論文 97-22-6-24

실행시간 예측가능한 실시간 메카니즘 제어언어의 구현기법

正會員 白 貞 鉉*, 元 裕 憲**

Implementation Technique of Execution Time Predictable Real-Time Mechanism Control Language

Jeong-Hyun Baek*, Yoo-Hun Won** *Regular Members*

요 약

본 논문에서는 실시간 메카니즘(mechanism) 제어 언어를 설계하고 그의 구현 기법을 제안 하였다. 프로그램형 제어기(PC), 수치제어기(NC), 분산 제어시스템(DCS) 및 로봇(Robots) 제어기와 같은 프로그램가능한 제어기들의 실시간 메카니즘 제어 프로그램은 수 백에서 수 천개에 이르는 센서(sensor)나 스위치에서 발생되는 전기적인 신호를 실시간에 동시 처리해야 하기 때문에 기존의 프로그래밍언어나 운영체제에서는 효율적인 처리가 불가능하였다. 그러므로 프로그래밍언어의 조건문, 반복문, 선택문과 같은 구문 구조에 시간제한 구조와 메카니즘 동기화 구조를 추가하고, 실행시간 예측기법을 개발함으로서 언어의 번역시간에 실시간 응용프로그램의 시간제한 구문의 타당성과 전체 프로그램의 실행가능성을 예측할 수 있도록 하였다.

또한, 기존의 운영체제는 프로그램의 크기가 방대하고 시스템운영으로 인한 과부하(overload)가 크기 때문에 내장형시스템의 환경에서 효율적으로 실행될수 없었다. 그러므로 주기 및 비주기 실시간 병행 태스크를 효율적으로 처리할 수 있도록 반복 실행(cyclic executive) 기법에 의하여 인터프리터를 설계함으로서 효율적으로 실행될 수 있도록 구현하였다.

ABSTRACT

In this paper, We designed real time mechanism control language and proposed execution time analysis technique. It was impossible to handle real-time mechanism control programs like programmable controller, numerical controller, distributed control system and robot controller with general purpose programming languages and operating

*중경공업전문대학 전산과 조교수
**홍익대학교 컴퓨터공학과 교수
論文番號:96407-1227
接受日字:1996年 12月 27日

systems because they have to process electric signals generated by thousands of sensors at the same time and in real time. So We made it possible to predict plausibility of time constraint constructs of a real time application program at compilation time by adding time constraint constructs and mechanism synchronization structure to conditional statement and iteration statement of a programming language and developing execution time analysis technique.

Ⅰ. 서 론

마이크로프로세서 제조 기술의 발전이 모든 산업분야의 자동화를 가속화하고 있다. 이들 마이크로 프로세서의 사용 형태는 범용 컴퓨터의 중앙처리 장치로 사용되기보다 대부분이 자동화기기와 같은 내장형 시스템들의 핵심 제어기로 내장되어 사용되고 있다. 이와 같은 내장형 시스템(embedded system)은 가정용 세탁기, 에어콘과 같은 전용 제어기에서 산업용 로봇이나 프로그램형제어기(PC : Programmable Controller), 분산 공정제어 시스템(DCS : Distributed Control System), 수치제어기(NC : Numerical Controller)등과 같은 프로그램형 범용 제어기에 이르기까지 점차 그 영역이 확대되고 있다.

이러한 내장형 시스템의 실시간 메카니즘제어 언어의 특징은 뛰어난 실시간 및 병행처리 기능을 요구하기 때문에 기존의 Fortran, Pascal, C 언어와 같은 순차적 프로그래밍을 지향하는 프로그래밍 언어는 실시간 병행처리 프로그램 작성시 운영체제나 병행처리 라이브러리에 의존함으로서 효율적인 프로그래밍이 불가능하였다. 그러므로, 80년대 초반부터 Modula, Ada, Occam등과 같은 실시간 병행처리 기능을 갖춘 프로그래밍 언어들이 개발되었으나, 미국 국방성에서 지원하는 Ada를 제외하면 거의 사용되지 않았다[2].

여러면에서 우수한 평가를 받는 Modula나 Ada, Occam과 같은 실시간 프로그래밍 언어가 폭넓게 사용되지 않는 이유는 첫째, Assembly나 C 언어에 익숙한 프로그래머들이 Modula나 Ada에 대해 거부감이 많았으며, 둘째 Modula나 Ada, Occam등이 프로세스의 병행성 표현 능력은 뛰어 나지만 실시간 표현 능력이 미흡 하였기 때문에 병행처리보다 실시간 처리를 중요시 했던 실시간 시스템의 개발자들에 의해서 외면되었고, 셋째, 이들 언어들이 운영체제의 기능과 깊숙이 의존된 함수나 라이브러리를 이용한 프리프로세서 형태로 구현되었기 때문에 실시간 및 병행처리 기능이 저하되었고, 프로그램의 실행시간을 정확히 예측할 수 없었기 때문이다[2].

따라서, 본 연구에서 제안하는 실시간 메카니즘 제어언어 RTL/MCS(Real-Time Language for Mechanism Control Systems)는 Modula나 Ada와 같은 기존의 우수한 실시간 프로그래밍언어들의 가장 큰 단점인 실시간 표현능력을 고급 개념의 시간제한구조(time constraint construct)[4, 5, 9]를 갖는 프로그래밍 문장 구조들을 설계하여 추가시킴으로서 실시간 시스템의 설계시에 시간제한 표현이 자연스럽고, 시간 제한과 관련된 실행과정을 가시적으로 나타내어 시간 측정을 위한 디버깅 과정이 필요없이 효율적으로 실시간 소프트웨어의 개발이 가능하도록 설계하였다.

또한 시간제한구문의 실행시간 분석 기법을 개발하여 실시간 프로그램의 시간제한구문들이 프로그래머가 명시한 시간 제한조건을 만족하는지를 분석하여 하드웨어적 실행과정 없이 프로그램의 안정성과 실행가능성을 예측할 수 있도록 하였다.

Ⅱ. RTL/MCS

2.1 RTL/MCS의 특성

2.1.1 Cycle Block

Cycle은 제어공정상 하나의 독립된 기능이나 상황과 관련된 제어문장들의 그룹으로 구성된다. 즉, 각 cycle들은 완전히 독립되어야 하며, 하나의 cycle 안의 제어문장들은 해당 cycle의 기능과 관련되어 의존성을 가질수 있다.

그러므로, 여러개의 cycle들이 모여서 하나의 RTL/MCS 프로그램을 이루며, 이때 시스템의 제어 공정에 따라서 각 cycle들이 활성화(active)된다. 또한 활성화된 cycle들은 cycle 리스트의 맨 처음(top)에서 시작하여 끝(bottom)까지 순차적으로 실행하며, cycle 리스트 전체를 한번 실행한 시간을 scan주기(scantime)라 하며, 프로그램이 멈출때(시스템이 정지할때)까지 활

성화된 cycle 리스트 전체가 반복실행된다.[6]

2.1.2 Cycle 상태

Cycle 블록은 운영체제의 프로세스(process)와 같이 실행 상태를 전이한다. 대기상태에있는 비활성화(inactive)된 cycle 리스트들은 프로그램의 start나 call 문장에 의하여 실행상태로 활성화(active) 된다. 활성화된 cycle 리스트는 동시에 여러개가 존재하여 반복실행되므로서 병행성을 갖게된다. 그러나 활성화된 cycle들 중에서 wait, delay, call, when, every와 같이 조건이나 시간제한을 기다리는 cycle들은 중단상태(suspend)로 전이하여 조건이 만족될 때까지 실행이 중지된다. 그러나 각 scan 주기마다 이들의 조건이 검사되어 조건이 만족되면 다시 활성화 상태로 전이되어 반복실행되고 cycle의 실행이 완료되거나 return, exit문에 의하여 반환이 이루어지면 cycle은 대기상태(Inactive)로 된다.

그러므로 RTL/MCS의 cycle들은 반드시 프로그램 문장에의해서 명시적으로 상태전이가 이루어지며, 이는 실시간언어의 기본 조건인 프로그램의 실행상태를 정확히 분석 및 예측할 수 있도록 해준다. 그림 1은 cycle들의 상태전이 조건과 과정들을 나타내준다.

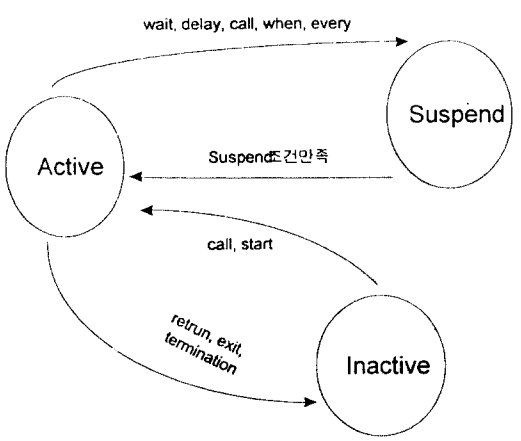


그림 1. Cycle의 상태전이

2.2 RTL/MCS의 구문정의

실시간 메카니즘제어 언어인 RTL/MCS는 C 언어의 기본 구조에 실시간 프로그래밍언어의 필수 조건인 시간제한 구조[4, 5]와 실행시간 예측에 필요한 경험적인 정보를 제공하기위한 문장구조[3, 8]들을 포함하고 있다. 이들 문장구조는 시간지연구조, 시간제한구조, 그리고 cycle을 제어하기위한 문장들로 구성된다. 그림 2에 RTL/MCS의 주요 문장구조의 EBNF[1]를 나타내었다.

2.2.1 시간 제한 구문

①delay:시간지연문(delay)은 워드수식(word_expression)으로 지정한 시간동안 cycle의 실행을 중지시킨다. 또한 delay until문은 워드수식으로 주어진 시간(calendar clock)까지 cycle의 실행을 중지 시킨다. 지연시간의 오차는 프로그램의 전체 scan 시간에 비례하여 커진다.

②within:시간제한(within)문은 문장들이 실행 되어야할 마감시간(deadline)을 지정한다. Timeout절은 문장들의 실행시간이 제한 시간을 초과하게 되면 즉시 실행된다.

③every:주기적 반복문(every)은 지정된 주기에 따라서 문장들이 반복적으로 실행된다. 즉, 워드수식으로 지정된 시간만큼 주기적으로 문장들이 실행되며, 이때의 주기는 문장들의 실행시간과는 관계없이 절대적인 시간 간격을 의미하며, 실행중인 cycle은 워드수식으로 지정된 시간이 경과 할때 까지 중지(suspend)된다.

2.2.2 메카니즘 동기화 구조

①wait:대기(wait)문은 비트수식(bit_expression)으로 주어진 상태가 참이 될때까지 cycle의 실행을 중지(suspend) 시킨다. 이때 중단된 cycle은 각각의 scan cycle마다 중단조건이 평가되어 결과가 참일때 cycle은 활성화되어 다음 스캔주기에 중단된 다음 문장부터 실행이 재개된다.

②condition:예외처리문(condition)은 비트수식이 참일때 실행된다. 비트수식은 매 scan 주기 마다 do 문의 문장들이 실행되기 전에 평가되고, 또한 cycle 실행을 중단 시키는 어떠한 문장을 포함하고 있을때 평가된다. 만일 비트수식의 평가 결과가 거짓이면 on exception 절이 실행된후 do문 이후의 문장들이 실행된다.

③when:When문은 조건문으로서 비트수식이 참일 경우 대응되는 문장을 실행한후 when문을 벗어난다. 그러나 조건절의 값이 모두 거짓이면 해당 cycle은 실

행이 중단되어 조건이 하나라도 만족되면 실행을 재개한다. 그러므로 when문은 Ada의 select문과 같이 사용되어 cycle들의 랑데뷰를 가능하게 해준다.

2.2.3 Cycle 제어문

①start: Start문은 해당 cycle을 활성화(active)시켜 실행큐(queue)에 추가하므로서 다음 scan 주기부터 실행된다. 활성화된 cycle 리스트들은 순차적으로 실행(scan)되기 때문에 활성화 시킨 cycle은 다음 sacn 주기에 실행된다. 또한 start문을 포함하는 기존의 cycle도 suspend 되지 않는다.

②call: 호출(call)문은 호출된 cycle을 활성화 시킨다. 동시에 호출한 cycle은 호출당한 cycle이 종료될때까지 실행이 중지되며, 호출당한 cycle이 종료되어 호출한 cycle의 실행이 재개될때에는 call 다음 문장부터 실행된다.

③exit: 종료(exit)문은 현재 실행중인 cycle을 종료시킨다. 만일 cycle이 call이나 start문에 의하여 다시 활성화되면 cycle은 exit 다음 문장부터 실행을 재개한다.

④return: Return 문은 현재 실행중인 cycle을 비활성화(inactive) 시킨다. 현재 실행중인 위치를 기억하지 않기 때문에 call이나 start문에 의하여 cycle이 다시 실행될때 cycle의 첫번째 문장부터 실행이 재개 된다.

```
delay_statement ::= DELAY '(' word_expression ')' ( MSEC | SEC | MIN )
                  | DELAY UNITIL word_expression ':' word expression
within ::= WITHIN word_expression ( MSEC | SEC | MIN ) [ ON TIMEOUT
           statement list ] DO statement_list
every ::= EVERY word expression ( MSEC | SEC | MIN ) statement list
        | EVERY word expression ( MSEC | SEC | MIN )
          WHILE '(' bit_expression ')' statement list
wait statement ::= WAIT '(' bit expression ')'
                   [ TIMEOUT number (MSEC | SEC | MIN) ]
condition_do ::= CONDITION '(' bit expression ')' ON EXCEPTION
                 statement_list DO statement list
when_statement ::= WHEN bit_expression ':' statement_list
                   ( OR bit_expression ':' statement list )
                   TIMEOUT word expression;
start_statemnet ::= START cycle name
exit_statement ::= EXIT
return_statement ::= RETURN
if_statement ::= IF '(' bit expression ')' statement_list
                 { elsif '(' bit_expression ')' statement_list }
while statement ::= WHILE '(' bit_expression ')' [TIMEOUT number
                  | COUNTOUT number] statement list
forever ::= FOREVER statement_list
set_statement ::= SET '(' bit_term { ',' bit_term } ')' ( ON | OFF | INV )
                | SET ')' bit_term { ',' bit_term } ')' TO bit_expression
let_statement ::= [ LET ] word_term '=' word_expression
word_expression ::= word_expression ( '+' | '-' | '*' | '/' | '%' )
                    word_expression | word_term
word_term ::= word_name | number | word_function
bit_expression ::= bit expression ( '&' | '|' | '!=' | '^' )
                   bit_expression | bit_term
bit_term ::= bit_name | bit_relation | bit_selector | cycle_name
           | bit_vector | bit_function
bit_relation ::= word_expression ('<' | '>' | '<=' | '>=' | '==' | '<>' )
                 word expression
statement_list ::= '{' { statement ';' } '}'
cycle ::= cycle_name '(' [ scan_period ] ')' statement_list
program ::= main_function { cycle }
```

그림 2. RTL/MCS 구문의 EBNF

2.3 예제 프로그램

다음의 프로그램은 본 논문에서 제안한 RTL/MCS의 프로그래밍 기법과 문장들의 기술방법을 설명하기 위한 캔음료를 생산하는 자동화라인을 제어하는 예제이다. 그림 3과 같이 컨베이어에 의해서 이동되는 캔에 position1에서 음료수를 주입하고 position2에서 스탬퍼를 이용하여 마개를 밀봉하는 작업을 한다.

그림 4의 프로그램에서 main()은 제어기의 전원스위치에 의해서 한번만 활성화되는 주함수이며, 변수 앞의 '+'와 '-' 기호는 센서의 on/off 상태를 나타낸다. 프로그램에서 사용되는 모든 변수들의 타입(type)과 선언은 제어기의 입출력 주소와 매핑되어야 하기 때문에 제어대상에 따라서 제어기의 환경설정 도구(configuration tool)에 의해서 생성되어 데이터화일로 컴파일러에 제공된다.

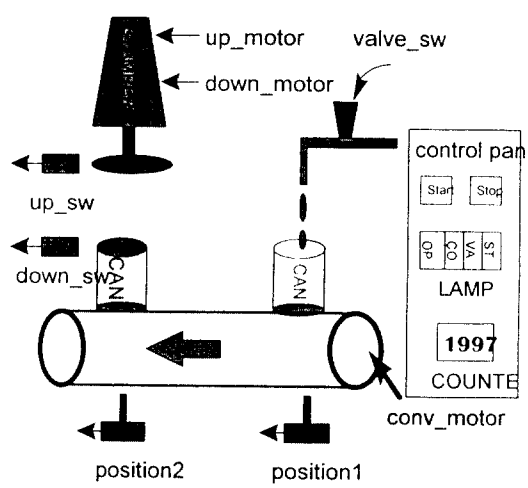


그림 3. 캔음료 생산공정

```
main()
{  counter = 0;
   wait(+start);
   if(+start) {
     wait(-start);
     set(operation) on;
   }
   set(up_motor) on;
   wait(+up_sw);
   start conveyer();
   start injector();
   start stamper();
   start monitor();
}
cycle conveyer()
{  while(+operation)
   {   set(full_can) off;
       set(conv_motor) on;
     wait(+position1);
     set(conv_motor) off;
     wait(+full_can);
     set(conv_motor) on;
     wait(+position2);
     set(conv_motor) off;
     wait(+down_sw);
   }
}
cycle injector()
{  while(+operation)
   {  wait(+position1);
      set(valve_sw) on;
      delay( 5) sec;
      set(valve_sw) off;
      set(full_can) on;
      wait(-position1);
   }
}
cycle stamper()
{  while(+operation)
   {  wait(+position2);
      set(down_motor) on;
      wait(+down_sw);
      set(up_motor) on;
      counter = counter +1;
      wait(-position2);
   }
}
cycle monitor()
{  every 500 msec while(+operation)
   { if(+stop)
        set(openation, op_lamp) off;
     set(co_lamp) to conv_motor;
     set(va_lamp) to valve_sw;
     set(st_lamp) to position2;
     counter_led = BCD(counter);
   }
}
```

그림 4. RTL/MCS 예제 프로그램

Ⅲ. RTL/MCS의 의미분석

이절에서는 RTL/MCS의 주요 구문들에 관한 의미분석과 중간코드로의 번역규칙을 기술한다.

3.1 within

within문을 구현하기 위해서는 실시간 타이머를 이용해야 하며, RTL/MCS의 중간코드에는 타이머를 설정하고 제거하기 위한 SetTimer와 DeleteTimer가 있다. SetTimer는 제한시간과 timeout handler를 설정할 수 있다. within문을 중간코드로 표현할 때는 구문을 시작하면서 타이머를 설정하고 구문을 마치면서 타이머를 소거함으로써 구현할 수 있다. 다음은 within문에 대한 중간코드의 번역과정이다.

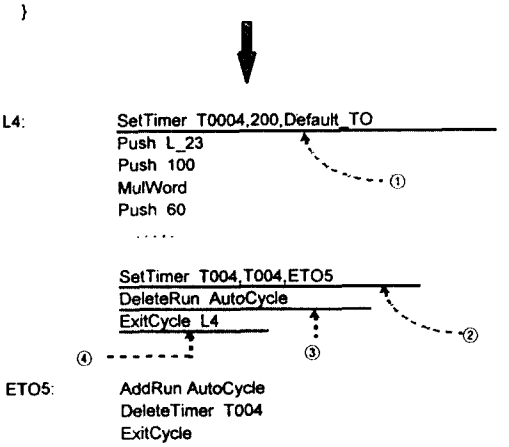
중간코드 SetTimer T002, 100, LT2에서 T002는 실시간 Timer의 ID이고 100은 제한시간, LT2는 timeout handler 루틴의 레이블이다. 그러므로 timeout이 발생하면 레이블 LT2 이하에 있는 중간코드들이 실행된다.

```
within 100 ms                          SetTimer T002,100,
on timeout                       LT2:  Jump L17
{ TimeOutCmd = DefaultTimeOutCmd;      Push L_47
  .....                                Pop R-reg
  .....                                Mov 0xEF04
}                                      Push 0
do {                                     .....
  B1Status = DefualtData;              Delete T002
  .....                                Jump L18
  .....                          L17:  Push L_19
}                                      Pop R_reg
                                         .....
                                       Delete T002
                                 L18:    .....
```

3.2 every

Every문은 설정된 시간의 주기를 가지고 반복적으로 cycle블럭을 실행한다. 따라서, every문 역시 Timer를 사용하는 중간코드를 이용하여 구현될 수 있다. 다음은 every의 중간코드번역 과정이다.

```
every 200 ms {
  D2OutData = D1InData /60 + temper;
  if ( D2OutData > D2OutData ) {
                . . .  . . .
  }
```

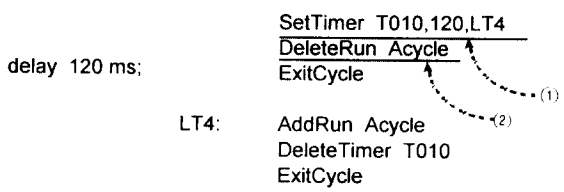


①의 SetTimer는 제한시간을 every문의 실행주기로 설정한다. 실행주기 내에 every문의 실행을 마치지 못하면 자동적으로 defualt timeout handler가 호출되어 프로그램의 실행이 정지된다. ②의 SetTimer는 every문이 설정한 주기의 간격으로 수행될 수 있도록 현재 실시간 타이머에 남아있는 값을 재설정(reload) 설정

하고 timeout handler에서 cycle을 기억된 위치에서부터 다시 실행할 수 있도록 한다. 그리고, ③의 DeleteRun을 이용하여 every문이 속한 cycle을 Ready 리스트에서 제거한다. ④의 ExitCycle은 다음번에 cycle을 실행할 때 시작위치를 every문의 시작점으로 설정하게 된다. ②의 timeout handler(ET05)는 중지된(suspend) cycle을 설정주기가 경과한후 활성화큐에 추가하여 cycle이 다시 실행될 수 있도록 한다.

3.3 delay

Delay문은 일정시간 동안 cycle의 실행을 중지시켜 cycle의 상태전이를 일으킨다. delay문의 구현도 실시간 타이머를 이용한다. 다음은 delay문의 중간코드 번역 과정이다.



위의 중간코드 번역예에서 ①의 SetTimer는 지정한 시간으로 Timer를 설정하고 ②의 DeleteRun이 Cycle을 활성화큐로부터 제거한다. ②의 timeout handler인 LT4는 다시 ACycle을 활성화큐에 추가하고 Timer를 제거한다.

Ⅳ. RTL/MCS의 실행시간 분석

4.1 RTL/MCS의 실행시간 분석기법

RTL/MCS의 각 문장 요소들에 대한 실행시간을 MET(Maximum Execution Time)라고 정의할때 RTL/MCS의 프로그램에 대한 최악 실행시간 WCET (Worset Case Execution Time)의 분석은 표 1에서 기술한 방법으로 계산할 수 있다.

표 1에서 보는바와 같이 단순한 문장들의 나열인 문장시퀀스는 각 문장들의 실행시간의 합이 된다. if문의 경우에는 조건을 만족하여 실행하는 then부분과 만족하지 않아 실행하는 else부분의 실행시간중 가장 큰 것과 조건 검사하는데 걸리는 시간의 합이 if문의 최악 실행시간이 된다. while문은 조건 검사시간과 몸체(body)의 실행시간의 합에 프로그래머가 지정한 경계(bound) 값을 곱한것과 while문을 빠져나올 때의 조건 검사시간을 더한 것이 while문의 실행시간이 된다.

본 논문에서는 Unix의 컴파일러 생성도구인 yacc를 이용하여 구문을 분석[1]하고 중간코드를 생성하면서 시간스택(timing stack)을 생성하여 문장들의 최악실행 시간과 제어프로그램의 scantime을 분석하였다. 4.2절에서 부터 시간스택을 이용하여 실행시간을 분석하는 과정을 조건문과 반복문 그리고 scantime 분석에 대하여 도식적으로 상세히 기술한다.

4.2 if문의 실행시간분석

본 논문의 실행 시간분석기 구현은 실행시간 스택

표 1. RTL/MCS 문장의 최악 실행시간 계산기법

구문	최악실행시간(WCET)
문장시퀀스	Σ MET(문장)i-1,n
if문	MET(조건절) + max(MET(then)i-1,n, MET(elif)i-1,n)
while문	MET(조건절) + LoopBound*(MET(body) + MET(조건절))
within	if(설정시간 > MET(body)) then WCET = MET(body) else WCET = 설정시간
every	if(설정주기 > MET(body)) then WCET = MET(body) else WCET = 설정주기
when	if(max(MET(조건절)i-1,n) > timeout) then WCET= timeout else WCET = max(MET(조건절)i-1,n)
condition	MET(조건절) + max(MET(on exception), MET(do의 body))

(timing stack)을 생성하여 문장들에 대한 중간코드생성시 각각의 코드에대한 실행시간을 중간코드 실행시간 테이블로부터 읽어 스택에 저장하고 합산하는 방법으로 구한다. 이때 적용되는 계산방법은 표 1에서 기술한 기법에의한다.

if문의 실행시간 분석은 문장을 parsing하기 전에 stack에 시간을 저장하기 위한 templete을 그림 5의 ①과 같이 push한다. 이 templete에 조건검사를 위해 필요한 문장들의 실행시간을 저장한다. 그리고, then부분을 parsing하기 전에 그림 5의 ②처럼 templete을 push한다. 이 templete에 then부분 문장들의 실행시간을 저장한다. 마찬가지로 else부분을 parsing하기전에 그림 5의 ③과 같이 templete을 push하고 이 templete에 else부분의 실행시간을 기록한다.

if문의 parsing이 끝나는 부분에서 그림 5의 ④와 같이 then부분의 templete과 else부분의 templete을 꺼내 큰 곳을 조건검사의 templete과 더해서 기존의 맨 위에 쌓여있던 templete에 더한다.

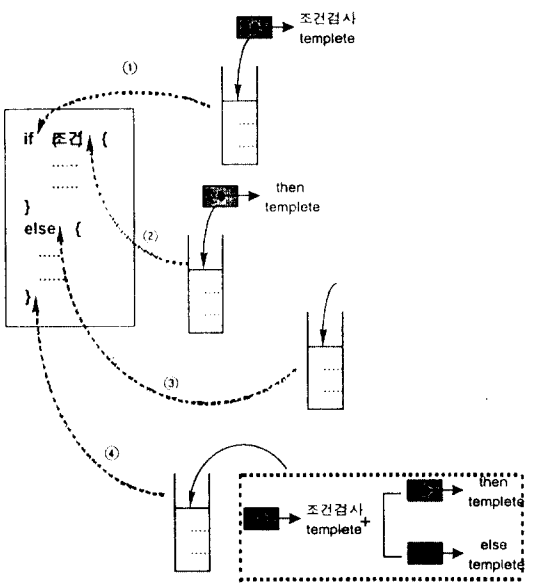


그림 5. if문의 실행시간 분석과정

4.3 while문의 실행시간분석

while문의 실행시간분석을 위한 시간스택의 운영방법은 다음과 같다. 그림 6의 ①에서 처럼 조건검사를 위한 문장들의 실행시간을 저장하기 위한 templete을 push하고 ②에서 몸체부분의 실행시간을 저장하기 위한 templete을 push한다.

while문이 끝나는 부분에서 조건검사의 templete과 몸체부분의 templete을 stack에서 꺼내서 두 templete에 while문의 bound를 곱한 것과 조건검사의 templete의 값을 더해 stack의 맨 위의 templete에 더한다.

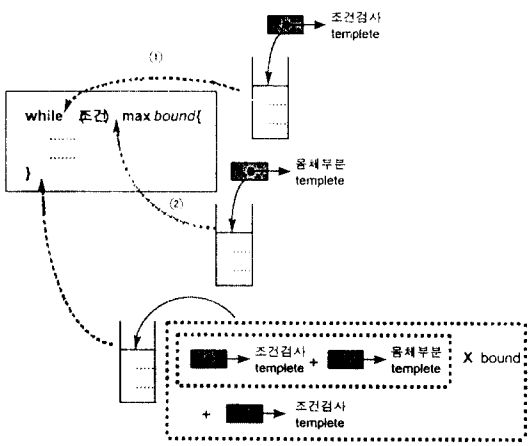


그림 6. while문의 실행시간 분석과정

4.4 스캔시간(Scantime) 분석

RTL/MCS로 작성된 실시간제어 프로그램의 실행가능성 분석은 작성된 응용프로그램이 허용하는 최대 시간오차인 Scantime을 프로그래머가 지정한 제한시간내에서 실행가능한지 분석하여 예측할 수 있다. 이는 각 scan주기에서 동시에 활성화(active)된 cycle들의 실행시간을 모두 더한것과 같다.

그러나 Scantime의 분석은 각 cycle문장안에 every, wait, delay, when등의 문장들이 cycle을 실행중단(suspend) 상태로 전이시키기 때문에 scantime에 영향을주는 각 cycle의 실행시간은 cycle내에 있는 모든 문장들의 실행시간이 아니라 어느 한 scan주기에 활성화되어 실행되는 문장들만 계산하기 때문에 scantime 분석을 위한 Cycle의 실행시간은 이 문장들을 경계로 나뉘어진다. 그러므로 나뉘어진 경계들의 실행시간 중에서 가장 큰 시간을 각 cycle별로 계산하여 더하면 전체 프로그램의 최악 Scantime이 된다. 따라서 이 최악 Scantime이 프로그래머가 명시한 scantime 값보다 작으면 작성된 응용프로그램은 해당제어기에서 실행가능하게 된다.

그림 7의 ①처럼 cycle문이 시작하면서 time templete을 스택에 Push한다. 이 templete에 각 문장들의 실행시간이 더해지게 된다. delay, every, wait문등을 만나면 그림 7의 ②, ③처럼 각각 time templete을 Push하여 각 templete에 실행시간의 결과가 저장된다. Cycle문의 끝에서 스택에 Push된 templete중에서 가장 큰 값을 갖는 것이 최악 Scantime 분석에 필요한 실행시간이 된다. 따라서, 실제의 최악 Scantime은 각 Cycle을 분석한 실행시간의 합에 언어처리기 커널의 부하(load)를 더한 것이 된다.

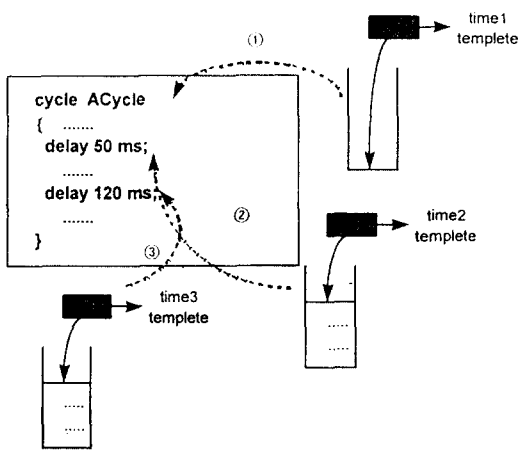


그림 7. Scantime 분석 과정

V. RTL/MCS의 구현

본 논문에서는 RTL/MCS를 Unix환경에서 gnu c++, yacc, lex를 이용하여 구현 하였다. Unix환경에서 구현한 RTL/MCS는 크게 세가지 부분으로 이루어졌다. 그림 8은 Unix환경에서 구현한 RTL/MCS의 구성을 보이고 있다.

언어처리기의 전반부(front-end)에서는 RTL/MCS의 문법으로 이루어진 원시코드(source code)를 읽어들여 중간코드를 생성한다[1]. 구문분석기에서는 RTL/MCS 구문의 문법을 검사하며, 특히 프로그래머가 within, every, delay등의 문장에서 내포(nested)하여 기술한 시간제한 문장들의 타당성이 검증된다. 의미 분석기에서는 무한루프나 반복문의 실행시간이 프로그래머가 지정한 scantime을 만족하는지 분석한다. 중간코드 생성기는 중간코드를 생성하고 중간코드 실행테이블을 참조하여 문장들의 실제 실행시간이 계산된다.

언어처리기의 후반부(back-end)에서는 중간코드를 인터프리터가 읽어들여 실행한다. 중간코드 인터프리터는 실시간클럭(real-time clock)을 이용하여 시간테이블(timing table)을 관리하여 시간 제한 문장들의 실행과, cycle들의 상태전이 감시하고 제어하여 병행적으로 실행시켜 준다.

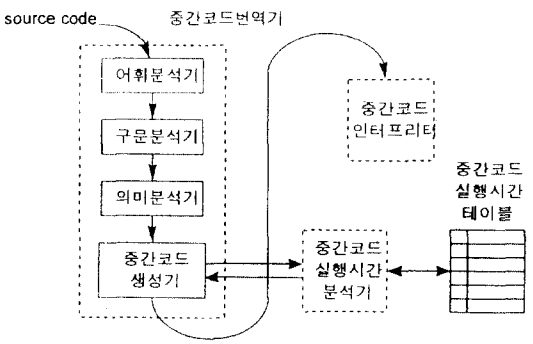


그림 8. RTL/MCS

5.1 cycle의 구현

RTL/MCS의 중간코드는 Cycle의 저장과 실행을 위해 CycleBlock과 Cycle 클래스를 정의하였다. CycleBlock은 각각의 Cycle을 저장, 실행하기 위한 클래스이고, Cycle은 CycleBlock형태로 표현된 각 Cycle들을 관리 및 실행시키는 역할을한다. 그림 9는 Cycle과 CycleBlock의 클래스 정의내용이다.

```
typedef struct CodeBlock_tag {
  Code Inst[IBUNITSIZE];
  struct CodeBlock_tag* next;
} CodeBlock;
Class CycleBlock {
  char CycleName[MAXIDSIZE];
  int no;
  int status;
  CodeBlock* IB;
public:
 CycleBlock(char *name);
 void AddInstruction(Code* code);
 void Exec();
};
```

```
class Cycle {
  CycleBlock* CBs[MAXCYCLE];
  int cycleno;
public:
  Cycle();
  void LoadCycleName(char* );
  void AddInstruction(Code* );
  void ExecCycleBlocks();
  void PrintCycle();
};
```

그림 9. Cycle과 CycleBlock의 클래스정의

CycleBlock의 멤버함수인 Exec는 한 개의 Cycle이 자기자신을 실행하기 위한 함수이다. 그리고, Cycle의 멤버함수인 ExecCycleBlocks는 Cycle이 저장하고 있는 Cycle 리스트인 CBs 내의 Cycle들을 실행하기 위한 함수이다. 현재 구현된 RTL/MCS의 중간코드 인터프리터는 라운드-로빈(Round Robin) 처리방법을 기본으로 반복실행(cyclic executive) 되도록 구현되었다. 따라서, ExecCycleBlocks의 기본적인 골격은 그림 10과 같다.

그림 10에서 처럼 실행대기큐(Ready Queue)로부터 실행될 cycle의 인덱스를 얻은 뒤 그 cycle을 대기큐의 맨뒤에 삽입하는 과정을 통해 라운드로빈 방식의 반복실행이 이루어진다. ChkTimerTable()를 통해 제한시간을 설정한 handler들 중에서 timeout된 cycle들을 실행한다. Wait나 when과 같은 조건대기문에서 설정한 조건을 변수테이블을 검사하여 그중에서 조건이 만족된 cycle들은 활성화(active) 시킨다.

```
void Cycle::ExecCycleBlocks()
{
    . . . . . .
    for(;;) {
      I - Ready.Delete();
      Ready.Add(i);
      ChkTimeTable();
      ChkVariableTable();
      CBs[i] -> Exec();
    }
      . . . . . .
}
```

그림 10. Cycle의 ExecCycleBlocks 멤버함수

5.2 TimerTable의 구현

Unix 환경에서는 프로그램의 실행시간을 gettimeofday 함수를 이용하여 마이크로초(micro second) 단위까지 측정할 수 있다. 그러므로 본 논문에서는 시간제한 문장들의 처리를 위하여 사용된 실시간 타이머는 TimerTable을 주기적으로 갱신하여 구현하며 타이머 테이블의 구조는 그림 11과 같다. 그림 11에서 TimerTable은 TimerTableNode라는 구조체의 포인터 배열이며 배열의 각 요소는 실시간 소프트웨어 타이머가 된다.

TimerTableNode에서 TimeName은 Timer의 이름을 나타내며 HandlerScope는 TimeHandler가 포함된 함수나 Cycle을 나타낸다. HandlerLabel은 Handler가 포함된 함수내에서 Handler의 위치를 나타내는 label을 저장한다. HandlerLocation은 Handler가 포함된 함수나 cycle내에서의 Handler의 첫 instruction의 시작위치이다. timeout은 타이머의 timeout이 발생하는 시간을 저장하는 변수로 Timer가 초기화될 때 값이 설정된다. timeout의 값은 gettimeofday()함수의 호출로 얻은 값을 이용하여 검사된다.

```
tydef struct TimerTableNode_tag {
    char *TimerName;
    char *HandlerScope;
    char *HandlerLable;
    char *HandlerLocation;
    long  timeout;
} TimerTableNode;

TimeTableNode *TimerTable[MAXTIMERTABLESIZE];
```

그림 11. TimerTable의 구조

5.3 결과분석 및 평가

본 논문에서 구현한 RTL/MCS의 컴파일러 및 인터프리터, 실행시간분석기는 Linux(Kernel 2.0.21)가 설치된 Intel80386(clock 33Mhz)을 CPU로 하는 PC에서 구현되었다. Linux 운영체제를 선정한 이유는 SUN 혹은 HP사의 UNIX 운영체제보다 운영체제에의한 과부하가 적으면서 마이크로초까지 시간측정이 가능하고 lex와 yacc와 같은 우수한 개발환경을 제공하기 때문이다.

5.3.1 중간코드의 실행시간 측정

표 2는 Linux(Intel80386) 시스템에서 구현한 RTL/MCS의 중간코드 인터프리터의 주요 중간코드들의 실행시간(단위는 ms)이다. 이는 각각의 중간코드들을 실행하기 전에 시스템의 실시간 클럭을 설정하고 중간코드의 실행후에 경과시간을 읽어서 계산하였으며 UNIX 시스템의 특성상 시간오차가 생기므로 10000회 실시하여 평균값을 구하였다.

표 2. RTL/MCS 중간코드의 실행시간

중간코드	실행함수	실행시간(ms)
PushWord (1)	Exec_Push_Word_I()	0.020
PushWord (2)	Exec_Push_Word_M()	0.047
PopWord	Exec_Pop_Word()	0.016
AddWord	Exec_AddWord()	0.031
MulWord	Exec_MulWord()	0.039
GTWord	Exec_GTWord	0.034
AddRun	Exec_AddRun()	0.052
DeleteRun	Exec_DeleteRun()	0.047
SetTimer	Exec_SetTimer()	0.165
DeleteTimer	Exec_DeleteTimer()	0.097
Jump	Exec_Jump()	0.064
JumpZ	Exec_JumpZ()	0.089

5.3.2 시간분석 및 실행결과

본 논문에서 구현한 실행시간 분석기로 응용 프로그램의 실행시간을 분석한결과 문장단위의 예측 성공비율은 매우 높다. 그러나 프로그램 전체의 실행시간과 실행가능성의 예측은 프로그래머가 제공한 경험적인 시간정보에 크게 의존하는 결과를 보였다. 그림 12는 RTL/MCS로 작성된 프로그램의 스캔시간 예측을 보여준다. 또한 그림 13은 cycle내의 문장단위의 실행시간 예측 과정을 나타내었다.

RTL/MCS로 작성된 예제프로그램을 본 연구에서 구현한 목적(target) 시스템에서 실행할 경우 예측한 시간과 비교할 때 약 10% 내외의 오차를 보였다. 이는 Linux가 시분할 시스템으로 구현되었기 때문으로 분석된다. 또한 운영체제의 정확한 부하(load)를 계산할 수 없었기 때문에 오차가 가변적이었다.

```
Cycle Name -> Acycle
Predicted Execution Time for ScanTime
:-> 67.427 ms
Cycle Name -> Bcycle
Predicted Execution Time for ScanTime
:-> 149.381 ms
Predicted ScanTime
:-> 216.808 ms
```

그림 12. 프로그램 Scan시간 분석

```
Statement Name -> (within 100 ms) in Acycle
Predicted Execution Time:: 59.120 ms
Statement Name -> (within 120 ms) in Acycle
Predicted Execution Time:: 49.814 ms
Predicted Execution Time for Acycle
:-> 245.179 ms

Statement Name -> (every 500 ms) in Bcycle
Predicted Execution Time:: 38.473 ms
Predicted Execution Time for Bcycle
:-> 69.354 ms
```

그림 13. 문장의 실행시간 분석

5.3.3 평가 및 향후계획

본 연구에서 설계 및 구현한 실시간 메카니즘제어언어는 비록 응용분야를 비교적 큰 시간오차를 허용하는 메카니즘제어로 한정하였지만 실시간 프로그래밍언어의 연구가 미흡한 국내에서 처음 시도 되었고 연구결과의 실용성이 높다는데 의미가 있다.

그러나 산업 자동화 분야에서 실용성을 확보하기 위해서는 무엇보다도 내장형시스템(embedded system) 개발에 적합한 목적 프로세서를 선정하여 이식하고, In-ciruit 에뮬레이터(emulator)나 로직 분석기(logic anlayzer)를 이용한 정확한 시간측정이 이루어져야 한다.

또한 RTL/MCS 커널의 실행시간을 분석하여 응용프로그램의 실행시간에 반영되어야 정확한 실행시간 예측이 가능해질 것이다.

따라서 현재 중소형 내장형 시스템의 프로세서로 가장 많이 사용되고있는 Intel8086 마이크로프로세서를 목적시스템으로한 RTL/MCS의 인터프리터의 보완과 이식을 추진하고 있다.

또한, 정확한 scan 시간의 예측을 위하여 실행시간 전반에 걸쳐 활성화되는 cycle들의 실행상태를 추적하는 알고리즘을 개발하고 있으며, 이는 RTL/MCS의 cycle이 프로그램 명령어에 의하여 명시적으로 시작(creation)과 종료(termination) 및 상태전이가 이루어지도록 설계하였기 때문에 가능할 것이다.

Ⅵ. 결 론

본 연구에서는 수치제어기(NC), 로봇제어기, 프로

그램형제어기(PLC)와같은 프로그램가능한 내장형시스템의 제어에 적합한 실시간 메카니즘 제어언어(RTL/MCS)를 설계하고 구현기법을 제안하였다. RTL/MCS는 실시간 프로그램 작성시 시간제한을 자유롭게 표현할 수 있고, 병행성을 가지고 있어 자동화기기들의 실시간 메카니즘 제어 프로그램을 효율적으로 작성할 수 있다.

또한, RTL/MCS의 실행시간분석기는 RTL/MCS의 시간제한 구문들의 실행시간을 분석하여 실시간 프로그램의 컴파일시간에 프로그램의 타당성과 실행가능성을 예측할 수 있으므로 반복적인 하드웨어적 시뮬레이션을 거쳐 프로그램의 정확성을 검증하던 기존의 실시간 제어언어들에 비하여 시간과 비용을 크게 절약할 수 있다.

따라서 본 연구는 현재 폭넓게 연구되고있는 실시간 시스템의 스케쥴링 기법의 연구와 실시간 프로그래밍언어 및 컴파일러 연구에 도움이될것으로 기대하며, 특히 선진국에 비하여 낙후되어 있는 국내의 메카트로닉스(mechatronics) 기술발전에 도움이 되기를 기대한다.

참 고 문 헌

1. Jeffrey D. Ullman, Ravi Sethi and Alfred V. Aho, Compilers: principles, techniques, and tools, Addison-Wesley,1986.
2. Burns A. and Wellings A.J., Real-time Systems and their Programming Languages, Addison-Wesley, 1990.
3. Roderick Chapman, Static Timing Analysis and Program Proof, Computer Science, University of York, PhD. thesis, 1995.
4. Seongsoo Hong, Compiler-Assisted Scheduling for Real-Time Application: A Static Alternative to Low-Level Tuning, Computer Science, University of Maryland, PhD. thesis, 1994.
5. M. Saksena, Parametric Scheduling for Hard Real-Time Systems, Computer Science, University of Maryland, PhD. thesis, 1994.
6. Harmon, M.G., T. Baker and D.B. Whalley, A Retargetable Technique for Predicting Execution Time, IEEE Real-Time Systems Symposium, USA: IEEE Press. p. 68-77, 1992.
7. Kligerman E. and A.D. Stoyenko, Real-Time Euclid: a language for reliable real-time Systems, IEEE Transactions on Software Engineering, SE-12(9): 941-949, 1986.
8. P. Puschner, Ch. Koza, Calculating the Maximum Execution Time of Real-Time Programs, Real-Time Systems, 1(2): 159-176, Sep. 1989.
9. T. Chung, H. Dietz, Language construction and transformation for hard real-time systems, Second ACM SIGPLAN Workshop on Language, Compilers, and Tools for Real-Time Systems, 1995.
10. Shaw A.C., Reasoning about Time in Higher Level Language Software, 1989, IEEE Transactions on Software Engineering, SE-15(7): 875-889.
11. Alexander Vrchoticky, Compilation Support for Fine-Grained Execution Time Analysis, ACM SIGPLAN Workshop on Language, Compilers, and Tools for Real-Time Systems, 1994.

백 정 현(Jeong Hyun Baek) 정회원

1985년: 홍익대학교 전자계산학과(이학사)

1987년: 홍익대학교 전자계산학과(이학석사)

1995년: 홍익대학교 전자계산학과 박사과정 수료

1987년: 현대중공업(주) 중전기연구소 연구원

1992~현재: 중경공업전문대학 전산과 조교수

※주관심분야: 프로그래밍언어(실시간 언어), 원격교육(Web-Based Instruction), 컴퓨터 교육방법론

원 유 헌(Yoo Hun Won) 정회원
1972년 : 성균관대학교 수학과졸업(B.S)
1975년 : 한국과학원 전자계산학과 1회 졸업(M.S)
1975년 : 한국과학기술연구소 근무
1985년 : 고려대학교 대학원 졸업 (Ph.D)
1986년 : 미국 RPI대학 교환 교수
1976~현재 : 홍익대학교 컴퓨터공학과 교수
※주관심분야 : 프로그래밍언어, 소프트웨어 엔지니어링, 객체지향언어, VLSI 디자인언어