2003년도 대한전자공학회 하계종합학술대회 논문집 제26권 제1호

재사용 및 내장 가능한 구성요소 기반 VHDL 분석기

박 상 헌, 손영석
호남대학교 정보통신공학과

Component-Based VHDL Analyzer for Reuse and Embedment

Sanghun Park and Youngseok Shon
Dept. of Information and Communication Eng., Honam University
E-mail : spark@honam.ac.kr and sys@honam.ac.kr

Abstract

As increasing the size and complexity of hardware and software system, more efficient design methodology has been developed. Especially design-reuse technique enables fast system development via integrating existing hardware and software. For this technique available hardware/software should be prepared as component-based parts, adaptable to various systems. This paper introduces a component-based VHDL analyzer allowing to be embedded in other applications, such as simulator, synthesis tool, or smart editor. VHDL analyzer parses VHDL description input, and performs lexical, syntactic, semantic checking, and finally generates intermediate-form data as the result. VHDL has full-features of object-oriented language such as data abstraction, inheritance, and polymorphism. To support these features special analysis algorithm and intermediate form is required. This paper summarizes practical issues on implementing high-performance/quality VHDL analyzer and provides its solution that is based on the intensive experience of VHDL analyzer development.

I. 서 론

하드웨어/소프트웨어 시스템의 규모가 커지고 복잡해짐에 따라 새로운 효율적인 설계방법들이 많이 연구되어 왔다. 주변 기술이 발전하고 응용 프로그램이 복잡해지면서 내장형 시스템에도 과거와는 다른 개발 방법들이 나타나고 있다. 과거에는 주로 가격이나 성능에 기준을 두어 시스템을 개발하였지만, 현재에는 개발기간을 줄이면서 복잡한 응용 프로그램을 쉽게 작성할 수 있는 효율적인 개발환경 구축에 중점을 두고 있다. 특히, 설계-재사용 (Design-Reuse)기술은 기존의 다양한 형태의 하드웨어/소프트웨어들을 통합하여 새로운 시스템을 개발하는 방식으로 빠르고 신뢰성 있는 제품개발을 가능하게 한다. 새로운 기능을 제외한 대부분의 기능들은 기존 하드웨어/소프트웨어를 재사용하고 기타 모듈은 부분적으로 수정하는 방식으로 시스템 개발이 가능할 것이다. 예로, 소프트웨어 기술이 급격히 변화하더라도 새롭게 개발되는 대부분의 소프트웨어의 핵심 부분은 기존에 존재하는 소프트웨어를 재사용하여 구현되어지는 경우가 많다. 이를 위해서는 하드웨어/소프트웨어들이 기계부품처럼 구성요소 기반(Component-Based)으로 정비되어 있어야 하며, 또한 일부 구성요소들은 다양한 제품에서 요구하는 기능을 제공함으로 재사용을 고려하여 개발, 유지, 및 관리되어야 한다 [1,2].

컴퓨터가 출현한 이후로 컴퓨터를 이용한 하드웨어/소프트웨어 개발방식은 비약적으로 발전하여 왔으며 생산품질과 개발기간단축에 크게 기여하여 왔다. 특히, VHDL[3,4]은 하드웨어 설계언어로 80년대 후반에 표준화가 이루어져 지금까지 학계 및 산업현장에서 널리 이용되어 왔다[5-9]. VHDL 분석기(analyzer)는 주어진 VHDL 표현을 읽어서 분석하여 정의된 데이터 모델에 맞는 중간형태의 정보를 만드는 기능을 갖는다. 분석기는 VDT (VHDL Developer's Toolkit)[10]에 포함되어 있는 기본적인 응용 툴로서, VDT를 이용하는 모든 다른 응용 툴들의 입력으로 사용되는 중간형태의 자료를 만드는 역할을 하는 이유로 매우 중요하다. 본 논문은 개발된 VHDL 분석기를 재사용 및 내장 가능한 소프트웨어 모듈로 정비함으로써 다양한 응용분야에 적용할 수 있게 함과 동시에 고기능 툴의 개발을 돕고자 한다. 예로, 최근의 소프트웨어 편집기는 사용자가 코드를 입력함과 동시에 문법과 문맥을 검사해줌으로써 코드입력 오류를 쉽고 빠르게 수정할 수 있게 한다. 내장 가능한 VHDL 분석 모듈이 구성요소 기반으로 준비되어 있으면 이와 같은 기능을 VHDL 편집기에 쉽게 적용 및 구현할 수 있게 된다.

본 논문은 VHDL 분석모듈을 포함하는 개선된 VDT 구조와 분석모듈의 주요 기능들에 대하여 기술한다. 다음 장에서는 기존의 VDT 구조에서 개선된 사항을 살펴본다. III장에서는 주요 분석기능으로 어휘소 및 문법검사 방법, 가시영역과 가시성을 처리하는 방법, 외부에 선언된 객체를 참조하는 방법, 중복선언 요

소들을 처리하는 방법을 설명한다. 마지막으로 구현에 대한 고찰과 함께 결론을 맺는다.

II. VDT 구조

VDT (VHDL Developer's Toolkit)는 VHDL 관련 응용 툴의 개발을 돕는 통합 환경으로 VHDL 원시코드로부터 중간형태의 자료 (Intermediate Form)를 생성하고 재생하고 관리하는 기능을 제공한다. 그림 1은 VDT 구조를 보여주고 있으며 기본적인 툴과 라이브러리 (Procedural Interface)로 구성되어 있다. 기본적인 툴로는 분석기, 재생기, 문맥검사기 등을 포함한다. 라이브러리는 기능별로 분류되어 있으며 또한 계층구조로 이루어져 있어서 쉽게 확장 및 개선이 가능하다. 라이브러리를 계층별로 구분하면 하위계층 (Primitive Layer)과 상위계층 (Application Layer)으로 이루어져 있다. 하위계층은 중간형태의 자료를 직접 생성하고 수정하고 삭제하는 기능을 담당한다. 상위계층은 VHDL 원시코드로부터 중간형태 자료로 변환하고 오류를 검사하고 다시 VHDL 원시코드를 재생하는 등의 기능을 한다. VDT에 포함된 기본 툴들은 대부분은 라이브러리 기능을 이용하여 간단히 구현된다. 예로, VHDL 분석기, 재생기, 문맥검사기는 각각 분석모듈 (Analysis Routines), 재생모듈 (Generate Routines), 검사모듈 (Check Routines)을 호출하는 것으로 쉽게 구현된다.

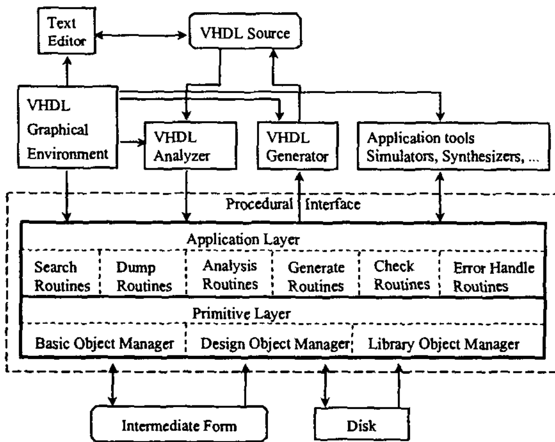

그림 1. VHDL Developer's Toolkit 구조
Fig. 1. Structure of VHDL Developer's Toolkit

기존의 VDT에서 VHDL 분석기능을 요소기반 재사용 가능한 모듈로 구현하기 위해서는 다음 두 가지의 개선작업이 필요하다. 하나는 분석기능 (Analysis Routine)을 라이브러리 형태로 만드는 것이고, 다른 하나는 중간형태 자료의 생성과 삭제가 실행 프로그램 내에서 자유롭도록 하는 것이다. 그림 1에 보인 개선된 VDT 구조는 상위계층에 분석기능을 포함하고 있으며 하위계층에서는 객체 관리 모듈에 객체 삭제 기능을 포함시키고 있다. 기존의 VDT는 객체 삭제 기능이 없었으며 중간형태 자료의 생성과정에서 발생하는 메모리 할당은 프로그램 수행 중에 강제로 반환하지 못하고 프로그램이 종료하면서 자동으로 반환되었다. 그러나 이런 상황은 분석기능을 다른 응용 툴에 내장하는 데 어려움이 있다. 예로, 분석기능을 내장한 VHDL 편집기가 여러 파일을 편집하고 분석하는 과정을 반복하면서 증가하는 메모리 할당은 편집기가 종료될 때까지 반환되지 않으며 쉽게 메모리 부족현상이 발생하게 된다. 요소기반 재사용 가능 분석모듈을 개발하기 위해서는 객체 삭제 기능이 반드시 필요하며 생성 기능만큼이나 많은 노력과 시간이 소요된다. 또한 VHDL 분석과정은 매우 복잡하고 많은 연산과 메모리를 사용하므로 메모리 누수현상이 없도록 체계적인 메모리 관리 알고리즘이 사용된다.

VHDL의 폭넓은 표현력을 지원하는 중간형태의 자료구조는 복잡한 형태를 필할 수 없다. 그림 2는 중간형태 자료표현을 위해 개발된 VDT 데이터 모델이다. 본 모델은 크게 세 객체 (Lib, Design, Basic)로 구성되어 있다. VHDL 원시코드의 문맥은 이들 객체의 복잡한 연결로 표현된다. VDT 하위계층 라이브러리의 세 모듈은 각각 이들 객체를 생성, 조작, 삭제하는 기능을 담당한다. 효율적인 메모리 사용과 효과적인 자료처리를 위해 모든 객체는 테이블로 관리되어 객체 삭제와 메모리 관리를 용이하게 한다.

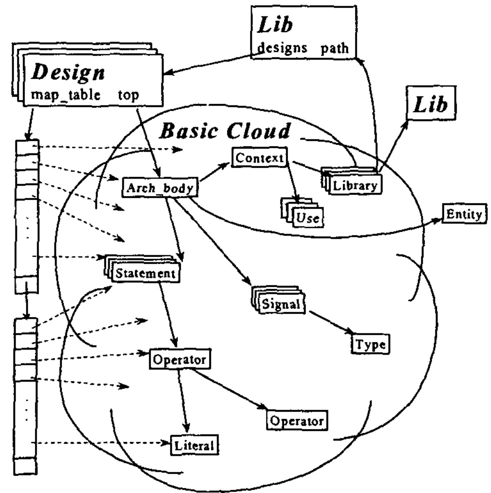

그림 2. VHDL Developer's Toolkit 데이터 모델
Fig 2. Data Model of VHDL Developer's Toolkit

III. 주요 분석 기능

VHDL 분석기는 주어지는 VHDL 표현 내의 정보를 가능한 많이 포함하도록 중간형태의 자료를 만들어야 한다. 예로, VHDL 재생기를 위해 VHDL 각 구문이 기술된 파일내의 라인번호를 그에 해당하는 각 객체에

기록하여 두며, 이로부터 각 객체들에 해당하는 VHDL 구문들의 상대적인 순서를 결정하여 원래의 VHDL 구문과 의미가 같은 내용을 재생할 수 있다. VHDL은 일반적인 프로그래밍 언어와 유사하며, 분석도 그들과 유사한 방식으로 이루어진다. 분석과정에서 여러 가지의 검사가 이루어지며 오류가 발생하면 사용자가 쉽게 파악할 수 있도록 적절한 오류상황을 출력해주어야 한다. VHDL 분석 과정에서 검사되어야 하는 것들로는, 어휘소(lexeme) 및 문법(syntax) 검사, 가시영역(scope) 및 가시성(visibility)의 검사, 수식(expression)에 대한 자료형 검사, 주어진 자료형이 가질 수 있는 값의 범위의 검사 등이 이루어진다.

1. 가시영역 및 가시성 분석

일반적인 프로그래밍 언어에서처럼 VHDL도 주어진 선언이 어떤 범위 내에서 보일 수 있는 가에 대한 가시영역(scope)과 주어진 위치에서 어떠한 선언들이 실제로 보이는 가에 대한 가시성(visibility)이 정의되어 있다. VHDL 구문들 중 선언문들이 쓰일 수 있는 범위를 선언영역(declarative region)이라고 하며 어떤 선언에 대한 가시영역은 일부 예외를 제외하면 그 선언이 시작하는 위치부터 선언영역의 끝까지가 된다. 주어진 선언은 그의 가시영역 내에서만 보이며 외부에서는 보이지 않는다. 즉, 주어진 선언에 대한 참조는 가시영역 내에서만 이루어져야 하며 외부에서 참조하는 경우에는 문맥 오류가 된다. 선언영역을 포함하는 VHDL 구문에 해당하는 VDT의 객체는 그 안에 선언된 구문들에 해당하는 객체들에 대하여 가시영역(scope) 객체의 의미를 갖는다. 이와 같은 처리는 주어진 선언의 가시영역은 그 선언이 시작하는 위치부터라는 가시영역의 정의에 위배된다. 그러나 실제로는 주어진 선언을 분석하고 나서 그에 해당하는 VDT 객체를 생성하고 이를 상위 객체의 심볼 테이블 (symbol table)에 등록하는 순서로 이루어지므로 선언에 대한 가시영역의 처리가 자연스럽게 이루어진다.

2. 외부 참조 분석

VHDL 구문들 중에 독립적으로 분석 단위가 되는 것을 design-unit이라고 한다. Design-unit은 분석된 후 주어진 design-library에 해당하는 디렉토리 내에 파일로 저장된다. 독립적인 design-unit들은 서로 간에 참조가 가능하도록 되어 있으며 일반적으로 use-clause를 통해 이루어진다. Design-unit의 분석을 위해서는 그에 해당하는 VHDL 구문에 나오는 여러 가지 이름들의 사용이 정당(legal)한가를 검사해야 하며 이런 이유로 참조하고 있는 다른 design-unit들을 조사해야 한다. Design-unit은 분석 후에 독립적으로 저장되는데 이때 외부참조로 인해 형성된 relationship을 어떻게 저장할 것인가에 대한 문제가 발생하게 된다. 즉, 이 상태로는 저장 후 다시 읽어 들일 때 외부참조에 대한 relationship을 복구할 수 있어야 한다. 이를 위해 중간형태의 자료를 저장할 때 외부 참조되는 객체가 어디에 포함되어 있는 객체인가에 대한 정보를 저장해야 한다. 즉, 외부 design-unit를 가리키기 위한 정보, 외부참조 되는 객체의 이름, 외부참조 되는 객체의 object-type 등을 저장하게 된다. 이렇게 하여 실제 design-unit이 저장될 때는 외부참조 되는 객체에 대한 위치만을 저장하고 relationship 자체는 저장하지 않는다. 저장된 자료를 읽어 들일 때는 외부 참조되는 객체에 대한 충분한 정보가 있으므로 쉽게 외부 객체로의 relationship을 생성할 수 있게 된다.

3. 문맥 환경 분석

독립적으로 분석되는 VHDL 구문의 단위인 design-unit은 entity-declaration, architecture-body 등으로 구성된다. 각 design-unit은 분석되어 library-unit이 되며, design-library 내에서 독립적으로 관리가 된다. 이때 분석은 context-clause에 명시되는 초기 명칭 환경 (initial-name-environment) 내에서 이루어진다. 어떠한 context에서 분석이 되었는가에 대한 정보를 저장하기 위하여 VDT 데이터 모델에서 각 library-unit 객체는 하나의 context 객체를 포함하게 하였다. Context-clause는 library-clause와 use-clause로 구성되는데, library-clause는 논리적인 라이브러리 이름을 명시하여 design-library가 포함하고 있는 library-unit들을 참조할 수 있도록 한다. Use-clause는 다른 library-unit 내의 declaration들을 명시하여 주어진 design-unit 내에서 직접 사용할 수 있도록 해준다.

Architecture-body의 context는 entity의 context-clause, entity 내에서 선언된 이름을 갖는 객체들, architecture-body의 context-clause 들로 구성된다. 즉, entity의 영역 안에서 보일 수 있는 객체는 architecture-body 안에서도 보일 수 있다. 그러므로 architecture-body를 분석하기 위해서는 entity의 context와 entity 내에서 선언된 이름을 갖는 객체들을 참조할 수 있도록 해야 한다. 이를 효과적으로 처리하기 위해 실제 구현에서는 entity의 context 및 선언 객체들을 외부참조를 통해 architecture-body의 context의 심볼 테이블에 등록한다.

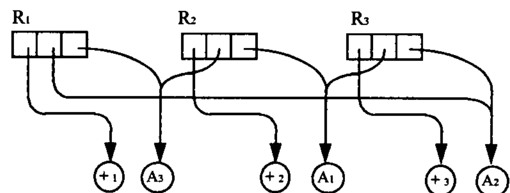

그림 3. 구문 "A+A"에 대한 내부 표현.
Fig. 3. Internal Representation of "A+A"

4. 중복 선언 분석

VHDL은 자료형을 엄격하게 정의하여 사용하는 언어이다. 또한 여러 선언들을 중복시키고 자료형을 통하여 이들을 구별하는 방식을 사용하고 있다. VHDL에

서는 operator, sub-program, enumeration-literal이 중복 선언될 수 있다. 중복해결 (overloading resolution) 이란 중복 선언된 것들을 구별하고, 주어진 상황에 맞는 것을 찾는 것이다. 이때 주어진 상황은 주로 허용되는 자료형이 된다. 예로, 다음과 같은 signal-assignment 문에 대한 분석 및 중복해결은 다음과 같이 두 과정으로 이루어진다.

S <= (A + A) + A;

첫째, 오른쪽 구문의 첫번째 + 연산자는 주어진 두 피연산자의 자료형을 허용하는 연산자로 이미 하나 이상 선언되어 있어야 한다. 둘째, 두 번째 + 연산자의 연산 결과 값의 자료형은 왼쪽 signal 'S'의 자료형과 같아야 한다. 위의 예에서 알 수 있듯이 분석 및 중복해결을 위해서는 서로 다른 두 과정의 작업이 필요하다. 첫 번째 과정은 주어진 가시영역 내에서 변수 및 연산자 등이 이미 하나 이상 선언되어 있는가를 조사해야 한다. 복잡한 수식에 대하여 이 과정은 상향접근(bottom-up) 방식으로 이루어지며, 이 과정을 통해 구문 "A+A"에 대해 생성되는 내부 표현은 그림 3와 같다. 그림에서 {A1, A2, A3}는 각각 {T1, T2, T3}의 자료형을 가진 피연산자를 가리킨다. 여기서 '+' 연산자가 인수로서 {(T1,T2), (T2,T3), (T3,T1)}와 같은 조합을 갖는 것들이 이미 선언되어 있다고 가정할 때, 구문 "A+A"에 대한 '+' 연산자는 그림에서와 같이 세 가지로 해석될 수 있다. 그림에서 {R1, R2, R3}는 각각 {T1, T2, T3}의 자료형을 반환하는 '+' 연산자라고 가정한다.

VI. 결론

재사용 가능한 VHDL 분석기는 발전된 개발환경시스템에 내장되는 것을 허용한다. VHDL을 분석하는 과정은 여러 가지의 검사와 중간형태의 자료를 만드는 작업으로 이루어진다. 중간형태 자료를 생성하는 과정은 VDT가 제공하는 접근루틴(access-routine)을 이용한다. 중간형태 자료구조 및 접근루틴은 객체지향기술을 이용하여 구현되어 있으며 VHDL 분석기 개발에 적합한 환경을 제공한다 [13]. 분석과정에서 검사들은 어휘소 검사, 문법 검사, 문맥 검사의 순서로 이루어진다. 특히, 중복해결 과정은 많은 계산과 시간을 소요하며 실제로 분석과정의 대부분의 시간을 이 과정에서 차지한다. 그러므로 효율적인 자원활용 및 짧은 분석시간을 위해서는 효과적인 중복해결의 처리가 필수적이다. 이를 위해 one-pass 분석 알고리즘 [14]을 사용하였다. VHDL 분석에 대한 검증을 위해 validation-suite을 이용하였다[15]. 완전한 검증은 현실적으로 불가능하며 가능한 한 많은 검증을 필요로 한다. 이와 같은 목적으로 미리 준비된 validation-suite에 들어있는 VHDL 구문들은 체계적이고 효과적인 검증을 위해 준비된 것이다. VHDL 분석기의 검증은 validation-suite을 충분히 실행해 봄으로써 안정하게 동작함을 확인하였다. 또한 이미 구현된 VHDL 생성기를 사용하여 분석된 중간형태 자료를 다시 VHDL 표현으로 재생하고 이를 다시 VHDL 분석기를 통해 생성된 중간형태 자료가 전의 것과 같음을 확인하는 방식으로 VHDL 분석기의 동작을 검증하였다.

참고문헌

[1] A. Sen, "The role of opportunism in the software design reuse process," *IEEE Trans. on Software Engineering*, vol. 23, no. 7, pp. 418-436, Jul 1997.

[2] H. Mili, F. Mili, and A. Mili, "Reusing software and research directions," *IEEE Trans. on Software Engineering*, vol. 21, no. 6, pp. 528-562, Jun 1995.

[3] *IEEE Standard VHDL Language Reference manual*, IEEE Std1076-1987, 1988.

[4] *IEEE Standard VHDL Language Reference manual*, IEEE Std1076-1993, 1994.

[5] D. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton, “Data management and graphics editing in the Berkeley design environment”, *in Proc. ACM/IEEE International Conference on Computer Aided Design*, Nov. 1986.

[6] L. F. Saunders, “The IBM VHDL design system”, *in Proc. ACM/IEEE Design Automation Conference*, pp. 484-490, Jun. 1987.

[7] L. M. Augustin, B. A. Gennart, Y. Huh, D. C. Luckham, and A. G. Stanculescu, “Verification of VHDL designs using VAL”, *in Proc. ACM/IEEE Design Automation Conference*, pp. 48-53, Jun. 1988.

[8] M. J. Chung and S. Kim, “An object-oriented VHDL design environment”, *in Proc. ACM/IEEE Design Automation Conference*, pp. 431-436, Jun. 1990.

[9] B. Harding, “HDLs: A high-powered way to look at complex designs”, *Computer Design*, vol.29, pp. 74-84, Mar. 1990.

[10] S. Park and K. Choi, *VHDL Developer's Toolkit 2.6: User's Guide & Reference*, Tech. Report. SNU-EE-TR-1997-5, 1997.

[11] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, 1988.

[12] B. Stroustrup, *The C++ Programming Language*, 2nd ed., Addison Wesley, 1991.

[13] W. Pree, G. Pomberger, “Object-Oriented versus Conventional Software Development: A Comparative Case Study”, Microprocessing and Microprogramming 35, 1992.

[14] W. F. Tichy, "Smart Recompilation", *ACM Trans. on Programming Languages and Systems*, vol. 8, no. 3, pp. 273-291, Jul. 1986.

[15] J. Armstrong, C. Cho, S. Shah, and C. Kosaraju, "The VHDL Validation Suite", *in Proc. ACM/IEEE Design Automation Conference*, pp. 2-7, Jun. 1990.