LOTOS 명세로부터 C++ 소스코드의 자동 생성

김 철 홍[†] · 천 윤 식[†] · 김 강 호[††]

요 약

정보통신용 소프트웨어 개발은 대용량, 고신뢰도, 고복잡도, 이질성 및 분산 환경이라는 특징을 가지고 있다. 이러한 시스템을 개발함에 있어서 개발 품질 및 적정 비용을 유지하기 위해서는 향상된 정형명세 기법과 이러한 기법을 지원하는 도구가 필수적이다. ISO 표준 정형 명세 언어인 LOTOS는 사용자의 요구사항이나 시스템 모형을 추상적이고 정형적으로 작성할 수 있도록 한다. 반면, 명세로부터 구현을 직접 도출하기는 쉽지 않다. 본 연구는 LOTOS 명세 언어로 작성된 명세로부터 C++ 코드를 자동 생성하여, 이를 PC 플랫폼상에서 실행하는 동작 모형을 통하여 시스템의 기능적 요구사항의 오류를 초기에 검출할 수 있는 프로토타이핑을 지원하는 LOTOS/C++ 코드 생성기(code generator)를 개발하는 것이다.

Automatically Generating C++ Source Code from LOTOS Specifications

Cheol-Hong Kim[†] · Yoon-Sik Cheon[†] · Kang-Ho Kim[††]

ABSTRACT

Communication systems can be characterized by high reliability requirement, complexity in development, and distribution of processes and resources. In developing these kinds of systems, formal methods can be fruitful in achieving high quality of resulting systems. LOTOS, a formal specification language, allows us to write models of the system in an abstract but rigorous manner. In this paper, we present a "code generator" that supports rapid prototyping by generating C++ code from LOTOS specifications. Execution of the generated code allows us to detect analysis and design errors at the early stage of system development life cycle.

1. 서 론

수학 기반의 엄격한 소프트웨어 개발 방법인 정형 기법은 최근 산업계에서 소프트웨어 개발에 실제 적용되기 시작하고 있다. 특히, 요구분석과 같은 개발 초기 단계에 가장 효과적인 것으로 보고되고 있다[2]. 요구분석의 오류가 설계 및 구현으로 전달되는 파급 효과를 방지할 수 있기 때문이다. 소프트웨어 오류의 80% 이상이 요구사항의 잘못된 이해에 기인 한다는 사실을 감안할 때, 바람직한 접근 방법이다. 하지만, 정형 기법이 효과적으로 적용되기 위해서는 수학적 소프트웨어 개발을 지원하는 지원 도구와 개발 환경이 필요하다. 특히, 정형 증명과 같은 이론 지향적인 도구보다는 실용적인 -- 예를 들면, 사용자의 요구사항이 명세에 제대로 반영되었는지 파악할 수 있는 확인 위주의 정량 도구가 필요하다. 코드 생성과 명세 시뮬레이션이

† 정 회 원 : 한국전자통신연구원 컴퓨터 · 소프트웨어기술연구소 소프트웨어공학연구부 선임연구원
†† 정 회 원 : 한국전자통신연구원 컴퓨터 · 소프트웨어기술연구소 소프트웨어공학연구부 연구원
논문접수 : 1998년 7월 13일, 심사완료 : 1998년 10월 9일

이런 부류에 속한다.

코드 생성이란 명세를 만족하는 실행 가능한 프로그램을 명세로부터 도출하는 것을 말한다. 예를 들어, L를 LOTOS 명세[3], C를 C++ 프로그램이라고 하자, LOTOS 명세 언어의 의미 함수 $M:L \rightarrow 2^C$를 다음과 같이 정의할 수 있다.

$$M(l) \triangleq \{c \in C \mid c \text{ satisfies } l\}$$

즉, 명세 l를 만족하는 모든 C++ 프로그램을 l의 의미로 정의한다. 따라서 코드 생성이란 임의의 명세 l이 주어 졌을 때 이를 만족하는 C++ 프로그램 $c \in M(l)$를 선택하는 행위로 볼 수 있다. 현실적으로는 기능적 요소뿐만 아니라 자원과 효율성 같은 비기능적 요소도 물론 고려되어야 한다.

일반적으로 코드 생성의 가장 큰 목적은 명세의 동작 모형이다. 명세를 실제 실행해 봄으로써 다양한 형태로 명세의 특성을 검사하고 분석할 수 있고, 명세의 작동 상황을 가시화 할 수도 있다. 명세의 동작 모형이 실제 구현 언어로 주어진다면, 실제 구현에도 사용할 수 있으며, 효율성 향상을 위해 조율이 필요할 수도 있다. 미 구현 모듈의 테스팅 스텁으로도 사용 가능하다. 최근 경향은 코드 생성을 기반으로 명세의 검증 및 증명 도구를 개발하는 추세이다. 따라서 코드 생성은 정형 지원 도구의 핵심 모듈로 사용될 수 있다.

대부분의 코드 생성기는 Prolog나 SML과 같은 논리 언어나 함수 언어를 사용한다. 하지만 단순한 동작 모형이 아니라 산업계 개발자가 실제 사용하기 위해서는 C/C++와 같은 실제 구현 언어로 주어져야 한다. 아직까지 C++를 지원하는 도구는 찾아보기 어렵다. 개발자가 C++로 이동하는 추세일 뿐만 아니라 객체 지향은 코드 생성에 많은 혜택을 줄 것으로 기대된다. 예를 들면, 생성된 코드의 모듈화, 재사용, 상속을 통한 조율과 수작업 코드와 연계 등을 들 수 있다.

본 논문에서는 서론에 이어 제 2장에서는 LOTOS에 대해 소개하고, 제 3장에서는 코드생성기와 관련된 연구를 소개하고, 제 4장에서는 LOTOS 명세로부터 자료부분의 코드를 생성하는 자료 컴파일러와 행위부분의 코드를 생성하는 행위 컴파일러의 설계와 구현을 설명한다. 마지막으로 제 5장에서는 코드 생성기의 향후 연구과제와 결론으로 맺는다.

2. LOTOS

LOTOS(Language Of Temporal Ordering Specification)는 개방형 분산 시스템 -- 특히, OSI의 서비스와 프로토콜 -- 명세를 위하여 ISO에서 개발한 정형 명세 언어이다[4]. 외부와 일어나는 상호작용의 시간적 순서를 정의함으로써 시스템을 명세 할 수 있다는 철학에 바탕을 두고 있다. 이를 위하여 Milner의 CCS (Calculus of Communicating Systems)와 Hoare의 CSP(Communicating Sequential Processes)와 같은 프로세스 대수학을 도입하였다. 상호작용시 자료 교환이 일어날 수 있는데, 교환되는 자료를 표현하기 위하여 대수방식의 정형 명세 언어 ACT ONE을 사용한다. 요약하면, 자료와 제어의 상호보완적인 명세를 위해 ACT ONE과 CCS/CSP라는 두 정형기법을 결합한 것이다.

```
specification VMSPEC[c,f,r]: noexit
  library
    Boolean, NaturalNumber
  endlib
  type Coin is
    Boolean (*| extern |*), NaturalNumber
    opns _-_ (*| name minus |*): Nat, Nat -> Nat
    eqns forall m, n: Nat
      ofsort Nat
        (n - 0) = n;
        succ(n) - succ(n) = n - m;
  endtype
  behaviour
    VM[c,f,r](0)
  where
  process VM[c,f,r](n: Nat): noexit :=
      c?m:Nat[m eq succ(0)]; VM[c,f,r](n + succ(0))
    []
      f[n ge succ(0)]; VM[c,f,r](n - succ(0))
    []
      (*| delay 2 |*)
      r!n[n gt 0]; VM[c,f,r](0)
  endproc
endspec
```

(그림 1) LOTOS 명세
(Fig. 1) LOTOS Speccification

(그림 1)은 LOTOS 명세의 간단한 예를 보여준다. 동전 입력, 커피 판매, 동전 반환의 기능을 가진 아주 간단한 커피 판매기를 명세 한다. 그림에서 type 절이 대수 논리를 바탕으로 하는 자료 정의 부분이고, behaviour절과 process절이 프로세스 논리를 바탕으로 하는 행위 정의 부분이다.

3. 관련 연구

대표적인 LOTOS 코드 생성기는 스페인의 마드리드 대학에서 개발한 TOPO이다[5, 9.3절]. TOPO는 코드 생성을 두 단계로 구분한다. 먼저 명세로부터 가상 머신(Virtual Machine) 코드라 불리는 중간 코드를 생성하고 이를 C 또는 ADA 코드로 변환한다. 이러한 접근 방법은 역할을 분담하고 기능을 모듈화 하여 도구의 확장 및 유지 보수가 용이하다. 자료부분은 개서 시스템(rewrite system)을 바탕으로 하며 행위 부분은 유한 오토마타를 바탕으로 한다. TOPO는 LITE 도구에 포함되어 있다. LITE에 포함된 또 다른 코드 생성기는 독일의 베를린 대학에서 개발한 COLOS이다[5, 9.5절]. COLOS는 TOPO와 달리 행위부분만을 지원한다. 자료부분은 C로 사용자에 의해 정의되고 구현되어야 한다. COLOS가 TOPO에 비해 다소 빠른데, 이는 인식할 수 있는 행위 표현식에 제한을 두며 자료 부분이 수작업으로 작성되기 때문이다.

자료 부분을 처리함에 있어서 대부분의 LOTOS 지원 도구는 대수 등식에 따라 값 표현식을 동적으로 개서한다. 개서 엔진은 등식에 독립적일 수도 있고 TOPO에서처럼 등식의 특성을 충분히 이용할 수 있도록 종속적일 수도 있다. 개서 시스템을 사용하지 않은 방법으로는 추상 머신을 위한 코드 생성을 들 수 있다. 이 방법은 패턴 매칭 기능을 갖고 있는 함수 언어에서 유래하나 LOTOS에도 적용될 수 있다. LOTOS 자료부분을 C 코드로 변환하는 Caesar라는 도구에서는 기존에 비해 전혀 새로운 방법을 취하고 있어 흥미롭다[7]. 패턴 매칭 컴파일러 알고리즘을 사용하여 대수 명세를 C 언어의 자료구조와 프로시져로 정적 컴파일한다. 생성된 코드는 결정 코드이다. 즉, 컴파일시 개서 규칙의 적용이 완벽하게 결정되므로 실행시 unification이나 backtracking은 전혀 일어나지 않는다.

4. LOTOS/C++ 코드 생성기

본 코드 생성기의 구조는 LOTOS 의미론을 반영한다. LOTOS 명세는 사건이 일어나는 시간적 순서를 묘사하는 행위 정의 부분과 자료 값의 특성을 묘사하는 자료 타입 정의 부분으로 구성된다. 자료 값 표현식의 의미 해석을 위하여 행위 부분이 자료 부분을 참조한다는 것을 제외하면 두 부분은 완전히 독립적으로 동작한다. 이는 코드 생성기를 자료 부분의 코드를 생성하는 자료 컴파일러와 행위 부분의 코드를 생성하는 행위 컴파일러로 나누어 독립적으로 개발할 수 있도록 한다.

4.1 자료 컴파일러

4.1.1 이론적 배경

(그림 2)는 LOTOS 자료 명세에 관한 간단한 예이다. 일반적으로 자료 명세는 소트 정의, 연산자 정의, 자료 등식의 세 부분으로 구성된다. 먼저 추상 값의 집합을 나타내는 소트를 정의하고 연산자라 불리는 소트 상에 정의된 전칭 함수를 정의한다. 연산자의 의미는 연산자 상호 관계를 등식으로 서술한다.

```
type Stack is Boolean, Natural
sorts Stack
opns
  empty: -> Stack
  push: Stack, Nat->Stack
  pop: Stack->Stack
  top: Stack->Nat
  isempty: Stack->Bool
eqns
  forall s:Stack, n, m:Nat
    ofsort Nat
       top(push(s,n)) = n;
    ofsort Stack
       pop(push(s,n)) = s;
       pop(empty) = empty;
    ofsort Bool
       isempty(empty) = true;
       isempty(push(s,n)) = false;
endtype
```

(그림 2) LOTOS 자료명세
(Fig. 2) LOTOS Data Specification

ACT ONE 명세의 수학 모형은 다종 대수(many-sorted algebra)이다. 특히, 초기 의미론(initial semantic)을 갖는데[8], 이는 수학적으로 잘 정립되었으나 명세의 실행 모형으로 사용될 수 없다. 따라서 텀 개서 시스템(term rewrite system, TRS)을 사용한다. 명세의 등식으로부터 개서 규칙을 도출하고 개서 규칙에 따라 텀을 개서한다. 예를 들면, 연산자 pop을 제약하는 두 등식은 다음과 같은 개서 규칙으로 변환할 수 있다.

```
pop(push(s,n)) -> s;
pop(empty) -> empty;
```

개서 시스템은 등식에 종속 혹은 독립적인 동적 개서 엔진이나 패턴 매칭 기반의 정적 컴파일 방법으로 구현할 수 있다. 예를 들면, 소트 Stack을 C++ 클래스 Stack으로 구현한다고 가정할 때, 연산자 pop는 다음과 같은 형태의 C++ 맴버 함수로 구현할 수 있다.

```
Stack Stack::pop()
{
  switch (this) {
    case "of the form push(s,n)";
      return s;
      break;
    case "of the form empty";
      return empty();
    break;
  }
}
```

대수론과 개서 시스템의 의미론적 차이에서 오는 이론적 한계가 있다. 대수학의 등식은 방향성이 없는데 반해, 개서 시스템에서는 방향성을 부여한다. 결과적으로 함수 평가가 종결되지 않을 수 있고 동일한 값을 상이하게 해석할 수도 있다. 기술적 용어를 빌리자면, Noetherian 특성과 ChurchRosser 특성을 가진 완전한 개서 시스템이 요구된다.

4.1.2 목표 코드 형식

생성되는 C++ 프로그램은 어떤 형식을 가져야 하는가? 구문적인 측면에서 보면, 소트는 C++ 클래스로 구현하고 연산자는 기본적으로 해당 클래스의 맴버 함수로 구현한다. 예를 들면, 앞 절의 Stack 명세는 다음과 같은 C++ 헤더 파일을 생성한다. (CVRef는 동적 메모리 관리를 위한 주형 클래스이다.)

```
CVRef<Stack> empty();
class Stack: public KSort {
public:
  Stack(int oid = 0, KLink *p = (KLink*) NULL) :
    KSort(oid,p) {}
  CVRef<Bool> isempty();
  CVRef<Nat> top();
  CVRef<Stack> pop();
  CVRef<Stack> push(const CVRef<Nat>&);
};
```

따라서 LOTOS 값 표현식 pop(push(empty, 0))에 해당하는 C++ 문장은 다음과 같다.

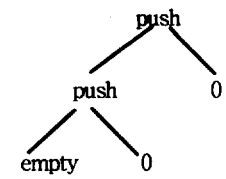
```
CVRef<Stack> s = empty()->push(zero())->pop();
```

프로그램 모듈화를 위해 생성된 모든 클래스는 커널 클래스라 불리는 시스템 클래스 KSort의 서브 클래스로 정의한다. 따라서 실행 가능한 완전한 C++ 프로그램은 자동 생성 코드와 커널 클래스가 결합된 형태이다. 커널은 자동 생성 클래스의 공통된 특성을 정의한다. 예를 들면, 추상 값을 구현하기 위한 자료 구조나 이를 조작하기 위한 함수를 제공한다. 특히, 내부 자료구조의 초기화, 대수명세의 초기 모형 구현, 추상 값의 스트링 변환, 동적 메모리 할당과 같은 서비스를 제공하며, 자동 생성 클래스의 조율과 수작업 코드와 연계를 위한 메카니즘도 제공한다.

4.1.3 추상 값 구현

추상 값을 어떻게 구현할 것인가? 코드 생성, 크게 보면 프로그래밍이란 추상 명세를 실행 가능한 형태로 구체화하는 과정이다. 두 가지 측면에서 구체화가 일어난다. 명세의 추상값이 C++ 자료구조로 표현되어야 하고 연산자가 C++ 함수로 변환되어야한다.

대수 명세에서 추상값은 값 표현식 --즉, 해당 소트를 결과 소트로 갖는 연산자의 적용으로 표현된다. 예를 들면, push(push(empty, 0),0)은 소트 Stack이라는 집합에 속하는 추상값이다. 추상 자료 값을 구현하는 방법은 다양하나 가장 일반적인 방법은 추상 구문 트리(abstract syntax tree, AST)를 사용하는 것이다. 예를 들면, 추상 값 push(empty,0))에 해당하는 추상 구문 트리는 다음과 같다.



구현 값 표현을 위해 모든 연산자에 식별자라는 고유 번호가 부여된다. 추상 문법 트리 구축을 위한 자료 구조와 관련 함수는 커널에 정의된다. 생성된 모든 클래스는 커널 클래스의 서브 클래스로 정의되어 이를 상속받는다.

4.1.4 연산자 변환

연산자는 C++의 맴버 함수로 변환된다. 자료 명세의 등식으로부터 개서 규칙만 도출되면, 맴버 함수로의 변환은 기계적으로 처리할 수 있다. 예를 들면, Stack 명세의 pop 연산자는 다음과 같이 변환된다. (switch 문의 oid()는 연산자 식별자를 나타낸다.)

```
CVRef<Stack> Stack::pop()
{
  CVRef<Stack> _v;
  switch (this->oid()) {
    case 15: /* push */
      _v = ((Stack*)this->arg()->val.deref());
      break;
    case 14: /* empty */
      _v = empty();
      break;
  }
  if (!_v) {
    KLink *_sl;
    _sl = new KLink(this);
    _v = new Stack(16,_sl);
  }
  return _v;
}
```

코드 생성의 핵심은 자료 등식으로부터 개서 규칙을 도출하는 것이다. 가장 일반적인 방법은 앞 절의 예에서와 같이 등식에 방향성을 부여하여 개서 규칙으로 간주하는 것이다. Caesar.ADT와 TOPO가 이 방법을 채택하고 있다. 결과적으로 생성된 개서 시스템이 원하는 특성을 가지지 않을 수도 있다. 이는 명세자의 책임이다. 원하는 특성이라 함은 시스템의 완전성을 말한다. 완전성 보장을 위해서는 Knuth-Bendix 알고리즘이나 이의 확장을 사용하여야 한다.

등식을 개서 규칙으로 간주하는 방법은 몇몇 제한 사항을 초래한다. 예를 들면, 등식의 조건절과 우변에 나타나는 모든 변수는 반드시 좌변에 나타나야 한다. LOTOS 명세에서는 전칭 변수가 사용될 수 있지만 개서 규칙을 구현하는 동작 모형은 전칭 변수를 포함할 수 없기 때문이다.

4.1.5 구현 종속 사항

코드 생성이란 실행 가능한 코드를 산출하기 위하여 추상 명세에 구현 종속적인 결정 사항을 첨가하는 행위라고 볼 수 있다. 구현 종속 사항은 어휘 관습과 같은 단순히 구문적인 요소에서부터 부분 구현과 같은 의미적인 요소까지 포함한다. 구현 종속적인 사항을 명세하기 위하여 주석을 사용한다[5][6]. 주석은 특수한 형태의 코맨트로 명세에 첨부되거나 GLAD를 사용하여 독립적으로 존재할 수 있다[9]. GLAD는 명세를 모듈화하고 라이브러리와 같은 외부 LOTOS 명세에도 구현 종속 사항을 기술할 수 있도록 한다. 예를 들면, Stack 명세의 코드 생성을 위해 다음과 같은 GLAD가 사용될 수 있다.

```
specification
  => (*| ldc #include "ldc_imp.h" |*);
sorts
  any => (*| lexical |*)
opns
  any: forall-> any => (*| lexical |*);
  top: stack -> Nat => (*| nonconstructor |*);
endspec
  => (*| ldc #include "ldc_impl.cpp" |*)
```

먼저 주석 lexical은 소트와 연산자의 이름을 각각 구현 클래스와 구현 함수의 이름으로 사용하라는 뜻이다. 주석 nonconstructor는 pop 연산자가 새로운 값을 생성할 수 없다는 뜻이다. 즉, top(empty)와 같은 값은 오류로 처리된다. 실제 C++ 코드도 추가될 수 있는데, ldc 주석이 그 예이다.

4.1.6 수작업 코드

소프트웨어 개발에 코드 생성기가 실질적으로 사용되기 위해서는 생성된 코드를 부분적으로 조율하고 대치하고 기존 혹은 수작업 코드와 연계하여 작동할 수 있어야 한다. 또한 수정 첨가되는 코드는 모듈화되어 유지 보수 관리가 용이하여야 한다. 예를 들면, 코드를 재 생성할 때 수정 첨가된 코드를 재 작업할 필요가 없어야 한다. 객체 지향의 상속 개념은 함수와 클래스

단위로 자동 생성 코드를 정제하고 일부분을 대치할 수 있도록 한다. 또한 extern 주석을 사용하여 연산자 혹은 소트의 구현을 자동 생성하지 않고 수작업 코드로 대치할 수 있다. 커널 클래스 USort는 자동 생성 코드와 수작업 코드의 연계를 위한 서비스를 제공한다. 예를 들면, 수작업 클래스의 자료구조에 대한 관습과 기능에 관한 프로토콜을 정의한다. 모든 수작업 클래스는 커널 클래스 USort의 서브 클래스로 구현되어야 한다.

4.1.7 동적 메모리 관리

명세는 수학적 도구를 사용하여 실세계를 개념화한 관념의 세계인 반면 프로그램은 수학적 계산이 물리적 자원을 사용하여 일어나는 현실 세계이다. 따라서 수학적 세계에서는 무시될 수 있는 시간, 자원, 비용 등이 중요한 고려 사항이 된다. 예를 들면, 명세의 추상값이 무한하고 재귀적 구조를 갖기 때문에, 구현값은 동적 메모리를 사용한다. 따라서, Garbage Collection을 제공하지 않는 C++에서 동적 메모리는 관리 대상이 되는 중요한 자원이다.

동적 메모리 관리를 위하여 Smart Pointer를 사용한다[1]. 즉, 해당 포인터가 지시하고 있는 객체의 참조에 관한 정보를 자동적으로 유지 관리한다. 참조 회수에 관한 정보는 사용자로부터 완전히 은닉되어 사용자에게는 투명하게 비쳐진다. 예를 들어, 변수가 설정될 때 해당 객체의 참조 회수는 자동적으로 조작되고 관리된다. 경우에 따라서는 기존 객체가 삭제될 수도 있다. Smart Pointer는 표현이 자연스럽고 사용이 편리하며, 성능이 좋고 작은 프로그래밍 부담을 준다. 또한 생성되는 코드는 간결하고 깔끔하며 효율성이 좋다.

4.2 행위 컴파일러

우리가 개발하는 행위 컴파일러는 LOTOS 명세로부터 C++ 코드를 생성하는 것이 목적이다. 컴파일러를 바닥부터 개발하는 것도 방법이지만, 우리는 TOPO 도구에서 생성한 중간 코드(가상 머신 코드)를 입력으로 하는 방법을 채택하였다. 이 방법은 문법/의미 검사 모듈을 TOPO의 것으로 대치할 수 있고, 잘 정의된 가상 머신 코드에서 목표 코드를 생성할 수 있으므로 개발 기간을 단축할 수 있다.

4.2.1 TOPO의 행위 컴파일러

TOPO의 행위 컴파일러는 프로그래밍 언어 코드를 생성하기전 단계인 중간 코드 형태로 확장 오토마타(automata)와 유사한 BUT(Behaviour UniT)의 집합을 생성한다. 이 오토마타는 프로세스 생성, 관리,통신 기능을 지원하는 전통적인 운영체계와 유사한 커널과 협력하여 작동한다. TOPO에서는 주로 프로세스들간의 통신에 관심을 두고 있다. 오토마타는 각 상태에서 통신 요구사항 - LOTOS 용어로 동기화 환경 - 을 커널에 알려 준다. 오토마타는 커널을 통하여 다른 오토마타에 행위(action)를 전달하고 동기화가 되면 오토마타는 다음 상태로 이동한다. 커널에 제공되는 행위는 하나의 오토마타와 연결되어 있다. 여러개의 행위가 LOTOS 행위 표현식에 의해서 동시에 제공될 때는 여러개의 오토마타가 존재하고, 각 오토마타는 하나의 행위를 기다리고 있게 된다.

커널은 동기화 요구와 행위 정보를 저장하고, 이 정보에 따라 동기화가 일어날 부분을 결정한다. 그리고, 해당 오토마타가 다음 상태로 이동하기 위해서 활성화된다[10].

[11]에 의하면 여러 가지 LOTOS 컴파일러가 있으나 완전한 LOTOS 문법을 지원하는 것은 TOPO의 방법뿐이다.

(1) 목표 코드 형식

TOPO는 LOTOS 프로세스 정의를 가상기계의 BUT으로 변환하고, BUT은 목표 언어인 C의 함수로 표현한다. 이 C 함수는 재진입(reentrant) 코드로서 동시에 여러 곳에 호출되어 실행될 수 있다. 그 함수의 실행 환경(context)은 함수를 호출하는 측에서 제공함으로 같은 코드를 다른 환경에서 실행할 수 있다. 이 기능은 하나의 BUT 정의에 대해서 여러개의 개체가 있을 수 있고, 각 개체는 자신만의 실행 환경을 가진다는 것을 의미한다[11]. C 함수의 내용은 오토마타를 표현하기 위한 여러개의 case 문으로 구성되어 있다. case 문에는 행위 정보 또는 구조 정보가 등록된다. 행위 정보란 실행을 원하는 게이트와 그에 딸린 상수, 변수를 통칭하는 것이다. 실행 내용이 행위(action)인 경우는 해당 커널 호출을 수행한 후 곧바로 커널 코드로부터 복귀(return)하고, 행위가 아닌 경우 즉, 선택(choice), 병렬 연산자(parallel operation)인 경우는 재귀적 함수 호출을 사용하여 여러개의 case 문을 실행하면서 커널 트리를 확장하는 작업을 한다.

```
hiding a, b, c in a
    a; ...
  |||
      ( b; ...
    []
        c; ...
      )
```

(그림 3) LOTOS 명세
(Fig. 3) LOTOS Specification

예를 들어, (그림 3)과 같은 LOTOS 프로세스 정의는 아래의 C 함수로 변환된다. LOTOS 표현식에서 하나의 행위는 C 함수에서 하나의 case 문으로, 행위의 내용은 커널 호출 함수로 변환된다. 함수에 전달되는 board 형의 변수 b는 함수의 커널 트리 연산에 필요한 노드 포인터이다. b->type은 함수 c1에 필요한 실행 환경을 갖고 있다.

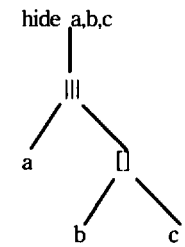
```
c1 (b)
board b;
{
  again;
  M = b->frm;

  switch (M->ent) {
  case 0: {
    b = mkhd(b,1);        /*hiding*/
    gts = b->fth->gte;
    gts[1] = TRUE;        /* gate a */
    gts[2] = TRUE;        /* gate b */
    gts[3] = TRUE;        /* gate c */
    M = b->frm;

    mkpi(b, 2, 4);        /* parallel interleaving */
    c1 (b->sons);
    b = b->sons->bth;
    goto again;
  }
  case 2: { /* action */
    mkad(b, 3, 1, 0, 0, -1, 0);
    break;
  }
  case 3: ...; break;
  case 4: {
    mkch(b, 8, 7);  /* choice */
    c1(b->sons);
    b = b->sons->bth;
    goto again;
  }
  case 5: {  /* action */
    mkad(b, 6, 2, 0, 0, -1, 0);
    break;
  }
  case 6: ...; break;
  case 7: {  /* action */
    mkad(b, 8, 3, 0, 0, -1, 0);
    break;
  }
  case 8: ...; break;
  }
}
```

위 함수의 첫 번째 case 문은 hide와 ||| 구조를 커널 트리에 반영하고, 재귀 함수 호출 c1(b->sons)와 goto 문을 사용하여 상태 2와 4를 실행한다.

(2) 커널 코드

커널 코드는 (1)에서 설명한 것과 같은 형식으로 생성된 코드를 실행하는데 필요한 코드로서 주기능은 BUT 행위간의 동기화를 조정하는 역할이다. 이 모듈은 미리 만들어져 있어서 사용자가 작성한 LOTOS 명세로부터 생성된 코드와 연결되어 완전한 실행가능한 프로그램을 만든다.

커널에서 사용하는 주된 자료구조는 커널 트리이다 (단 한 개의 instance만 존재한다.). 커널 트리의 잎 노드는 하나의 BUT 행위를 갖고, 그 행위가 발생할 경우 실행되어야할 오토마타에 관한 모든 정보가 기록되어 있다. 중간 노드는 LOTOS 표현식에 나타난 행위들의 구조 정보를 갖는다. LOTOS 용어로 설명하면, 커널 트리의 각 노드는 hiding, enabling, disabling, parallel composition, choice, 그리고 relabelling 타입을 갖는다[10][11]. (그림 3)을 사용하여 커널 트리를 구성하면 (그림 4)와 같다.



(그림 4) 커널 트리
(Fig. 4) kernel tree

커널은 커널 트리의 초기화 작업(첫 트리 확장)을 수행한 후, 아래의 과정을 입력에 의한 후보가 없을 때까지 반복한다. 아래의 과정 중에서 핵심이 되는 부

분은 트리 계산과 행위의 실행이다.

1. 트리 계산으로 실행 후보를 선택한다.
2. 실행 후보 중 의미 있는 것만 선택한다. 게이트는 동기화 되었지만 게이트에 딸린 값이 결정되지 않은 경우는 동기화되지 않은 것으로 판단, 의미없는 것으로 간주한다.
3. 후보들 중 하나를 임의로 선택하여 실행한다. 하나의 입력에는 여러개의 행위가 동기화되어 실행될 수 있다.
4. 트리 계산에 사용되었던 임시 정보를 삭제한다.

(3) 트리 계산

커널 트리의 확장 작업이 완료되어 트리가 안정되면, 잎 노드에 저장된 행위 정보를 트리의 중간 노드 정보에 따라 실행되어야할 부분을 찾는 것을 트리 계산이라 한다. 트리 계산 알고리즘은 [10]을 참조하기 바란다.

이 알고리즘의 핵심은 트리 구조를 따라 잎 노드에서 뿌리 노드로 이동하면서 잎 노드에 저장된 행위 정보를 중간 노드의 타입에 따라 계산하고 결과를 부모 노드로 전달하는 것이다. 뿌리 노드까지 전달된 행위들은 실행 후보가 되는 것이다. 단, hiding의 경우는 예외로, hiding 리스트에 포함된 게이트의 경우 곧바로 실행 후보에 포함된다.

(4) 행위의 실행

트리 계산에 의해 선택된 BUT은 다음 상태로 이동하고 그 상태에서 커널에 주어지는 행위 정보를 받는다. 계산 결과가 여러개의 행위이면 임의로 하나를 선택하여 실행하고 나머지는 무시한다. 행위의 실행은 1단계로 행위가 포함된 문맥에 따라 트리를 간단화하고, 2단계는 오토마타에 정의된 다음 상태로 이동하고, 3단계는 그 상태에 정의된 정보를 커널에 전달하는 과정을 의미한다.

하나의 행위가 실행되면, 그에 따라 커널 트리를 변화시키게 된다. 예를 들어, 선택문에 포함된 행위중 하나가 실행되었다면, 선택문의 다른 행위들은 의미가 없어지므로 커널 트리에서 삭제된다. 이 과정을 간단화(simplification)라고 하는데 아래의 규칙을 따른다 [10]. *로 표시된 부분은 트리 계산에 의해서 선택된 행위이고, g는 게이트, δ는 exit 게이트이다. 규칙 ④의 ⇒는 규칙의 실행 조건으로 B_1^*의 게이트가 exit인 경우를 나타낸다.

$$B_1^* \ []B_2 \quad -> \quad B_1^* \qquad ①$$

$$B_1^* \ []B_2^* \quad -> \quad B_2^* \qquad ②$$

$$B_1[>B_2^* \quad -> \quad B_2^* \qquad ③$$

$$(g = \delta) => B_1^* >>B_2 \quad -> \quad B_2 \qquad ④$$

행위 실행의 마지막 단계로 트리의 확장이다. 아래의 예에서 상태 0의 행위가 실행되어 다음 상태 4로 이동하는 경우 트리의 확장이 일어난다. 상태 4의 경우, 행위가 아니라 행위들간의 구조를 정의하는 부분이므로 이 정보를 커널 트리에 반영하는 것이 트리의 확장이다.

```
c1 (b)
board b;
{
  again;
  M = b->frm;
  switch (M->ent) {
    case 0: { ...  /* hiding */ }
    case 4:
      mkch (b, 5, 7);    /* choice */
      c1 (b->sons);
    b = b->sons->bth;
    goto again;
  }
  ...
}
}
```

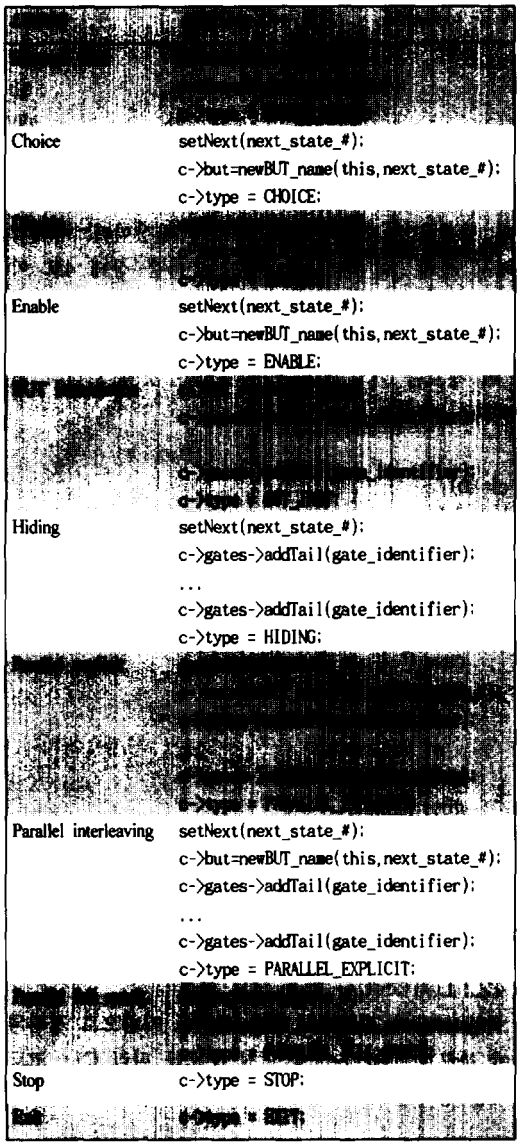
4.2.2 LOTOS/C++ 목표 코드 형식

목표 코드가 TOPO의 중간 코드를 기반으로 정의되어야 하므로 TOPO 가상 머신 구조를 따라 C++ 코드 형식을 정의한다.

(1) BUT

중간 코드를 C++로 표현할 때, 가장 중심이 되는 문제는 BUT을 어떻게 표현할 것인가이다. 우리는 TOPO의 BUT을 C++의 class로, BUT의 행위는 그 class의 메소드로, BUT의 자료는 private data로 표현한다. TOPO에서는 함수, 프레임 등으로 흩어져 있는 구조를 class 구조를 사용하여 한 곳에 모았고, 간결한 코드 생성이 가능하도록 노력하였다. (그림 5)는 LOTOS 각 노드에 대응되는 C++ 코드의 형식을 정의한 것이다.

실제 코드에서는 BUT class에 BUT이 가질 수 있



(그림 5) LOTOS 행위부분의 C++ 변환 형식
(Fig. 5) Transforming LOTOS operators into C++ code; *n* is the number that indicates the next state

는 기본 기능을 묘사하였다. offer 맴버 함수는 오토마타의 행위를 커널에 전달하는 역할을, values 변수는 BUT에서 사용되는 변수의 값을 저장하는 역할을 담당한다.

```
class BUT {
privates:
  int next;   /* next entry point */
protected:
  Offer* offer;
  CVRef<XSort> values[MAX_NUM_VAL];
public:
    BUT(int n = 0);
    BUT(BUT *obj);
  Cmd* run() {
    wipeOutOffer();
    offer = new Offer;
    return _run();
  };
virtual Cmd* _run();
virtual char* name() { return ""; };
  void setNext(int n) { next = n; };
   int getEntryPoint() { return next; };
       ...
      ~BUT();
};
```

LOTOS 명세의 프로세스로부터 생성되는 BUT은 BUT class를 상속하고 자신만의 특성 즉, 이름, 변수, 변수의 소트, 오토마타 등을 추가로 묘사한다. 오토마타는 _run() 함수로 표현한다. (그림 3)이 포함된 프로세스의 이름을 "proc"라고 가정하고 C++ 코드로 변환한 것은 다음과 같다.

```
class Proc: public BUT {
public:
      Proc(); BUT(0) {};
      Proc(Buffer *obj); BUT(obj) {};
  char* name() { return "Proc"; };
  Cmd* _run();
};

Cmd* Proc:: _run()
{
  Cmd *c = new Cmd;
  switch (getEntryPoint()) {
  case 0:   /* hiding */
    setNextState(1);
    c->gates->addTail(1);
    c->gates->addTail(2);
    c->gates->addTail(3);
    c->type = HIDING;
    break;
  case 1:   /* parallel interleaving */
    setNextState(2);
    c->but = new Proc(this, 4);
    c->type = PARALLEL_INTER;
    break;
```

```
    case 2:   /* action */
      setNextState(3);
      offer->setGateId(1);
      c->type = EXTERN_OFFER;
      break;
    case 3: ...; break;
    case 4:   /* choice */
      setNextState(5);
      c->but = new Proc(this, 7);
      c->type = CHOICE;
      break;
    case 5:   /* action */
      setNextState(6);
      offer->setGateId(6);
      c->type = EXTERN_OFFER;
      break;
     case 6: ...; break;
     case 7:   /* action */
      setNextState(8);
      offer->setGateId(3);
      c->type = EXTERN_OFFER;
      break;
     case 8: ...; break;
     }
     return c;
}
```

각 case 문에서 주로 하는 작업은 행위 정보를 커널에 전달하는 것이다. 사용한 자료값, 다음 상태 번호, 게이트 정보는 BUT 객체 내부에 저장하여 커널에 전달하고, 새로이 생성된 BUT 객체, 실행할 연산의 종류는 Cmd 객체를 사용하여 전달한다.

(2) 동치 검사

커널이 게이트의 동기화를 검사하기 위해서 게이트에 제공된 값의 동치성을 검사해야 한다. 이 부분을 커널에 불박이 코드로 넣을 수 없기 때문에 LOTOS 명세에 해당하는 목표 코드를 생성할 때마다, 그 코드에 맞는 동치 검사 모듈을 생성해야 한다.

이 문제는 커널 코드가 미리 만들어져 있고, LOTOS 명세에 해당하는 코드가 이후에 만들어지기 때문이다. 커널이 작성될 때는, 어떤 소트(sort)가 있는지 알 수 없으므로, 특정 소트를 대상으로 코드를 만들 수 없고, 모든 소트의 포괄 소트(generic sort)만을 알 수 있다. 자료 컴파일러에서 정의한 포괄 소트는 XSort이다[1]. XSort로 저장된 두 값을 x, y라고 가정하고, 두 값이 동일한지 검사하려면 x->equal(y) 문장을 실행하면 된다. equal은 자료 컴파일러가 각 소트에 미리 정의한 메소드이다. equal은 generic class의 가상함수로 정의되어 있으므로 x는 실제 타입으로 실행되지만 y는 여전히 XSort 타입으로 인식된다.

우리는 equal이라는 독립함수를 사용하여 이 문제를 해결하였다. 아래의 예제 코드에서도 볼 수 있듯이 generic 타입으로 저장된 값이 동치 검사를 위해서 실제 타입으로 변환된다. 우리가 채택한 방법은 자료 컴파일러를 수정하지 않고 해결하는 최선의 방법으로 생각한다. 자료 컴파일러가 위 문제점을 해결하기 위한 방법을 제시할 수 있으나 그럴 경우, 자료 컴파일러의 설계를 행위 컴파일러의 커널만을 위해서 변경되어야 하므로 좋지 않다.

```
#define C(sort, ref) ((sort*) ref.deref())

int equal(int,sort,CVRef<XSort>k x, CVRef<XSort>k y)
{
   switch (sort) {
     case NAT: {
          CVRef<nat> n0(C(nat,x));
          CVRef<nat> n1(C(nat,y));
          return n0->equal(n1); }
    case STACK: { ... }
   }
   return NO;
}
```

(3) 조건식(predicate)

게이트의 동기화 검사에 사용되는 조건문 검사는 앞에서 설명한 동치 검사보다 더 어려운 문제를 안고 있다. 동치검사의 문제는 이미 알고 있는 메소드 equal()을 어떻게 처리하느냐의 문제이지만, 조건식의 문제는 커널이 만들어질 때, 전혀 알 수 없는 메소드를 어떻게 처리하느냐의 문제이다. 조건식에 사용되는 연산자들은 커널이 만들어질 때에 알 수 없다. 이 문제를 해결하는 방법 역시 동치검사의 방법과 같다. 코드를 생성하는 단계에서는 LOTOS 명세에 나타난 모든 정보를 알 수 없으므로 조건식에 어떠한 메소드가 사용되어야 하는지 알 수 있다. 코드 생성 단계에서 각 조건식에 번호를 부여하고 각각을 독립함수로 구현하는 것이다. 조건식 함수 내에서는 커널로부터 받은 값들을 원래의 타입으로 변환하여 값을 계산할 수 있다.

```
int prd000(CVRef<XSort> _value[], BUT* b)
{
  CVRef<bool> t(true());
  CVRef<nat> n0(C(nat, _value[0]));
  CVRef<nat> n1(C(nat, _value[1]));

  return t->equal(n0->gt(n1));
}
```

4.2.3 LOTOS/C++ 커널 코드

커널은 BUT의 행위 정보를 받아서 동기화될 행위를 선정하고, 그 행위를 실행하도록 하는 기능을 수행한다. 이 코드는 미리 만들어져 있고 사용자가 작성한 LOTOS 명세로부터 생성된 코드의 실행을 지원한다.

커널의 가장 중요한 기능은 행위 정보를 커널 트리로 관리하면서 동기화를 실현하는 것이다. 우리가 사용하는 커널 트리 개념과 그와 관련된 알고리즘은 TOPO의 것을 사용했으므로 본 절에서는 우리가 개발한 커널의 특징만을 설명한다.

우리가 개발한 커널의 특징은 생성 코드의 형식에서 드러난다. 앞의 예에서도 언급되었지만, TOPO가 제시한 코드 형식은 생성된 코드가 직접 커널 함수를 호출하여 커널 트리 연산을 수행하도록 설계되어 생성된 코드가 복잡하고 읽기 어려운 반면, 우리가 설계한 코드는 오토마타의 각 상태 정보만 나열하도록 되어 있어 간결하고 읽기 쉽다. 커널 트리에 관련된 연산은 커널이 담당하도록 한 결과이다. 이렇게 함으로써 생성 코드를 재사용 할 경우 코드를 이해하기 쉽고, 수정이 용이하다. 커널 코드에 포함된 커널 트리 확장 부분의 골격은 아래와 같다.

```
int KernelTree::expandTree(Board* b) {
   Cmd* cmd = b->run();   /* BUT 실행 */
   switch (cmd->type) {
      case HIDING:
           ...;
           expandTree(b->getChild());
           break;
      case ENABLE:
           ...;
           expandTree(b->getChild());
           break;
      case PARALLEL_EXPLICIT:
           ...;
           expandTree(b->getChild());
           expandTree(b->getChild());
           break;
      ...
   }
}
```

5. 결 론

대용량 고신뢰도, 복잡성, 분산 환경이라는 특징을 지닌 정보 통신용 소프트웨어 개발에서 최소의 비용으로 최고의 품질을 지닌 제품을 생산하기 위해서는 향상된 정형 명세 기법과 이러한 기법을 지원하는 도구가 필수적이다. 도두들 중에서 코드 생성기는 다른 상위 도구의 기반이 될 뿐만 아니라 그 자체로도 유용한 도구이므로 큰 비중을 두고 개발되고 있다.

TOPO는 C와 ADA 코드를 생성하고 있으나 요즘 산업계에서 널리 사용되고 있는 C++ 언어를 지원할 필요가 있다. 이론적으로 C가 C++의 부분 집합으로 C++가 사용되는 곳에서 C를 사용할 수 있으나, 실제적으로는 통합이 쉽지 않으므로 별개의 C++ 코드 생성기를 개발하는 것은 의미있는 일이다.

C++는 C와는 개념적으로 완전히 다른 언어이므로 C 코드를 생성하는 방법을 단순하게 적용하는 방법은 적합하지 않다. TOPO의 C 코드생성기를 분석하여 C++에 적합한 목표 코드 형식을 정의했으며, 또한 생성된 코드의 간결함을 위해서 노력하였다. 생성된 코드가 단순히 실행 코드를 만들기 위한 중간 코드가 아니라 다른 모듈에서 재사용되거나 필요에 따라 수정되어야 하므로 생성된 코드의 간결한 형식은 중요한 문제이다.

본 코드 생성기는 스페인 마드리드 대학에서 개발한 TOPO를 기반으로 개발된 것이다. TOPO에서 제공하는 전위(Front-End) 도구와 중간 표현 형태인 AST (Abstract Syntax Tree)를 사용한다. TOPO를 비롯한 대부분의 코드 생성 도구들은 C, ADA, 함수 언어를 생성하고 있다. 본 도구에서는 목적언어로 C++를 생성하여 객체지향 개념을 도입하고, LOTOS 프로세스의 자료와 행위를 하나의 클래스로 정의함으로써 자연스럽게 변환할 수 있고, 생성된 코드가 모듈화됨으로써 코드를 수정하거나 재사용이 용이하게 한다. 자동 생성된 코드가 효율성면에서 수작업으로 작성된 코드보다 떨어지나, LOTOS 명세로부터 신속하게 코드를 생성하여 실행함으로써 시스템 개발 초기에 사용자의 요

구사항 만족성 여부를 검사할 수 있다. 현재 LOTOS를 향상시킨 E-LOTOS의 표준화 작업이 아직 미완성 단계에 있으며, 표준화 작업이 완성되면 E-LOTOS로부터 소스코드를 생성할 수 있도록 본 코드 생성 도구를 확장할 계획이다.

참 고 문 헌

[1] 천윤식, 김강호, 박원규, 김철홍, "자동 생성된 프로그램에서 동적 메모리 관리", '97 추계학술발표 논문집, 제4권, 제2호, pp.712-716, 1997.

[2] Juan Bicarregui, Jeremy Dick, and Eoin Woods, "Quantitive analysis of an application of formal methods," In M-C Gaudel and J. Woodcok, *FME '96 :Industrial Benefit and Advances Formal Methods, Third International Symposium of Formal Methods Europe, Oxford, UK, March 1996 Proceedings, volume 1051 of Lecture Notes in Computer Science,* pages 6073, Springer-Verlag, 1996.

[3] T.Bolognesi and E.Brinksima, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems,* 14(1):25-59, 1987.

[4] ISO. Information Processing Systems - Open Systems Interconnection - LOTOS: A Formal Description Technique Based on the Temporal Ordeing of Observational Behaviour, International Standard 8807. ISO, 1988.

[5] M.Caneve and E.Salvatori, editors, "LITE User Manual," LOTOSPhere Consortium, 1992.

[6] J.A. Manas and T. de Miguel, "From LOTOS to C," In K.J. Turner, editor, Proceedings of the 1st International Conference on Formal Description Techniques(FORTE '88), University of Stirling, Scotland, pp.247-261, North-Holland, September 1988.

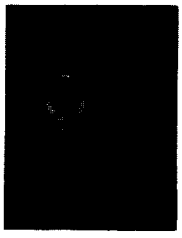
[7] Hubert Garavel, "Compilation of LOTOS abstract data types," In S.T. Vuong, editor, Proceedings of the 2nd International Conference on Formal Description Technique(FORTE '89), Vancouver, Canada, pp.147-162, North-Holland, December 1989.

[8] Jan de Meer, Rudolf Roth, and Son Vuong. "Introduction to Algebraic Specification Based on the Language ACT ONE." Computer Networks and ISDN Systems, 23(5):363-392, February 1992.

[9] Marcelino Veiga, "GLAD: A General Language to Annotate Data," Technical report, Dept. of Telematics Engineering, Technical University of Madrid, Madrid, Spain, October 1994.

[10] Jose A. Manas, Joaqin Salvachua, and Toma de Miguel, "The TOPO implementation of the LOTOS multiway Rendezvous." Department Ingenieria Telematica Technical University *of Madrid,* 15 Jan. 1991.

[11] Eric Emile Dubuis, "Compiling the Behaviour Part of LOTOS," Ph.D thesis, Swiss federal institute of technology ZURICH, 1993.



김 철 홍

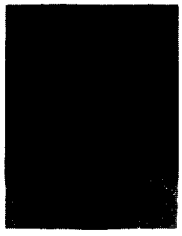
e-mail : kch@etri.re.kr

1982년 충남대학교 문과대학(학사)

1993년 성균관대학교 정보처리학과(석사)

1983년~현재 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 소프트웨어공학연구부 선임연구원

관심분야 : 소프트웨어 재사용, 소프트웨어 설계 방법론, 분산처리



천 윤 식

e-mail : cheon@etri.re.kr

1989년 고려대학교 전산과학과 졸업(학사)

1991년 Iowa주립대학교 전산과학과 졸업(이학석사)

1992년~현재 Iowa주립대학교 전산과학과 박사과정 재학

1995년~현재 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 소프트웨어공학연구부 선임연구원

관심분야 : 정형방법, 객체지향프로그래밍, 소프트웨어공학

김 강 호

e-mail : khk@etri.re.kr

1993년 경북대학교 컴퓨터과학과 졸업(학사)

1996년 경북대학교 대학원 컴퓨터과학과 졸업(이학석사)

1996년~현재 한국전자통신연구원 컴퓨터·소프트웨어기술연구소 소프트웨어공학연구부 연구원

관심분야 : 소프트웨어 재사용, 객체지향설계, 저장시스템