ToyLotos/Ada : 실시간 Ada 소프트웨어 개발을 위한 객체행위 시뮬레이션 시스템

이 광 용[†] · 오 영 배[††]

요 약

본 논문에서는 기존 객체행위 설계방법에 의해 개발된 '시각적 실시간 객체모델'의 설계의미를 검증·확인하기 위한 시뮬레이션 기반 지원 시스템을 제안한다. 이 시스템은 실행 가능한 Ada 코드 생성에 의한 객체 프로세스들간의 동적인 상호작용을 시뮬레이션할 수 있게 하며, 실제 시스템 구현에 앞서 여러 가지 논리적, 시간적 문제들을 검출할 수 있게 한다. 또한, 시뮬레이션에 의해 검증·확인된 명세서로부터 Ada 프로토타입 코드를 직접 생성시켜 준다. 이 시스템은 Visual C++ 버전 4.2로 개발되었다. 그리고, 시뮬레이션 코드로 Ada를 사용하였는데, 이것은 Ada의 병행 행위 및 시간 표현력 등의 실시간 시스템의 표현력에 있어 기존 언어들에 비해 우수성을 가지고 있기 때문이다. 이 작업은 방법론 기반 시각적 모델과 자동화된 정형기법 기반 시뮬레이션 시스템의 연결, 그리고 자동화된 명세 검증·확인 기술의 실현이라는 점에서 기여한다.

ToyLotos/Ada : Object-Behavior Simulation System for Developing a Real-time Ada Software

Kwang-Yong Lee[†] · Young-Bae Oh[††]

ABSTRACT

This paper presents a simulation-based system for verification and validation(*V&V*) of design implication of the *Visual Real-time Object Model* which is produced by existing *object's behavior design method*. This system can simulate the dynamic interactions using the *executable Ada simulation machine*, and can detect various logical and temporal problems in the *visual real-time object model* prior to the real implementation of the application systems. Also, the system can generate the *Ada prototype code* from the validated specification. This system is implemented by Visual C++ version 4.2. For simulation, this system is using the Ada language because Ada's real-time expression capabilities such as *concurrent processes, rendezvous, temporal behavior expression*, and *etc*, are competent compared to other languages. This work contributes to a tightly coupling of methodology-based visual models and formal-based simulation systems, and also contributes to a realization of automated specifcation *V&V*.

1. 서 론

소프트웨어 개발 방법론 기술은 소프트웨어 개발 생산시에 적용하는 구체적인 절차, 모델, 기법을 다루는 기술로서 소프트웨어 개발 생산성을 향상시키고, 소프트웨어 품질을 높여 소프트웨어 개발을 경제적 및 질적으로 향상시키기 위한 것이다. 예를 들어, OMT[1]나 SA/SD[2] 방법들은 크고, 복잡한 시스템을 효율적으로 개발하기 위해 소프트웨어 생명주기의 전 단계를 지원하는 절차, 모델 및 기법들로 구성된 방법론들이다. 이

※ 본 연구는 1997년 정보통신부 연구비 지원에 의한 결과임.
† 정 회 원 : 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 선임연구원
†† 정 회 원 : 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 선임연구원 영역기반재사용기술 과제책임자
논문접수 : 1998년 5월 18일, 심사완료 : 1999년 6월 10일

방법들을 잘 활용하게 되면 누구나 쉽게 이해할 수 있으면서 품질이 좋은 소프트웨어를 쉽게 개발할 수 있게 되며 또한 유지보수가 용이한 시스템을 구축하게 된다.

그러나, 현재까지 제시된 소프트웨어 개발 방법론 기술들의 문제 중의 하나는 비정형성에 있다. 기존 방법론들에서는 현재 모두 그 나름대로의 어느 정도 정형화된 그래픽 모델을 제시하고 있지만 아직까지는 완전한 형태의 정형적인 모델로 보기 어렵다. 그리고, 대부분의 방법론들에서는 그래픽 표현력을 이용하여 시각적으로 이해하기 쉬운 시스템 모형을 그리는데 그 주안점을 두고 있는 형편이다. 이 결과, 소프트웨어 개발 방법론 기술에서는 실시간 시스템과 같은 시스템 개발에 있어서 반드시 필요로 하는 보다 정형화된 표현에 의해 모호함이나 상반된 것을 발견하는 기능이나, 자동 혹은 반자동으로 시스템을 개발할 수 있는 기능, 혹은 수학적인 방법에 의해 시스템을 검증할 수 있는 기능 등을 제공하지 못하고 있다.

본 논문에서는 "실시간 소프트웨어 개발을 위한 객체행위 설계 방법"[3]에서 제안한 "시각적 실시간 객체모델"의 설계의미를 검증·확인하기 위한 시뮬레이션 기반 지원 시스템인 "*ToyLotos*(*T*imed *O*bject s*Y*stem for *Lotos*)"를 제안한다. 시각적 실시간 객체모델의 설계의미를 검증·확인하기 위해 정보통신 시스템 개발 과정에서 고신뢰성 보장을 위한 실행 가능한 정형명세 방법으로 이용되고 있는 LOTOS 정형 명세 기법을 활용하였다. 실시간 시스템을 위한 여러 가지 명세언어들 가운데 LOTOS 정형 명세 기법을 사용하는 이유는 LOTOS(Language fOr Temporal Ordering Specification)는 ISO의 OSI(Open Systems Interconnection) 표준정의를 위해 개발되어진 정형 명세 기술(formal description technique)로서 1989년도에 표준 정형 명세 언어(ISO 8807)로 채택되었다[4]. 그리고, 무엇보다도 LOTOS는 본 논문에서 제안한 시각적 실시간 객체모델의 '객체 프로세스' 표현에 있어서 활용성이 높기 때문이다. 특히, 소프트웨어 생명주기 단계 중 설계단계에서 정형 명세 언어가 갖추어야할 요건인 시험성 혹은 시뮬레이션성, 모듈화 능력, 추상화 표현력, 검증성, 자료 표현력, 교육·훈련시 용이성 측면에서 살펴볼 때 LOTOS는 ESTELLE, SDL 정형명세 언어와 함께 좋은 모델 명세 언어로 추천되고 있다[5]. 그러나, 본 논문에서는 기존의 LOTOS 정형 명세 문법을 수정하여 사용하였는데 그것은 시각적 실시간 객체모델에 반영된 객체구조정보와 시간정보를 손쉽게 표현할 수 있도록 하기 위해서이다. ToyLotos 모델명세 언어는 기본적으로 객체합성, 객체상속(단일, 다중 상속), 가시성 메소드 등의 모듈화 명세개념[6,7]과 LOTOS의 병행 프로세스들 간의 메시지 정의 표현력을 결합하였으며, 시간표현 기능[10,11,12]을 추가하였다. 또한, 이 ToyLotos 모델명세 언어로 작성된 명세서들은 궁극적으로는 시뮬레이션을 위해 Ada 언어로 변환된다.

논문의 구성은 다음과 같다. 2장에서는 시각적 실시간 객체모델의 표현 및 특징에 대해 간략히 검토하고, 3장에서는 이 시각적 실시간 객체모델의 설계의미를 표현하기 위한 ToyLotos 모델명세 언어와 그 시맨틱스에 대하여 설명한다. 그리고 나서, 4장에서는 ToyLotos 명세를 입력으로 하여 수행되는 ToyLotos 시뮬레이션 시스템의 시스템 아키텍쳐 및 주요 적용기술에 대해 논하고, 5, 6장에서는 각각 간단한 실험예제 및 관련연구들과의 비교·평가를 통해 제안한 시스템의 유용성 및 보완사항에 대해 살펴보기로 하겠다.

2. 시각적 실시간 객체모델의 표현 및 특징

실시간 시스템의 모형화에서 중점적으로 고려해야할 부분으로는 구조, 행위, 그리고 시간 관점들을 어떻게 잘 표현하여 시각적 모델로 만들어내는가에 있다. 본 논문에서는 실시간 시스템의 구조, 행위, 그리고 시간을 표현하기 위한 그래픽 모델로 (그림 1, 2)와 같은 객체구조도와 객체행위챠트를 사용한다. 이 실시간 객체모델은 논문 [3]에서 제안한 객체행위설계 방법에 의해 작성된다.

먼저, 실시간 객체모델의 표기법을 살펴보면 다음과 같다. (그림 1)에서는 기본적으로 객체와 메소드는 사각형으로, 프로세스는 타원으로 표시한다. 또한 메소드 호출은 한 객체 프로세스로부터 다른 객체 메소드로의 화살표로 표시하고, 흰 원과 검은 원을 가진 화살표로 자료와 사건 흐름을 구분하여 표시한다. 그리고, 메소드 호출 화살표의 모양과 화살표 라벨(label)에 의해 waiting, timeout, balking 실행 타이밍과 오류처리 화살표를 구분하여 나타내며, 프로세스의 주기적인 실행과 비주기적인 실행형태에 따라 삼각형이 첨가된 타원으로 표시한다.

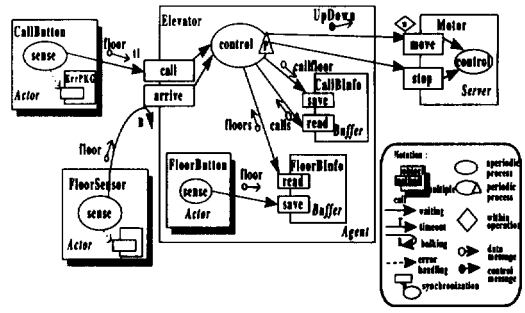
한 객체의 프로세스에서는 다른 객체의 세부적인 메

소드와 프로세스를 알 필요가 없고 오로지 인터페이스에 관련된 메소드들만 알면 되므로 모델에는 외부 프로세스에서 보이는(visible) 메소드들만을 경계선에 그린다. 객체들 사이의 인터페이스는 화살표로 표현하되 메소드 호출(method invocation)의 의미를 가지며 그 위에 자료나 사건흐름을 표현한다. 자료와 사건 흐름은 객체 프로세스로부터 메소드로의 입출력 형태에 따라 In 타입(○→)흐름, Out 타입(←○)흐름, 그리고 In Out 타입(←○→)흐름이 있다. 각각의 객체 하단에는 객체의 행위를 액터(Actor), 에이전트(Agent), 서버(Server), 자료객체(Data Object) 형태로 구분하여 표시하고, 프로세스 안에는 프로세스 수행 기능 및 상태를 표시하여 시각적으로 전체적인 시스템 행위를 판단할 수 있도록 하였다.

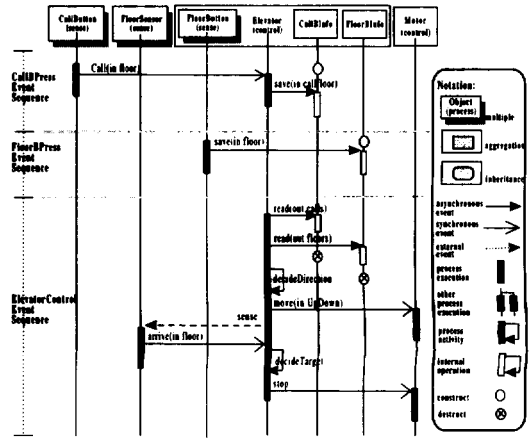
그리고, (그림 2)는 시스템의 행위(상호작용) 순서를 시간순서로 모형화한 예인데 기본적으로 프로세스 실행상태는 검고 굵은 실선으로, 오퍼레이션의 수행은 희고 굵은 실선으로 표시하며, 객체들간의 상호작용은 시간순서로 챠트의 위에서부터 아래로 나열하는데 비동기적인 호출은 닫힌 화살촉을 가진 화살표로 동기적 호출은 열린 화살촉을 가진 화살표로 표시한다. 또한, 내부 프로세스 활동 혹은 오퍼레이션은 각각 (그림 2)의 표기법에서처럼 자기자신의 프로세스 혹은 오퍼레이션을 호출하는 형태로 표현한다. 그리고, 객체의 동적인 행위 즉, 객체의 생성과 소멸 시점을 나타내기 위해 동그라미(◯)와 동그라미 안에 x표(Ⓧ)를 가진 형태로 표시한다. 소프트웨어 시스템으로는 구현이 되지는 않지만 실제 시스템 외부의 물리적 혹은 기계적 작용에 의해 두 객체 사이의 시간적인 의존성이 존재하는 것을 표시하기 위해 점선 화살표를 이용한다. 객체행위챠트 좌측에는 시스템의 사건그룹(event group)들이 나타나는데, 하나의 사건그룹은 외부 객체로부터의 자극에 대해서 시스템 객체들이 반응하는 일련의 메소드 호출 및 내부 오퍼레이션들의 집합을 의미한다. 이 객체행위챠트를 그리는 방법은 객체행위챠트 좌측에는 모형화할 시스템의 사건그룹(event group)들을 표시하고, 챠트의 상단에는 객체들의 행위특성에 따라 액터는 좌측에, 서버들은 우측에, 그리고 에이전트 혹은 중간객체들은 챠트의 중간에 위치시켜서 그린다.

(그림 1)과 같은 객체구조모델은 기존 객체모델(e.g., Rumbaugh et al.의 객체모델[1])에서의 객체 클래스, 집단화, 연관화, 일반화, 객체 다중성 등의 표현의미는 그대로 가지고 있으면서, 실시간 시스템의 병행성, 시간성, 정확성 등의 실시간 특성을 만족시키도록 하였다. 구체적으로 병행 특성을 표현하기 위해 객체 프로세스를 도입하였으며, 시간 특성을 표현하기 위해 프로세스 주기, 프로세스 우선순위, 메소드 실행 타이밍 등의 표현기법들을 추가하였다. 그리고, 실시간 시스템의 비정상적인 행위에 대한 대처를 위해 오류처리 루틴들의 표시와 그 루틴에의 접근에 대한 표기법을 첨가하였다.

또한, (그림 2)의 객체행위챠트는 (그림 1)과 같이 객체구조모델에서는 외부 객체의 메소드 호출에 반응하는 객체의 활동을 표시하지 못하고 있는 점을 해결하기 위한 방안으로 개발되었다. 이 객체행위모델을 개괄적으로 살펴보면 Jacobson의 상호작용도[8]와 비슷한 형태를 갖고 있으나, 이 모델과의 근본적인 차이는 객체구조모델의 구조적 의미 즉, 객체 캡슐화와 상속,



(그림 1) 실시간 객체모델 I - 객체구조도



(그림 2) 실시간 객체모델 II - 객체행위챠트

연관, 다중, 그리고 객체간의 호출/피호출 관계 등의 의미는 그대로 이 행위챠트에 표현되면서 추가적으로 객체간의 상호작용(이 행위챠트에서는 호출/피호출에 의한 메시지 전달관계)이 순차적으로 나열이 되는 것이다.

3. ToyLotos 모델명세언어 및 시맨틱스 정의

2장에서 제시한 시각적 실시간 객체모델은 사용자가 이해하기 쉬운 형태로 만들어져 있지만 기계가 이해할 수 있는 형태는 아니다. 따라서, 실제로 시각적 객체구조 및 행위 모델로 모형화된 시스템이 과연 사용자가 의도한대로 수행이 되는가를 점검하기 위해서는 이를 기계가 이해할 수 있는 형태로 바꾸어 주어야 한다. 이 장에서는 시각적 객체구조 및 행위 모델로부터 기계가 이해할 수 있는 형태인 명세언어의 문법 및 그 행위 시맨틱스에 대하여 간략히 설명한다.

3.1 모델명세언어

본 논문에서 제시하는 명세언어는 시각적 실시간 객체모델을 효과적으로 표현할 수 있는 형태로 구성되어 있다.

먼저, (그림 1)과 같은 객체구조모델은 전체적으로 볼 때 하나의 시스템을 설명하고 있다. 따라서, 이 시스템을 표현하기 위해 아래와 같은 시스템 명세서를 사용한다. 이 시스템 명세서는 표준 Lotos 명세의 시스템 명세서와 구조가 많이 닮았으나 표준 Lotos 구조에서는 프로세스의 표현에 중점을 둔 것에 비해, 제안한 시스템 명세서에서는 객체 클래스들을 표현할 수 있게 하였다.

```
system system_name [list of gates] : functionality :=
  (* standard library & information type definitions *)
      library ... endlib
      type  ...  endtype
      type  ...  endtype
      ...
  (* system behavior definitions *)
      behavior
        (* behavior expression *)
      where
        (* class definitions *)
        class  ...  endclass
        class  ...  endclass
        ...
endsys
```

다음으로 실시간 객체모델의 핵심이 되는 개별 객체들을 표현하기 위하여 클래스 명세서를 제공한다. (그림 1)의 개별 객체들 각각에 대하여 아래와 같은 클래스 명세서를 이용하여 작성한다. 이 클래스 명세서에는 객체들의 집단, 상속, 연관 구조와, 객체들 사이의 메시지 상호작용, 그리고 객체 내부에 포함된 프로세스에 대하여 기술할 수 있게 되어 있다.

```
class  class_name  is
  (* information type & behavior characteristics definition *)
    type   ...   endtype;
    subtype  ... endtype;
    ...
    a_kind_of
        (* behavior characteristics *)
  (* internal or external visible method definitions *)
    extern_visible
        (* messages definitions *)
    intern_visible
        (* messages definitions*)
  (* aggregation, inheritance, and using relations definitions *)
    using
        (* list of classes *)
    containing
        (* list of classes *)
    inherit_from
        (* list of classes *)
  (* process behavior definitions *)
    process  ...  endproc
    process  ...  endproc
    ...
endclass
```

한편, 실시간 시스템의 객체 행위는 아래와 같은 프로세스 정의 구문을 사용하여 표현한다.

```
process  process_name [ list of gates ] : functionality :=
  (* behavior expression *)
where
  (* data type definitions *)
endproc
```

이 프로세스 명세는 (그림 2)와 같은 객체행위챠트에 묘사된 행위 시맨틱스를 표현하는데 이용된다. 객체행위챠트의 행위는 기본적으로 동기적/비동적 메시지의 나열과 외부 객체 프로세스로부터의 동기적 메시지 입수에 의한 선택적 행위에 의해 묘사된다. 그러나, 객체의 행위 형태는 Actor, Agent, Server와 같은 객체행위 패턴으로 제한이 되게된다. 구체적으로 하나의 메시지는 "게이트!사건!프로세스!객체!파라메터들..."과 같이 느낌표(!) 분리자로 구분이 되며 객체 프로세스에서의 하나의 사건과 그 사건을 입수할 수 있는 게이트 채널로 구성되어있다. 그리고, 프로세스 이름에 따라 동기적/비동기적 메시지를 구분하는데 특별히 비동기

적인 메시지는 'self'라는 예약어를 사용한다. 그리고, 외적으로 관찰 가능한 메시지와 내부 프로세스에서만 들여다 볼 수 있는 메시지로 구분이 되는데 이것은 게이트 이름에 따라 내부 메시지는 'i'로 구분이 되고, 외부 메시지는 사용자가 정의한 이름으로 구분이 된다. 구체적인 행위 표현문법은 다음 <표 1>에서와 같다.

〈표 1〉 객체 행위 표현문법

행위정의 의미	문 법
행위정지	stop
실행탈출	exit
선택행위	B_1 [] B_2
선택적 동기화행위	(g!B_1 -> (...) [] g!B_2 -> (...) ...)
메시지	gate!event!process!object!parameters...
외부가시성메시지	g!B
내부가시성메시지	i!B
조건	[CE] -> (B)
실행연기	wait t
동기화 요청	entry
timeout 동기화 요청	entry g!B when t -> (...)
balking 동기화 요청	entry g!B when balking -> (...)
주기적 실행	every(t) do B end
시간내 실행	within(t) do B when t -> B end
범례 : B, B_1, B_2 : 행위 표현식 g, g_1, g_2, ... , g_n : 게이트 CE : 조건식	

3.2 객체 시맨틱스 정의

2장에서는 실시간 객체모델의 비정형적 의미에 대해 간단히 살펴보았다. 이 절에서는 실시간 객체모델의 행위 시맨틱스에 대해 아래와 같이 간단한 식을 이용하여 설명한다. 이 식의 의미는 행위식 B 중에서 x 행위를 수행하게 되면, 행위식은 B에서 B′ 상태로 변경됨을 나타내는 것으로 LTS(Labelled Transition System)[9]를 표현하는 표현식으로 사용된다.

$$B \text{ -}x\text{ }\rightarrow B'$$

이 행위 도출식에서, x는 하나의 행위(action)를 나타내고, B와 B′은 행위식(behavior expression)을 표현한다.

객체 행위 시맨틱스

앞서 설명한 시각적 실시간 객체모델과 객체행위식과의 대응관계를 살펴보면 다음에서와 같이 시각적 모델에서의 "process execution"은 아래 행위식들에서의 B, B1, B2로, 또 "객체프로세스"는 P1, P2로 각각 대응이 되어 표현되어 있다.

* 선행행위: i;B or g;B

선행행위(action prefix)는 하나의 사건이 어떤 행위식보다 앞서 선행해서 수행됨을 표현하는데 사용된다. 이 선행행위의 의미는 아래와 같은 하나의 행위식으로 표현된다. 이 식의 의미는 행위 "μ;B"는 동작 μ를 수행하고 나서 "B" 행위로 변환됨을 나타낸다.

$$\mu;B \text{ -}\mu\text{ }\rightarrow B$$

* 선택: B1 [] B2

이 선택 행위식은 두 가지 가능한 행위식 중에서 하나를 선택하여 수행하는 것을 표현한다. 행위표현식 B1 [] B2는 현재 동작이 B1의 처음 동작인가 아니면, B2의 처음 동작인가에 따라 프로세스가 B1처럼 동작하는 프로세스 혹은 B2처럼 동작하는 프로세스를 생성하는 의미를 갖고 있다. 이러한 의미는 아래와 같은 두 가지 행위식에 의해 표현된다.

$$B1 \text{ -}\mu\text{ }\rightarrow B1'$$
implies
$$B1 \text{ [] } B2 \text{ -}\mu\text{ }\rightarrow B1'$$

$$B2 \text{ -}\mu\text{ }\rightarrow B2'$$
implies
$$B1 \text{ [] } B2 \text{ -}\mu\text{ }\rightarrow B2'$$

* 일반 병렬행위: P1 |[g,...,g]| P2

이 병렬 행위식은 동기화 게이트 집합이라고 부르는 사용자 정의 게이트 집합에 의해 두 개 이상의 프로세스가 결합되어 동시에 동작할 수 있음을 표현한다. 만약, P1 |S| P2가 병렬 행위식을 표현할 때, 아래 규칙에 의해 병렬 행위의 의미가 표현된다. 이 행위식은 세 번째 규칙에서와 같이 S에 속하는 게이트에서의 동작은 P1과 P2에서 함께 동작하여 일어날 수 있음을 표현한다. 그리고, 그 이외의 게이트에서의 동작은 첫 번째와 두 번째에서와 같이 P1 혹은 P2 각각에서 일어날 수 있음을 표현하고 있다. 여기에서, S는 $\{g_1,...,g_n\}$의 사용자 정의 동기화 게이트들의 집합을 의미하고, δ는 프로세스의 성공적 행위 종결을 의미하고 있다.

$$P1 \text{ -}\mu\text{ }\rightarrow P1' \text{ and } \mu \notin S$$
implies
$$P1 \text{ |S| } P2 \text{ -}\mu\text{ }\rightarrow P1' \text{ |S| } P2$$

$P2 - \mu \rightarrow P2'$ and $\mu \notin S$
implies
$P1 \ |S| \ P2 - \mu \rightarrow P1 \ |S| \ P2$

$P1 - g \rightarrow P1'$ and
$P2 - g \rightarrow P2'$ and $g \in S \cup \{\delta\}$
implies
$P1 \ |S| \ P2 - g \rightarrow P1' \ |S| \ P2'$

* 순수 병렬행위: B1 ||| B2

동기화 게이트의 집합 S가 공집합({})일 경우에 병렬 연산자 |S|를 |||로 나타내며, 위의 병렬 행위식에서 성공적 종결을 제외한 세 번째 규칙은 절대 적용되지 않으며, 나머지 두 규칙으로 순수 병렬 행위의 의미가 표현되는데 이 두 규칙은 두 구성요소 프로세스가 독자적으로 행하는 동작을 설명한다.

* 행위정지: stop

ToyLotos에서 stop으로 표시되는 행위정지(terminate)는 프로세스가 더 이상 외부와 상호 작용할 수 없는 상태에 도달하였음을 나타낸다.

* 실행탈출: exit

ToyLotos에서는 프로세스의 실행탈출을 exit으로 표현하고, exit을 stop 전에 일어나는 특별한 종결동작으로 해석한다. 이 특별한 종결 동작을 δ라 부르며 다른 모든 정상적인 동작들과 구별이 된다.

$exit - \delta \rightarrow stop$

객체 시간 시맨틱스

객체구조도에 표현된 프로세스 주기(삼각형), 마감시간(다이아몬드), timeout 혹은 balking과 같은 동기화 사건(화살표)은 아래의 행위식들에서와 같이 각각 every, within, synchronous event의 의미를 갖는다. 객체 시간 시맨틱스의 표현에서도 객체 행위 시맨틱스 표현에서와 같이 P는 객체 프로세스를, B, B1, B2는 객체 프로세스의 행위를, 그리고 M1, M2는 객체행위챠트에서의 하나의 메시지를 나타낸다.

* every I (periodic process) : P~every(Φ)

이 행위식의 의미는 특정한 객체 프로세스 P의 주기적인 프로세싱 즉, every(Φ)를 표현하는 것으로 다음 식에서와 같은 의미를 갖고 있다. 즉, 어떤 시점 T에서 프로세스 P에 대하여 행위집합 μ가 수행되고 난 후의 $T+\Phi$ 시점에는 P' 상태에 도달하게되어 다시 프로세스 P가 수행할 수 있는 준비상태에 도달하는 것을 의미한다.

$P - \mu \ \wedge \ t=T \rightarrow P' \wedge \ t=T+\Delta(\mu)$ and
$P' - \varepsilon \ \wedge \ t=T+\Delta(\mu) \rightarrow P'' \wedge \ t \leq T+\Delta(\mu+\varepsilon)$ and
$\Phi=\Delta(\mu+\varepsilon)$
implies
$P\tilde{}every(\Phi) - \mu \wedge t=T \rightarrow P''\tilde{}every(\Phi) \wedge t=T+\Delta(\mu+\varepsilon)$

여기에서, μ는 프로세스 P의 행위집합 즉, $\{\mu_1, \ldots, \mu_n\}$를 의미하며, ε는 프로세스 P'이 blocked 상태에 있기 위한 LTS 시스템의 특별한 사건을 나타낸다. 그리고, $\Delta(\mu+\varepsilon)$는 μ행위집합을 수행하기 위해 필요로 하는 시간($\Delta(\mu)$)과 ε행위에 의해 프로세스가 blocked 상태에 빠져있는 시간($\Delta(\varepsilon)$)의 합을 의미하는 것으로 Φ가 된다.

* every II (periodic operation) : every(Φ) do B

이 표현식의 의미는 위의 주기적인 프로세스(P~every(Φ))와 마찬가지로 주기(Φ)에 따라 반복적으로 수행하게 되는 행위를 표현한다.

$B - \mu \ \wedge \ t=T \rightarrow B' \ \wedge \ t=T+\Delta(\mu)$ and
$B' - \varepsilon \ \wedge \ t=T+\Delta(\mu) \rightarrow B'' \ \wedge \ t \leq T+\Delta(\mu+\varepsilon)$ and
$\Phi=\Delta(\mu+\varepsilon)$
implies
every(Φ) do $B - \mu \ \wedge \ t=T \rightarrow$ every(Φ) do $B'' \ \wedge$
$t=T+\Delta(\mu+\varepsilon)$

여기에서, μ는 행위집합 B 즉, $\{\mu_1, \ldots, \mu_n\}$를 의미하며, ε는 행위식 B'이 blocked 상태에 있기 위한 LTS의 특별한 사건을 나타내고, $\Delta(\mu+\varepsilon)$는 μ행위를 수행하기 위해 필요로 하는 시간($\Delta(\mu)$)과 ε행위에 의해 프로세스가 blocked 상태에 빠져있는 시간($\Delta(\varepsilon)$)의 합을 의미한다.

* within : within(Φ) do B1 when timeout → B2

이 표현식의 의미는 Φ시간 내에 B1 행위식이 끝나지 못했을 때 B2 행위식을 수행하는 것을 표현한다. 이 within 표현식의 의미는 다음과 같은 행위도출식에 의해 표현된다. 즉, 첫 번째 식에서와 같이 B1 행위식에 대하여 사건 μ가 Φ시간 내에 수행이 되면 B1' 상태에 도달하나, 만약 두 번째 식에서와 같이 사건 μ

를 수행하는데 있어 Φ시간을 넘기게 되면 B2′ 상태에 도달하게 됨을 의미한다.

$B1 - \mu \wedge t=T \rightarrow B1' \wedge t \leq T+\Delta(\mu)$ and $\Phi=\Delta(\mu)$
implies
"within(Φ) do B1 when timeout → B2" $- \mu \wedge t=T$
$\rightarrow B1' \wedge t \leq T+\Delta(\mu)$

$B2 - \mu \wedge t=T \rightarrow B2' \wedge t>T+\Delta(\mu)$ and $\Phi=\Delta(\mu)$
implies
"within(Φ) do B1 when timeout → B2" $- \mu \wedge t=T$
$\rightarrow B2' \wedge t>T+\Delta(\mu)$

* timeout, balking event

- timeout event : entrycall *M1* when(Φ) → B

이 표현식은 호출행위 *M1*이 Φ시간일 때까지 수행이 되지 않으면 B 행위를 수행해야함을 표현하는 것으로 다음 식에서와 같은 의미를 갖고 있다. 즉, 첫 번째 식에서와 같이 현재 시점 T에서 행위 μ 즉, *M1*이 T+Φ 시간 내에 처리가 되면 다음 행위 B2에 도달하는 것으로 정상적으로 entrycall이 수행되는 것을 의미하나, 그렇지 않고 두 번째 식에서처럼 T+Φ 시점을 넘기게 되는 경우에는 정상적인 행위 B2를 수행하는 대신에 B1 행위를 수행됨을 의미한다.

M1;$B2 - \mu \wedge t=T \rightarrow B2 \wedge t \leq T+\Delta(\mu)$ and $\Delta(\mu)=\Phi$
implies
"entrycall *M1* when(Φ)→B1;B2" $- \mu \wedge t=T \rightarrow B2 \wedge t \leq T+\Phi$

M1;B1 $- \mu \wedge t=T \rightarrow B1 \wedge t>T+\Delta(\mu)$ and $\Delta(\mu)=\Phi$
implies
"entrycall *M1* when(Φ)→B1;B2" $- \mu \wedge t=T \rightarrow B1;B2 \wedge t>T+\Phi$

- balking event : entrycall *M1* when balking → B

이 표현식은 위 타임아웃 사건(timeout event)의 특수한 형태로 호출행위 *M1*이 즉각적으로 수행이 되지 않으면 B 행위를 수행해야 하는 것을 표현한다. 즉, 다음 첫 번째 식에서 처럼 현재시각 T 시점에서 호출행위 *M1*에 대해 μ사건이 T시점과 거의 동일한 시점에서 발생하게 되면 정상적으로 *M1′* 상태에 도달하게 되어 정상적인 처리가 됨을 의미 하지만, 두 번째 식에서 처럼 μ를 수행함에 있어 T 시점 이상에서 발생하게 될 경우에는 *M2* 행위식을 처리해야 하는 것을 의미한다.

M1;$B2 - \mu \wedge t=T \rightarrow B2 \wedge t=T+\Delta(\mu)$ and $\Delta(\mu) \cong 0$
implies
"entrycall *M1* when balking→B1;B2" $- \mu \wedge t=T$
$\rightarrow B2 \wedge t=T$

M2;$B1 - \mu \wedge t=T \rightarrow B1 \wedge t>T+\Delta(\mu)$ and $\Delta(\mu)>0$
implies
"entrycall *M1* when balking→B1;B2" $- \mu \wedge t=T$
$\rightarrow B1;B2 \wedge t>T+\Delta(\mu)$

* waiting : wait(Δ); B

이 표현식의 의미는 Δ 시간동안 행위를 정지하고 하고 있다가 Δ 시간이 지난 후에는 B 행위를 수행하는 것을 표현하는 것으로 다음과 같은 의미를 갖는다. 즉, 행위식 "wait Δ;B"에 대해 현재 시점 T에서 ε 사건에 의해 Δ시간동안 행위정지 한 후, T+Δ 시점에는 행위식이 B가 된다.

wait(Δ); $B - \varepsilon \wedge t=T \rightarrow B \wedge t=T+\Delta$

여기에서, ε는 현재 프로세스를 Δ 시간동안 blocking 하기 위한 행위지연 오퍼레이션이다.

4. ToyLotos 시뮬레이션 시스템

이 장에서는 ToyLotos 시뮬레이션 시스템에 의해 제공되는 주요 기능들과 시스템 아키텍처에 대해 설명한다. 이 시스템은 Visual C++ 버전 4.2로 개발되었으며 Windows 95 환경에서 Stand-alone형으로 수행된다.

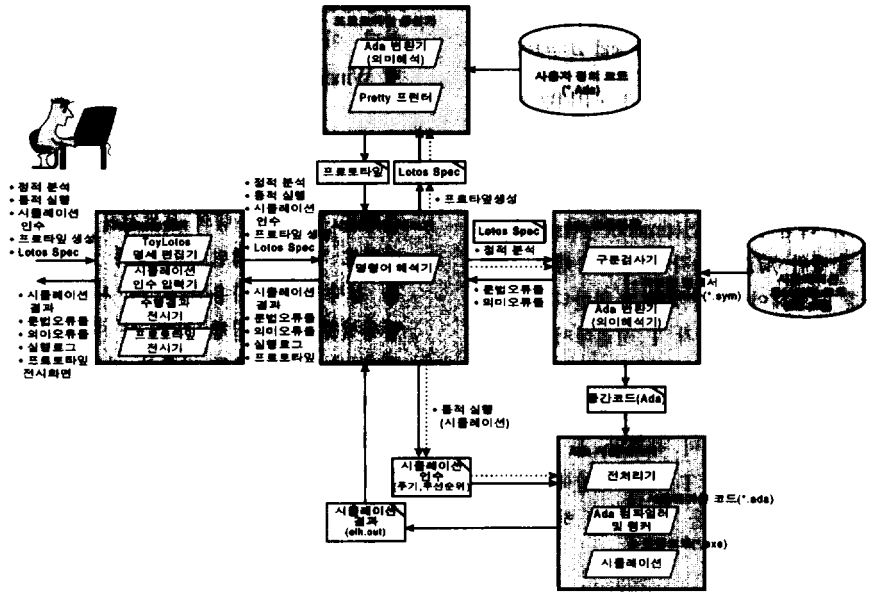
4.1 시스템 아키텍처 및 주요기능

ToyLotos 시스템은 명세서를 입력으로 받아 실제 실행을 통해 시스템의 유효성을 확인하기 위한 시뮬레이터로 명세된 행위가 사용자 요구사항에 부합하는지를 검사하는데 사용된다. 이 시스템의 특징은 시스템 명세서를 직접 실행 시켜 볼 수 있게 함으로써 개발 대상 시스템에 대한 정확한 이해와 함께, 사용자와 개발 팀원들간의 명확하고 애매함이 없는 의견을 도출해 내도록 지원하는 것이다. 또한, 작성된 Lotos 명세서의 직접적인 수정이 없이도 프로세스에 대한 주기, 우선순위와 같은 시간 인수들을 변경하여 시뮬레이션할 수 있게 함으로써 시스템의 스케줄링 가능성(schedulability)을 동적으로 분석할 수 있게 지원하는 특징을 갖고 있다.

ToyLotos 시스템은 (그림 3)의 시스템 아키텍쳐도에서와 같이 크게 5가지 종류의 서브시스템 객체들로 구성되어 있다. ToyLotos GUI는 사용자와의 직접적인 상호작용을 위해 제공되는 부분으로 시뮬레이션 결과를 여러 가지 다양한 윈도우들에 전시하는 역할을 담당한다. 시뮬레이션 제어기는 ToyLotos GUI를 통해 사용자로부터 입력된 명령과 자료를 Ada 변환엔진, 시뮬레이터, 프로토타입 생성기로 가공 처리하여 전달하여주는 작업과 명령어 실행결과를 GUI에 전시해 주는 일종의 중간자적인 역할을 담당한다. 그리고, Ada 변환엔진은 모델 명세서를 입력으로 받아들여 구문검사를 통해 문법오류를 점검하고 의미해석기와 시뮬레이션 모니터링 코드를 이용하여 문법오류가 없는 명세서를 1차 시뮬레이션이 가능한 Ada 코드로 변환한다. 여기서, 시뮬레이션 모니터링 코드는 시뮬레이션 중에 발생한 사건과 중간 결과들을 동적으로 수집하기 위한 모니터링 코드로 구성되어 있으며, ToyLotos 시스템에서는 'elh_exe(...)'라는 모니터링 함수를 이용하여 시뮬레이션 코드의 적절한 부분에서 시뮬레이션 정보를 수집하게 한다. 또한, Ada 시뮬레이터는 ToyLotos 시뮬레이션 시스템의 핵심기능을 담당하는 부분으로 시뮬레이션 시간, 주기, 우선순위와 같은 시뮬레이션 인수와 Ada 변환엔진에 의해 생성된 중간코드를 입력으로 받아 실시간 실행이 가능한 코드를 생성시켜 동적으로 실행시켜볼 수 있는 장치이다. ToyLotos 시뮬레이션 시스템에서는 실시간 인수들을 자유롭게 변경하면서 만족된 결과가 도출될 때까지 시뮬레이션해볼 수 있게 하였다. 프로토타입 생성기는 시뮬레이션 결과 요구사항에 만족된 모델 명세서에 대하여 시뮬레이션 모니터링 코드가 빠진 나머지 부분과 사용자가 직접 정의한 사용자 기능들을 이용하여 실제 Ada 코드를 생성하는 장치로 만들어 졌다.

4.2 Ada 변환엔진

ToyLotos 시뮬레이션 시스템에서는 Ada 변환을 위해 구문중심변환(syntax-directed translation) 기법을 이용하여 객체 구조 및 행위 명세를 Ada 코드로 변환한다. 이 구문 중심 변환 기법은 명세서를 탐색하는 동안에 해당 구문의 시맨틱스에 해당하는 코드로 변환해내는 기법이다. 객체구조 명세의 시맨틱스를 Ada코드로 변환하기 위한 변환규칙은 다음 <표 2>에서와 같다. 이 표로부터 유추할 수 있는 Ada 변환행위들을 객체구조 명세 문법에 나타내보면 다음 (그림 4)에서와 같다. 이로부터 ToyLotos 시스템의 해석과정에서 구문트리를 형성하는 동안 Ada 변환행위들을 수행하



(그림 3) ToyLotos 시뮬레이션 시스템 아키텍쳐

게 됨으로써 다음과 같은 "StationManager" 클래스를 Ada 코드로 변환하게 된다.

```
class StationManager is
        a_kind_of
                relay
        extern_visible
                Sm!arrive!manage!StationManager,
                Sm!depart!manage!StationManager,
                Sm!call!self!StationManager
        process manage [] : noexit :=
        ...
        endproc;
endclass
```

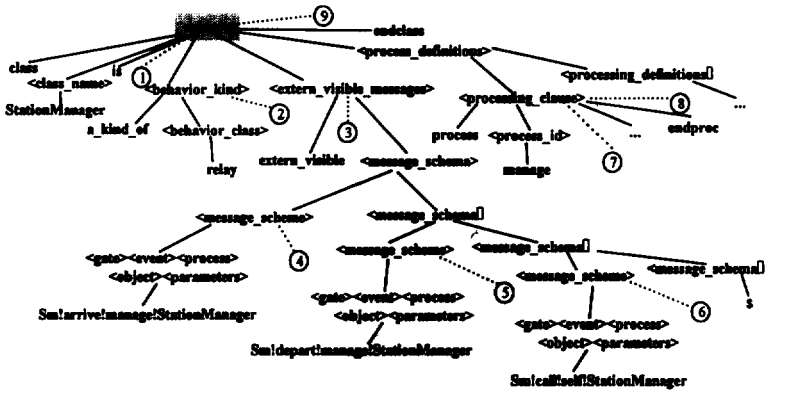
〈표 2〉 객체 구조 명세의 Ada 변환규칙들

Object Structure Specification	Ada 변환규칙
class	Package
synchronous visible message	Entry Call in Task Spec
asynchronous visible(self) message	Procedure or Function
process	Task
message parameters	Parameters
containing classes	Package in Package with comment "composite"
inherit classes	Package in Package with comment "inherit"
using classes	With, Use
behavior kind	Comment

```
<class> -> class <class_name> is
           {printf("Package %s is",<class_name>);}
             <behavior_kind>
             <external_visible_messages>
             <process_definitions>
           endclass
           {printf("End %s",<class_name>);}

<behavior_kind> -> a_kind_of <behavior_class> |e
<behavior_class> -> actor    {printf("-- a_kind_of actor");}
                  | agent    {printf("-- a_kind_of agent");}
                  | server   {printf("-- a_kind_of server");}
                  | data     {printf("-- a_kind_of data");}
                  | buffer   {printf("-- a_kind_of buffer");}
                  | relay    {printf("-- a_kind_of relay");}
                  | transporter {printf("-- a_kind_of transporter");}
                  | external    {printf("-- a_kind_of external");}

<external_visible_messages> -> extern_visible
                               {msgTyp = EXTERN;} <message_schema>
                             |e
<message_schema> -> <message_scheme> <message_schema'>
<message_schema'> -> ',' <message_scheme> <message_schema'> |e
<message_scheme> -> <gate><event><process><object><parameters>
                    {
                     if( msgTyp == EXTERN ) {
                       if( "<process>" != SELF )
                         fprintf(tmpFp,"entry %s%s;",<event>,<parameters>);
                       else
                         fprintf(tmpFp,"Procedure %s%s;",<event>,<parameters>);
                     }
                    }

<processing_definitions> -> (<processing_clause>
                           | <type_declaration> <processing_clause>)
                             <processing_definitions'>
                           | e

<processing_definitions'> -> ';' ( <processing_clause> |
                                   <type_declaration> <processing_clause> )
                             <processing_definitions'>
                           | e

<processing_clause> -> process <process_id>
                       ['(' <gate_list>')'][':'] ':'
                        <behavior_[nuc]tionality>
                        ( 'priority' '(' <priority_num> ')') ':='
                       {printf("Task %s is",<process_id>);
                        readFile(tmpFp, Entries);
                        printList(Entries);}
                        <behavior_expression>
                        <type_expression>
                       endproc
                       {printf("End %s",<process_id>);}
                     | e
```

(그림 4) 구문중심 Ada 변환

(그림 5)는 Ada 코드 변환과정을 구문트리로 나타낸 것이며, 이 파스트리를 깊이우선탐색(depth first search)하는 동안에 Ada 코드 변환행위들(①~⑨)을 만나게 되면 그 시점에서 적절한 코드 변환행위를 수행한다. 예를 들어, 이 그림에서 첫 번째로 class 오퍼런드를 만나는 동안에는 어떤 변환행위가 없기 때문에 다음



① {printf("Package %s is\n",<class_name>)}
② {printf("-- a_kind_of %s ",<behavior_class>)}
③ {msgTyp = EXTERN;}
④ {fprintf(tmpFp,"entry %s%s\n", <event>,<parameters>);}
⑤ {fprintf(tmpFp,"entry %s%s;\n", <event>,<parameters>);}
⑥ {printf("Procedure %s%s\n", <event>,<parameters>);}
⑦ {printf("Task %s is\n", <process_id>);
readFile(tmpFp,Entries);
printList(Entries); }
⑧ {printf("End %s;\n", <process_id>);}
⑨ {printf("End %s;\n",<class_name>);}

(그림 5) StationManager 클래스의 Ada 변환과정

오퍼런드를 방문한다. 다음으로 <class_id>에 대한 서브트리에서 식별자 식별행위를 수행한 다음, 변환행위가 필요 없는 is 오퍼런드를 방문한다. 그리고 나서, 첫 번째 구조 시맨틱스 변환행위인 "{ printf("Package %s is\n",<class_name>); }"를 수행한다. 이러한 방법으로 파스트리의 모든 노드들을 방문하는 동안에 객체 구조 시맨틱스와 일치하는 Ada 코드를 아래와 같이 생성하게 된다.

```
Package StationManager is
        -- a_kind_of  relay
        Procedure call;
        Task manage is
                entry arrive;
                entry depart;
        End;
End StationManager;
```

그렇지만, 실제 Ada 변환엔진에서는 1차 구문분석과정에서의 심볼테이블생성 및 구문오류 식별과 2차 의미해석과정에서의 시맨틱 오류들을 찾기 위해 여러 번의 파스트리 탐색을 수행하게 된다.

다음으로, 객체 구조 시맨틱스의 Ada 변환 기법과 마찬가지로 객체 행위 시맨틱스에 대해서도 구문중심 변환기법을 적용한다. 파스트리를 탐색하는 동안에 적절한 변환행위들을 수행한다.

객체행위명세를 Ada의 태스크로 변환하는 행위들을 살펴보면 다음 <표 3>에서와 같다. 객체행위 명세에서 행위 시맨틱스를 나타내는 구문들은 <표 3>에서 보는 것과 같이 순서를 나타내는 선행행위, 행위 선택, 두 프로세스들간의 동기화 행위, 행위 탈출, 행위 정지 등이 있다. 이들은 각각 Ada의 순서, selective accept 문, entry call & accpet, procedure 호출문, exit, 혹은 terminate 문으로 변환이 된다.

〈표 3〉 객체 행위 명세의 Ada 변환규칙

Object Behavior Specification	Ada 변환규칙
순서(;)	순서(;)
반복문(loop, noexit)	반복문(loop 문)
조건문(condition)	선택문(if or case 문)
동기화 입력에 의한 행위표현(g!e!p!o!..->...)	accept 문
대안([])과 동기화 입력(g!e!p!o!..->...)	selective accept문
동기화호출(g!e!p!o!...)	entry 호출문
비동기적호출	procedure 호출문
행위탈출(exit)	행위탈출(exit 문)
행위정지(stop)	행위정지(terminate 문)

한편, 시간 행위 시맨틱스의 Ada 변환규칙들은 <표 4>에서와 같은 시간특성을 가진 구문들에 대하여 적

〈표 4〉 시간 행위 시맨틱스의 Ada 변환규칙

Object Temporal Behavior Syntax	Ada Production Rules

within <time> do
  <stmt1>
when timeout -> <stmt2>
end
	declare
	  timeout:exception;
	  task timecheck;
	  task within_stmt is
	  entry start;
	  end;
	  task body timecheck is
	  withint : calendar.time;
	  time : duration := t;
	  begin
	  withint := calendar.clock+time;
	  loop
	  if calendar.clock >= withint then
	  raise timeout;
	  end if;
	  select
	  within_stmt.start;
	  or
	  delay 0.0;
	  end select;
	  end loop;
	  exception
	  when timeout =>
	  abort within_stmt;
	  <stmt2>
	  end timecheck;
	  task body within_stmt is
	  begin
	  loop
	  select
	  accept start;
	  <stmt1>
	  or
	  terminate;
	  end select;
	  end loop;
	  end within_stmt;
	begin
	  null;
	end;

every <time> do
  <stmt>
end
	declare
	  time_interval : duration:= t;
	  cur_time : calendar.time;
	  next_time : calendar.time;
	begin
	  loop
	  delay next_time - cur_time;
	  <stmt>
	  next_time :=
	  next_time+time_interval;
	  cur_time := calendar.clock;
	  end loop;
	end;

entrycall <event_term> when <time> -> <stmt> end	select <class_id>.x.a; or delay t; <stmt> end select;
entrycall <event_term> when balking -> <stmt> end	select <class_id>.x.a; else <stmt> end select;
wait <time>	delay <time>;
priority(<priority_num>)	pragma priority(<priority_num>);

용이 된다. 객체 행위 명세에서의 "wait t"는 t 시간단위 동안의 프로세스 실행 연기를 의미하기 때문에 이 행위명세는 Ada의 "delay t" 코드로 직접 변환가능하다. 그리고, every 문의 의미는 일정 시간간격으로 계속 동일 오퍼레이션을 반복수행하는 의미이므로 <표 4>에서와 같이 Ada의 delay 문과 next_time을 이용하여 구현한다. within 문의 경우에는 timecheck 태스크와 within 내의 행위를 수행시켜주는 두 개의 태스크에 의해 수행된다. within_stmt 태스크가 수행하는 도중에 타임아웃이 발생하게 되면 즉시 timecheck 태스크의 stmt2가 수행되고 within_stmt 태스크가 종료하도록 되어 있다. 그리고, timeout 동기화 사건의 경우에는 "select ... or delay t; ... end select;"를 이용하여 구현되었으며, balking의 경우에는 "select .... else ... end select;" 문을 이용하였다.

5. 적용실험

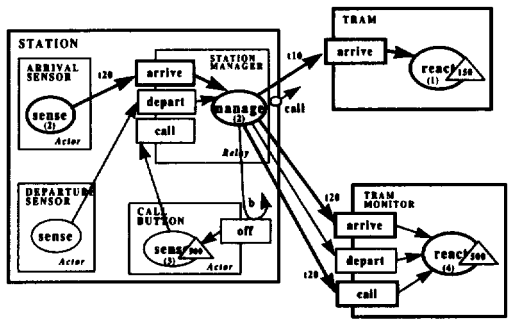
5.1 실험예제 : 순환궤도열차 시스템

본 논문에서는 객체행위해석기의 유용성을 판단하기 위하여 아래와 같은 간단한 실험 예제에 적용하였다.

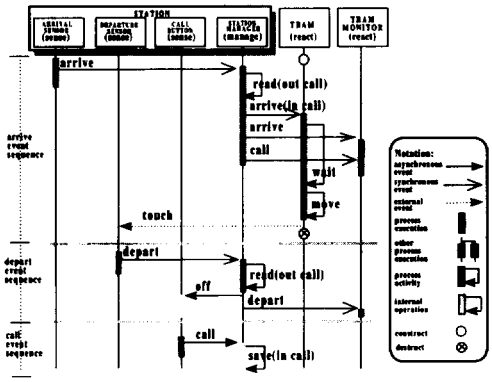
순환궤도 열차 시스템(Tram System)은 열차(Tram), 순환궤도(Track), 몇 개의 역(Station)들 및 열차감시 시스템(Tram Monitor)들로 구성되어 있는 시스템이다. 열차는 궤도를 따라 한 방향으로 움직이며 열차 내부에 정지버튼(Stop Push Button)이 있고, 역에는 열차 호출버튼(Call Push Button)이 있어서 승객(Passenger)이 정차(Stop) 혹은 승차(Taking)하고 싶을 때 그 버튼(Button)을 누르게 되면, 역을 지나가는 열차는 잠시 정차 후 자동적으로 출발한다. 역에는 열차가 지나가는 것을 인식하기 위해서 열차감지 센서(Sensor)가 있으며, 순환궤도열차 시스템에서 이루어지는 모든 움직임은 열차감시 시스템실(Tram Monitor Room)로 전달되어 통제가 이루어질 수 있도록 되어있다.

5.2 시각적 실시간 객체모델 및 정형명세

논문 [3]에서 제시한 "객체행위 설계기법"을 적용하여 객체구조도와 객체행위차트를 작성하면 다음 (그림 6, 7)에서와 같다. (그림 6)에서 순환궤도 열차시스템에는 STATION, TRAM, TRAM MONITOR와 같은 세 가지 주요 객체들로 구성되어 있고, STATION 객체에는 또다시 4가지 서브 객체들로 구성되어 있음을 볼 수가 있다. (그림 7)에서 볼 때, 이 시스템에는 도착감지처리, 출발감지처리, 그리고 열차 호출의 3가지 주요 사건들로 구성되어 있다. 이 순환궤도 열차시스템의 설계에 있어서 호출버튼에 의한 사건의 경우에는 비동기적인 사건형태로 설계되었다.



(그림 6) 순환궤도 열차 시스템의 실시간 객체구조도



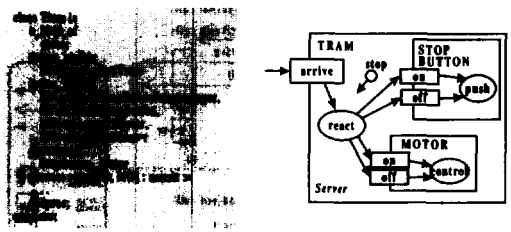
(그림 7) 순환궤도 열차 시스템의 객체행위챠트

(그림 6)의 순환궤도 열차 시스템의 객체구조모델로부터 STATION 시스템의 명세서를 작성해보면 다음과 같다. 이 시스템 명세서에서 보듯이 STATION 서브시스템에서는 STATION, ARRIVAL SENSOR, DEPARTURE SENSOR, CALL BUTTON, STATION MANAGER와 같은 객체들로 구성되어 있음을 알 수 있다.

```
system STATION_SYSTEM [StTr, StTm] : noexit :=
  (* standard library & information type definition *)
    library text_io, ada_io endlib;
    type Bool is (t, f) endtype;
    subtype Call_t is Bool endtype;
    c : Call_t;
    ...
  (* system behavior definitions *)
    behavior
      ...
      STATION[[StTr,StTm]|TRAM[[StTr,StTm]|TRAM_MONITOR
      ...
```

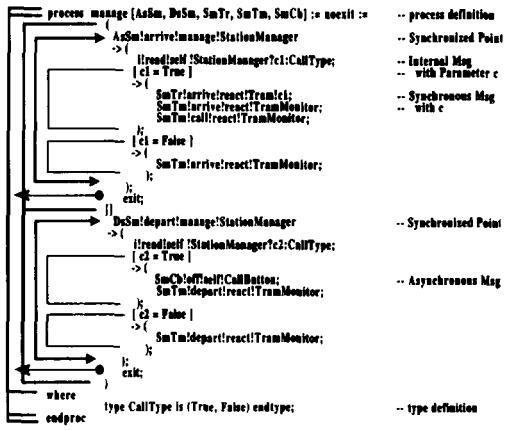
```
...
where
  class STATION is ... endclass;
  class ARRIVAL_SENSOR is ... endclass;
  class DEPARTURE_SENSOR is ... endclass;
  class CALL_BUTTON is ... endclass;
  class STATION_MANAGER is ... endclass;
  class TRAM is ... endclass;
  class TRAM_MONITOR is ... endclass;
endsys
```

객체구조모델에서 합성클래스는 다른 클래스들을 구성품으로 갖고 있는 클래스로 정보를 은닉하고 있는 모듈화 의미를 담고 있다. 다음 Tram의 경우는 StopButton과 Motor를 프리미티브 객체로 갖는 합성클래스이다. 이 합성클래스의 이러한 모듈화의 의미를 ToyLotos 모델 명세 언어로 표현해보면 아래에서처럼 intern_visible문과 containing문을 사용하여 바꾼다. 아래 객체 예에서 외부에 보여줄 수 있는 메시지는 arrive이고, 내부에서 직접 볼 수 있는 메시지들은 각각 StopButton의 on, off와 Motor의 on, off 메시지들이다.



(그림 8) 객체 클래스 명세

(그림 7)과 같은 객체행위챠트에 표현되어 있는 객체들간의 상호작용들을 각각의 객체 프로세스들에 대하여 순차적으로 나열하여 표현한다. 이 때, (그림 9)의 프로세스 명세에서 처럼 동기적/비동기적 메시지 스키마에 따라 객체행위챠트의 메시지들을 차례대로 구분하여 표현하며, 프로세스의 기능특성을 구분하여 반복적인 형태(noexit)와 비반복적인 형태(exit), 그리고 주기적인 형태(every)로 구분하고, 외부 요청 메시지에 대한 응답 행위들(g!e!p!o!para...-> (...))을 필요에 따라서는 선택적 행위 시맨틱스([])로 표현한다. 그리고 경우에 따라서는 조건문([condition]->(...))을 첨가하여 완전한 프로세스 로직을 기술한다. 예를 들어, (그림 7)의 객체행위챠트에서 STATION MANAGER의 manage 프로세스는 arrive라는 메시지와 depart라는 메시지에 의해 선택적 행위를 수행해야 하므로 (그림 9)의 프로세스 명세에서 보듯이 선택적 행위 시맨틱스를 갖도록 하였다.



(그림 9) 프로세스 명세(초기버젼)

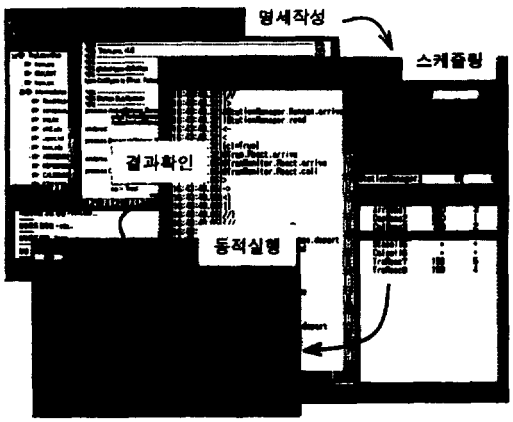
5.3 실행(시뮬레이션)

ToyLotos 시뮬레이션 시스템의 Ada 변환엔진을 이용하여 Lotos 명세 언어로 작성된 행위정의 구문을 Ada 코드로 자동 변환한 후 이것을 시뮬레이터에게 전달하여 실제로 실행시켜 봄으로써 모델의 수행성능 및 기능성을 검증한다. (그림 9)의 프로세스 명세는 ToyLotos 시뮬레이션 시스템에 의해 자동으로 다음 Ada 코드에서와 같이 시뮬레이션 모니터링 코드(elh_exe(*))가 첨가된 코드로 변환되며, 궁극적으로 수행가능한 형태의 실행코드로 변환된다.

```
1  Task body StationManager_Manage is
2    c1:Call_t;
3    c2:Call_t;
4  begin
5    loop
6      select
7        accept Arrive do
8          elh_exe(6,"@Arrive");
9          elh_exe(6,"@end");
10         StationManager_self.Read(c1);
11         if c1=t then
12           elh_exe(6,"@Arrive");
13           Tram_React.Arrive(c1);
14           elh_exe(6,"@Arrive");
15           TramMonitor_React.Arrive;
16           elh_exe(6,"@Call");
17           TramMonitor_React.Call;
18         elsif c1=f then
19           elh_exe(6,"@Arrive");
20           TramMonitor_React.Arrive;
21         end if;
22       end Arrive;
23     or
24       accept Depart do
```

```
25          elh_emit(6,"@Depart");
26          elh_emit(6,"!Read");
27          StationManager_self.Read(c2);
28          if c2=t then
29             elh_emit(6,"@Off");
30             CallButton_self.Off;
31             elh_emit(6,"@Depart");
32             TramMonitor_React.Depart;
33          end if;
34          if c2=f then
35             elh_emit(6,"@Depart");
36             TramMonitor_React.Depart;
37          end if;
38       end Depart;
39    end select;
40  end loop;
41 end StationManager_Manage;
```

또한, 실시간 시스템의 개발에서 있어서 프로세스 스케줄링은 중요한 작업중의 하나인데, 프로세스 스케줄링을 위해서 주기적인 프로세스, 비주기적인 프로세스, 그리고 시간제약이 필요 없는 배경 프로세스들을 종합적으로 고려하여 프로세스들의 수행순서, 병행성, 우선 순위, 주기 및 마감시간의 배정이 필요하다. 이러한 작업을 지원하기 위하여 ToyLotos 시뮬레이션 시스템에서는 (그림 10)과 같은 도구를 이용하여 프로세스들의 우선 순위와 주기를 자유자재로 변경시키면서 시스템 성능을 동적으로 관찰할 수 있게 하였다. 이 실험에서는 프로세스 스케줄링을 위해 레이트모노토닉 알고리즘을[13] 적용하여 주기가 적을수록 우선 순위를 높게(우선 순위 값을 크게) 배정하였다. (그림 10)에서 볼 때, Tram과 TramMonitor 객체의 프로세스 우선 순위가 높고, 그 다음이 Arrival Sensor, Call Button, Departure Sensor의 순이다.



(그림 10) 프로세스 스케줄링 및 동적실행

그리고, 그 실행이력화일에 수집된 수행 결과는 "전체보기 윈도우"에 표시되는데 동기적 사건의 발생(@)과 그 사건의 입수(&), 비동기적 사건의 발생(@)과 실행(?), 그리고 내부 사건의 발생(!)과 실행(?) 등의 수행결과를 시간순서로 보여주고 있고, 그 실행이력의 좌측에는 사건 발생 시점의 현재시각을 표시하고 있으며, 첫 번째 줄에는 프로세스 명칭이 나타나 있다.

(그림 11)과 같은 실행이력을 분석해 봄으로써 여러 프로세스들에서 시간에 따라 적절히 메시지를 발생시키고 있는 지와 수행과정이 적절한지, 그리고 마감시간에 충족되는지를 판단한다. 프로세스들의 수행성능은 메시지들의 생성 시간간격을 계산함으로써 분석하고, 참여하는 프로세스들의 수행순서를 살펴봄으로써 프로세스 스케줄링이 적절한지를 판단한다.

time	AS1	DS2	CS3	TR4	TM5	SM6
...			...	...	...	
11:18:41.21			@Call			
11:18:41.21						
11:18:41.70	① @Arrive					
11:18:41.70						② &Arrive
11:18:41.70						③ !Read
11:18:41.70						
11:18:41.70						④ @Arrive
11:18:41.70				⑤ &Arrive		
11:18:41.70				⑥ !Wait		
11:18:41.70						
11:18:41.70		⑧ @Depart				
11:18:41.98				⑦ !Move		
11:18:41.98						
11:18:41.98						**@Arrive**
11:18:41.98					**&Arrive**	
11:18:41.98					**!Arrive**	
11:18:41.98						
11:18:41.98						**@Call**
11:18:41.98					**&Call**	
11:18:41.98					**!Call**	
11:18:41.98						
11:18:41.98						&Depart
11:18:41.98						!Read
11:18:41.98						
11:18:41.98						**@Depart**
11:18:41.98					**&Depart**	
11:18:41.98					**!Depart**	
11:18:41.98						
11:18:42.20	⑨ @Arrive					
11:18:42.20						&Arrive
...			...	...	...	

(그림 11) 실행이력화일(elh.out)

예를 들어, (그림 11)의 실행결과에서 ARRIVAL SENSOR (AS1)의 Arrive 감지신호 생성(@)은 ① (41.70초)과 ⑨(42.20초) 사이 즉, 0.50초마다 발생함을

알 수 있다. 그리고, 이 예의 경우, 다음과 같은 비정상적인 시점에서의 행위에 의한 오동작에 대한 적절한 조치가 없음을 발견할 수 있다.

- STATION에 TRAM이 도착하여 순환궤도 열차 시스템의 STATION 서브시스템에서 Tram 도착 감지 처리(①-⑦의 처리)가 종료되지 않았는데, DEPARTURE SENSOR에 의해 Tram 출발(⑧)이 감지가 되었을 경우와 같이 예외적인 상황에 대한 적절한 조치가 없다.

6. 관련연구 및 Lotos 지원도구들과의 비교

정형화 기술 초기의 프로토콜 명세 언어는 대부분 FSM(Finite State Machine) 방법에 기반한 것이었으나, 곧이어 EFSM(Extended Finite State Machine)에 기반한 언어 예를 들면 SDL, ESTELLE가 나왔으며, 현재에는 OSI 프로토콜 및 서비스 정의를 위해 Lotos와 같은 언어가 제시되었다.

Lotos는 EFSM에 기반한 언어들과는 다른 개념을 갖고 있다. 기본적으로 OSI 표준 프로토콜을 정의하게 만들어졌으며, 프로토콜 뿐만 아니라 OSI의 서비스에 대해서도 기술할 수 있게 만들어졌다. 그리고, Lotos의 추상화 능력 때문에 구현에 독립적인 명세를 생성할 수 있으며, Lotos의 강력한 수학적 배경 때문에 정밀한 정형 시맨틱스를 갖는다.

<표 5>는 본 논문에서 제안한 ToyLotos와 기존 Lotos 지원도구들과의 비교·평가를 위해 정형도구의 11가지 지원 기능 측면에서 도구들의 특징을 요약한 것이다. 이 표에서 보듯이, 현재에는 11가지 기능 모두를 지원하는 도구는 존재하지 않고 있으며 프로토콜 명세의 정확성을 테스팅하기 위한 자동 테스트 케이스 생성 부분의 지원을 위한 기능 측면에서는 소수의 도구밖에 없는 것으로 조사되었다. 그리고, 이 표에서 보는 모든 도구들은 상용화된 도구들이 아니며 학교나 연구소 등에서 기술 실현을 위한 실험용 도구로 개발된 것으로 기존의 CASE 도구나 다른 모형화 지원도구에서 처럼 많은 실용화 노력이 필요한 것으로 밝혀졌다.

각각의 도구들의 세부 특징에 대하여 살펴보면 다음과 같다. 우선, LOLA(LOtos LAboratory[14,15])는 스페인 마드리드 대학에서 개발된 Lotos를 위한 실험환경으로 제공된 변환도구이다. 이 도구는 처음에는 Pascal로 설계되었으며, 현재에는 C로 변환 되어있다. 이 도구는 명세의 확장 및 인수확장 시뮬레이션, 단계 시뮬레이션, 그리고 내부행위 루프의 제거와 같은 기능들을 제공한다. TOPO[14,16]도 LOLA와 마찬가지로 스페인 마드리드 대학에서 개발된 도구로 Lotos 명세서로부터 실제 산출물을 개발할 때 지원하기 위한 도구로 만들어 졌다. 이 도구는 명세의 의미분석 및 자료타잎 의존성 분석, 상호참조 기능들을 제공한다. 그리고, 프로토타잎으로 C 혹은 Ada 코드를 생성한다. TOPO는 자료타잎과 행위를 컴파일하기 위한 컴파일러로써 제공된다.

〈표 5〉 Lotos 지원도구들의 기능 비교

지원기능 \ 지원도구	LOLA	TOPO	SMILE	LITE	ISLA & SVELDA	ToyLotos
명세편집						○
구문 및 의미 검사		○			○	○
시뮬레이션	○		○		○	○
변환	○		○		○	○
도구관리				○		○
테스트생성						
테스트실행	○					
검증	○					
컴파일		○			○	○
방법론지원						○
프로토타입 생성		○				○

SMILE(SyMbolic Interactive Lotos Execution)[14,17]은 네덜란드 트웬티 대학에서 개발된 도구로써 Lotos를 위한 심볼릭 평가 도구이며 Lotos 명세가 가지고 있는 구조를 해석하여 시뮬레이션 트리를 구성한 후 각각의 노드에 대해 평가하는 작업을 사용자와 대화식으로 할 수 있게 한다.

LITE(Lotos Integrated Tool Environment)[14,18]는 일종의 Lotos를 위한 통합도구 환경으로 ESPIRIT 프로젝트에서 여러 가지 단위도구들을 하나로 통합하기 위해 특별히 만들어졌다. 이 도구는 윈도우 기반 사용자 인터페이스를 제공한다. 여러 가지 다양한 도구들을 하나로 통합하기 위해 공통내부표현(Common Internal Representaion)이라는 단일 내부표현을 갖고 있다.

ISLA(Interactive System for Lotos Application)[19]와 SVELDA(System for Validating and Executing

System for Lotos Data Abstractions)[20]는 캐나다 오타와 대학에서 개발한 Lotos 기반 해석기로 각각 Lotos 행위해석기와 ADT 해석기이다. 이 해석기는 Lotos 문법과 시맨틱스에 따라 Lotos 명세서를 점검하고, 명세서가 정확하다고 밝혀지면 동일한 시맨틱스를 가진 프롤로그 언어로 변환시켜 수행시킨다. 이 도구의 이러한 기능들은 모두 C로 작성이 되었다.

마지막으로, 본 논문에서 제안한 ToyLotos는 LITE와 같이 기존 도구들을 관리하기 위한 환경으로 개발되었으며, TOPO나 ISLA&SVELDA에서 처럼 기본적으로 Lotos 문법 및 정적 시맨틱스를 점검하여, 만족된 명세에 대하여 Lotos 시맨틱스를 특정 언어(Ada)로 변환하여 시뮬레이션할 수 있게한다. 그리고 또한, 다른 도구들과 다르게 특정 방법론 기반 지원도구로써 기존 방법론 기반 시각적 실시간 객체모델을 위해 만들어졌다. 이것은 시스템 모듈화 및 추상화 능력을 높게하여 개발자와 사용자 모두에게 시스템 이해 및 생산성을 높이는데 기여할 수 있겠다.

7. 결론 및 향후연구 방향

본 논문에서는 객체행위 설계 방법에 의해 개발된 시각적 실시간 객체 구조 및 행위 모델의 설계의미를 검증·확인하기 위한 ToyLotos/Ada 시뮬레이션 시스템을 제시하였다.

이 시스템은 시각적 객체 구조 및 행위 모델로부터 명세서를 생성하여 이를 Ada 코드로 변환하여 객체프로세스들간의 동적인 상호작용을 시뮬레이션할 수 있게 한다. 그리고, 실제 시스템 구현에 앞서 여러 가지 논리적·시간적 이상행위들을 동적으로 시뮬레이션하여 검출할 수 있도록 지원한다.

또한, "순환궤도 열차 시스템"이라는 간단한 사례연구를 통해 적용해본 결과로는 시각적 실시간 객체모델을 정형명세언어를 이용하여 명세화하고, 이를 Ada로 변환하여 시뮬레이션할 수 있었다. 그러나, 아직까지는 수행결과가 문자형태로 저장이 되어있기 때문에 개발자가 분석하기에는 좋은 형태가 아닌 것으로 나타났다. 그렇지만, 수행결과를 각각의 개별 객체별로 살펴보게 함으로써 시스템 이해에 도움이 되었다. 그리고, 기존 Lotos 지원도구들과의 비교에서 볼 때 다른 도구들과 마찬가지로 아직까지는 상용화된 도구는 아니지만, 지원도구가 가지는 여러 가지 지원기능을 많이 갖고 있으며, 특히, 다른 도구와 다르게 특정 방법론 기반 시각적 실시간 객체모델에 근거한 도구이며, 아울러 시간이 첨가된 Lotos의 실행모델이라는 것이다.

그러나, 지금은 Lotos 명세로부터 Ada코드만을 생성하여 실행시키는 시스템으로 구성되어 있는데 이를 C++혹은 Java와 같이 널리 응용이 되고 있는 코드를 생성하도록 확장이 필요하다. 그리고, 현 시스템은 모델의 시맨틱스에 맞는 실제 실행코드 생성에 의해 시뮬레이션을 하지만, 향후에는 가상의 시뮬레이션 모델, 예를 들면 확장된 유한상태기계(EFSM)와 같은 가상기계에 의해 시스템의 시맨틱스를 분석하는 기법의 연구도 필요하다. 그리고, 현재 기존 표준 Lotos에 시간 개념을 추가하는 연구들이 Lotos 개발 부류들에서 최근 수년간 연구하고 있는데 우리도 이러한 부분의 기술력 확보에 노력을 기울여야할 것으로 판단된다.

참 고 문 헌

[1] James Rumbaugh, Michael Blaha, Wiliam Premeriani, and Fredrick Eddy, William Lorensen, Object Oriented Modeling and Design, Prentice-Hall, 1991.

[2] Ward, P. T. and Mellor, S. J., Structured Development for Real-Time Systems, Vol. I, II, III, Yourdon Press, 1985.

[3] 이광용, 정기원, "실시간 Ada 소프트웨어 개발을 위한 객체행위설계 방법", 정보과학회 논문지(B), 제24권, 제1호, 1997년 1월, pp.43-61

[4] ISO, IS 8807, Information Processing Systems-Open Systems Interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behavior, May 1989.

[5] Mark A. Ardis, et al., "A Framework for Evaluating Specification Methods for Reactive Systems *Experience Report*," IEEE Transaction on Software Engineering, Vol.22, No.6, Jun. 1996, pp.378-389

[6] Angelo Morzenti and Pierluigi San Pietro, "An Object-Oriented Logic Language for Modular System Specification," 5th European Conference on Object-Oriented Programming, Jul. 1991, pp. 39-58

[7] Angelo Morzenti and Pierluigi San Pietro, "Object-Oriented Logical Specification of Time-Critical Systems," ACM Transaction on Software Engineering and Methodology, Vol.3, No.1, Jan. 1994, pp.56-98.

[8] Ivar Jacobson, Object-Oriented Software Engineering : A Use Case Driven Approach, Addison-Wesley, 1992.

[9] Henk Eertink, Simulation Techniques for the Validation of LOTOS Specification, Proefschrift Enschede, The Netherlands, 1994.

[10] Shem-Tov Levi and Ashok K. Agrawala, Real-Time System Design, McGraw-Hill, 1990.

[11] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[12] C.A.R. Hoare, "Communicating Sequential Processes," Communication of the ACM, Vol.21, No. 8, Aug. 1978, pp. 666-677.

[13] Lui Sha and John B. Goodenough, "Real-Time Scheduling Theory and Ada," IEEE Computer, Apr. 1990, pp.53-62.

[14] L. Moonen and A.Ebrahim, "Overview of the specification language LOTOS," Dec. 1994, Available by web server from HTTP://wwwtios.cs.utwente.nl/lotos/#paper

[15] Juan Quemada, Santiago Pavon, Angel Fernadez, "Transforming LOTOS Specifications With Lola: The Parameterized Expansion," FORTE'88, North-Holland, 1988, pp.45-54.

[16] J. A. Manas, Tomas de Miguel, Joaquin Salvachua, and Arturo Azcorra, "Tool support to implement LOTOS formal specifications," In J. Quemada, editor, Special Number on Tools for FDTs, Vol.25(7). Computer Networks and ISDN systems, 1993.

[17] P. van Eijk and H. Eertink, "Design of the Lotosphere Symbolic Lotos Simulation," FORTE'89, Madrid, Spain, North-Holland, Nov. 1989, pp. 577-580

[18] P. van Eijk, "The Lotosphere Integrated Tool Environment *Lite*," FORTE'91, Australia, Nov. 1991, pp.471-474

[19] M. Haj-Hussein, An Interactive System for LOTOS Applications(ISLA), Master's thesis, University of Ottawa, Ottawa, Canada, Nov. 1988

[20] Pierre Ashkar, Symbolic Execution of LOTOS Specifications, Master's thesis, University of Ottawa, Ottawa, Canada, Jan. 1992

이 광 용

e-mail : kylee@etri.re.kr

1991년 숭실대학교 전자계산학과 졸업(학사)

1993년 숭실대학교대학원 전자계산학과(공학석사)

1997년 숭실대학교대학원 전자계산학과(공학박사)

1996년~1998년 숭실대학교 생산기술연구소 연구원

1997년~1998년 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 박사후연수연구원(Post-Doc.)

1999년~현재 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 선임연구원

관심분야 : 소프트웨어공학, 실시간시스템, 객체지향 모델링/시뮬레이션/디버깅, 정형기법, 분산처리, 인공지능

오 영 배

e-mail : yboh@etri.re.kr

1982년 고려대학교 기계공학과 졸업(학사)

1995년 인하대학교 전자계산공학과(공학석사)

1995년~1997년 미국캘리포니아대학(UCI) 객원연구원

1999년 정보처리기술사

1983년~현재 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 선임연구원, 영역기반소프트웨어재사용기술 과제책임자

관심분야 : 컴포넌트기반소프트웨어공학, 소프트웨어재사용기술, 정형기법, 실시간시뮬레이션