SpecCharts로부터 합성 가능한 Synchronous VHDL 코드 생성기 설계 및 구현

윤 성 조[†]·최 진 영[††]·한 상 용[†††]·이 정 아[††††]

요 약

가상 프로토타입(Virtual Prototyping; VP) 방법론을 이용하면 내장형 시스템을 설계하고 구현할 때에 비용을 절감하면서 제품의 개발기간을 단축할 수 있다. VP는 S/W component, H/W component 그리고 S/W 와 H/W를 연결하는 Interface component로 구성되어 진다. VP의 구성 요소중 H/W component를 구현하는 방법은 여러 가지가 있으나 시스템 명세 언어로부터 하드웨어 컴포넌트로 구현하는 방법을 고려하고자한다. 그러나 시스템 명세 언어로부터 생성된 H/W component 용 VHDL 코드는 항상 합성 가능한 코드라고 할 수 없다. 본 논문에선 시스템 명세 언어로부터 검증을 용이하게 하는 하드웨어 구현을 위하여 명세언어로써 SpecCharts를 이용하고 이로부터 동기적 의미론을 만족하는 합성 가능한 VHDL 코드를 생성하는 방법론을 제시한다.

Synthesizable Synchronous VHDL Code Generator Design and Implemetation from SpecCharts

Seong-Jo Yun[†] · Jin-Young Choi[††] · Sang-Yong Han[†††] · Jeong-A Lee[††††]

ABSTRACT

We are using a methodology of virtual prototype(VP) which can reduce costs and developement time to the market. VP is composed of S/W component, H/W component, and interface component which links H/W to S/W. There are many methods of realizing H/W components, but we adopt a method which translates from system specification into hardware description in VHDL.

In this paper, we present design and implementation of code generator from SpecCharts as system specification language to a VHDL code which can be synthesized and verified. The verification becomes feasible when the hardware satisfies synchronous semantics, which we call, Synchronous VHDL.

1. 서 론

현재 많은 내장형 시스템(Embedded System)을 구현하기 위한 방법론으로 VP(Virtual Prototyping)을 이용하고 있다. VP로 설계된 시스템 요소들을 하드웨어 코드나 소프트웨어 코드로의 변환 부분이 필요하게 된다. 이 과정은 가상 현실로부터 해당 시스템의 정형명세로 이루어지는 시뮬레이션을 거쳐 요구사항에 관한 정형 명세를 검증 후 하드웨어 코드나 소프트웨어 코드 변환단계를 거쳐 합성 과정을 통해 실제 시스템과 동일한 기능을 수행하는 제품을 생성하게 해준다. VP

※ 이 논문은 1998년도 학술진흥재단 연구비의 지원을 받아 연구되었음.(과제번호 : 1998-016-E00060)
† 준 회 원 : 조선대학교 대학원 전자계산학과
†† 정 회 원 : 고려대학교 컴퓨터학과 교수
††† 정 회 원 : 중앙대학교 컴퓨터공학과 교수
†††† 정 회 원 : 조선대학교 컴퓨터공학부 교수
논문접수 : 2000년 5월 10일, 심사완료 : 2000년 10월 27일

은 특히 마이크로프로세서나 주문형 프로세스가 포함된 내장형 시스템의 설계 및 구현에 용이하다는 장점으로 생명 주기를 단축시키고 제품에 신뢰성을 높이고 있다.

내장형 시스템을 개발할 때에는 시장진입시점(Time-to-Market)의 효율적인 운영이 매우 중요하다. VP를 이용하여 시스템을 개발하는 경우, 시장진입시점을 줄일 수 있기 위하여, 시스템 명세로부터 구현 가능한 시스템으로의 체계적인 방법론이 더욱 더 필요하게 된다.

현재까지 많은 연구를 통해 시스템 명세 언어를 통해 시스템을 기술하고 모델링하는 부분에 대한 연구는 꾸준히 진행되어 왔다. 본 논문에서는 시스템의 요구사항을 검증을 통하여 시스템 명세 언어인 SpecCharts로 명세된 형태를 cycle-level 추상화를 통한 VHDL 코드로의 변환을 통해 동기적 의미론(semantics)을 만족하도록 하여 동기적 VHDL 코드를 생성한다. 이렇게 생성된 VHDL 코드가 합성가능한 RTL(Register-Transfer-Language)-level 및 behavior-level의 코드로 변환하게 하면 검증이 용이하게 되고 시스템 명세로부터 합성 단계까지의 과정을 자동화 시킬 수 있다.

이에 대한 방법론으로 시스템 명세 언어인 Spec Charts로부터 합성가능한 VHDL 코드를 생성하기 위해 계층이 없는 flat한 구조의 process들로 코드를 생성하여 SpecCharts가 가지는 세 가지 behaviors-sequential, concurrent subbehavior behavior 그리고 code behavior-가 지닌 의미론을 지원하는 VHDL 코드 생성 방법을 제안하였다.

그리고, 동기적 VHDL 코드를 생성하기 위해 W. Baker에 의해 규명된 동기적 VHDL subset을 적용하여 시스템 명세 언어인 SpecCharts로부터 합성가능한 동기적 VHDL 코드로 변환하는 방법을 제시하였다. 이를 통해 결과적으로 합성 가능한 동기적 VHDL 코드로 변환이 되어지는 코드 생성기를 설계하고 구현한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 코드 생성 및 구현을 위한 요구사항에 대하여 구체적으로 살펴보겠다. 3장에서는 시스템 명세 언어 중에 하나인 SpecCharts를 고찰하고, 4장에서는 동기적 VHDL의 subset에 대해 살펴볼 것이다. 5장에서는 SpecCharts로부터 VHDL 코드 생성에 대해, 6장에서는 합성가능한 동기적 VHDL로 변환기 설계에 대한 내용을 다룬다. 변환된 VHDL 코드를 동기적 VHDL로 변환에 대하여, 7장에서는 컴파일러 생성 도구인 lex와 yacc를 이용하여 입력된 코드를 토큰링하고 파싱하는 것에 대하여 살펴보고, 마지막으로 8장에서는 결론 및 향후 방향에 대해서 언급한다.

2. 코드 생성기 설계 및 구현을 위한 요구사항

2.1 코드 생성기 설계를 위한 전제 조건 및 요구사항

코드 생성기 설계를 위해 먼저 필요한 요구사항은 개발하기 위한 시스템을 효율적으로 해당 모듈의 의미론에 맞게 기술했는지 검증할 필요가 있다. 이는 시스템 설계시에 사용된 명세 언어가 갖는 의미론과 변환 후 VHDL 코드가 갖는 의미론이 같아야 제대로 된 변환을 했다고 말할 수 있기 때문이다. 이를 위해서, 본 논문에서는 다음과 같은 가정을 한다. 전제적으로 변환 과정을 위해 필요한 SpecCharts 원시 코드를 전체 변환기의 입력으로 받으며 이는 시스템 디자이너가 기술한 명세 내용을 정형 명세를 통해 검증된 형태로 생성한 SpecChart 코드로부터 시작함을 가정한다.

SpecCharts로부터의 코드 생성시 생성되는 VHDL 구문의 제한을 두기 위한 방법으로 제한적인 VHDL 구문을 정의하여 본래의 의미론을 위배되지 않게 제한하여야 한다. 또한 표현이 어렵거나, 의미론에 위배된 경우엔 해당 구문들에 대한 새로운 구문에 대한 정의를 하여야 한다. 이는 동기적 VHDL 코드를 제한하는 부분에서도 동일하게 적용되어진다. 이를 위하여 본 논문에서는 동기적 VHDL에 대해 4장에 언급될 것이다.

합성 가능한 모델은 VHDL 구문에선 매우 제한적으로 사용되어 진다. 합성의 단계는 변환(translation)단계와 최적화(Optimizing) 단계로 나뉜다. 본 연구의 합성 가능한 코드생성의 경우는 Behavior-level이나 RTL-level에 대한 구분을 두지 않고 SpecCharts에 기반하여 원시 소스를 이용한 변환까지로 합성방안을 제한한다.

2.2 기존의 SpecCharts로부터 코드 생성에 대한 연구

먼저 SpecCharts를 작성하기 위한 편집도구가 필요하다. 하지만, 현재까지 구현된 SpecCharts editor는 거의 없다. SpecCharts 대표적인 editor로 D. Gajski가 만든 Xspeccharts라는 도구를 통해 GUI환경에서 누구나 쉽게 SpecCharts를 생성할 수 있다. 이 Xspeccharts란 도구는 원래 HW/SW 통합설계를 위한 목적으로 Specsyn이란 도구에 패키징되어 있는 도구이다. HW/SW 통합설계 도구인 Specsyn에서도 sc2vhdl 이라는

VHDL 코드 생성기가 있다. 하지만 이 코드 생성기는 시스템 명세 언어인 SpecCharts에서 시뮬레이션이 가능한 VHDL 코드만이 생성될 뿐이지 합성이 가능하지 않는 실정이다.

sc2vhdl을 이용해 VHDL 코드를 생성하는 경우 코드의 구성은 guard signal을 이용하여 다중 block 구문으로 구성되어지게 하였고, 각 상태들(states)를 표시하는데 inactive, excuting, complete 상태로 구분하여 각 상태를 표시하였다. 이를 통해 시스템 전체의 상태를 이 세 가지 상태로 표시하게 하였고, 이를 이용하여 상태전이가 일어나도록 하였다. 또한 내부적인 subtype을 통해 새로운 type 선언을 지원하였으며 이는 signal을 register 형태로 표현하여 합성에 불가능한 요소가 되었다.

이렇게 sc2vhdl 시뮬레이션을 위한 코드 생성 구조로 되어 있으므로 CAD tool을 이용하여 합성하기가 사실상 불가능하다. 그러므로 본 논문에서는 자동적으로 합성 가능한 VHDL 코드 생성에 대한 방안에 대해 고찰하고 이를 통해 실제로 VHDL 코드로의 코드 생성기에 대한 설계와 구현에 대해 살펴본다.

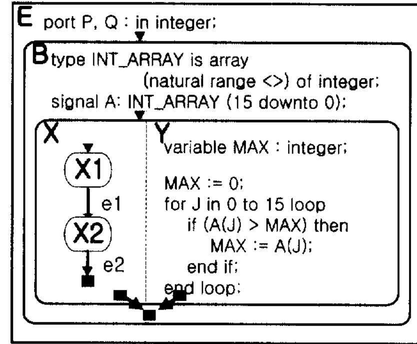
3. SpecCharts에 대한 고찰

3.1 SpecCharts의 특징

SpecChart는 내장형 시스템 명세를 위해서 D. Gajski가 제창한 것으로 프로그램-상태 기계(Program-state machine;PSM)와 VHDL를 결합한 형태로 PSM을 이용한 시스템 명세와 각 상태들에 대한 행위 기술을 VHDL로 표현할 수 있게 해준다.

SpecCharts가 지원하는 특징들은 다음과 같다.

- 첫째, D.Harel이 주창한 StateCharts의 시각적인 효과와 VHDL처럼 textual하게 표현이 가능하다.
- 둘째, 행위의 계층성을 지원하여 순차적(sequential)이거나 병행적(Concurrent)한 행위를 기술할 수 있다.
- 셋째, 상태간의 전이를 TOC(transition on completion) arcs를 통해 지원한다. 수행 중 예외 상황을 TI(transition immediately) arcs를 통하여 지원한다.
- 넷째, 상태(프로세스)간의 통신을 공유 메모리(shared memory), 메시지 전달(message passing)을 통하여 지원한다.

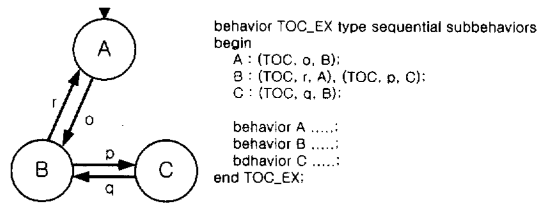


(그림 1) SpecCharts의 표현법

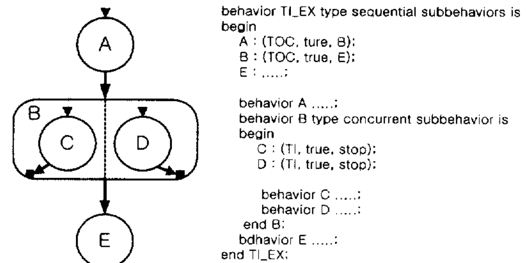
또한, VHDL과 유사성을 가지고 있어서 다음과 같은 특성들을 지닌다. 프로그래밍 구성체(programming constructs)이고, 구조적인 계층성(Structural hierarchy)을 갖으며 동기화와 타이밍을 지원한다.

SpecCharts의 상태전이는 TOC와 TI arcs 두 행위(behavior)로 나타난다. TOC는 조건에 관계된 상태에 연결되어 있는 프로그램이 완결되었을 때 상태전이를 발생하게 하고, TI는 해당 조건이 만족하는 그 순간 상태전이를 발행하게 하는 차이를 갖는다.

3.2 SpecCharts의 계층성과 behaviors



(a) TOC arc를 이용한 상태 전이



(b) TI arc를 이용한 상태 전이

(그림 2) SpecCharts의 상태전이와 behavioral 계층성

SpecCharts에서 계층성은 내포된 행위들(nested behaviors)을 통해 표현되어 지며, 각각의 행위들은 순차적(sequential)과 병행적(concurrent) 하위행위들(subbehaviors)로 분해할 수 있다. 그러므로, SpecCharts의 구조는 층계 구조 형태로 이루어져 있으며 각 층계들은 두 가지의 형태로 구성된다. 그 중 하나는 OR-level의 성격을 지닌 순차적 하부행위의 타입이 되고, 나머지 하나는 And-level의 성격인 병행적 내부행위의 타입을 갖는다. 층계 구조의 터미널 노드의 성격을 지닌 잎노드는 code라는 타입으로써 실제 수행되는 행위들이 기술되어 진다.

3.3 SpecCharts의 의미론(semantics)

SpecCharts의 원래의 의미론은 Statecharts의 discrete-semanics와 VHDL의 discrete-event semantics의 조합으로 이루어진다. VHDL-1076과 Statecharts의 semantics는 전달 이산 사건(propagation discrete event)에 근거하여 형성된다. 즉, Speccharts의 표현의 편리함은 시스템 표현 시에 언어의 문법적인 측면을 고려하였다고 말할 수 있다.

4. Synchronous VHDL subset

4.1 Synchronous VHDL에 대한 정의

synchronous VHDL은 기존의 full VHDL에서 동기적 의미론(synchronous semantics)를 만족시킬 목적으로 Wendell Baker에 의해 VHDL에 제한성을 부여하여 규명되었다. 그 이유로는 VHDL이나 Verilog 언어의 경우에는 discrete event 의미론으로 해석되는 경우에 동기적 의미론 성질을 표현하기가 어렵기 때문이다. Baker가 주장한 바에 의하면 유한 오토마타 VHDL 시뮬레이터를 통해 생성된 VHDL 코드와 실제의 전체의 VHDL 시뮬레이터를 통해 나온 결과를 동등하게 할 수 있다라고 하였다. 여기서 유한 오토마타 VHDL 시뮬레이터는 추상적(abstract) 시뮬레이터로 그 기능은 유한적인 메모리 공간에서 정해진 수행 시간 내에 동작하도록 하였다. synchronous VHDL subset은 유한 상태의 의미론을 기초로 생성된 의미있는 방법론이다.

4.2 Synchronous VHDL subset 를 위한 제한사항

synchronous VHDL subset을 규명하기 위하여 전체의 VHDL에 대한 4가지 제한 사항을 적용하여야 한다.

- 첫째, 이벤트들을 관리하는 시그널 큐들의 길이를 하나의 크기로 제한한다. 즉, VHDL상에서 waveform 대입문의 사용이나 전달(transport) 또는 관성(inertial) 지연(delay)이 임의의 상태들의 수를 적용하기 때문에 제약을 두어야 한다.
- 둘째, 프로세스 안의 순차적 코드는 스택을 사용하지 않아야 한다. 즉, 중첩된 함수(프로시저)나 재귀 호출, 동적 크기의 객체 타입(array type), 지역적인 정적(static) 객체들의 사용을 제한하여야 한다.
- 셋째, 동적 저장장소 할당 사용을 제한하여야 한다. 즉, 힙(heap) 영역을 사용하지 않아야 한다. VHDL 구문상의 access type과 new operator 같은 문장은 제한해야 한다.
- 넷째, 시그널 전달(propagation) 패스들을 반드시 인과관계(causal)로 구성하여야 한다. 즉, 무한적인 발산(oscillation)들을 허용하지 않아야 한다.

위 4가지 제한 사항은 크게 유한적인 메모리 요구 측면과 유한적인 반응(reaction) 계산(computation) 분야로 정리할 수 있다. 이 4가지 제한 사항을 통해 전체의 VHDL 구문에서 위에 제약들에 문제되는 구문들을 배제하고 synchronous VHDL subset을 설정하도록 한다.

5. SpecCharts로부터 VHDL 코드 생성

5.1 SpecCharts와 VHDL 구문의 비교

3장에서 보는 바와 같이, PSM(Program State Machine)의 특성을 가지는 SpecChart의 특성은 VHDL의 기능적인 면을 다 표현할 수 있을 뿐만 아니라, 이를 graphical한 형태로도 표현할 수 있다는 점이다. <표 1>에서 정리된 내용을 볼 수 있다.

SpecCharts의 구조는 VHDL 구조와 거의 유사한 형태로 구성되어 있다. (그림 3)를 보면 한눈에 비교가 될 것이다. 일단 Entity 선언부에 사용되어지는 port statements는 VHDL 코드와 동일한 구조를 가진다. 또한 VHDL 코드형태와 마찬가지로 Entity block 다음에 architecture block이 구성되어 있다. architecture 구조는 architecture 내부에서 사용되는 signal과 하부 behavior 구조로 되어 있다. behavior의 type은 3장에서

보다시피 sequential, concurrent subbehavior 와 code 로 구성된다. (그림 2)에서 보다시피 계층성을 갖는 sequential과 concurrent 한 subbehavior 구조를 볼수 있다. 이는 모두 non-terminal state 들이다. SpecCharts 에서 실제 실행문(statements)들이 존재하는 부분이 terminal state인 code behavior 이다. 그러므로, VHDL 코드로 변환 시에 고려할 사항은 실제 code behavior 와 계층성을 가지고 있는 sequential, concurrent behavior에 대한 변환 부분이 중요하게 된다.

〈표 1〉 VHDL, StateCharts, SpecCharts 비교

	VHDL	Statecharts	SpecCharts
State Transitions	×	○	○
Behavioral Hierarchy	△	○	○
Concurrency	○	○	○
Program Constructs	○	×	○
Exceptions	×	○	○
Behavioral completions	○	×	○

○ : fully supported, △ : partially supported × : not supported

```
entity RootE is
        port (
                in_a : in bit ;
                in_b : in bit ;
                output : out bit          );
end;

architecture RootA of RootE is
begin
behavior Root type code is
begin
        output <= in_a and in_b;
end Root;
end RootA;
```

(그림 3) SpecCharts 코드 구조

5.2 SpecCharts로부터 VHDL로 변환

entity 구조의 코드 생성은 특별한 변환없이 파싱(parsing)을 통해 VDHL로 직접 변환이 가능하다. 이는 SpecCharts와 VHDL 공히 거의 유사한 구문 구조를 가지고 있기 때문이다. 파싱과정에 대해서는 7장에서 자세히 살펴보도록 한다.

architecture definition에 해당하는 부분은 entity definition과 동일하게 파싱과정을 거쳐 그대로 VHDL 구문으로 변환하면 된다.

architecture body에 선언되는 세 가지 behavior는 경우에 따라서 다르게 변환된다. 세 가지 behavior 타입 중 제일 간단한 형태는 code behaviors로 시스템 디자이너가 기술한 실제 statements들로 구성된다. 이들 statements 는 VHDL 구문 구조와 동일 한 구조를 가지고 있으므로 직접 변환이 가능하다. 그러므로 파싱과정을 통해 code behavior의 definition을 파싱하게 되면 실제 code behavior에 해당하는 process를 하나 생성하고 code behavior에 존재하는 statement를 process 구문안에 내용으로 변환시키면 code behavior의 코드 생성이 마무리 된다.

두 번째로 sequential subbehavior를 VHDL로 변환하는 경우이다. SpecCharts로 구성되어진 behavior의 구조는 다중 중첩된 계층 구조로 구성되어 있다. 이를 flat한 구조의 process 구조로 변환 시키는데 중점을 둔다. 물론 계층성을 유지하면서 VHDL 코드로 변환시킬수 있다. 이는 SpecSyn에서 제안된 방법으로 다중 block구조를 이용하면 계층성을 이용하여 VHDL 코드로 변환할 수 있다. 그러나 다중 block 구문을 생성한 경우 합성가능한 모델이라고 말할 수가 없기 때문에 본 논문에선 block구문을 통해 VHDL 코드로 변환하는 부분에 대하여는 배제하였다. 그러므로 계층적으로 표현된 SpecCharts의 구조를 flat한 형태로 펼치는 부분과 상태전이를 위한 TOC, TI 아크(arch)에 대한 고려를 하여야 한다.

```
process ( 동기 siganl )
  variable definition
begin
  if ( 동기 signal 검출 여부) then
    statements
  end if;
end process;
```

(그림 4) sequential subbehavior를 위한 process 코드 구조

먼저 SpecCharts의 구조를 flat한 VHDL 코드로 변환하는 과정에 대한 것은 Top-level로부터 파싱을 해나가면서 flat한 프로세스 구조로 생성을 해나가는 과정을 통해 해결할 수 있다. 계층구조를 파싱하면서 분석하고 각각의 behavior에 process를 매핑하는 구조로 생성해 나간다. TOC, TI arch의 경우는 process 구문

의 sensitivity signal list을 이용하여 동기 signal을 부여하여 동기적으로 수행될 수 있도록 구현한다. 그러므로 TOC, TI arch하나마다 상태전이를 위한 Sensitivity signal을 mapping하여 코드를 생성하면 된다.

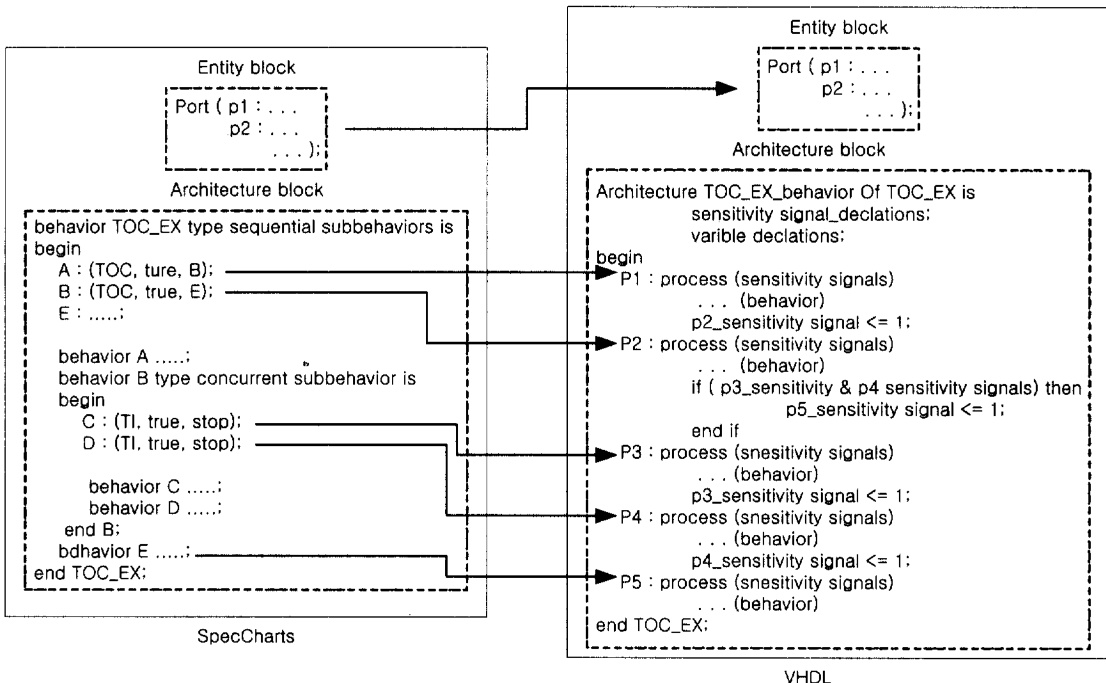
```
[ 호출한 프로세스 ]
P0 : process (동기1, 동기2 siganls )
  variable definition
begin
  statements
    if ( 동기1 & 동기2 신호 검출 여부) then
      activate next process;
    end if;
end process;
......

[ 병행적으로 수행되는 프로세스 P1, P2 ]
P1 : process ( 동기 siganl_2 )
  variable definition
begin
  statements
    if ( 동기 신호 signal_2 검출 여부) then
      behavior exit;
    end if;
end process;
```

```
P2 : process ( 동기 siganl_1 )
  variable definition
begin
  statements
  if ( 동기 신호 signal_1 검출 여부) then
    behavior exit;
  end if;
end process;
```

(그림 5) concurrent subbehavior를 위한 process 코드 구조

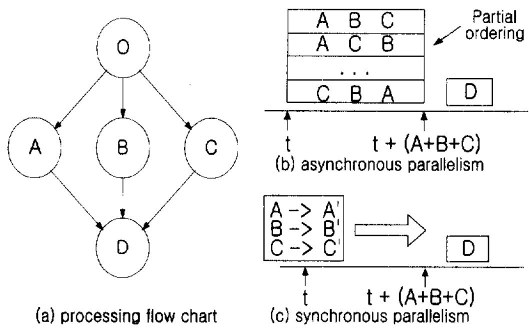
세 번째로 concurrent subbehavior를 VHDL로 변환하는 경우는 sequential subbehavior와 같이 process 두 개를 병행으로 수행되도록 sensitivity signal list를 이용하여 병행적으로 수행되게 하고, 이를 호출한 프로세서는 두 개의 병행적으로 수행된 프로세서들의 작업이 종료될 때 까지 기달렸다가 다음 수행할 프로세스에게 활성화 시키는 과정을 통해 병행성을 지원 할 수 있다. 또한 flat하게 펼치지 않고 중첩된 구조를 그대로 block 구문으로 매핑하는 방법을 통하여 구현할 수 있다. 이 때 주의할 점은 GUARD signal을 사용하지 않고 중첩구조를 구성해야 한다.



(그림 6) SpecCharts를 VHDL로 변환

6. 합성 가능한 synchronous VHDL로 변환기 설계

VHDL에서 지원하고 있는 병행처리 방법은 비동기적인 형태를 가진다. 그러하기 때문에 동기적 VHDL 코드를 구성하기 위해서는 VHDL의 discrete event 의미론을 만족할 수 있는 시뮬레이터를 구성해야 한다.



(그림 7) 비동기적과 동기적 병행처리 방법 비교

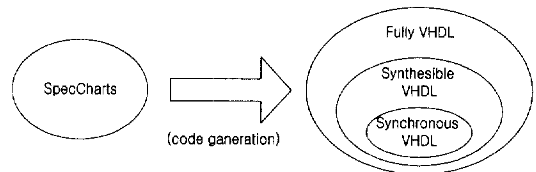
비동기적 병렬처리 방법론은 대기된 값들과 프로세스들의 리스트를 큐로 유지하여 순차적으로 처리해 나가는 방법을 사용한다. VHDL에서 병행처리는 비동기적 방법에 의해 처리되어 진다. 이는 event partial ordering을 통하지 않고서는 연계된 상태전이를 지원할 방법이 없기 때문이다.

동기적 병행처리 방법은 비결정성(non-determinism)을 허용하거나 제공하지 않는다. 이는 가능한 사건들이 단지 유한 개의 수이고 각 상태들은 단지 명시적인 지연만이 존재하고 모든 계산은 즉각적으로 수행된다는 동기적 가설(synchrony hypothesis)에 기반한 방법론이다.

SpecCharts는 시스템 모델링을 할 수 있도록 구성되어 있으며 또한 프로그래밍한 구조로 되어 있어 VHDL 코드로 쉽게 생성할 수 있다. 하지만, VHDL에서 지원하는 동기화 방식은 asynchronous 방식을 기본적으로 지원하므로 synchoronous 방식을 지원하는 subset으로 제한하여야만 실제로 synchronous VHDL 코드로 정의 될 수 있다. 이미 4장에서 살펴본 W. Baker 가 제안한 전체 VHDL set에서 4가지 제한 사항에 해당되는 구문들을 필터링 과정을 통해 VHDL 코드를 생성한다.

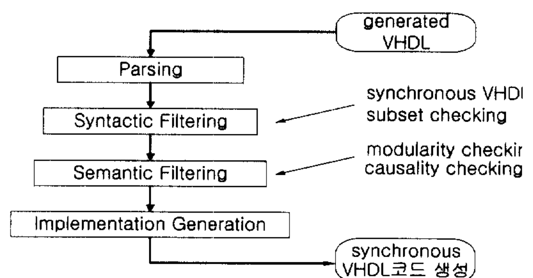
〈표 2〉 synchronous VHDL을 위한 제한사항

- Dynamic memory allocation
  access_type_definition
  : **ACCESS** subtype_indication;
  allocator
  : **NEW** subtype_indication
  | **NEW** qualified_expression;
- Manipulate the number of drivers attached to a signal
  disconnect_specification
  : **DISCONNECT** guarded_signal_specification
  AFTER expression ';';
- Transport delay
  _optional_transport
  : /* NULL */ | **TRANSPORT** ;
- Inertial and transport delay
  _optional_after_expression
  : /* NULL */ | **AFTER** expression
- Interface from the system to the OS
  file_declaration
  : **FILE** identifier ' : ' subtype_indication IS ... ;
  file_type_definition
  : **FILE OF** type_mark;
- Waveform assignment use transport model
  : waveform **WHEN** condition **ELSE** ...
- Unconstrained arrays
- Use of the timeout clause in the wait statement



(그림 8) SpecCharts로부터 VHDL 코드 생성

그럼 synchronous VHDL subset을 적용하기 위한 filtering 단계는 (그림 9)와 같다. 이러한 단계를 통해 생성한 VHDL을 효율적으로 동기적 VHDL 구문으로 변환시킬 수 있다. 또한 VHDL 코드가 자동 생성됨으



(그림 9) synchronous VHDL을 위한 filter 구성

로 원치 않는 코드 생성시에 해당 코드에 대한 검증작업이 필요로 하게 되기 때문에 중복적인 필터링을 수행한다.

7. lex & yacc을 이용한 토큰링 & 파싱

실제, 코드 생성을 위해 SpecCharts 코드 분석이 필요하다. 이를 위해 본 연구에서는 원시 코드의 Scanning 기능에는 Lex를 파싱 기능에는 yacc을 써서 구현하였다. Lex tool은 실제 파일로부터 입력받아 해당 지정된 토큰 정보에 따라 입력 받은 내용을 토큰링을 하여 yyparse()라는 함수를 통해 구문분석을 할 수 있는 yacc에게 yytext라는 배열을 이용하여 현재 토큰된 정보를 넘겨준다. 토큰된 정보를 받은 yacc는 주어진 구문정보에 대한 BNF 구조로 구성된 형식에 맞는지 검사하는 순으로 입력 스트림이 끝날 때 까지 반복 수행한다. 또한 lex와 yacc는 C 언어를 삽입하여 해당 부분에 대한 처리도 가능하다. 이러한 유연성 때문에, 컴파일러를 설계하는데 많이 쓰이는 도구이기도 하다. 또한 중요한 것은 코드 생성을 위해 파싱 단계에서 SpecCharts에서 기술된 상태 및 계층 정보에 대한 부분을 감지하여 이를 생성시 참조 할 수 있도록 해야 한다.

```
[\t ]+  { }
\n      { lineno++; }
"entity"{ return ENTITY; }
"is"    { return IS; }
"port"  { return PORT; }
"in"    { return IN; }
"out"   { return OUT; }
"bit"   { return BIT; }
"integer"       { return INTEGER; }
"signal"{ return SIGNAL; }
"variable"      { return VARIABLE; }
"end"   { return EEND; }
"architecture" { return ARCHITECTURE; }
"behavior"      { return BEHAVIOR; }
"type"  { return TYPE; }
"sequential"  { return SEQUENTIAL; }
"concurrent"  { return CONCURRENT; }
"subbehaviors" { return SUBBEHAVIORS; }
"code"  { return CODE; }
"of"    { return OF; }
"begin" { return BBEGIN; }
"and"   { return AND; }
"or"    { return OR; }
"<="    { return ASSIGN_S; }
":="    { return ASSIGN_V; }
[a-zA-Z][a-zA-Z0-9]* { return IDENT; }
[0-9]+  { return NUMBER; }
.       { return yytext[0]; }
```

(그림 10) 구문 scaning을 위한 basic lex 코드

(그림 10)과 (그림 11)에 나타난 것은 lex에 대한 토큰링할 수 있는 정보와 간단한 SpecCharts를 BNF표기법을 이용해 구성하여 파싱할 수 있도록 만든 코드이다.

```
mini_vhdl : entity_def architecture_def ;
entity_def :        ENTITY   entity_lable IS {
                    port_body
            EEND ';' ;
entity_lable : IDENT ;
port_body :          PORT '(' port_def')' ';' ;
port_def : port_def_e | port_def ';' port_def_e ;
port_def_e :
 | IDENT ' : ' direct_op data_type sigvar_init
            ;
direct_op :  IN | OUT ;
data_type : BIT | INTEGER ;
architecture_def : ARCHITECTURE architecture_label
            OF entity_lable IS
            BBEGIN behavior_def behavior_part
            EEND architecture_label ';' ;
architecture_label : IDENT ;
behavior_def :
            | behavior_def_e
            | behavior_def_e ';' behavior_def_e ;
behavior_def_e :
          | IDENT ' : ' direct_op data_type sigvar_init ;
behavior_part : BEHAVIOR behavior_label
            TYPE behavior_type IS data_init_part
            BBEGIN statement_part
            EEND behavior_label ';' ;
behavior_label : IDENT ;
behavior_type : type_seqcon SUBBEHAVIORS
            | CODE ;
data_init_part : init_e | init_e ';' init_e ;
init_e : type_sigvar IDENT ' : ' data_type sigvar_init ;
type_sigvar : SIGNAL | VARIABLE ;
sigvar_init : | ASSIGN_V NUMBER ;
type_seqcon : SEQUENTIAL | CONCURRENT ;
statement_part : statement
         | statement_part ';' statement ;
statement : | assign_st;
assign_st : IDENT ASSIGN_S exp ;
exp : exp '+' term | exp '-' term | term ;
term : term AND factor | term OR facter | factor ;
factor : IDENT | NUMBER | '(' exp ')' ;
```

(그림 11) 구문 parsing을 위한 간단한 yacc 코드

8. 결론 및 향후 연구 방향

본 논문에서는 내장형 시스템을 설계하고 구현하는데 방법론으로 이용되는 VP를 고려하여 HW component를 구현하기 위하여 검증이 용이하고 동기적인 VHDL 코드를 생성하는 방법에 대해 살펴보았다. 이를

위해 시스템 명세 언어 중 SpecCharts로부터 합성 가능한 동기적 VHDL 코드를 생성하는 코드 생성기를 제안하였다. 그리하여 시스템 디자이너가 SpecCharts 에디터인 Xspecchart editor를 통해 SpecCharts로 시스템을 명세하면 그 동기적 의미론을 만족하는 합성가능한 VHDL 코드가 생성된다. 이러한 검증을 용이하게 하는 동기적 의미론을 만족하는 VHDL 코드를 생성하기 위하여 주어진 VHDL 코드에서 synchronous에 위배되는 구문을 배제하고 syntactic filtering과 semantic filtering 과정을 통해 동기적 VHDL 코드를 생성하였다.

전체적인 코드 생성기 구조는 최초의 SpecCharts로부터 lex & yacc 도구를 이용한 SpecCharts의 스캐닝과 구문 분석을 통해 control flow를 분석하고 이를 바탕으로 SpecCharts가 가지고 있는 계층성과 세가지 행위 모델(sequential, concurrent subbehavior, code behavior)들을 구현하기 위한 각 상태에 대한 sensitivity signal list들을 갖는 합성가능한 형태의 flat한 process 구조로 변환하였다. 이어서 VHDL의 동기적 성질을 위배하는 구문구조를 제약시켜 동기적 VHDL 코드로 변환하고 다시 동기성 위배 여부 및 인과성, modularity를 재검사하여 동기적 변환을 한층 강화하도록 하였다.

이렇게 하여 정형화된 형태로 HW component를 생성하는 코드 생성기를 구현함으로서 시스템 명세 언어인SpecCharts를 통해 검증이 용이한 동기적 모델의 시스템을 설계하고 구현하는데 보다 더 효율적인 공정을 제공하게 한다. 이는 VP가 제안하고자 하는 목적에 부합되고, 하드웨어 구성 면에서도 합성 가능한 VHDL 코드 생성 과정의 자동화로 시스템을 설계부터 실제 합성까지의 공정을 빠르게 하여 제품의 시장 진입 시간을 단축시킬 수 있다는 장점을 가지게 된다.

하지만 Wendell Baker가 주장했던 synchronous VHDL을 통한 자동 코드 생성기는 존재하지 않고 추상적인 시뮬레이터만 존재하는 현실이므로 기존의 sc2vhdl 생성기와의 비교 분석적 관점에서 보기보다는 기존의 VHDL 코드를 생성하는 목적에서 발전된 형태로 원시 코드가 가지는 동기적 의미론에 부합되는 동기적 VHDL 코드를 생성하는데 의의를 둔다.

또한 현재 SpecCharts로 구성된 코드 전체가 변환되는 합성 가능한 부분에 대한 연구는 본 논문에서 제한적인 상황을 두어 살펴보았다. 이는 의미론에 부합하는 합성에 대한 VHDL 코드 구조의 유연성이 부족한 시점에서 전체 경우에 대한 코드 생성기를 표현할 수 없는 문제점을 가지고 있기 때문에 process 구조로 표현될 수 있는 형태의 구문을 변경 가능한 형태의 코드 변환을 제안하였다. 이를 통해 생성된 코드는 부분적 합성만 지원한다고 말할 수 있지만 전체 합성 코드의 아주 제한적인 일부분이라고 말할 수 있다. 하지만 전체 합성 코드를 위한 방법론을 제시하기 위해서는 합성 가능한 VHDL 코드 모두가 동기적 성질을 만족하는 코드이다고 말할 수 없기 때문에 전체 합성의 제한을 두게 되었다. 동기성을 부여할 수 있는 부분적인 합성을 위해 SpecCharts의 3가지 behavior들을 각각 경우에 맞는 process 구조로 변화시키는 합성 방안을 제안하였다.

향후 연구 방향으로는 시스템 디자이너가 설계한 부분에 대한 자동 오류 검정 기능 및 설계시 필요한 시스템 자원에 대한 자동 트래킹(tracking) 기능과 VHDL 코드 생성시에 flat한 구조가 아닌 계층성을 이용한 구조와의 비교 성능 분석을 통해 최적의 코드 생성 기법 검증에 대한 연구를 진행되어야 할 것이다.

참 고 문 헌

[1] D. D. Gajski, F. Vahid, S. Narayan and J. Gong, Specification and Design of Embedded Systems, Prentice Hall, 1994.

[2] Frank Vahid, Sanjiv Narayan, Daniel D. Gajski, "SpecCharts : A VHDL Front-End for Embedded Systems," IEEE Transactions on computer-aided design of integrated circuits and systems. Vol.14. No.6. June 1995.

[3] S. Narayan, F. Vahid, D. D. Gajski, "Modeling with SpecCharts," Technical Report, 1992.

[4] W. C. Baker, An Application of a Synchronous / Reactive Semantics to the VHDL Language, M.S. Report, Department of EECS, University of California, Berkeley, January 1993, UCB/ERL M93/10.

[5] Wendell C. Baker, Richard Newton, "The Maximal VHDL Subset with a Cycle-Level Abstraction," EURO-DAC, 1996.

[6] IEEE Standartd VHDL Language Reference Maual, IEEE, 1987, 345 East 47th., New York, NY 10017 USA. Std 1076-1987.

[7] IEEE Standartd VHDL Language Reference Maual, IEEE, 1993, Std 1076-1993.

[8] D. Harel, "STATECHARTS : a Visual Formalism for Complex Systems," In Science of Computer Programming Vol.8, north Holland, 1987.

[9] John R. Levine, Tony Mason and Doung Brown, lex & yacc, O'REILLY, 1995.

[10] Yanbing Li, Miriam Leeser, "HML, a Novel Hardware Description Language and Its Translation to VHDL," IEEE Transactions on very large scale integration(VISI) systems. Vol.8. No.1. February 2000

윤 성 조

e-mail : sjyun@rain.chosun.ac.kr

1999년 조선대학교 정보과학대학 전자계산학과 졸업(학사)

1999년~현재 조선대학교 대학원 전자계산학과(석사)

관심분야 : Embedded System, RT-OS, Linux

최 진 영

e-mail : choi@formal.korea.ac.kr

1982년 서울대학교 컴퓨터공학과 졸업

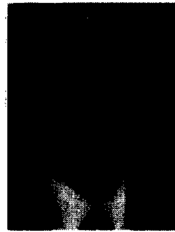
1986년 Drexel University Dept. of Mathematics and Computer Science 석사

1993년 University of Pennsylvania Dept. of Computer and Information Science 박사

1993년~1996년 Research associate, University of Pennsylvania

1994년~현재 고려대학교 컴퓨터학과 부교수

관심분야 : 컴퓨터이론, 정형기법(정형명세, 정형검증), 실시간 시스템, 분산프로그래밍 언어, 소프트웨어 공학



한 상 용

e-mail : hansy@cau.ac.kr

1975년 서울대학교 공과 대학 졸업(학사)

1977년~1978년 KIST 연구원

1984년 미네소타 대학교 컴퓨터 공학과(공학박사)

1984년~1994년 IBM 연구소 연구원

1995년~현재 중앙대학교 컴퓨터공학과 교수

관심분야 : CAD, Virtual Prototyping, 망 접속

이 정 아

e-mail : jeong@rani.chosun.ac.kr

1982년 서울대학교 컴퓨터공학과(학사)

1985년 미국 인디애나주립대학 컴퓨터학과(석사)

1990년 미국UCLA 컴퓨터공학과(박사)

1990년~1995년 미국 휴스턴주립대학 전기전산공학과(조교수)

1993년~1994년 미국 국립초전도가속기연구소(객원연구원)

1995년~현재 조선대학교 컴퓨터공학부(교수)

관심분야 : Computer Arithmetic, Application-Specific Processor Design, (Re)Configurable Computing