라서 XML 문서를 처리할 수 있도록 코드를 갖는다. SAddType 클래스의 evaluate() 메소드에서 리턴 타입은 SML 문서의 synthesized 원소에 기술된 타입을 사용하며, 매개 변수는 inherited 원소에 기술된 타입을 사용한다. SAddType의 evaluate() 메소드에서 자식 원소로 갖는 exp 원소의 값을 더해서 리턴한다.

XsCC는 자동적으로 XML 컴파일러를 생성하기 때문에 XML 컴파일러를 개발하는 비용과 노력을 경감시켜 줄 수 있을 것이다. XML 컴파일러의 개발 효율성을 평가하기 위해서 개발자가 직접 XML 컴파일러를 작성하는 경우에 XsCC를 이용해서 XML 컴파일러를 생성하는 경우를 비교하여 보았다. 표 8은 수작업으로 SAX API를 사용해서 XML 컴파일러를 작성한 경우와 XsCC를 이용해서 exp.sml 파일을 작성한 코드의 라인 수를 비교한 것이다. 표 8에서 볼 수 있듯이 XsCC를 사용하는 경우에 직접 작성하는 것에 비해 1/3의 노력으로 XML 컴파일러를 작성할 수 있다.

표 7 생성된 의미 클래스 코드

```
public class SAddType extends AddType ... {
    public Double evaluate(Object p) {
        return (Double)((SExpType)exp1).evaluate(p) +
            (Double)((SExpType)exp2).evaluate(p);
    }
    ...
}
```

표 8 개발의 용이성 비교

	수작업 컴파일러	XsCC 자동 생성
라인 수	130	39

수작업으로 작성한 컴파일러와 XsCC를 통해서 생성된 컴파일러의 성능을 비교해보면, 자동적으로 생성된 컴파일러가 속도 면에서 떨어진다. 다음 그림 12는 2개의 컴파일러의 성능을 측정한 것이다. 테스트는 펜티엄 4의 2.80GHz CPU와 992MB의 램을 사용하는 컴퓨터에서 수행하였다. 테스트를 위한 XML 컴파일러는 XML 문서를 파싱하고, 수식을 계산한 결과를 출력하도록 하였고, XML 파일의 크기는 다음 그림에서 X 축으로 표현하였다. X 축 파일의 크기 단위는 바이트이고, Y 축 시간 단위는 1/1000초이다.

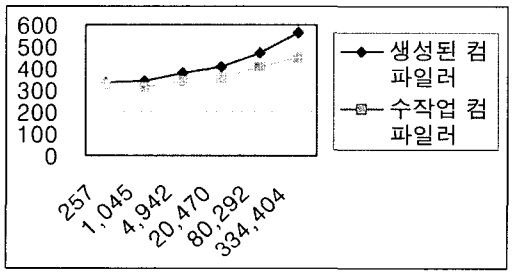

그림 12 성능 비교

그림 13은 XML 컴파일러가 사용하는 메모리(힙 메모리)의 사용량을 체크한 것이다. X 축은 사용된 XML 파일의 크기(단위: 바이트)이고, Y 축은 메모리 사용량(단위: MB)이다. 메모리 사용량을 측정하기 위해서는 JDK 1.5에 포함된 jconsole[21]을 사용하였다.

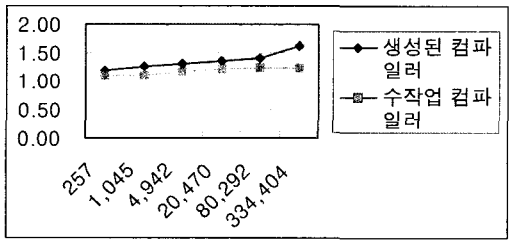

그림 13 메모리 사용량 비교

XsCC를 사용하는 경우에 XML 컴파일러를 쉽게 생성할 수 있는 반면에 성능과 메모리 사용면에서는 수작업으로 생성한 컴파일러에 비해서 떨어지는 것으로 나타났다. 이것은 XsCC를 통해서 생성되는 클래스들이 유연성을 위해서 복잡한 구조를 갖는 것에 비해 수작업 컴파일러는 비교적 간단하기 때문에 메모리 사용량과 성능 면에서 차이가 나타나는 것으로 판단된다. XsCC의 컴파일러 생성에서 최적화에 관한 문제는 향후에 연구가 이루어져야 할 것이다.

5. 결 론

XML은 데이타를 표현하기 위한 표준으로서 산업계 전반에 걸쳐서 폭넓게 사용되고 있기 때문에 XML 문서를 처리할 수 있는 XML 컴파일러의 중요성도 더욱 커지고 있다. 특히 DTD 대신에 XMLSchema를 활용하면, XML 응용을 보다 정확히 표현할 수 있으며, 재사용성을 적극 활용할 수 있다. 따라서 향후 XMLSchema의 사용이 증대되고, 궁극적으로 XMLSchema를 위한 XML 컴파일러의 필요성이 증대될 것으로 예측되기 때문에 본 논문에서는 XMLSchema를 사용하는 경우에 XML 컴파일러를 자동적으로 생성하기 위한 방법을 소개하였다.

본 논문에서는 XMLSchema를 위한 컴파일러를 자동적으로 생성하기 위해서 XMLSchema에 객체지향 속성 문법을 적용하는 방법을 소개하였다. XMLSchema에 객체지향 속성 문법을 적용하기 위해서 XMLSchema의 데이타 타입을 데이타 타입 클래스라는 클래스로 표현하고, 의미 처리를 위해서 이 클래스로부터 상속받는 의미 클래스에 evaluate() 메소드를 구현하는 방법을 사용한다. 의미 정보는 별도의 SML이라는 XML 파일에 기술한다.

또한 XML 컴파일러 생성기는 XML 문서를 처리하기 위해 필요한 TreeBuilder, SAX Handler 등을 생성한다. 생성된 XML 컴파일러는 파싱 및 정적인 의미 분석은 기존 범용 파서를 사용하고, 의미 분석 및 코드 생성 등은 컴파일러 생성기를 통해서 생성된 클래스에서 수행한다. XML 컴파일러는 파싱이 끝나면, XML 문서를 의미 클래스의 인스턴스로 구성된 트리로 표현하고, 트리를 순회하면서 XML 문서를 사용자의 의도에 맞게 처리한다. XsCC를 통해서 생성된 XML 컴파일러는 수작업으로 작성하는 컴파일러에 비해 효과적으로 개발할 수 있다는 장점을 가지고 있지만, 성능 면에서는 약간 떨어지는 문제점도 가지고 있다. 따라서 XML 컴파일러를 최적화시키기 위한 연구가 향후에 이루어져야 할 것이다.

본 논문에서 제시한 XML 컴파일러 생성기와 이를 구현한 XsCC는 기존에 지원하지 않았던 XMLSchema를 지원한다는 점에서 의의를 가진다. 향후에는 XsCC의 기능을 더 추가해서 XMLSchema의 전체 기능을 지원할 수 있도록 할 것이다. 또한 XMLSchema가 상속과 프로그래밍 언어에서 문법 상속의 상관관계를 연구

하고, XMLSchema에서 상속되는 경우에 좀더 효과적으로 처리할 수 있는 방안에 관한 연구를 수행할 것이다.

참 고 문 헌

[1] Eric Johnson, "XML Usage Patterns," In *Proc. of XML Europe*, 2001, available at http://www.gca.org/papers/xmleurope2001/papers/html/sid-01-5.html.

[2] 최종명, 유재우, "객체지향 속성 문법을 이용한 XML 문서 처리기 생성기," *정보과학회논문지 B*, 31권 2호, pp. 224-234, Feb., 2004.

[3] 최종명, 유재우, "XCC : 객체지향 속성 문법과 SML을 이용한 XML 컴파일러 생성기," *정보처리학회논문지 A*, 11권 2호, pp. 149-158, Apr., 2004.

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

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

[6] 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/.

[7] W3C, *XML Schema Part 1: Structures*, available at http://www.w3.org/TR/xmlschema-1/.

[8] Makoto Murata, Donwon Lee, Murali Mani, "Taxonomy of XML Schema Languages Using Formal Language Theory," In *Proc. of Extreme Markup Languages*, Aug., 2001.

[9] Dongwon Lee and Wesley W. Chu, "Comparative Analysis of Six XML Schema Languages," In *Proc. of ACM SIGMOD*, Sep., 2000.

[10] The Simple API for XML (SAX), http://www.saxproject.org/.

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

[12] XSL Transformations (XSLT) Version 1.0, available at http://www.w3.org/TR/xslt.

[13] 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.

[14] 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/.

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

[16] XML Path Language (XPath), available at http://www.w3.org/TR/xpath.

[17] Sun, "Java Architecture for XML Binding," available at http://java.sun.com/xml/jaxb/.

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

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

[20] T. J. Parr and R. W. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," In *Software Practice and Expreience*, Vol. 25, No. 7, pp. 789-810, Jul., 1995.

[21] Mandy Chung, "Using JConsole to Monitor Applications," Sun Developer Network (SDN), 2004, available at http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html.

최 종 명

1992년 숭실대학교 전자계산학과(학사) 1996년 숭실대학교 컴퓨터학과(석사). 2003년 숭실대학교 컴퓨터학과(박사). 2004년~현재, 목포대학교 정보공학부 컴퓨터공학전공 조교수. 관심분야는 멀티패러다임 시스템, XML, 유비쿼터스 컴퓨팅

박 호 병

1999년 숭실대학교 전자계산학과(학사) 2002년 숭실대학교 컴퓨터학과(석사). 2004년 숭실대학교 컴퓨터학과(박사수료). 2006년~현재, 고등기술연구원 선임연구원. 관심분야는 컴파일러, 프로그래밍 환경, XML