論文 97-34C-7-1

VHDL 기술의 점진적 분석

(Incremental Analysis of VHDL Descriptions)

安 泰 均 *, 金 求 學 **, 朴 相 憲 *, 崔 起 榮 *

(Taekyoon Ahn, Koo Hak Kim, Sanghun Park, and Kiyoung Choi)

요 약

VHDL로 회로를 설계하여 시뮬레이션할 때에는 분석 및 가공의 두 과정을 거치게 된다. 이 과정에 걸리는 시간을 줄이면 설계 시간을 단축시킬 수 있다. 이를 위하여 본 논문에서는 VHDL 기술의 점진적 분석 및 가공 알고리듬을 제시한다. 어떤 VHDL 기술이 설계 도중 수정되면 이 알고리듬으로 재분석 및 재가공이 되어야 할 설계 단위의 수를 최소화 할 수 있으며, 따라서 설계 시간을 단축시킬수 있다. 실험 결과를 보면 제시된 방법이 기존의 방법보다 분석은 평균 네 배, 가공은 평균 1.25배의 빠른 속도로 수행됨을 알 수 있다.

Abstract

VHDL simulation requires both analysis and elaboration processes. Reducing the time taken by these processes shorten design cycles. We propose an incremental analysis and elaboration algorithm for VHDL, which minimizes the number of design units to be re-analyzed and re-elaborated after an incremental change, thereby reducing the design cycle time. Experimental results show about four times performance improvement in analysis and 1.25 times in elaboration over the conventional method.

I. 서 론

VHDL은 회로를 설계할 때 가장 많이 쓰이는 대표적인 하드웨어 기술 언어이다. VHDL로 하드웨어를 설계할 경우 대개는 최종적으로 설계가 끝날 때까지 반복적으로 시뮬레이션 또는 합성을 하게 되며 이 과정에서 분석 및 가공이 매번 실행된다. 따라서 설계 시간을 줄이기 위해서는 VHDL 기술의 분석과 가공이 빠른 시간 내에 이루어져야 한다.

VHDL에는 어떤 설계 단위(design unit)가 외부의 설계 단위에 있는 선언을 참조할 수 있도록 콘텍스트의 개념이 포함되어 있다. 설계 도중에 VHDL 기술에 수정이 가해졌을 때 기존의 VHDL 관련 툴은 재분석되어야 할 설계 단위 또는 원시 파일을 이 콘텍스트를 통하여 결정한다. 즉 어떤 설계 단위가 수정되면 콘텍스트에 그 설계 단위 내부에 선언된 식별자를 이용한다고 명기해 놓은 모든 외부의 설계 단위를 재분석하게 된다. 이 방법은 UNIX 시스템의 MAKE 프로그램[1]이 프로그래밍 언어를 컴파일할 때 쓰는 방법과 유사하다. 실제로 대부분의 VHDL 시스템은 Make-file을 생성하여 분석할 때 MAKE 프로그램을 이용한다[2]. 즉 모든 설계 단위가 한번씩 분석된 후, VHDL 시스템은 각 설계 단위의 콘텍스트와 계층 구조를 보고 설계 단위간의 관련성을 파악하여 Make-

* 正會員, 서울大學校 電氣工學部
(School of Electrical Engineering, Seoul National University)
** 正會員, LG半導體 技術 硏究所
(DA Center, R & D, LG Semicon Co.)
※ 이 논문은 1994년도 교육부 학술 연구 조성비에 의하여 연구되었음.
接受日字:1996年4月3日, 수정완료일:1997年7月1日

file을 생성한다. 그 이후에는 재분석이 필요할 때에는 사용자가 MAKE 프로그램을 수행함으로써 관련된 모든 설계 단위를 자동으로 재분석하게 된다. 이 경우에는 어떤 설계 단위가 수정되면 콘텍스트에 의해 관련되어 있는 설계 단위와 함께 수정된 설계 단위보다 설계 계층상의 상위에 있는 모든 설계 단위들이 재분석되어야 한다.

그 이유는 MAKE 프로그램의 방식은 기본적으로 UNIX 파일 시스템의 타임 스탬프를 쓰고 또한 상위의 설계 단위가 수정된 설계 단위를 인스턴스로 쓸 때에 그 수정이 접속부(interface)의 조건에 부합하는지를 검사해야하기 때문이다. 따라서 한 설계 단위가 조금만 수정되더라도 전체의 설계를 다시 재분석해야 하는 경우도 생기게 된다.

Tichy는 이러한 문제를 프로그래밍 언어의 컴파일러에서 해결하고자 하였다.[3] 그가 만든 컴파일러에서는 원시 프로그램을 좀 더 자세히 살펴보고, 가해진 수정이 실제로 어의적인 영향을 미치는 컴파일 단위를 찾는다. 프로그래밍 언어에서는 어떤 컴파일 단위가 다시 컴파일되어야 할 것인가를 기본적으로 결정하는 것은 콘텍스트뿐이다.

예를 들면, C 언어에서 어떤 헤더 파일이 수정되었을 때 지시자인 "#include"를 사용하여 직접 또는 간접적으로 수정된 헤더 파일을 참조하는 프로그램 파일이 일차적인 재컴파일 후보가 된다. MAKE 프로그램을 쓰면 이 재컴파일 후보는 모두 재컴파일되나 Tichy의 컴파일러에서는 원시 프로그램의 식별자의 변경 사항을 분석하여 재컴파일되는 컴파일 단위의 수를 줄인다. VHDL에서는 설계 단위 간의 관련성이 일반적인 프로그래밍 언어보다 복잡하다. Package declaration과 body, entity declaration, architecture body, configuration declaration이 서로 관련성을 갖고 있다. 식별자의 선언뿐만 아니라 설계의 계층도 설계 단위 간의 관련성을 결정한다. 따라서 [3]의 아이디어를 VHDL에 적용하기 위해서는 이러한 관련성을 다룰 수 있도록 알고리듬을 확장하는 일이 필요하다.[4]

다음 장에서는 기본적인 아이디어를 예를 들어 설명하고 3 장에서는 제안된 알고리듬을 설명하는데 필요한 몇 가지 정의를 기술한다. 4 장에서는 각각 점진적 분석 알고리듬을 설명하고 마지막으로 실험 결과를 기술하고 결론을 맺는다.

Ⅱ. 점진적 분석의 개요

일반적인 프로그래밍 언어에서는 컴파일러가 원시 파일을 기계어로 번역한다. VHDL은 하드웨어 기술 언어이기 때문에 VHDL 컴파일러는 원시파일이 어떻게 이용되는가에 따라 하는 일이 달라진다. 설계자가 회로를 시뮬레이션하고자 하면 VHDL 컴파일러는 시뮬레이션을 할 수 있는 실행 코드나 자료 구조를 생성하게 된다. 설계자가 게이트 단계의 네트리스트로 합성하고자 하면 VHDL 컴파일러는 시뮬레이션의 경우와는 다른 형태의 자료 구조를 만들어 내야 한다. 따라서 VHDL의 컴파일 과정을 분석(analysis)과 가공(elaboration)의 두 단계로 분리하는 것이 효과적이다. 분석 단계에서는 분석기가 문법과 어의를 검사하고 오류가 없으면 분석된 VHDL 기술을 중간 형태(intermediate form)로 바꾼다. 가공 단계에서는 시뮬레이터나 합성기가 중간 형태를 읽어서 하고자 하는 일에 적합한 형태의 자료 구조로 변환하게 된다. 중간 형태가 최종 목표가 아니기 때문에 중간 형태의 형식은 VHDL 툴을 설계하는 사람에 따라 달라진다. 본 논문에서는 중간 형태가 원시 VHDL 기술에 완전히 대응될 수 있다고 가정한다. 즉 분석 단계에서는 중간 형태를 간략화하거나 수정하지 않고 각 설계 단위 간의 관계를 중간 형태에서도 그대로 유지한다. 그리고 VHDL 기술의 가공은 시뮬레이션을 위해 이루어지는 경우를 다룬다.

VHDL 기술을 처음 분석할 때에는 모든 설계 단위가 분석되어야 한다. 설계를 수정할 필요가 있을 때에는 설계자는 VHDL 원시 기술을 변경하고 VHDL 분석기를 다시 실행한다. 기존의 VHDL 분석기는 변경된 원시 파일의 콘텍스트를 검사하여 변경된 파일과 관련성이 있는 원시 파일이 어떤 것인지를 가려낸다. 그리고 변경된 파일과 관련된 파일을 분석 순서(order of analysis)에 따라 분석한다.

예를 들면 그림 1에서 Traffic_Package라는 package declaration에서 Color이라는 enumeration type declaration의 literal 리스트에 Unknown이라는 네 번째 요소가 첨가된 후 분석된다고 가정하자. 그러면 TLC라는 entity declaration은 use 절에서 보는 바와 같이 수정된 package declaration 내에 선언된 식별자를 사용할 수 있으므로 재분석되어야 한다. Specification이라는 architecture body도 그의 enti-

```
package Traffic_Package is
  type Color is (Green, Yellow, Red);
  type State is ( . . . );
end Traffic_Package;

use work.Traffic_Package.all;
entity TLC is
  port(
    Highway_Light: out Color;
    Farmroad_Light: out Color;
    . . . );
end TLC;

architecture Specification of TLC is
  . . .
end Specification;

entity TB_TLC is
end TB_TLC;

architecture Test of TB_TLC is
  . . .
begin
  UUT: TLC port map( . . . );
  . . .
end Test;

configuration Config of TB_TLC is
  for Test
    for UUT: TLC use work.TLC;
    end for;
  end for;
end Config;
```

그림 1. 교통 신호 제어기의 VHDL 기술
Fig. 1. A VHDL description of traffic light controller

ty declaration의 콘텍스트를 그대로 물려받으므로 또한 재분석되어야 한다. 분석이 끝나면 재분석된 설계 단위는 모두 재가공되어야 한다. 만약 다른 설계 단위가 TLC라는 entity declaration을 인스턴스로 가져다 쓴다면 TLC보다 상위에 있는 모든 설계 단위는 접속부를 검사하기 위해 다시 컴파일되어야 한다.

점진적 분석기에서는 수정된 package에 대한 중간 형태의 구 버전과 신 버전을 비교하여 변경된 식별자의 리스트를 만든다. 위의 예에서 변경된 식별자는 Color인데 이는 TLC 내에서 쓰이고 있다. 그러나 architecture body인 specification은 그 식별자를 참조하지 않으므로 점진적 분석기는 이를 재분석하지 않는다. 재분석되는 설계 단위의 수가 줄어들면 재가공되어야 할 설계 단위의 수도 함께 줄어든다.

VHDL에서는 이러한 예가 설계 계층의 표현에서도 나타난다. 가령 그림 1에서 TLC 의 port 중 일부가 수정되었다고 가정하자. 그러면 기존의 방법에서는 TLC 외에도 TLC를 인스턴스로 써서 설계 계층을 나타내고 있는 Config라는 configuration declaration도 또한 재분석하게 된다. 만약 설계 계층의 단계가 깊고 최하단 entity declaration이 수정된다면 상위에 있는 모든 configuration declaration과 configuration specification이 재분석되어야 한다. 그러나 실제로는 Config가 TLC의 port에 선언되어 있는 식별자를 쓰지 않으므로 재분석되지 않아도 무방하다. 점진적 분석기는 이를 파악하고 수정된 TLC 만을 재분석한다.

Ⅲ. 정 의

VHDL로 설계를 한 후, 그 설계를 변경하기 위해서는 VHDL 기술을 고쳐야 한다. VHDL 기술의 변경에는 외부와는 관련이 없이 그 설계 단위 내부에서 구조 또는 행위를 변경할 수도 있고 접속부를 변경하거나 외부 설계 단위에서 쓰는 함수 등을 변경할 수도 있다. 그리고 어떤 변경이 외부 설계 단위에 영향을 미치다면 그 변경이 식별자 선언의 변경이며 외부 설계 단위가 변경된 식별자를 사용하여 기술되었기 때문이다. 어떤 변경이 외부의 어느 설계 단위에 영향을 미치는지를 판단하기 위해 다음과 같은 집합들을 정의한다.

어떤 설계 단위 u가 주어졌을 때, 참조 집합 REF_u는 다음과 같이 정의된다.

- REF_u : u에서 사용되나 u 안에서 선언되지 않은 모든 식별자의 집합. Entity의 이름은 그 entity declaration 내에서 선언되어 있는 것으로 간주한다.

VHDL에서 식별자는 다른 설계 단위를 거쳐서 간접적으로 참조될 수 없다. 즉 어떤 설계 단위가 use 절을 써서 한 package declaration을 참조한다면 그 설계 단위는 그 package declaration 내에 선언된 식

별자만 이용할 수 있다. 그 package declaration의 use 절을 통하여서는 참조가 불가능하다. 이 참조 집합은 매번 분석할 때 만들어진다.

어떤 설계 단위 u가 주어졌을 때, 관련 집합 DEP_u는 다음과 같이 정의된다.

- DEP_u : u에 선언된 한 개 또는 그 이상의 식별자를 참조하고 있는 모든 설계 단위의 집합.

VHDL에서는 use 절 이외에도 두 설계 단위가 관련성을 갖게되는 또 다른 여러 가지 경우가 있다. Architecture body는 그것의 entity declaration에 선언된 식별자를 자동으로 참조할 수 있게 된다. 마찬가지로 package body는 그것에 대한 package declaration에 선언된 식별자를 자동으로 참조할 수 있게 된다. 어떤 configuration declaration이 한 architecture body를 바인딩(binding)할 때 그 architecture body 내에 선언된 식별자 중 일부를 쓸 수 있다. 따라서 관련성은 use 절 뿐만 아니라 이러한 내재된 관계에서도 찾아야 한다. 설계의 계층 구조는 component configuration의 entity aspect에 의해 정의되며 이도 관련 집합을 만드는데 고려되어야 한다. 어떤 설계 단위 u와 그의 수정된 버전 u'이 주어졌을 때, 다음의 세 집합을 정의할 수 있다.

- ADD_u : u'에는 선언되어 있으나 u에는 선언되지 않은 식별자의 집합.
- DEL_u : u에는 선언되어 있으나 u'에는 선언되지 않은 식별자의 집합.
- MOD_u : u와 u'에 모두 선언되어 있으나 외부의 설계 단위가 참조할 때 그 내용이 달라진 식별자의 집합. u가 entity declaration이고 그 접속부가 변화되었을 때에는 그 entity 자체가 수정되었다고 간주하고 그 이름을 이 집합에 포함시킨다.

이 세 집합은 u에 대한 중간 형태와 u'에 대한 중간 형태를 일일이 비교함으로써 만들어진다.

모든 수정이 수정된 식별자를 사용하고 있는 다른 설계 단위에 영향을 미치는 것은 아니다. 예를 들어 초기값 표현이 주어진 어떤 signal이 entity declaration 내에 정의되어 있을 때, 그 초기값 표현이 수정되더라도 architecture body를 비롯한 다른 설계 단위에는 영향이 없다. 왜냐하면 그 초기값 표현은 entity declaration의 중간 형태만 변화시키며, 다른 설계 단위에서 그 signal을 사용하더라도 그 signal의 이름, 형태 만이 관련되어 있기 때문이다. 집합 MOD에 포함되지 않는 예들은 다음과 같다.

- signal의 초기값 표현의 수정
- physical type declaration의 abstract literal의 수정
- physical type declaration의 각 element의 순서의 변화
- alias declaration의 이름의 변화
- attribute specification의 표현의 수정
- disconnection specification의 시간 표현의 수정
- subprogram 내의 declaration 및 statement의 수정

수정 집합 $CHANGE_u$는 다음과 같이 위의 세 집합에 의해 정의된다.

- $CHANGE_u = ADD_u \cup DEL_u \cup MOD_u$

관련 집합과 수정 집합은 매 분석이 끝난 후에 만들어진다

Ⅳ. 점진적 분석 알고리듬

어떤 설계 단위을 처음 분석할 때에 점진적 분석기는 각 설계 단위의 참조 집합을 만들고 설계 단위 간의 관련성을 조사하여 관련 집합을 만든다. 그런데 어느 설계 단위가 현재 분석되고 있는 설계 단위를 참조하고 있는지를 분석기가 알지 못한다. 따라서 현재 분석되는 설계 단위가 참조하는 설계 단위의 참조 집합에 현재의 설계 단위를 첨가하게 된다. 이후 어떤 설계 단위가 수정되어 재분석이 필요할 때 점진적 분석기는 새로운 버전의 참조 집합을 만들고, 그 설계 단위가 참조하는 다른 설계 단위의 참조 집합에 현재의 설계 단위가 포함되어 있는지를 검사하고 포함되어 있지 않으면 첨가한다. 그리고 수정된 설계 단위와 이전의 설계 단위의 중간 형태를 비교하여 수정 집합을 만들게 된다. 수정된 설계 단위에 문법상 또는 어의상

오류가 없으면 그 수정에 의하여 영향을 받는 설계 단위가 어떤 것들인지를 찾는 과정이 진행된다.

그 과정은 다음의 세 단계로 이루어진다. u_2라는 설계 단위가 u_1이라는 설계 단위에 관련성을 가진다고 가정하면, 즉 u_1 내에서 선언된 식별자를 u_2에서 쓴다고 가정하면, 다음의 수정 분석을 수행함으로써 u_1의 수정에 의해 영향을 받는 설계 단위들을 찾을 수 있다.

1. $ADD_{u1} \cap REF_{u2} \neq \phi$ 이면, 분석기는 오류 메시지를 출력하고 u_2를 재분석하지 않는다. 이는 만약 수정 전에 u_2에는 오류가 없고 이 검사의 결과가 참이면 결국 u_1에 새로이 추가된 어떤 식별자와 같은 이름의 식별자가 u_2가 관련성을 가지고 있는 다른 어떤 설계 단위에 이미 선언되어 있어서 충돌하게 되기 때문이다.
2. $DEL_{u1} \cap REF_{u2} \neq \phi$ 이면, 분석기는 오류 메시지를 출력하고 u_2를 재분석하지 않는다. 이 검사가 참이면 u_2가 u_1 내에 선언된 어떤 식별자를 참조하고 있는데 그 식별자가 u_1에서 삭제된 것이다. 따라서 u_2는 선언되지 않은 식별자를 쓰고 있는 것이 되고 이는 곧 오류가 된다.
3. $MOD_{u1} \cap REF_{u2} \neq \phi$ 이면, u_2는 재분석된다. 이 경우에는 u_1 내에 선언되어 있는 어떤 식별자를 u_2가 쓰고 있는데 그 식별자가 수정되었으므로 u_2는 재분석되어서 수정 후의 어의가 u_2에서 여전히 적합한지를 검사하여야 한다.

점진적 분석의 과정을 그림 2에 나타내었다. 첫 번째 if 문에서 package body가 제외된 것은 package body가 다른 설계 단위에 의해 참조될 수 없으므로 점진적 분석을 할 필요가 없기 때문이다. 2장의 교통신호 제어기에 대한 예에서는 REF_{TLC} = {Color}, $DEP_{Traffic\_Package}$ = {TLC, Specification}, $MOD_{Traffic\_Package}$ = {Color}이므로 $MOD_{Traffic\_Package} \cap REF_{TLC}$ = {Color}가 되어 entity declaration TLC가 재분석된다. 그러나 그 외의 교집합은 모두 공집합이므로 다른 설계 단위는 재분석되지 않는다.

VHDL의 각 설계 단위 간의 관련성이 복잡하기 때문에 설계자가 디버깅을 하는 과정에서 혼란스러운 경우가 있다. 설계자가 어떤 설계 단위를 수정하면 그 수정 때문에 새로 고쳐야 할 설계 단위가 어떤 것들인지를 그가 항상 기억하기는 힘들다. 위의 검사는 설계

```
incremental_analysis(curDU){
  read_intermediate_form(oldDU);
  analyze_design_unit(curDU);
  if(curDU is not a package body){
    update_dependency_set(oldDU, curDU);
    make_change_set(oldDU, curDU);
    if(oldDU exists){  // if not first time analysis
      for(each design unit U in DEP_curDU){
        if(ADD_oldDU ∩ REF_U is not empty)
          print an error message;
        if(DEL_oldDU ∩ REF_U is not empty)
          print an error message;
        if(MOD_oldDU ∩ REF_U is not empty)
          add U in designUnitToBeAnalyzed;
      }
    }
  }
  for(each U in designUnitToBeAnalyzed)
    analyze_design_unit(U);
}
```

그림 2. 점진적 분석 알고리듬
Fig. 2. Incremental analysis algorithm.

자가 수정의 영향으로 새로 고쳐야 할 설계 단위들을 파악하고자 할 때에도 매우 유용하게 사용될 수 있다. 어떤 수정이 다른 설계 단위에 적합하지 않은 영향을 미친다면 분석기는 오류 메시지를 출력하여 수정되어야 할 설계 단위를 설계자에게 제시할 수 있다.

지금까지는 중복(overloading) 문제에 대하여 언급하지 않았다. VHDL에서는 enumeration literal, 연산자, 함수의 중복이 가능하다. 구문 분석기는 enumeration literal과 연산자를 식별자로 인식하지 않는다. 중복을 고려하기 위해서는 앞 장의 집합에 대한 정의에서 식별자의 의미를 확대하여 이들을 포함시켜야 한다. 어의를 검사하지 않고서는 분석기가 중복을 해석(resolve)할 수 없을 뿐더러 중복이 되었는지도 판별할 수 없다. 따라서 참조 집합은 enumeration literal, 연산자, 함수에 대한 중복의 해석이 끝난 후에 만들어져야 한다. 식별자를 비교하기 쉽게 하기 위해 중복의 해석이 끝난 연산자와 함수의 이름에는 매개변수의 형식 이름을 덧붙이고 enumeration literal의 이름에는 enumeration 형식(type)의 이름을 덧붙인다.

재분석된 설계 단위는 시뮬레이션하기 전에 재가공되어야 한다. 점진적 분석에 의해 재분석된 설계 단위의 수가 줄어들어 재가공되는 설계 단위의 수도 또한 줄

어 든다. 그리고 설계의 계층 구조를 가공하여(elaboration of hierarchy) 네트리스트를 만드는 작업은 기존의 방법과 점진적 방법에서 모두 필요하다. 현재 구현된 시뮬레이터는 이 작업이 바인딩 정보를 포함하고 있는 configuration declaration과 architecture body 내의 configuration specification을 재가공함으로써 이루어지도록 설계되어 있다. 그래서 시뮬레이터는 재분석된 설계 단위 외에도 그보다 설계 계층에서 상위에 있는 configuration declaration 및 specification을 최상위 설계 단위까지 재귀적으로 재가공한다.

V. 실험 결과

점진적 VHDL 분석기와 가공기를 포함하는 VHDL 시뮬레이터는 C++ 언어로 작성되어 UNIX 시스템을 기반으로 하는 SUN SparcStation 20에 구현되었다. 구현을 쉽게 하기 위해 IVDT(ISRC VHDL Development Toolkit)[5]가 사용되었다. IVDT는 중간 형태를 다루는 클래스를 제공하며, 이 중간 형태는 VHDL 원시 기술에 완전히 대응된다. 분석기는 IVDT가 제공하는 멤버 함수 중 객체를 생성하는 함수를 써서 중간 형태를 만들게 된다. 시뮬레이터는 자료를 참조하는 멤버 함수를 써서 가공한다. 시뮬레이터는 가공 도중에 VHDL 코드의 동작에 해당하는 C 프로그램을 생성한다. 따라서 가공 과정은 C 프로그램의 생성과 컴파일 과정을 포함한다. 생성된 C 프로그램을 컴파일하기 위해서 UNIX의 C 컴파일러를 썼다. 분석기는 VHDL 1076-1987을 모두 지원하지만 시뮬레이터는 현재로서는 그 중 일부분만을 지원한다. 그러나 시뮬레이터가 지원하는 VHDL의 일부분은 VHDL의 구문 중 많이 쓰이는 것들을 모두 포함한다. 시뮬레이션 알고리듬은 컴파일드 코드 이벤트 구동 방법[6]을 썼다.

각 설계 단위는 각 중간 형태의 파일에 분리되어 관리된다. IVDT에서는 외부의 설계 단위에 선언된 식별자가 외부 참조 객체에 의해 표현된다. 분석기가 현재 분석되고 있는 설계 단위 내에 선언되지 않은 식별자를 만나게 되면 다른 설계 단위에서 그 식별자를 찾아서 외부 참조 객체로 중간 형태에 저장한다. 외부 참조 객체는 만들어질 때마다 참조 집합에 더해진다. 각 설계 단위의 참조 집합과 관련 집합은 따로 따로 파일에 저장된다. 수정 집합은 첫 번째 분석할 때를 제외하고 매 분석이 끝난 후에 중간 형태가 저장되기 전에 만들어진다.

실험에서 사용된 설계의 예는 행위 단계와 구조 단계를 섞어서 기술한 discrete cosine transformer가 사용되었다. 원시 파일은 약 2600 줄이며 76개의 설계 단위를 포함하고 있다. 분석은 파일 단위로 이루어지며 한 entity declaration에 대한 architecture body와 configuration declaration은 그 entity declaration과 같은 파일에 기술되어 있다. 한 package에 행위 단계로 기술된 기본적인 컴포넌트들이 선언(component declaration)되어 있으며 이 컴포넌트들을 사용하는 설계 단위는 이 package를 참조하게 되어 있다. 모든 설계 단위들이 이미 한 번씩 분석이 되어 있는 상태에서 다음의 네 종류의 수정을 가하여 실험하였다.

- 수정 1 : package에 선언된 한 컴포넌트의 접속부를 수정하였다.
- 수정 2 : 한 entity declaration의 접속부를 수정하였다.
- 수정 3 : entity declaration은 그대로 두고 architecture body의 행위 기술 중 일부를 수정하였다.
- 수정 4 : 위의 세 수정을 동시에 가하였다.

실험 결과로부터 기존의 방법과 제안된 알고리듬의 성능을 비교하여 표 1에 나타내었다. 오버헤드 시간은 참조 집합을 만들고 관련 집합을 고치고 수정 분석을 하는 시간에 해당한다. Entity의 접속부가 수정되면 (수정 2) 수정된 접속부가 바로 상위에 있는 설계 단위에서도 계속 적합한지를 검사하기 위해 상위의 설계 단위도 재분석되어야 한다. 기존의 VHDL 분석기는 어떠한 수정이 가해졌는지를 알지 못하기 때문에 모든 상위 설계 단위들을 재귀적으로 재분석한다. 그러나 본 논문의 방법에서는 접속부가 수정되었다는 것을 알고 바로 상위의 설계 단위만을 재분석한다. 수정 3에서는 영향을 받는 다른 설계 단위는 없지만 entity declaration이 수정된 architecture body와 같은 파일에 있기 때문에 기존의 분석기는 수정 2에서와 마찬가지로 동작하게 된다. 그러나 새로운 분석기에서는 더 이상의 다른 설계 단위를 분석하지 않는다. 분석

표 1. DCT에 대한 성능 비교
Table 1. Performance comparison for DCT.

수정	수정 1		수정 2		수정 3		수정 4	
방법	기존	점진적	기존	점진적	기존	점진적	기존	점진적
영향받은 DU 수	31	2	4	2	3	0	34	3
영향받은 파일 수	11	1	3	2	3	0	13	2
재분석된 파일 수	12	2	3	2	3	1	14	4
재분석된 DU 수	34	4	9	6	9	3	40	10
분석 시간	15.1s	2.6s	11.9s	3.4s	12.0s	2.5s	15.8s	4.0s
오버헤드 시간		0.2s		0.2s		0.1s		0.4s
성능 향상	5.8		3.5		4.8		4.0	
재가공된 DU 수	34	6	9	7	9	5	40	12
가공 시간	25.2s	13.2s	17.4s	15.0s	15.2s	13.0s	39.2s	15.5s
성능 향상	1.90		1.16		1.17		2.53	

표 2. 평균 성능 향상
Table 2. Average speedup.

설계	DCT		MIPS Processor		LZ77 Data Compressor		Pipelined FPU		평균	
설계 단위 수 줄 수	76 2600		155 4200		79 2100		26 820		84 2400	
방법	기존	점진적	기존	점진적	기존	점진적	기존	점진적	기존	점진적
평균 분석 시간	12.1s	2.9s	14.6s	3.4s	11.7s	3.0s	7.8s	2.8s	11.6s	3.0s
평균 성능 향상	4.17		4.29		3.90		2.79		3.87	
평균 가공 시간	16.6s	14.4s	23.4s	17.4s	16.7s	14.0s	12.2s	9.4s	17.2s	13.8s
평균 성능 향상	1.15		1.34		1.19		1.30		1.25	

시간은 원시 파일의 크기나 각 설계 단위 간의 관련성에 많이 좌우되므로 파일이나 수정의 수에는 비례하지 않는다. 실험 결과에 의하면 속도 향상이 분석에서는 3배에서 6배 정도이고 가공 과정에서는 최고 2.5배 정도이다. 일반적으로 설계의 크기가 커지면 속도 향상은 더 커질 것이다.

표 2에는 DCT를 포함한 4가지의 설계에 대한 결과가 나타나있다. 모든 예에서 entity declaration과 그에 대한 architecture body, configuration declaration은 한 파일에 있으며, 임의의 한 파일에 임의의 수정을 가했을 때의 분석 및 가공 시간의 평균 값을 구하였다. 분석 시간은 평균 약 4배 정도 빨라지며 가공 시간은 평균 1.25배이다. 가공 시간이 분석 시간보다 성능 향상이 적은 이유는 기존의 방법과 점진적 방법에서 모두 설계 계층을 가공하는 과정이 필요하기 때문이다.

이와 같은 점진적 분석기는 설계자가 설계 단위들 사이의 연관성을 파악하는 데도 도움을 준다. 예를 들면 수정 1을 가하였을 때는 어느 설계 단위가 수정된 요소의 선언을 사용하는지를 알기 힘들다. 설계가 더 복잡해질수록 이러한 상황에서 설계자는 더욱 혼란스러워진다. 그러나 점진적 분석기는 자동적으로 수정에 의해 영향을 받는 설계 단위를 재분석하고 그 설계 단위가 가해진 수정에 적합하도록 고쳐지지 않았다면 구문 오류나 어의 오류 메시지와 함께 수정 분석의 결과, 즉, 식별자의 삭제, 추가, 수정 중 하나를 출력하게 된다. 이 메시지를 보고 설계자가 관련된 설계 단위를 명확히 파악하고 그 오류의 원인을 쉽게 알 수 있다.

VI. 결 론

본 논문에서 기술한 점진적 분석 및 가공 알고리듬은 잉여의 과정을 제거함으로서 디버깅 시간을 효과적으로 줄일 수 있다. 또한 알고리듬이 간단하기 때문에 VHDL 시스템 개발자는 쉽게 이 알고리듬을 구현할 수 있다. 분석과 가공 시간이 줄어들면 컴파일 속도는 빠르나 이식성에는 문제가 있는 native compiled code simulation[7] 방식과 비교하여 C 프로그램 생성 방식이 갖는 단점을 줄이는 데도 효과적이다. 또한 본 알고리듬은 식별자에 대한 구문 오류나 어의 오류가 있을 때 수정 분석에 의해 자세한 오류 원인을 파악하여 설계자에게 알려준다. 더욱이 그 오버헤드가 적기 때문에 최악의 경우에도 기존의 방법과 같은 수준의 시간에 실행 되도록 구현될 수 있다.

참 고 문 헌

[1] S. Feldman, "Make-A program for maintaining computer programs", *Softw. Pract. Exper.*, vol. 9, no. 3, pp. 255-265, Mar. 1979.

[2] Synopsys, Inc., *VHDL System Simulator Command Reference Manual Version 3.0*, 1992.

[3] Walter F. Tichy, "Smart Recompilation", *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 3, pp. 273-291, July 1986.

[4] T. Ahn, K. H. Kim, S. Park, and K. Choi, "Incremental Analysis and Elaboration of

VHDL Description", *Proceedings of Asia Pacific Conference on Hardware Description Languages*, pp. 128-131, 1996.

[5] D. H. Ko and K. Choi, "IVDT:A VHDL Developer's Toolkit", *KITE Journal of Electronics Engineering*, vol. 5, no. 2, pp. 56-63, Dec. 1994.

[6] Y. S. Lee and P. M. Maurer, "Two new techniques for compiled multi-delay logic simulation", *Design Automation Conference Proceedings*, pp. 420-423, 1992.

[7] "Leapfrog, High-Performance VHDL Simulation", *Cadence Design Automation Show '93*, Cadence Design Systems, Inc., pp. 244-247, 1993.

저 자 소 개

安 泰 均(正會員)

1969년 5월 17일생. 1991년 한국과학기술원 전기 및 전자공학과(공학사). 1993년 서울대학교 전자공학과(공학석사). 1995년 서울대학교 전자공학과 박사과정 수료. 주관심분야는 CAD, VLSI 설계 등임.

朴 相 憲(正會員)

1992년 전남대학교 전자공학과(공학사). 1994년 서울대학교 전자공학과(공학석사). 1996년 서울대학교 전자공학과 박사과정 수료.

金 求 學(正會員)

1991년 영남대학교 전자공학과(공학사). 1993년 영남대학교 전산공학과(공학석사). 1993년 ~ 현재 LG 반도체(주) 기술연구소 근무.

崔 起 榮(正會員) 第 34卷 C編 第 3號 參照

현재 서울대학교 전기공학부 부교수