VHDL 분석 프로그램의 최적 구현

박상헌* · 이양원**

*호남대학교 정보통신공학과

목 차

Ⅰ. 서 론

컴퓨터가 출현한 이후로 컴퓨터를 이용한 하드웨어/소프트웨어 개발방식은 비약적으로 발전하여 왔으며 생산품의 품질과 개발시간단축에 크게 기여하여 왔다. 특히, VHDL[1][2]은 하드웨어 설계언어로 80년대 후반에 표준화가 이루어져 지금까지 학계 및 산업현장에서 널리 이용되어 왔다[3-7]. VHDL 분석기(analyzer)는 주어진 VHDL 표현을 읽어서 분석하여 정의된 데이터 모델에 맞는 중간형태의 정보를 만드는 기능을 갖는다. 분석기는 VDT(VHDL Developer's Toolkit)[8]에 포함되어 있는 기본적인 응용 툴로서, VDT를 이용하는 모든 다른 응용 툴들의 입력으로 사용되는 중간형태의 자료를 만드는 역할을 하는 이유로 매우 중요하다. 우선 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 분석과정에서 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.

다음 장에서는 어휘소 및 문법검사를 수행하는 VHDL 분석기의 형성과정을 설명한다. Ⅲ장에서는 가시영역과 가시성을 처리하는 방법에 대해 설명한다. Ⅳ장에서는 다른 design-unit 내에 선언된 객체를 참조하는 방법을 설명한다. Ⅴ장에서는 VHDL에서 외부참조를 위해 정의된 context-clause 구문의 의미를 살펴보고 실제 구현에서 이와 관련된 처리방법을 설명한다. Ⅵ장에서는 VHDL의 중복선언 요소들에 대해 살펴보고 이를 해석하여 올바른 중간형태 자료를 만드는 방법을 설명한다. 마지막으로 구현에 대한 고찰과 함께 결론을 맺는다.

II. 구현 방법

VHDL은 일반적인 프로그래밍 언어와 매우 유사하며 분석하는 과정도 유사하다. 일반적으로 프로그래밍 언어의 분석에서 어휘소(lexeme)와 문법(syntax)의 검사를 위하여 각각 LEX와 YACC 같은 소프트웨어 유틸리티를 많이 이용한다. VHDL의 분석을 위하여 이와 유사한 유틸리티인 FLEX와 BISON을 이용하였다. 이들을 이용한 분석기의 형성과정은 그림 1과 같다.

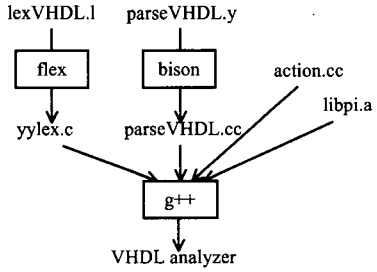

그림 1. 분석기 형성과정

그림 1에서 "yylex.c"는 어휘소를 분석하는 루틴이 들어 있는 C/C++[9,10] 코드로 "lexVHDL.l"로부터 FLEX를 통해 생성된 것이다. "parseVHDL.c"는 문법을 검사하는 루틴이 들어 있는 C/C++ 코드로 "parseVHDL.y"로부터 BISON을 통해 생성된 것이다. "action.cc"는 문법 검사가 통과된 후에 주어진 VHDL 표현에 해당하는 중간형태의 자료구조를 만들기 위한 루틴이 들어있다. 여기에는 중간형태의 자료구조를 만들기 위한 루틴 외에 문법검사를 하는 루틴 이름을 갖는 객체에 대한 가시성(visibility) 및 가시영역(scope) 검사, 일부분의 문맥(semantic) 검사 등을 포함한다. 실제로 중간형태를 만드는 루틴은 중간형태의 자료구조를 접근하는 함수들인 procedural-interface를 통해 이루어지며 분석과정에서는 이들을 사용하여 중간형태의 자료구조를 만든다. "libpi.a"는 procedural-interface를 라이브러리로 형성해 놓은 파일이다. VHDL 분석기는 GNU C++ 컴파일러를 통해 위의 파일들을 컴파일 및 링크하여 생성한다.

III. 가시영역 및 가시성

일반적인 프로그래밍 언어에서처럼 VHDL도 주어진 선언이 어떤 범위 내에서 보일 수 있는가에 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 테이블에 등록하는 순서로 이루어지므로 선언에 대한 가시영역의 처리가 자연스럽게 이루어진다.

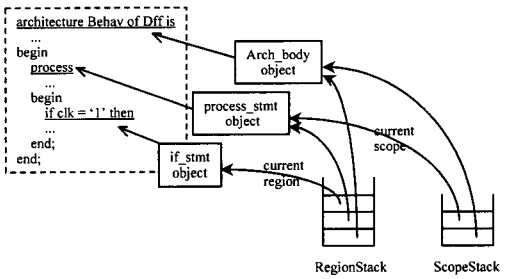

그림 2. 가시성 검사 알고리즘

실제 구현에서 구문의 검사를 하는 프로그램은 파서 생성기인 BISON에 의해 생성된 프로그램을 사용하며 이 프로그램은 재귀적인(recursive) 방법으로 문법을 검사한다. VHDL 표현의 분석에 있어 가시성 검사를 위해 이점을 이용하고 있으며, 이에 대한 자세한 알고리즘은 그림 2와 같다.

프로그램이 재귀적으로 실행되는 상황에서 현재의 가시영역 및 과거의 가시영역을 효과적으로 다루기 위하여 스택(stack) 구조의 자료구조를 사용하였다. 그림 2에서 Scope-Stack은 가시영역을 다루기 위한 스택이고 Region-Stack은 현재 영역을 다루기 위한 스택이다. 여기서 영역과 가시영역의 차이점은, 가시영역을 나타내는 객체가 될 수 있는 것은 자신이 이름을 갖고 있으며 또한 그 안에 이름을 갖는 객체가 선언될 수 있는 것에 해당한다. 반면 영역을 나타내는 객체가 될 수 있는 것은 그 안에 declaration 또는 statement를 포함할 수 있는 것에 해당한다. 예로, 그림에서 if-statement는 영역에 해당하는 객체이지만 가시영역에 해당되는 객체는 아니다. 그러므로 if-statement 내에 나오는 sequential-statement들은 분석과정에서 그에 해당하는 객체가 생성되어 if-statement에 해당하는 객체에 추가된다. 반면 if-statement 내에 나오는 이름들을 찾으려면 그에 해당하는 가시영역인 process-statement에서 찾게 된다. 스택에서 최상위의 엔트리(entry)는 현재의 영역에 해당하는 객체를 가리키고 있으므로 스택으로부터 현재의 영역을 쉽게 알아낼 수 있다. 그림에서 보듯이 항상 Region-Stack은 Scope-Stack 보다 스택의 높이가 같거나 높게 된다.

IV. 외부 참조

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. 그림 3은 design-unit들 사이의 참조과정을 보인 것이다.

그림 3은 복잡한 형태로 이루어진 이름을 분석하는 과정을 보인 것이다. 그림 3의 왼쪽과 같이 외부참조를 포함한 VHDL 구문이 주어졌을 때, 밑줄 쳐진 부분에 해당하는 중간형태자료 표현은 그림의 오른쪽과 같다. 밑줄 쳐진 구문에서 "A"는 signal-name으로 entity 객체에 해당하는 design-unit에 들어 있으며, selected-name의 suffix로 나오는 "Day"는 "P1" package 내에 record-type의 element이다. 이 구문은 분석을 통해 그림과 같이 외부참조 즉, 같은 design-unit 내에 있지 않은 객체를 참조하게 된다.

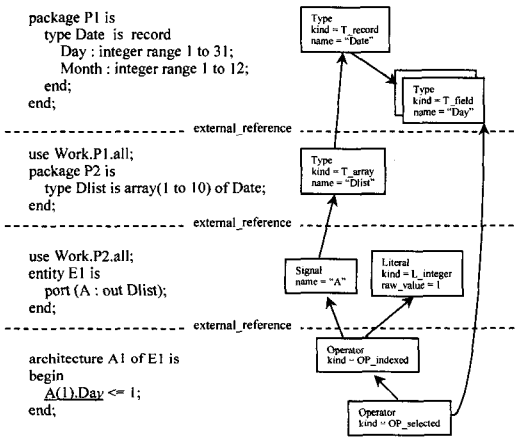

그림 3. 외부 참조 과정

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.

V. CONTEXT CLAUSE

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. 그림 4는 architecture-body의 context-clause가 어떻게 분석되는 가를 보인 것이다.

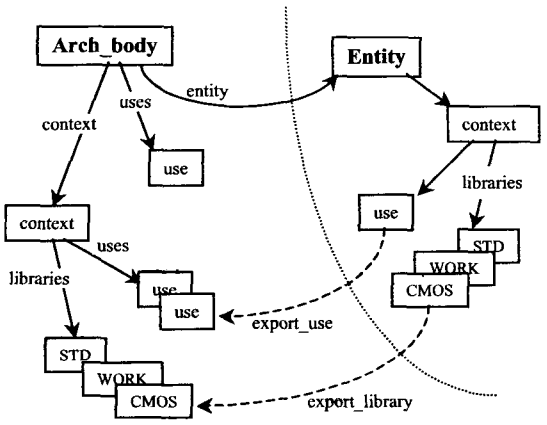

그림 4. Context clause 분석

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. 그림에서 보듯이 architecture-body의 context 객체는 entity의 context 객체가 포함한 library 객체 및 use 객체들을 포함하고 있다. 그림에서 "STD", "WORK" library 객체들은 표준 VHDL에서 내부적으로 선언된 것으로 간주되는 구문에 해당하는 것이다.

VI. 중복 해결

VHDL은 자료형을 엄격하게 정의하여 사용하는 언어이다. 또한 여러 선언들을 중복시키고 자료형을 통하여 이들을 구별하는 방식을 사용하고 있다. VHDL에서는 operator, subprogram, 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 첫 번째 '+' 연산자는 주어진 두 피연산자의 자료형을 허용하는 연산자로 이미 하나 이상 선언되어 있어야 한다. 둘째, 두 번째 '+' 연산자의 연산 결과의 자료형은 왼쪽 signal 'S'의 자료형과 같아야 한다.

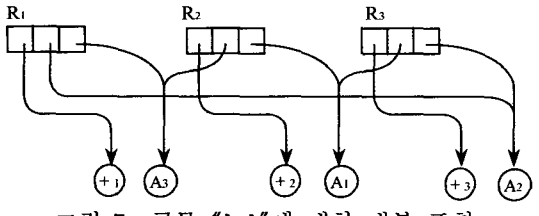

그림 5. 구문 "A+A"에 대한 내부 표현

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 5와 같다. 그림에서 {A_1, A_2, A_3}는 각각 {T_1, T_2, T_3}의 자료형을 가진 피연산자를 가리킨다. 여기서 '+' 연산자가 인수로서 {(T_1,T_2), (T_2,T_3), (T_3,T_1)}와 같은 조합을 갖는 것들이 이미 선언되어 있다고 가정할 때, 구문 "A+A"에 대한 '+' 연산자는 그림에서와 같이 세 가지로 해석될 수 있다. 그림에서 {R_1, R_2, R_3}는 각각 {T_1, T_2, T_3}의 자료형을 반환하는 '+' 연산자라고 가정한다.

그림 6은 구문 "(A+A)+A"에 대한 내부 표현이다. 이 구문의 두 번째 '+' 연산자도 첫 번째 '+' 연산자와 마찬가지로 {(T_1,T_2), (T_2,T_3), (T_3,T_1)}와 같은 인수의 조합에 대한 것으로 이미 선언되어 있으므로 그림에서와 같은 세 가지로 해석될 수 있다. 여기서 {Q_1, Q_2, Q_3}는 {T_1, T_2, T_3} 자료형을 반환하는 '+' 연산자라고 가정한다.

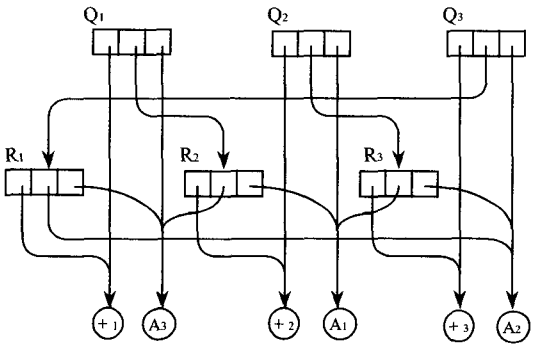

그림 6. 구문 "(A+A)+A"에 대한 중간형태 표현

두 번째 과정은 첫 번째 과정에서 중복 될 수 있는 변수 및 연산자 등을 주어진 상황에 맞는 것으로 선별하는 과정이다. 복잡한 수식에 대하여 이 과정은 하향접근(top-down) 방식으로 이루어진다. 구문 "S <= (A+A)+A"에서 오른쪽 수식의 연산 값에 대한 자료형은 signal 'S'의 자료형과 같아야 한다. 그러므로 {Q_1, Q_2, Q_3}로 해석될 수 있는 상황에서 signal 'S'와 같은 자료형을 반환하는 것으로 결정된다. 마찬가지로 {R_1, R_2, R_3}로 해석될 수 있는 두 번째 '+' 연산자는 위에서 필요로 하는 자료형을 반환하는 것으로 결정된다. 이때 이 과정을 통해서 단 하나의 것으로 해석되지 않고 두 가지 이상의 것으로 해석되는 불명확한(ambiguous) 경우가 생기면 오류상황이 된다. 위의 예에서 signal 'S'의 자료형이 'T_1'이라면 'Q_1'이 선택되며, 따라서 'R_2'와 'A_3'가 피연산자로 선택되고 이어서 'A_3'와 'A_1'이 'R_2'의 피연산자로 선택된다. 그러므로 "(A+A)+A"의 세 피연산자의 자료형은 각각 'T_3', 'T_1', 'T_3'로 결정된다.

VII. 결 론

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[11]. 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[12]을 사용하였다. VHDL 분석에 대한 검증을 위해 validation-suite을 이용하였다[13]. 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.

참 고 문 헌

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

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

[3] 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.

[4] L. F. Saunders, "The IBM VHDL design system", in Proc. 24th ACM/IEEE Design Automation Conference, pp. 484-490, Jun. 1987.

[5] L. M. Augustin, B. A. Gennart, Y. Huh, D. C. Luckham, and A. G. Stanculescu, "Verification of VHDL designs using VAL", in Proc. 25th ACM/IEEE Design Automation Conference, pp. 48-53, Jun. 1988.

[6] M. J. Chung and S. Kim, "An object-oriented VHDL design environment", in Proc. 27th ACM/IEEE Design Automation Conference, pp. 431-436, Jun. 1990.

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

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

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

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

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

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

[13] J. Armstrong, C. Cho, S. Shah, and C. Kosaraju, "The VHDL Validation Suite", in Proc. 27th ACM/IEEE Design Automation Conference, pp 2-7, Jun. 1990.

저 자 소 개

박상헌

1992년 2월 전남대학교 전자공학과 졸업(공학사)
1994년 2월 서울대학교 전자공학과 졸업(공학석사)
1999년 2월 서울대학교 전기공학부 졸업(공학박사)
1999년 4월 ~ 2002년 8월 하이닉스반도체 선임연구원
2002년 9월 ~ 현재 호남대학교 전임교수
※관심분야: VLSI/SoC 설계alc 자동화, 내장형 시스템, 논리합성, 구조합성

이양원

1982년 2월 중앙대학교 전자공학과 졸업(공학사)
1989년 2월 서울대학교 제어계측공학과 졸업(공학석사)
1996년 2월 포항공과대학교 전자공학과 졸업(공학박사)
1982년 ~ 1987년 국방과학연구소 선임연구원
1996년 3월 ~ 현재 호남대학교 정보통신공학과 부교수
※관심분야: DSP, 추정 및 필터이론, 인터넷 응용제어