Journal of the Korea Academia-Industrial cooperation Society
Vol. 12, No. 6 pp. 2840-2850, 2011

DOI : 10.5762/KAIS.2011.12.6.2840

시스템 레벨 설계에서 리팩토링을 적용한 단계적 상세화

김현종[1], 공헌택[1], 김치수[1*]
[1]공주대학교 컴퓨터공학부

A stepwise refinement method by applying Refactoring in system level design

Hyun-Jong Kim[1], Heon-Tag Kong[1] and Chi-Su Kim[1*]
[1]Division of Computer Engineering, Kongju University

요 약 임베디드 시스템을 설계하기 위해서는 하드웨어 설계와 소프트웨어 설계를 동시에 진행하여 마지막으로 그들을 통합하여 시스템을 구축하는 하드웨어/소프트웨어 공동 설계라는 설계 방법을 사용한다. 본 연구에서는 기존의 임베디드 시스템 설계의 문제점을 분석하고 그 해결 방안으로 SpecC 기술을 이용한 시스템 레벨 설계 방법을 제안한다. 또한 시스템 레벨 설계의 단계적 상세화를 실현하기 위해 소프트웨어 공학의 리팩토링 기술을 적용하여 체계적이고, 구체적인 단계적 상세화 방법을 제시한다.

Abstract Programmers can use the Hardware-Software Co-Designing, in which they design a hardware system and software system simultaneously and then unify them, to design an embedded system. This study analyzes the problems of designing an embedded system and suggests applying the system-level design technique, which uses SpecC technology, as a solution to the problems. Also, it suggests systematic and concrete stepwise refinement by applying Refactoring technology in software engineering as a way to make the use of system-level stepwise refinement.

Key Words : Refinement, Refactoring

1. 서론

일반적으로 임베디드 시스템이란 하드웨어와 소프트웨어가 결합해 특정기능을 수행하는 시스템을 의미한다. 따라서 임베디드 시스템은 하드웨어와 소프트웨어 양쪽 모두의 설계가 필요하다. 이같은 시스템의 설계는 일반적으로 하드웨어/소프트웨어 공동 설계라는 설계 방법이 이용되고 있다. 여기서 하드웨어/소프트웨어 공동 설계란 하드웨어 설계와 소프트웨어 설계를 동시에 진행하여 마지막으로 그들을 통합하여 시스템을 구축하는 설계 방법이다. 그러나 현재, 시스템 규모의 증대에 대해서 설계 생산성의 향상이 따라 잡지 못하고 있고, 단기화하고 있는 개발 기간 내에 개발이 완료되지 못하고 있으며, 단기 개발에 의한 검증 부족이 품질의 유지를 곤란하게 한다고 하는 문제가 있다. 이러한 문제를 해결하는 방법은 크게 2가지이다[1]. 하나는 설계의 추상 레벨을 끌어올리는 것으로, 설계 생산성을 향상시키고 개발을 단기화하고 조기 검증을 실현하는 것이다. 또 하나는 과거의 설계 자산을 재사용함으로써 설계가 필요한 부분을 축소하고, 개발을 단기화하고 재사용하는 부분의 검증을 생략하는 것이다. 두 가지 모두 일반적인 해결 방법이다.

최근에는 이 문제의 해결 방법으로 시스템 레벨 설계라는 설계 방법에 많은 연구가 되고 있다[2]. 이 설계 방법은 기존의 하드웨어 및 소프트웨어 공동 설계 방식보다 높은 설계 추상 레벨에서 설계를 시작하고 구현을 위한 단계적으로 설계를 상세화 하는 것이다. 여기서 단계적인 상세화는 시스템의 외부 동작을 바꾸지 않고 시스템의 내부 구조를 재구성하여 시스템의 다양한 측면(시간의 개념과 각 구조 구성 요소의 인터페이스 등)을 상세화 하는 작업이다. 이 작업은 추상적인 시스템 레벨을 구체적인 시스템 레벨로 상세화 한다. 단계적 상세화는, 시스템의 외부 동작을 바꾸지 않는 상세화 순서를 확립하

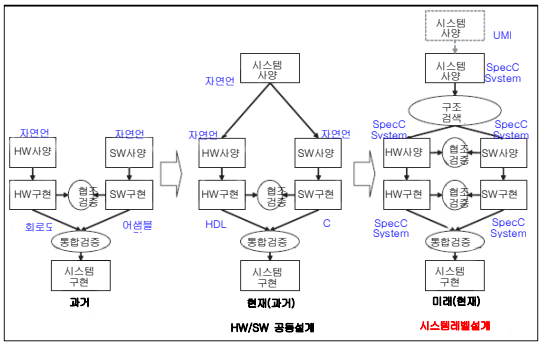
*교신저자 : 김치수(cskim@kongju.ac.kr)
접수일 11년 04월 19일 수정일 11년 06월 03일 게재확정일 11년 06월 09일

는 것과 동시에, 그 상세화 순서를 공식화할 필요가 있다. 따라서 본 연구에서는 단계적 상세화를 실현하기 위해 소프트웨어 공학의 리팩토링 기술을 적용하여 체계적이고 구체적인 단계적 상세화 방법을 제시한다.

2. 관련 연구

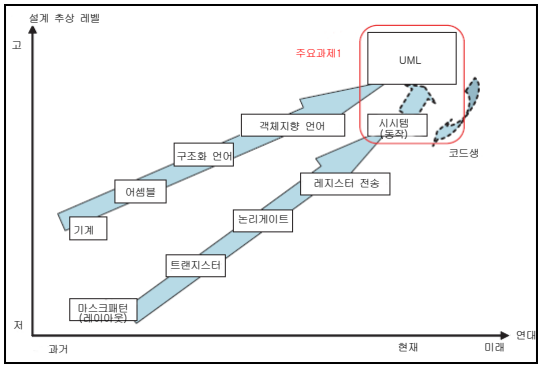
2.1 연구의 범위

하드웨어/소프트웨어 공동 설계는 대상 시스템을 위한 하드웨어 및 소프트웨어를 병행하여 설계하는 방법이다. 다음 [그림 1]은 하드웨어/소프트웨어 통합 시스템 설계의 흐름을 나타내고 있다. (System C 언어[3,4])



[그림 1] HW/SW 공동 설계의 흐름
[Fig. 1] HW / SW co-design flow

왼쪽 그림은 초기 설계 방법, 중간 그림은 하드웨어/소프트웨어 협조 설계, 오른쪽 그림은 시스템 레벨 설계 방법이다[5,6].



[그림 2] 미래의 HW/SW 통합 시스템 설계
[Fig. 2] Future of the HW / SW co-design

본 연구에서는 오른쪽 그림의 시스템 레벨 설계를 위해 SpecC방법론을 사용하는 경우 그림 2에서 나타낸 것처럼 UML 다이어그램을 입력으로 받아 동작 가능한 SpecC사양 모델로 변환하는 방법을 제시하였다.

2.2 SpecC 설계 방법론

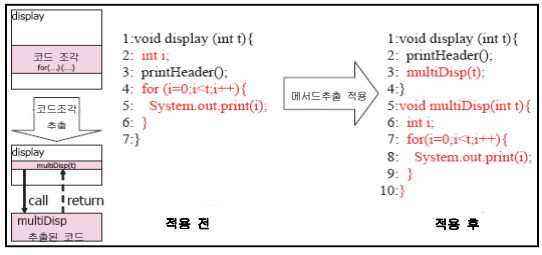
SpecC 설계 방법은 시스템 레벨 언어 SpecC를 기반으로 하는 시스템 레벨 설계 방법의 구체적인 예이다[7]. SpecC 언어는 하드웨어와 소프트웨어를 모두 포함하는 시스템을 설명하기 위해 C 언어를 확장한 시스템 레벨 언어이다. 구체적으로는 시스템의 모델링에 필요한 개념 즉, 동작 계층, 구조 계층, 동시성, 동기화, 예외 처리, 타이밍, 상태 전환의 처리가 가능하도록 C언어를 확장하고 있다. SpecC 설계 방법은 시스템의 모델을 사양 모델, 구성요소 어셈블리 모델, 버스 중재 모델, 버스 기능 모델, 구현 모델로 구분되는 5개의 추상 레벨로 구분하고 있다.

본 연구에서는 이 5개의 추상 레벨 중 최상위 모델인 사양 모델을 UML다이어그램으로부터 변환하여 추출해 내는 과정을 제시하였다.

2.3 리펙토링

리팩터링은 프로그램의 가독성, 모듈화, 재사용성, 유지 보수성 향상을 위해 프로그램 개선을 목적으로 프로그램의 외부 동작을 유지하면서 프로그램의 내부 구조를 재구성하는 기술이다[8]. 리팩터링은 정규화하지 않은 프로그램에 대한 재작성 규칙의 카탈로그이다. 재작성 규칙은 이름, 요약, 동기, 단계, 예 등 5개의 항목으로 구성되며, 리팩터링 규칙이라고 부른다. 여기서 이름은 리팩터링 규칙의 이름을 나타내고, 요약은 리팩터링을 필요로 하는 상태 및 리팩터링하는 것의 요약이다. 또한 동기는 리팩터링을 하는 이유와 리팩터링 해야 하는 상황에 대해 언급한다. 그리고 단계는 리팩터링의 실시 방법을 간략하게 언급하고, 예는 프로그램이나 UML의 다양한 다이어그램을 사용하여 설명된 리팩터링 사용 예를 나타내고 있다.

다음 그림 3은 리팩토링 규칙의 하나인 메소드 추출의 예를 나타내고 있다.



[그림 3] 메서드 추출의 예
[Fig. 3] Example of Extract Method

이 결과를 위한 과정은 다음과 같다.

<1>이름: 메소드 추출

<2>요약: 한 덩어리로 할 수 있는 코드의 조각이 있는 경우 코드의 조각을 메소드로 하고, 거기에 목적을 나타내는 이름을 붙인다.

<3>동기: 메소드가 메인 프로그램으로부터 분리되면, 다른 메소드로부터 그것을 사용할 수 있을 가능성이 증가할 수 있고 오버라이드(override)도 하기 쉽기 때문이다.

<4>단계

① 새로운 메소드를 작성해 메소드의 의도에 맞추어 명명한다.

② 추출되는 코드를 원래 메소드에서 새로운 추출할 메서드에 복사한다.

③ 추출되는 코드 중 원래 메소드에서 범위가 로컬 변수에 대한 참조를 검색한다.

④ 추출되는 코드에서만 임시 변수가 사용되고 있다는 것을 알아낸다. 그렇다면, 그것을 추출할 메소드의 임시 변수로 정의한다.

⑤ 추출되는 코드가 그 지역 범위 변수를 변경하는지 확인한다. 변경되는 변수가 1개라면, 추출한 코드를 문의 메소드로서 취급해 결과를 관련된 변수에 대입한다.

⑥ 추출하는 방법이 읽는 로컬 범위 변수를 추출할 메서드의 매개 변수로 전달한다.

⑦ 원래 메소드에서 추출한 코드를 추출할 메서드에 대한 호출로 대체한다.

⑧ 컴파일하고 테스트 한다.

<5> 예: 그림 3.1에 나타내었다.

일반적으로 긴 메소드는 가독성이 낮고 모듈성이 나쁜 경우가 많다. 이러한 메서드는 재사용 및 유지 보수성을 크게 감소시켜 개발 효율을 악화시키는 원인이 된다. 이러한 메서드에서 새 메서드로 코드를 추출함으로써 원래 긴 메서드가 짧아지고 가독성이 향상된다. 또한 새로 가져온 메서드는 다른 방법으로 다시 사용이 가능하다. 결과적으로 가독성, 모듈 방식, 유지 보수성이 향상됨과 동시에 개발 효율성도 높아진다.

2.4 다른 재구성 기법

모델 변환이란 수학이나 논리학에 근거하는 사양 기법에 대해서 수학이나 논리학의 변환 규칙을 적용하여 원하는 모델을 도출하는 기법이다[10]. 이 기법은 수학과 논리학의 배경을 가지고 있으므로 변환의 유효성을 엄밀하게 보장한다. 그러나 기본적으로 실행 불가능하므로 테스트를 할 수 없다. 모델의 예로서 수식 모델, 차트 변형 이론의 그래프 모델, 시간상의 논리 등이 있다.

본 연구에서는 시스템 레벨 설계 단계적인 상세화를 실현하기 위해 리팩터링 기법을 사용하였는데 그 이유는 다음과 같다. 단계적인 상세화는 항상 실행이 가능할 필요가 있다. 따라서 모델 변환은 실행 불가능하여 후보로부터 제외했다. 또한 시스템 레벨 설계를 대상으로 하는 시스템의 규모는 폭발적으로 증가하고 있기 때문에 적용 대상 시스템 규모가 큰 편이 바람직하다. 따라서 소규모에 적합한 프로그램 변환도 후보로부터 제외하였다.

본 연구에서는 시스템 레벨 설계의 단계적 상세화 과정에서 상세화의 전후로 외부 동작이 변하지 않도록 리팩토링의 재구성 기술을 사용 하였다.

3. 리팩터링에 근거한 단계적 상세화

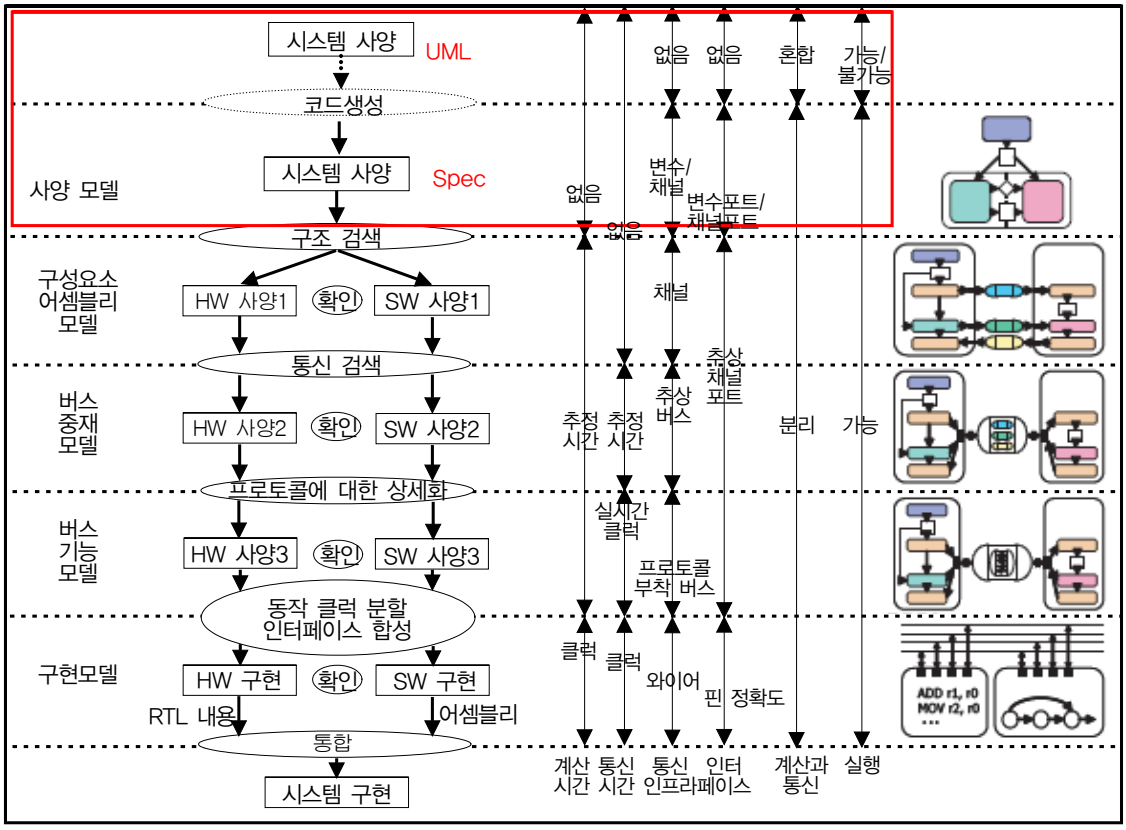
일반적인 시스템 레벨 설계의 재구성은 구조 검색, 통신 합성, 구현 등으로부터 이루어진다. 따라서 각각의 재구성에서는 규칙과 절차에 따라 시스템 기법을 재구성한다. 그러나 현재의 시스템 레벨 설계에서는 규칙이 나타내고 있는 단계에 머물러 있으며, 외부 동작을 유지하는 재구성의 구체적인 순서는 나타나지 않는다는 문제 점을 가지고 있다. 따라서 본 논문에서는 프로그램의 외부 동작을 유지하면서 프로그램의 재사용성, 가독성, 유지 관리 용이성 등을 향상시키는 방향으로 프로그램의 내부 구조를 재구성하는 리팩터링 기술을 적용하여 시스템 레벨 설계의 단계적인 상세화를 실현한다.

SpecC 설계 방법에서 본 논문의 연구 범위는 그림 4와 같다. 즉, UML에 의한 실행 불능의 다이어그램을 시스템 레벨 언어인 SpecC을 이용한 SpecC사양 모델로 변환하는 것이다. 또한 구체적으로 리팩토링 기술을 이용하여 재구성하는 과정은 그림 5와 같다. 즉, UML다이어그램으로부터 실행가능한 SpecC사양 모델을 변환하는 과정은 다음과 같다.

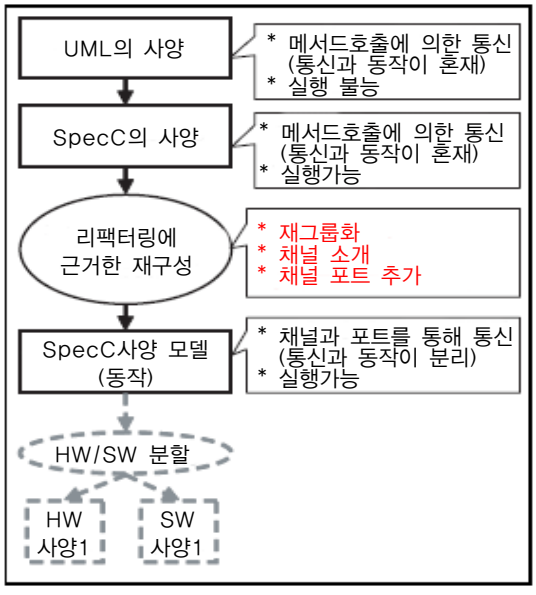
[단계 1] SpecC 사양으로의 변환

SpecC 사양은 UML의 클래스다이어그램과 시퀀스다이어그램을 SpecC에서 제공하는 인터페이스 메커니즘을 상속을 사용하지 않는 객체지향 프로그램 형태의 사양이다. 구체적 방법은 클래스를 동작으로 매핑하고, 다른 동작에서 호출되는 메서드는 인터페이스로 구현한다. 이렇게 하면 실행불능인 사양을 실행가능 사양으로 상세화된다. 여기에서는 리팩터링 규칙 "자기 캡슐화 필드"를 사용하여 다음에 계속될 재구성을 쉽게 할 수 있다.

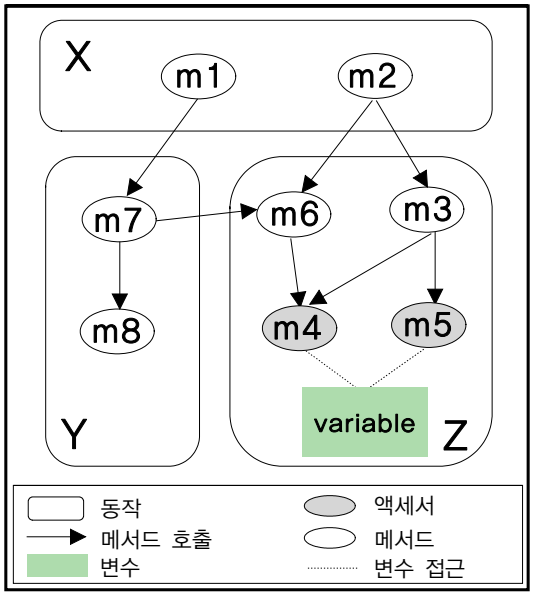
따라서 이 사양 이후는 실행 가능한 형태로 변환된다. 그 결과는 그림 6과 같다.



[그림 4] SpecC 설계 기법의 제안 기법의 위치
[Fig. 4] SpecC position of the proposed method of design techniques



[그림 5] 제안된 방법에 대한 상세도
[Fig. 5] The details of the proposed method

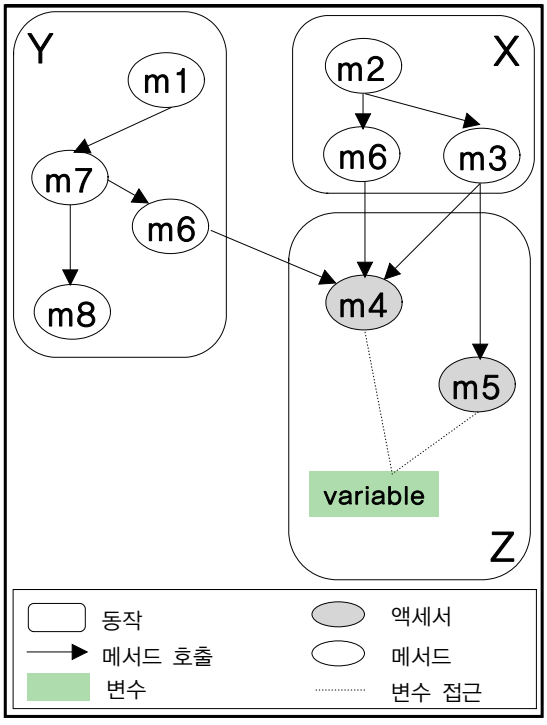


[그림 6] SpecC에 의한 시스템 기록
[Fig. 6] Recorded by the system in SpecC

이 과정의 구체적 절차는 다음과 같다.

① 클래스 이름과 동일한 이름의 동작을 새로 정의한다.
② 정의된 동작에 main 메소드를 추가한다.
③ 인터페이스를 새롭게 정의한다.
④ 공용 메서드를 인터페이스에 선언한다.
⑤ 새로 정의한 동작에 새롭게 정의한 인터페이스를 구현한다.
⑥ 인터페이스의 메서드를 동작에 추가한다.
⑦ 소유하는 측의 동작 포트 목록에 소유되는 측의 동작이 구현하는 인터페이스 형식의 인수를 추가한다.
⑧ 추가한 인수 메서드를 호출하는 처리를 소유하는 측의 동작 메서드에 추가한다.

위의 절차에 따라 수행한 결과는 그림 7과 같다.



[그림 7] 재그룹화에 의한 시스템 기록
[Fig. 7] Regroup by defining processings system records

[단계 2] 재그룹화 사양

이 단계는 객체 지향 모델링이 아닌 다른 그룹의 방침에 따라 동작을 재그룹화한다. 그룹 정책이란 예를 들면 어떤 기능을 수행하고 있는 일련의 메서드 여기서 호출은 동일한 동작을 그룹화하거나, 일반 동작을 공유하는 것이다. 본 논문에서는 "메서드 이동", "필드 이동", "메서드 인라인", "이름 바꾸기", "메서드 추출"과 같은 리팩터링 규칙을 사용한다. 그 결과는 그림 8과 같다.

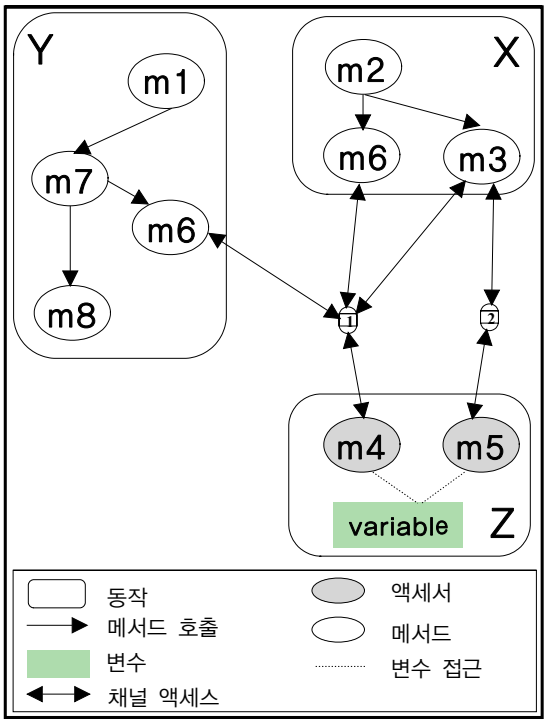
```
 1:interface YIF{void m7();};
 2:interface ZIF{void m3(); void m6();};
 3:behavior Z() implements ZIF{
 4:   int variable;
 5:   void m4(int a){variable=a;}
 6:   int m5(){return variable;}
 7:   void m3(){int a; /*...*/ m4(20); a=m5(); /*...*/}
 8:   void m6(){/*...*/ m4(10); /*...*/}
 9:   void main(){/*...*/}
10:};
11:behavior Y(ZIF z) implements YIF{
12:   void m8(){/*...*/}
13:   void m7(){/*...*/ z.m6(); m8(); /*...*/}
14:   void main(){/*...*/}
15:};
16:behavior X(YIF y, ZIF z){
17:   void m1(){/*...*/ y.m7(); /*...*/}
18:   void m2(){/*...*/ z.m3(); z.m6(); /*...*/}
19:   void main(){/*...*/}
20:};
```

[그림 8] SpecC에 의한 시스템 기록 코드
[Fig. 8] SpecC code written by the system

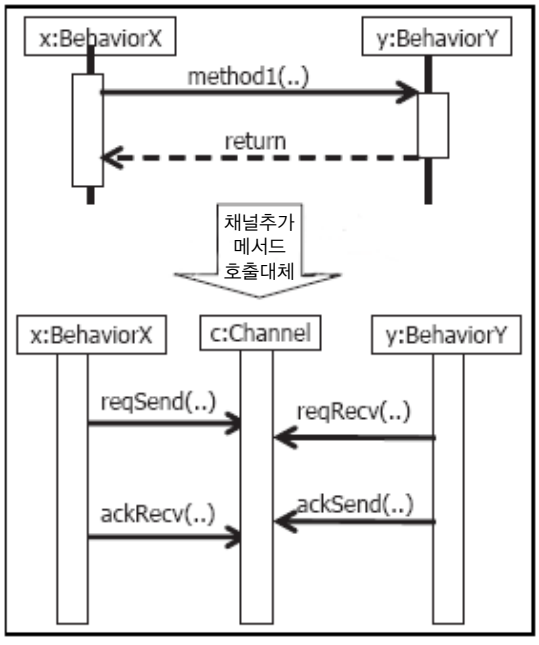
[단계 3] 채널 추가 사양

이 단계에서는 통신 인프라로 채널을 추가하고, 메서드 호출 채널 액세스로 대체한다. 즉, 모든 동작의 메소드 호출을 채널 메서드 호출로 옮겨 놓고, 채널을 통해서 서로 통신한다. 본 논문에서는 시스템 레벨 설계를 위한 새로운 리팩터링 규칙을 제시한다. 그 절차는 다음과 같다. 첫째, 동작의 메소드 호출을 인수의 송신, 인수의 수신, 결과의 송신, 결과의 수신 4개로 구분한다. 둘째, 채널은 위에서 언급한 4가지 메서드를 인터페이스로 구현한다. 마지막으로, 송신자측의 메소드 콜을 인수의 송신과 결과의 수신에 옮겨 놓고, 수신자 측의 메소드 콜은 인수의 수신과 결과의 송신에 옮겨 놓는다. 이상의 재구성에 의해, 통신 인프라에 관해서「없음」으로부터 「채널」에 추가되어 계산과 통신이 분리된 사양을 얻는다. 그 결과는 그림 9와 같다.

또한 동작 메소드의 호출 부분을 포트에 액세스 하도록 대체한 결과는 그림 10과 같다. 여기서 그림의 빈 화살표의 순서는 동작 BehaviorX가 동작 BehaviorY의 메소드 method1를 호출해, 그 결과를 돌려주고 있는 것을 나타내고 있다. 이 메소드 호출을 포트 액세스에 고쳐 쓰는 것으로 그림의 빈 화살표 아래에 나와 있는 순서가 된다. 그리고 메소드를 송신하는 측(BehaviorX)은 인수를 건네주는(reqSend)것, 결과를 받는(ackRecv)것 2개 동작으로 나누어진다.



[그림 9] 채널이 추가된 시스템 기록
[Fig. 9] System records added Channel



[그림 10] 메서드 호출 대체에 따른 변화
[Fig. 10] Replace the method call dependence

또 메소드를 수신하는 측(BehaviorY)도 인수를 받는(reqRecv)것과 결과를 건네주는(ackSend)것, 2개의 동작으로 나누어진다. 채널 Channel은 인수송신(reqSend), 결과수신(ackRecv), 인수수신(reqRecv), 결과송신(ackSend) 4개의 메소드를 구현한다. 그리고 그림 11은 동작의 메서드를 직접 호출하는 것으로 서로 통신하는 시스템 구현을 나타내고 있다.

```
1:interface SlaveIF{                          //behavior-interface
2:  int getArea(int height, int width);
3:};
4:behavior Slave() implements SlaveIF{
5:  int getArea(int height, int width){    //behavior-method
6:    return height * width;
7:  }
8:  void main(void){}
9:};
10:behavior Master() {
11:  int height = 3, width = 5, area = 0;
12:  Slave slave();
13:  void main(void){
14:    area = slave.getArea(height, width); // behavior-method call
15:  }
16:};
```

[그림 11] 메서드 호출의 예
[Fig. 11] Examples of the method call

[단계 4] 포트 추가 사양

이 단계에서는 채널과 동작을 접속하기 위해서 포트를 추가한다. 여기서 채널 메서드 호출을 포트 액세스로 교체함으로써, 동작 인터페이스 부분에 대한 "없음"에서 " 채널 포트"가 추가된 사양으로 바뀌게 된다. 채널 포트의 추가는 동작 리팩터링 규칙 "매개 변수 추가", "매개 변수 제거"를 적용 한다. 즉, 모든 동작은 채널 포트를 사용하여 채널과 접속된다. 따라서 채널 액세스는 포트 액세스로 옮겨져 모든 동작은 포트에 액세스 하는 것으로 서로 통신한다. 이 포트를 추가한 사양의 특징은 계산 시간 없이, 통신 시간 없이, 통신 인프라는 채널, 인터페이스는 채널 포트, 계산과 통신은 분리, 실행 가능함으로써 SpecC 설계 방법 사양 모델이 부과 제약 조건을 만족시킨다. 따라서 포트를 추가한 사양은 SpecC 설계 기법 사양 모델에 해당한다. 그 결과는 그림 12와 같다. 그리고 그림 11의 메소드 호출을 상세화 함으로써 그림 13을 얻을 수 있고, 통신 인프라 및 동작 인터페이스 부분을 각각 "채널"과 "채널 포트"에 추가하는 절차는 다음과 같다.

① 인터페이스를 새롭게 정의한다.
② 정의하는 인터페이스에 4개의 메서드 (void reqSend (), void reqRecv () void ackSend (), void ackRecv ())를 추가한다.

```
1:channel Ch(){
2:  int data;
3:  void write(int a){data=a;}
4:  int read(){return data;}
5:};
6:behavior Z(Ch ch1, Ch ch2){
7:  int variable;
8:  void m4(){variable=ch1.read();}
9:  void m5(){ch2.write(variable);}
10:  void main(){/*...*/}
11:};
12:behavior Y(Ch ch){
13:  void m8(){/*...*/}
14:  void m6(){/*...*/ ch.write(10); /*...*/}
15:  void m7(){/*...*/ m6(); m8(); /*...*/}
16:  void m1(){/*...*/ m7(); /*...*/}
17:  void main(){/*...*/}
18:};
19:behavior X(Ch ch1, Ch ch2){
20:  void m3(){int a; /*...*/ ch2.write(20); a=ch1.read(); /*...*/}
21:  void m6(){/*...*/ ch2.write(10); /*...*/}
22:  void m2(){/*...*/ m3(); m6(); /*...*/}
23:  void main(){/*...*/}
24:};
```

[그림 12] 채널이 추가된 시스템 기록 코드
[Fig. 12] System records code added Channel

```
1:#define INVALID -99999
2:interface GetAreaIF{                                  //channel-interface
3:  void reqSend(int height, int width);
4:  void reqRecv(int *height, int *width);
5:  void ackSend(int area);
6:  void ackRecv(int *area);};
7:channel GetAreaCh() implements GetAreaIF{
8:  int height=INVALID, width=INVALID, area=INVALID;
9:  event req1, req2, ack1, ack2;
10:  bool lock=false; event release;                   // for concurrent access control
11:  void reqSend(int arg1, int arg2){                 //channel-method
12:    while(lock){wait(release);} lock=true;         // for concurrent access control
13:    height=arg1; width=arg2; notify(req1); wait(req2);}
14:  void reqRecv(int *arg1, int *arg2){              //channel-method
15:    while(height == INVALID){wait(req1);}
16:    (*arg1)=height; (*arg2)=width;
17:    notify(req2); height=INVALID; width=INVALID;}
18:  void ackSend(int arg){                           //channel-method
19:    area=arg; notify(ack1); wait(ack2);}
20:  void ackRecv(int *arg){                          //channel-method
21:    while(area==INVALID){wait(ack1);}
22:    (*arg)=area; notify(ack2); area=INVALID;
23:    lock=false; notify(release);}};                //for concurrent access control
24:behavior Slave(GetAreaIF getAreaCh){
25:  int getArea(int height, int width){return height * width;}
26:  void main(void){
27:    int height = 0, width = 0, area = 0;
28:    getAreaCh.reqRecv(&height, &width);            //receive argument
29:    area = getArea(height, width);                 //original processing
30:    getAreaCh.ackSend(area);}};                    //send result
31:behavior Master(GetAreaIF getAreaCh) {
32:  int height=3, width=5, area=0;
33:  void main(void){
34:    getAreaCh.reqSend(height, width);              //send argument
35:    getAreaCh.ackRecv(&area);}};                   //receive result
```

[그림 13] 포트 액세스의 예
[Fig. 13] Examples of port access

③ 동작 메소드의 인수로 reqSend(), recRecv()를 인수에 추가한다. reqRecv()의 인수는 포인터 형식으로 한다.

④ 동작 메소드의 반환 형식을 검사하여 ackSend()와 ackRecv()를 인수에 추가한다. ackRecv() 인자는 포인터 형식으로 한다.

⑤ 채널을 새롭게 정의한다.

⑥ 정의한 채널에 정의하는 인터페이스를 구현시켜 인터페이스에 선언하고 있는 4개의 메소드를 추가한다.

⑦ 추가한 메서드의 내부를 채운다.

⑧ 동작 메소드를 송신측과 수신측의 포트 리스트에 위에서 정의한 인터페이스형의 인수(채널 포토)를 추가한다.

⑨ 동작 메소드의 송신 코드를 채널 포토 액세스 (reqSend()와 ackRecv())에 다시 쓴다.

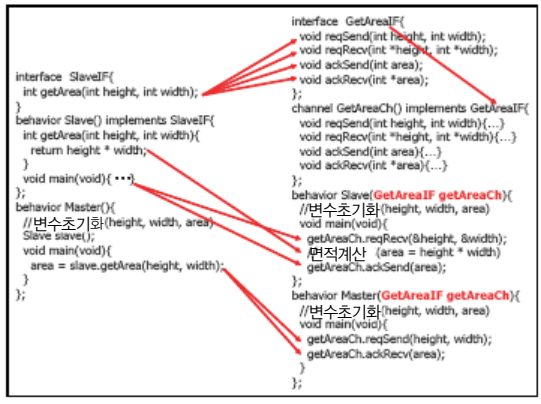
⑩ 동작 메소드를 수신 측의 동작에 채널 포토 액세스 (reqRecv()와 ackSend())를 추가해 그 사이에 불리고 있던 동작 메소드의 호출을 추가한다.

⑪ 동작 메소드를 송신하는 측의 동작의 포트 목록으로부터 수신하는 측의 동작이 구현하고 있는 인터페이스 형식의 인수를 삭제한다.

⑫ 동작의 인터페이스 구현을 삭제한다.

그림 13에서 24번째, 31번째 행의 GetAreaIF는 채널 포트를 나타내고, 28번째 행(getAreaCh. reqRecv), 30번째 행(getAreaCh. ackSend), 34번째 행(reqSend), 35번째 행의 ackRecv은 각각 채널 포트에 접근하고 있는 것을 나타내고 있다.

최종적으로 리팩토링 기술을 이용하여 변환된 새로운 코드는 그림 14와 같다.

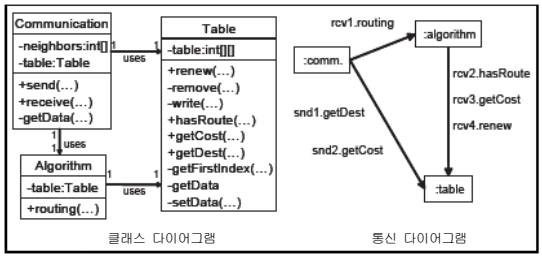


[그림 14] 새로운 리팩터링으로 코드의 변화
[Fig. 14] New refactoring code changes

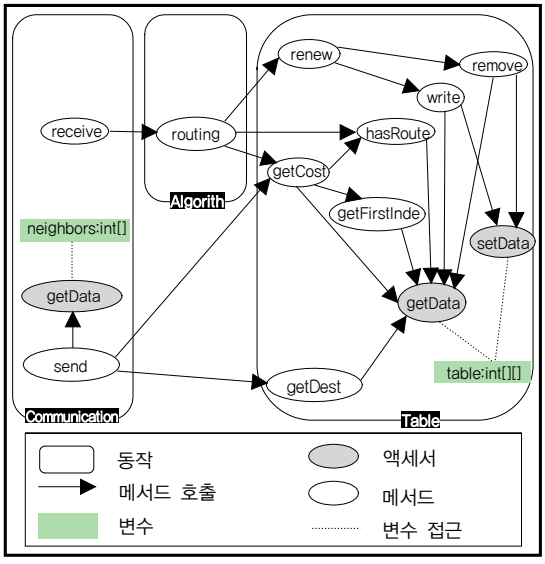
여기서 새로운 리팩터링 코드의 변화는 화살표로 나타내었다. 그리고 동작 인터페이스로 선언되고 있던 메소드가 채널 인터페이스내에서 4개에 분해되어 선언되었고, 채널 인터페이스를 구현하는 채널 GetAreaCh가 추가되었다, 또한 동작 Slave의 동작이 인수를 받는 reqRecv와, 면적의 계산, 결과를 건네주는 ackSend의 3가지로 구분되었고, 동작 Master의 동작이 인수를 건네주는 reqSend와 결과를 받는ackRecv의 2가지로 구분되었다. 또한 동작 Master와 Slave에 채널 포트가 추가되고 있다는 것을 알 수 있다.

4. 실험 및 결과

본 연구는 제안한 방법의 유효성을 확인하기 위해 인터넷 공유기를 예제로 설계했다. 그림 15는 인터넷 공유기의 시스템 구조를 클래스 다이어그램과 협력 다이어그램으로 표현했다. 그림 16은 그림 15를 [단계 1] 과정을 거쳐 생성된 그래프이다.

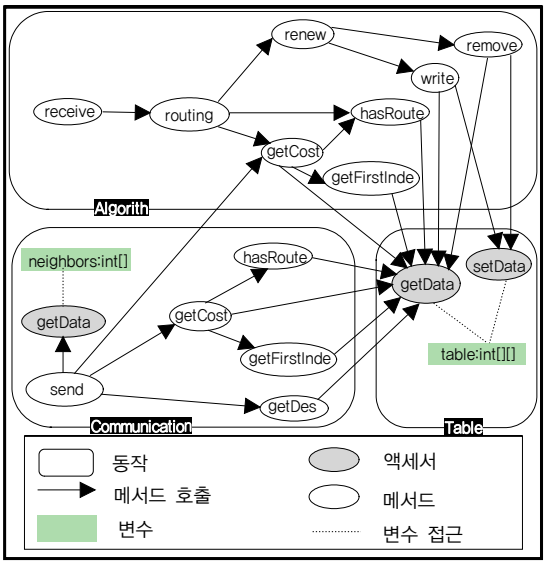


[그림 15] UML을 사용한 예제 사양
[Fig. 15] Specification using UML

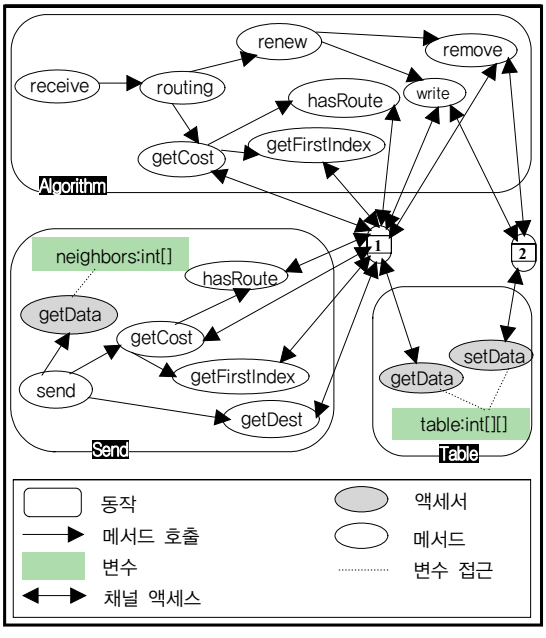


[그림 16] SpecC를 사용한 예제 사양
[Fig. 16] Using the example SpecC Specification

이 그림을 통하여 클래스와 동작이 대응되는 것을 알 수 있고, 각 클래스의 메소드가 동작의 메소드와 일치하고 있는 것을 알 수 있다. 또한 메소드 중 액세스 범위가 공용적인 메소드는 동작내에서 외부로부터 호출되어지는 메소드로 바뀌었다. 그림 17은 그룹화 과정을 통하여 재그룹화된 그래프를 나타내고 있다.

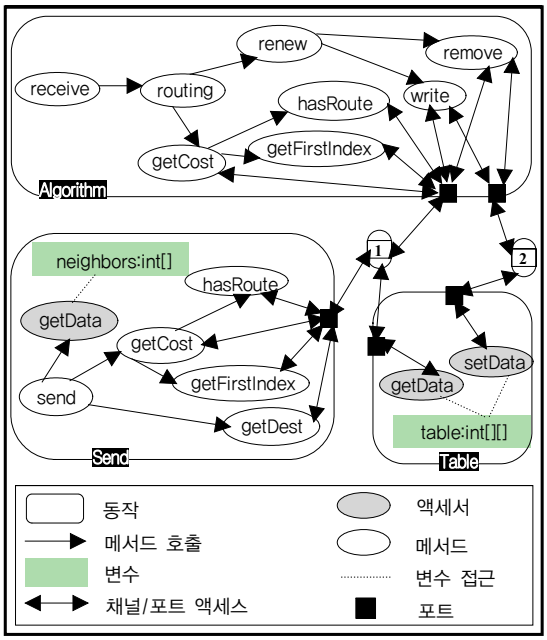


[그림 17] 재그룹화된 예제 사양
[Fig. 17] Regroup qualified sample specifications



[그림 18] 채널이 추가된 예제 사양
[Fig. 18] Specifications added channels

그룹화 과정에서 일련의 메소드 호출은 동일한 동작에 그룹화와 관련된 기능을 정리하고, 공유되는 메소드는 호출측의 동작에 복사/이동 기능 단위로 그룹화가 되고, 엑세스와 변수는 경로 정보의 데이터 구조가 복잡함으로 이동하지 않는다. 그림 18은 메소드 호출을 채널 액세스로 옮겨놓은 그래프를 나타내고 있다.



[그림 19] 포트가 추가된 예제 사양
[Fig. 19] Example specifications appended port

그림 19는 동작에 채널 포트를 추가한 모델이다.

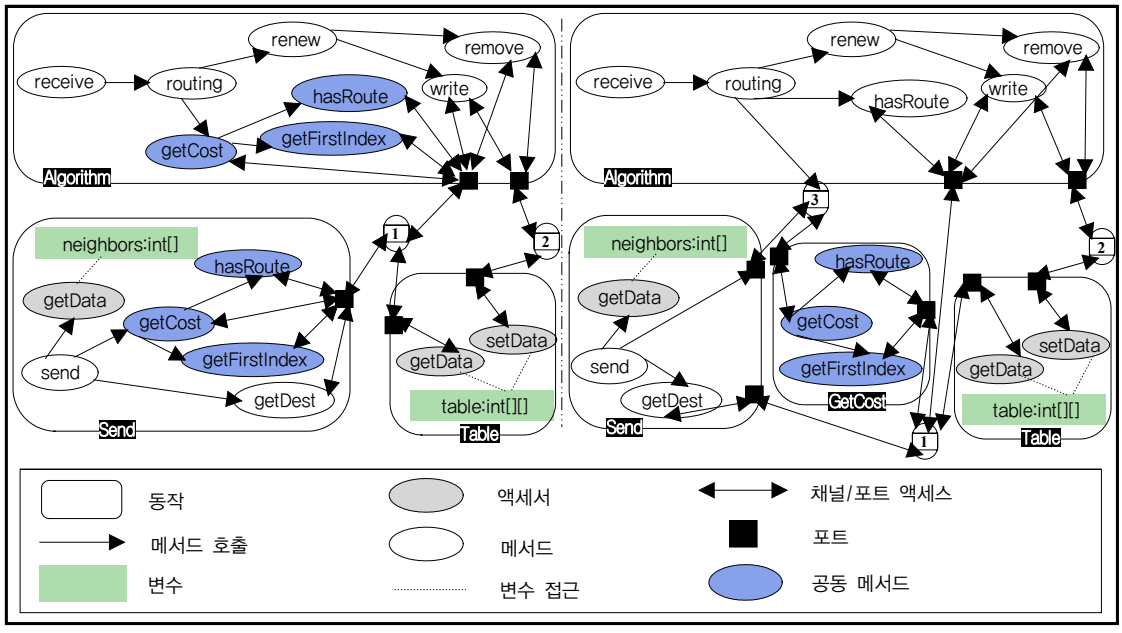
이 모델은 SpecC 설계 기법 사양 모델의 제약 조건을 만족시킨다. 본 논문을 통하여 제안된 방법은 UML 다이어그램으로 작성된 모델을 입력으로 받아 SpecC프로그램을 이용하여 SpecC설계 사양 모델을 만들어낸다. 그림 20의 오른쪽 그림은 공통적인 동작을 공유하는 규칙에 따라 도출되는 사양 모델이다. 이 사양 모델은 필요한 자원이 적고 각각의 동작이 단순하게 되는 장점을 가지고 있지만, 공유하는 동작이 해제될 때까지 기다려야한다는 단점도 가지고 있다. 예를 들면 라우터 설계의 경우, 저비용을 선호하는 경우에는 공통적인 동작을 공유하는 조직이 적당하다. 한편, 속도와 성능을 우선한다면 공통적인 동작을 각자 소유하는 구성이 적당하다.

예제를 통한 실험 결과, 사양 모델은 최초의 실행가능 사양과 같게 동작했다. 이것은 제안된 방법이 외부 동작을 유지하면서 시스템을 상세화 할 수 있다는 것을 보여주고 있다. 또한 상세화를 했어도 버그는 발견되지 않았다. 이것은 제안된 방법이 리팩터링을 기초로 하고 있기 때문이며, 기존 리팩터링 규칙의 대부분이 단계적으로 상세화에 사용할 수 있음을 나타내고 있다.

5. 결론 및 향후 연구

본 연구에서는 리팩토링 기술을 이용하여 시스템 레벨 설계의 단계적 상세화를 실현하였다.

리팩터링에 근거한 단계적인 상세화는 상세화 프로세



[그림 20] 사양 모델에 해당하는 다른 예제 사양
[Fig. 20] Another example specification corresponds to the specification model

스의 어느 시점에서, 어떤 리팩터링 규칙을, 어떤 순서로 적용하는지를 결정하는 것이 중요하다. 본 연구에서는 UML 다이어그램을 입력으로 받아 SpecC 설계 기법 사양 모델을 생성하는 상세화 프로세스를 제안하였다. 그 과정은 기능별 그룹화, 채널의 추가, 포트의 추가 순서로 상세화 하였다.

본 연구를 통하여 지금까지는 시스템 레벨 언어를 알아야 설계할 수 있었던 임베디드 소프트웨어 설계를 보다 쉽게 일반 프로그래머들이 UML 다이어그램을 작성하면 시스템 레벨 설계까지 할 수 있다는 가능성을 보여주었다. 이것은 하위 수준의 시스템 설계를 시스템 언어를 가지고 설계해야만 하는 어려운 문제를 해결하게 되고, 설계의 생산성 또한 매우 클 것으로 기대된다.

향후 연구로는 본 연구에서 제안한 방법을 실제 시스템에 적용할 수 있도록 보완 및 확충하고, 제안된 방법의 효과와 현실성을 검증하는 것이다. 또한 설계 지원 도구들을 개발하여 특별한 지식이나 기술이 없이도 단계적으로 상세화가 가능하도록 하는 것이다. 그 결과 단계적 상세화에 필요한 시간을 줄일 수 있고, 결국 설계 기간과 개발 기간 단축에도 많은 기여를 할 것으로 생각된다.

References

[1] Daniel D. Gajski, Jianwen Zhu, Rainer D ¨ omer, Andreas Gerstlauer and Shuqing Zhao, "SpecC : Specification Lanugage and Methodology", Kluwer Academic Publishers, 2000

[2] Manjote S. Haworth, William P. Birmingham "Towards optimal system -level design, 2004

[3] 'Torsten Gr ¨ otker, Stan Liao, Grant Martin, and Stuart Swan, "System Design with SystemC, "Kluwer Academic Publishers, 2002

[4] Diederik Verkest, IMEC, Leuven, Belgium, Joachim Kunkel, Synopsys, Mountain View, CA, Frank Schirrmeister, Cadence Design Systems, San Jose, CA "System Level Design Using C++ ", '00 Proceedings of the conference on Design, automation and test in Europe

[5] P. Boulet, J.-L. Dekeyser, C. Dumoulin, and P. Marquent "MDA for SoC Embedded Systems Design, Intensive Signal Processing Experiment "SIVOES - MDA workshop at UML2003, pp.20 - 24, 2003.

[6] Stephen J. Mellor, John R. Wolfe, Campbell McCausland, "Why System - on - Chip Needs More UML like a Hole in the Head ", Proceedings of the Design, Automation, and Test in Europe (DATE05), Vol.2, pp.834 - 35, 2005.

[7] Andreas Gerstlauer, Rainer D ¨ omer, Junyu Peng and Daniel D. Gajski, "System Design : A Practical Guide with SpecC "Kluwer Academic Publishers, 2001.

[8] Borba, P. "An Introduction to Software Product Line Refactoring" Lecture Notes in Computer Science, Vol.-No.6491 pp.1-26, 2011

[9] Clark, D. Chen, M. Tucker, J. "Automatic Program Translation" INTERNATIONAL SYMPOSIUM ON MULTIMEDIA SOFTWARE ENGINEERING, Vol.6 pp.265-272, 2004.

김 현 종(Hyun-Jong Kim) **[정회원]**

- 1999년 2월 : 공주대학교 교육대학원 수학교육과 졸업(교육학 석사)
- 2002년 8월 : 공주대학교 교육정보대학원 교육정보관리과 졸(교육정보학 석사)
- 2009년 3월 ~ 현재 : 공주대학교 컴퓨터공학과 박사과정

<관심분야>
소프트웨어공학, 소프트웨어 설계

공 헌 택(Heon-Tag Kong) **[정회원]**

- 1984년 : Northeast Missouri State Univ. 전산학과(학사)
- 1987년 Utah State Univ. 전산학과(석사)
- 1998년 : 단국대학교 전산통계학과(박사)
- 1988년 ~ 1990년 : 한국국방연구원 전산체계연구부 근무
- 1990년 ~ 현재 : 공주대학교 컴퓨터공학부 교수

<관심분야>
병렬 알고리즘, 병렬처리 컴퓨터, 데이터베이스

김 치 수(Chi-Su Kim) **[정회원]**

- 1984년 : 중앙대학교 전자계산학과 졸업(학사)
- 1986년 : 중앙대학교 대학원 전자계산학과 졸업(석사)
- 1990년 : 중앙대학교 대학원 전자계산학과 졸업(박사)
- 1990년 9월 ~ 1992년 8월 : 공주교육대학교 전임강사
- 1992년 ~ 현재 : 공주대학교 컴퓨터공학부 교수

<관심분야>
소프트웨어 개발 방법론, 소프트웨어 설계