SPARK Examiner를 이용해 ANSI-C프로그램의 안전성을 분석하기 위한 C언어의 제약 조건과 변환 방법

(Restrictions and translation rules of ANSI-C language for analyzing integrity of C program using SPARK Examiner)

김 진 섭 † 차 성 덕 ††
(Kim, jin-sup) (Cha, Sung-deok)

요 약 일반적으로 C언어는 고신뢰도를 요구하는 소프트웨어에 적합하지 않다고 알려졌음에도 불구하고 적지 않은 수의 안전성이 중요시되는(safety-critical) 시스템들이 C언어로 구현되었거나 C언어를 기반으로 개발되고 있다. 본 연구의 목적은 C언어의 안전한 subset을 정의하고 이 subset 언어로 작성된 프로그램을 SPARK Ada로 변환하여 SPARK의 분석 도구들을 사용해 프로그램의 안전성을 분석하는 데 있다. SPARK은 Ada의 안전한 subset으로 고신뢰도를 요하는 시스템을 구현하는데 성공적으로 사용되어 왔다. SPARK으로 변환된 C 프로그램은 SPARK 수준의 안전성을 갖게 되며 SPARK의 분석 도구인 Examiner를 통해 프로그램의 정확성 검증 등의 분석을 할 수 있다. 본 연구에서는 엘리베이터 컨트롤러 사례 연구를 통해 정의한 subset이 현실적인 시스템을 구현하기에 부족하지 않음을 발견하였으며, SPARK Ada로의 변환을 자동화해주는 변환기를 구현하였다.

키워드 : SPARK, 프로그램 분석, 프로그램 변환

Abstract The C language is widely adopted for safety-critical systems. However, it is known that the C language is an unsuitable choice for safety-critical system since the C language includes several bad language features such as heavy use of pointers. The aim of this work is to define safe subset of the C language and translate the subset into the SPARK Ada so that we can verify the program' s safety using SPARK analysis tools. SPARK is a safe subset of Ada and has been successfully applied to high integrity system development. The C program translated into SPARK has the same integrity level as SPARK, and the program correctness can be verified by using Examiner which is a SPARK analysis tool. An elevator controller case study is presented and is used to demonstrate the potential use of our approach to implement a realistic system. We also developed a translator that automatically translates C code into SPARK in accordance with the translation rules.

Key words : SPARK, Program Analysis, Program Translation, Safety-critical

1. 서 론

안전성이 중요시되는 (safety-critical) 시스템은 설계상의 오류로 인한 사고가 치명적인 결과로 이어지기 때문에, 이러한 시스템에서 사용되는 소프트웨어는 높은 수준의 신뢰도를 필요로 한다. 소프트웨어의 신뢰도에 영향을 주는 요소들은 여러 가지가 있는데 구현 단계에서 선택하는 프로그래밍 언어 역시 많은 영향을 주는 요소 중 하나이다[1]. 일부 프로그래밍 언어가 가진 요소들은 다른 언어들보다 더 문제를 일으킬 여지가 많은데, 어떤 요소들은 프로그래머가 의도한 것과 다르게 행동할 수 있고, 분석이나 검증을 어렵게 할 수도 있기 때문이다. 따라서 이러한 요소들을 포함하고 있는 프로그래밍 언어는 안전성이 중요시되는 소프트웨어를 구현하는데 적합하지 않다.

† 학생회원 : 한국과학기술원 전자전산학과
jskim@salmosa.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전자전산학과 교수
cha@salmosa.kaist.ac.kr
논문접수 : 2002년 8월 21일
심사완료 : 2003년 2월 21일

C언어는 많은 분야에서 널리 사용되고 있는 언어이지만 일반적으로 안전성이 중요시되는 시스템에 적합하지 않다고 알려져 있는데 C언어가 가지고 있는 여러 특징들이 고신뢰도를 요구하는 소프트웨어에 사용할 때 문제가 발생할 가능성을 높이기 때문이다. 그럼에도 불구하고, 예산상의 이유로 또는 선택할만한 다른 언어가 없어 적지않은 수의 안전성이 중요시되는 시스템들이 C언어로 구현되었거나 구현되고 있다.

C 언어를 사용해 좀 더 안전하게 프로그램 하기 위한 방법으로 산업계의 엔지니어들에 의해 여러 가지 지침(guideline)들이 만들어져 왔다.[2] 이것은 C언어가 가지고 있는 문제들을 분석하고 그것을 피하기 위해 노력해 온 사람들의 오랜 경험에 의해 얻어진 것으로, "어떤 방법으로 더 안전한 코드를 작성할 것인가?" 라는 질문에 대한 대답을 긴 목록으로 만든 것이다. 여기에는 변수나 함수의 작명 관습(naming convention), 구문 제한 규칙, 포인터나 union 타입 등을 사용할 때 따라야 할 제약 조건들이 포함된다. 비록 이러한 지침(guideline)들이 코드의 안전성을 완벽히 보장하지는 못하지만, 여러 프로젝트에서 프로그램 코드의 에러를 감소시키는 데 성공적으로 기여해 왔다.

Lint 계열의 전통적인 분석 도구들은 C 코드에 대한 간단한 분석을 수행한다. 구문 분석과 기본적인 제어 흐름(control flow) 분석에 더하여 최근의 LC/PC Lint같은 프로그램은[3] 간단한 정적 분석(static analysis) 기능을 가지고 있어 array bound violation이나 division-by-zero의 가능성이 있을 경우 이를 경고해 준다.

좀 더 안전한 프로그램을 만들기 위해 C의 불안전한 특징중 하나인 포인터나 배열을 오류 없이 사용하게 하기 위한 방법 역시 여러 분야에서 연구되고 있다. 정보보호 분야에서 몇 개의 연구가 진행되었는데 CCured[4]와 Cyclone[5]등이 대표적인 예이다. 이들은 포인터를 그 사용 방법에 따라 몇 가지 종류로 분류하고 각각의 포인터 종류에 따라 필요한 run-time 체크 루틴을 삽입하는 방법으로 발생할 수 있는 거의 모든 메모리 관련 에러들을 검사할 수 있다.

그러나 높은 수준의 안전성이 필요한 소프트웨어의 경우, 코딩 스타일에 대한 지침(guideline)을 제공하거나 에러의 가능성에 대해 경고해 주는 것으로는 부족하며 소프트웨어가 실제로 실행되기 이전에 오류가 없음을 분석하고 검증할 수 있어야 한다. 에러가 발생했을 때 이를 분석하여 소프트웨어의 수정에 반영할 수 있는 일반적인 경우와는 달리 안전성이 중요한 시스템에서는 run-time 에러 자체가 허용되지 않기 때문이다.

특정한 종류의 에러가 발생하지 않음을 보장하는 것에 더하여 안전성이 중요한 시스템의 소프트웨어는, 의도한 대로 동작한다는 것 또한 검증되어야 한다. 다시 말하면 프로그램의 정확성이 검증되어야 한다.

Carre등은[6] 고신뢰도를 요하는 시스템에 적합한 프로그래밍 언어가 만족해야 할 기준을 제안하고 그 기준을 만족하도록 Ada의 subset을 정의하였다. SPARK[6]이라는 이름의 이 subset은 안전한 프로그래밍 언어가 만족해야 할 조건을 갖추고 있으며, 코드에 대한 분석 도구인 Examiner를 통해 프로그램이 의도한 대로 동작하고 실행중에 특정한 종류의 에러를 발생시키지 않음을 검증할 수 있다.

만약 C와 같은 행위를 갖는 SPARK코드를 만들 수 있다면 그 자체로 언어의 안전함을 보일 수 있고 변환된 C프로그램이 SPARK이 제공하는 수준 만큼의 안전성과 신뢰도를 갖게 할 수 있으며 Examiner와 같은 프로그램 분석 도구들을 사용해 프로그램의 정확성을 검증할 수 있다. 본 논문에서는 이를 위해 C 언어로 작성된 코드를 SPARK으로 변환하는 방법을 제안하고, 이 변환 규칙을 적용하여 자동으로 C코드를 SPARK으로 변환해 주는 변환기를 구현하였다.

그림 1 에서는 논문 연구의 전체 구조를 보여준다.

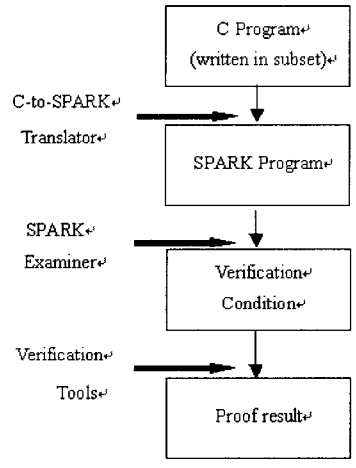

그림 1 C 프로그램 분석의 Framework

2. SPARK과 SPARK Examiner

SPARK은 Ada의 안전한 subset으로 SPADE tool[6]을 이용해 분석을 할 수 있는 Ada의 특성들을 추출하여 만들어진 것이 SPADE Ada Kernel 즉, SPARK이

다. SPARK은 Ada의 안전한 subset과 프로그램에 대한 정보 제공을 위한 주석으로 구성된다. 이런 두 가지의 요소로 구성된 프로그램은 SPARK Examiner를 통해서 분석되어진다. Examiner는 프로그램의 데이타 및 컨트롤 플로우를 분석하고, verification condition을 생성하여 증명 툴을 사용해 검증할 수 있게 한다.

```
procedure swap(x, y : in out integer)
--# derives x from y &&
--#          y from x;
--# post x = y~ && y = x~
is
    temp : integer;
begin
    temp := x;  x := y;  y := temp;
end swap;
```

위 예제는 간단한 SPARK 프로그램의 형태를 보여준다. SPARK 프로그램은 제한 규칙 하의 Ada 코드와 함께 '--#'로 시작하는 주석으로 구성되어 있는데, Examiner는 주석에 명시된 조건들을 코드가 만족하는지 검증하게 된다. 이 예제에서는 변수간의 의존성(dependency)을 분석하기 위해 'derive' 주석을 사용하였고, 변수들의 값이 올바로 결정되었는가를 보기 위해 'post' 주석을 사용하였다. Examiner는 이 코드를 분석하여 변수 x, y가 서로 의존 관계가 있으며 각 변수의 값은 다른 하나의 초기값(x~; y~)과 같음을 확인하고 명시된 주석이 올바름을 알려 준다.

본 논문에서 정의하는 C의 안전한 subset 모델은, SPARK으로 변환하여 Examiner를 통해 검증을 할 수 있다. 다음은 Ada의 subset을 선택할 때 고려되어진 특성들 중 몇 가지이다[6].

Logical Soundness - 프로그래밍 언어는 논리적인 응집성이 있어야 하고 모호해서는 안 된다.

Complexity of formal language definition - 언어의 정형적인 정의는 컴파일러 설계자가 정확한 컴파일러를 만들 수 있고, 프로그램을 검증을 수월하게 할 수 있을 만큼 충분히 간단해야 한다.

Expressive power - 복잡한 함수를 힘들지 않게 기술할 수 있고 컴파일러에 의해 자동으로 검사될 수 있어야 한다.

Security - 실행 시간 전에 언어에서 금지하는 것들에 대한 위반 사항을 찾아내어 보고해 줄 수 있어야 한다.

Bounded space and time - 힙과 같은 동적인 저장공간에 대한 모든 요구 사항들은 정적으로 분석되어질 필요가 있다.

이러한 엄격한 요구 사항들로 인해 결과적으로 SPARK은 full Ada에 비해 제한적이 된다. 그러나 이러한 제한들은 SPARK을 Ada의 안전한 subset으로서 고신뢰도를 요하는 안전성이 중요시되는 시스템에 적합한 언어가 되게 해 준다.

실제 산업계에서도 SPARK의 접근 방법을 사용해 성과를 올린 예가 있다. 전자 상거래를 위한 전자 카드에 사용되는 운영 체제, 헬기의 착륙 제어 장치, 화물 수송기의 운항 제어 시스템 등, 100,000 라인 이상의 코드 크기를 갖는 큰 규모의 시스템에 사용되어 개발에 성공적으로 기여해 왔다[7].

그림 2는 전체적인 SPARK tool의 검증 프로세스를 나타낸 것이다.

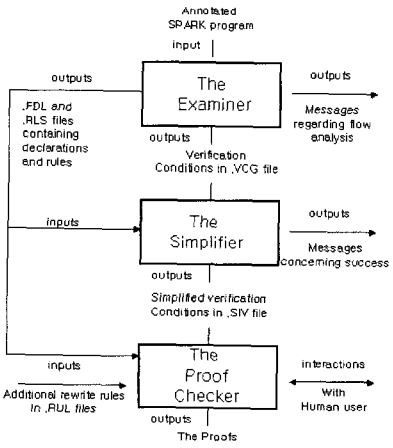

그림 2 SPARK 검증 프로세스

3. C언어의 안전성 분석과 SPARK으로의 변환

C언어의 특징들 가운데는 실행 중에 에러를 발생하게 할 가능성이 있는 잠재적인 위험 요소들과 프로그램이 의도한 대로 행동한다는 것을 확인하기 위한 분석을 어렵게 하는 요소들이 있다. Haton은[8] 이러한 요소들을 구체적으로 분석하여 문서화했는데 C언어에 적절한 제한을 가한다면 그러한 요소들을 피할 수 있다. 또한 C언어로 작성된 프로그램을 SPARK으로 변환하기 위해서는 그것을 SPARK의 구조나 구문으로 표현할 수 있어야 하는데, 이를 위해서도 SPARK으로 변환할 수 없

는 것들에 대한 제한이 필요하다. 본 연구에서는 ANSI C [9]에서 제공하는 모든 특징들에 대해 분석하면서 제한되어야 하는 요소를 찾고 SPARK으로의 변환 방법에 대해 정의하였다. 주목할 만한 몇 가지 특징들은 다음과 같다.

3.1 타입

C언어가 가지고 있는 안정성을 위협하는 특징 중 하나는 타입 모델이 엄격하지 못하다는 것이다. 서로 다른 타입들 간의 전환이 암시적으로 일어나고 컴파일시에 타입 검사를 하지 않기 때문에 타입 에러가 일어날 가능성이 매우 높아진다. 예를 들어 다음과 같은 경우 일반적으로 기대되는 것과는 다른 결과를 보인다.

```
unsigned a = 6;
int b = -20;
...
if (a+b > 6 ) {  // actually TRUE
...
```

그림 3 타입 에러를 갖는 C프로그램의 예

이 예제가 가지고 있는 문제는 if문의 조건이 실제로는 참(TRUE)으로 계산된다는 것이다. C가 int 타입을 자동으로 unsigned 타입으로 바꾸기 때문에 변수 b의 값은 조건식 안에서 전혀 다른 값이 되어 버린다.

따라서 C언어를 강한 타입 모델을 갖는 언어와 흡사하게 만들기 위해 제한을 가해야 한다. 우선 암시적인 타입 전환을 금지하고 모든 타입 전환은 캐스트 연산자를 통해 명시적으로 이루어지게 한다. 명시적인 타입 전환 없이 서로 다른 타입간의 연산을 금지한다. 이렇게 하는 것은 컴파일 시간에 모든 변수의 타입이 결정되게 하여 타입 에러를 검사할 수 있게 해 준다.

또한 위의 예제에서 설명한 것처럼 한 가지 부류의 타입은 하나만 존재하는 것이 바람직하다. 같은 정수를 표현하는 방법이 int와 unsigned가 서로 다르기 때문에 두 타입 간의 전환이 문제가 되었기 때문이다. 그러므로 가장 기본이 되는 세 가지 타입 - int, char, float - 만을 사용하기로 한다.

이에 대한 제한 사항들을 다음과 같이 정리할 수 있다.

제한 1. short, long, unsigned, double 타입을 사용할 수 없다.

제한 2. 타입 전환은 반드시 cast 연산자에 의해 명시적으로 표시되어야 하고 int와 float, int와 char간에만 가능하다 (열거형, 구조체, 배열 타입과도 변환될 수 없다.)

제한 3. 서로 다른 타입의 피연산자는 형 변환을 하지 않고는 함께 계산될 수 없다.

3.2 연산자

C에서 제공하는 모든 연산자들 또한 그대로 사용될 수 없으며 그대로 SPARK으로 변환되지 못한다. 연산자에 대한 제한 사항의 첫 번째 이유는 C에 불리안 타입이 없기 때문이다. 0이 아닌 모든 정수와 0을 각각 True와 False로 대신해 사용하기 때문에 논리 연산이나 비교 연산의 피연산자로 정수형만이 사용되게 된다. 그러나 SPARK에는 불리안 타입이 존재하며 정수형과는 엄격히 구별되므로 논문에서 제안하는 subset에서도 두 타입 간의 혼용을 막을 필요가 있다. 이를 해결하기 위한 방법은 C에도 가상의 불리안 타입이 존재한다고 가정한 뒤 각각의 연산자들이 어떤 타입 아래서 정의되는지를 재정의하는 것이다. 다음의 표는 불리안 타입과 정수형을 구분했을 때 각 연산자를 정의한 것이다.

표 1 피연산자와 연산 결과의 타입에 따른 분류

operator	numerical	relational	equality	logical
	+, -, *, /, %	<, <=, >, >=	==, !=	&&, ||
operand type	int float	int float char	int float char boolean	boolean
result type	int float	boolean	boolean	boolean

표 1을 참조해 보면 불리안 타입과 정수형을 구분하기 위해 다음과 같은 제한 사항이 필요함을 알 수 있다.

제한 4. relational operator(<, <=, >, >=)와 equality operator(==, !=), 그리고 logical operator(&&, ||)의 연산결과는 불리안 타입으로 간주하며 relational operator(<, <=, >, >=)와 numerical operator(+, -, *, /, &)의 operand로 사용될 수 없다.

예를 들어, 어떤 C 프로그래머는 a의 값이 b보다 큰 경우를 세기 위해 다음과 같은 코드를 만들 지 모른다.

count = count + (a > b);

그러나 논문에서 제안하는 subset에서는, a > b 의 결과를 불리안 타입으로 간주하므로 정수 변수 count와의 연산은 금지된다. 제한 사항을 만족하면서 같은 의미를 지니도록 다음과 같이 수정하여야 한다.

if(a > b)

```
count = count + 1;
```

연산자에 대한 제한 사항의 두 번째 이유는 특정한 축약형 연산자가 갖는 부수 효과(side-effect)때문이다. 다음의 예를 고려해 보자.

```
x = i++ + a [ i ] ;
```

i++은 거의 모든 C프로그래머가 빈번하게 쓰는 표현으로 i의 값을 하나 증가시키는 부수 효과(side-effect)를 갖는다. 이 문장은 안전성과 관련된 문제를 갖고 있는데 오른쪽의 i++항과 왼쪽의 a[i]중에서 어느 쪽의 값을 먼저 계산하느냐에 따라 덧셈의 결과가 달라진다. 그 결과 수식의 계산 순서가 엄격하게 정의되지 않는다면 프로그램의 결과를 정확히 예측할 수 없게 된다. 그러나 C를 포함한 대부분의 언어들은 수식의 계산 순서와 같은 것들에 대해 아무 것도 정의하고 있지 않다. 하지만 우리가 유의해 보아야 할 것은 부수 효과(side-effect)가 있는 수식의 사용을 금지한다면 정해지지 않은 계산 순서로 인한 모호성을 극복할 수 있다는 점이다. 한쪽의 값을 계산하는 행위가 다른 쪽의 계산 결과에 영향을 미치지 않기 때문이다. 실제로 SPARK에서도 수식 계산 순서를 규정하고 있지 않지만 부수 효과(side-effect)를 막음으로 수식 계산 과정에서의 모호성을 없앴다.

C의 연산자들 중 부수 효과(side-effect)를 유발할 수 있는 것은 앞서 예에서 설명되었던 증가, 감소 연산자(++, --)이다. 따라서 이 연산자의 사용은 금지되어야 한다.

제한 5. 증가, 감소 연산자 (++, --)를 사용할 수 없다.

연산자에 대한 마지막 제한 사항은 할당 연산자에 관해서이다. C언어를 처음 배우는 사람들이 혼란을 느끼는 부분 중 하나는 할당문 자체가 값을 갖는다는 것인데 이것은 숙련된 C프로그래머에게는 간결한 코드를 작성하게 해 주기도 한다. 예를 들어 다음의 프로그램에서는 while문의 종료조건을 비교할 때 i에 값을 할당하고 나서 그 값을 종료조건과 비교하게 된다.

```
while( (i = get_Value()) != 0) {
        if ( i > 10 )
. . .
}
```

여기서 발생할 수 있는 문제는 증가, 감소 연산자와 마찬가지로 수식 안에 변수의 값을 수정하는 부수 효과(side-effect)가 포함된다는 것이다. 할당문을 값을 갖는 수식으로 인정했을 때 발생하는 모호성의 예는 쉽게 생각할 수 있다.

```
x = (a = 0) + a;
```

그러므로 부수 효과(side-effect)를 갖는 수식을 제거하기 위해, 즉 expression과 statement를 구별해 사용하기 위해 할당문은 수식 안에서 사용하지 말아야 할 것이다.

제한 6. 할당문(assignment statement)는 자체로 값을 가지지 않으며 수식 가운데 포함될 수 없다.

부수 효과(side-effect)를 일으킬 수 있는 또 다른 예로는 값을 리턴하는 함수에서 전역 변수의 값을 수정하는 경우이다. 이 경우도 수식의 값을 계산하는 행위가 수식에 포함된 다른 변수의 값에 영향을 줄 수 있으므로 부수 효과(side-effect)에 의한 문제가 발생할 수 있다. 따라서, void 타입이 아닌 모든 함수의 본체 안에서는 전역 변수의 값을 수정할 수 없어야 한다.

제한 7. 값을 리턴하는 모든 함수(void 타입을 제외한)는 함수 본체 내에서 전역 변수의 값을 수정할 수 없다.

3.3 구조화되지 않은 프로그램

구조적 프로그래밍은 정확하고 신뢰할 만한 프로그램을 만들기 위해 오래 전부터 강조되어온 원칙이다. 많은 프로그래밍 에러의 원인이 계획되지 않은 goto문의 남용이라는 것은 널리 알려진 사실이며 대부분의 프로그래머는 goto문을 사용하지 않을 지도 모른다. 그러나 goto문은 결코 안전한 언어의 기준에 부합되지 않으며 goto문의 사용을 금지하는 것은 언어의 안전성을 높이기 위해 가장 먼저 해야 할 일 중에 하나이다. SPARK에서도 statement에 label을 달거나 goto를 사용할 수 없다. 또한 다중 entry/exit point를 갖는 반복문(loop) 역시 구조화되지 않은 프로그램이 되게 하므로 반복문 내에서 continue와 break문을 금지하였다.

제한 8. goto 문을 사용할 수 없다.

제한 9. 반복문 내부에서 continue문을 사용할 수 없다.

제한 10. 반복문 내부에서 반복문을 벗어나기 위해 break문을 사용할 수 없다.

3.4 분리 컴파일

C에서는 함수들의 프로토타입, 상수, 사용자 정의 타입 등이 선언된 헤더 파일과, 선언된 함수가 실제로 구현된 소스 파일이 하나의 라이브러리 단위를 구성한다. 다른 파일에서 선언된 함수나 타입을 사용하기 위해서는 그 파일의 헤더를 '#include'라는 전처리기 지시자를 사용해 참조하게 된다. SPARK에서도 이와 비슷한 방법으로 분리 컴파일이 이루어지는데, Ada가 객체 지향적 요소를 지니고 있는 언어이기 때문에, 각각의 라이브

러리 단위들은 클래스와 유사한 개념인 패키지라는 단위로 묶이게 된다. 패키지는 일반적으로 패키지 명세와 패키지 본체의 두 가지로 구성된다. 패키지 명세는 그 패키지에서 사용되는 함수의 프로토타입과 사용자 정의 타입, 상수, 전역 변수가 선언되어 있고, 패키지 본체에서는 함수들의 실제 코드를 정의한다. 다른 패키지에서 선언된 항목을 사용하기 위해서는 'with' 절을 사용해 접근하고자 하는 패키지의 이름을 프로그램의 처음에서 명시해 준다. 따라서 라이브러리 단위의 변환은 매우 자연스럽게 이루어질 수 있는데 C 소스 파일 이름을 패키지 이름으로 사용하여 헤더 파일을 패키지 명세로, 소스 파일을 패키지 본체로 변환할 수 있다.

그러나, 헤더를 갖지 않은 소스파일은 패키지를 사용해 변환할 수 없다. 이 경우 패키지 명세 없이 코드를 바로 변환해야 하는데, 이 때 몇 가지 제한 사항이 필요하다.

제한 11. 헤더 파일이 없는 소스 파일에는 단 하나의 함수 정의만 있어야 한다

SPARK의 프로그램 단위는 패키지와 서브프로그램으로 구성되는데 서브프로그램은 하나의 function(또는 procedure)를 말한다. 전체 SPARK 프로그램 가운데 서브프로그램은 단 하나만 존재해야 하며 이것은 당연히 메인 서브프로그램이 되어야 한다. 따라서 헤더 파일이 없는 소스 파일이 SPARK의 서브프로그램으로 변환되기 위해서는 main 함수(또는 그에 해당하는 다른 함수) 하나만 포함하고 있어야 한다.

고려해야 할 다른 하나는 C와 SPARK에서 다른 라이브러리 단위에서 선언된 항목들을 사용하는 방법의 차이이다. C의 #include 지시자는 헤더 파일의 내용을 그대로 치환해 넣으므로 다른 파일에서 선언된 여러 가지 항목들의 이름을 그대로 쓸 수 있지만 SPARK의 with 절은 어떤 패키지를 사용하겠다는 것을 컴파일러에게 알리는 것 이상의 일을 하지 않는다. SPARK에서 다른 패키지에서 선언된 항목에 접근하기 위해서는 패키지 이름을 접두사로 앞에 붙여 주어야 한다. Ada에서는 use 절을 사용하여 패키지 이름을 생략할 수 있지만 SPARK에서는 패키지 이름을 생략하는 것이 항목들이 어디에서 선언되었는지 알 수 없게 하여 프로그램을 읽고 이해하기 어렵게 한다는 이유로 use 절의 사용을 금지한다. 따라서 다른 파일에서 선언된 항목이 변환될 때는 그 항목이 선언된 패키지 이름을 접두사로 앞에 붙여 주어야 한다. 이를 위해 코드를 변환하기 전에, 프로그램에서 참조하는 모든 헤더 파일에서 선언된 식별자(형 이름, 함수 이름, 상수 이름, 전역 변수 이름, 열거성 상수 이름)와 그 식별자가 선언된 파일 이름을 관련지어 주는 검색 테이블을 만들고, 실제 변환시 변환 대상이 되는 식별자가 테이블 안에 있다면 해당하는 파일 이름을 패키지 접두사로 삼아 식별자 앞에 추가한다.

분리 컴파일에 대한 변환 예는 다음과 같다.

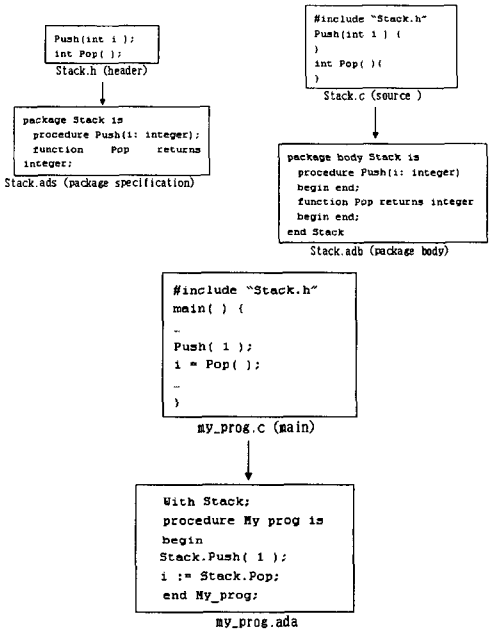

그림 4 라이브러리 유닛과 라이브러리 함수의 변환 예

3.5 포인터

앞에서 살펴보았듯이 포인터는 C에게 엄청난 융통성을 제공하지만 그만큼 C를 위험하게 만드는 요소이다. 그러나 포인터를 금지하는 것을 C를 절름발이 언어로 만들 수 있는데 참조에 의한 호출(call by reference)과 같은 것들은 포인터 없이 구현할 수 없기 때문이다. 그러나 SPARK에서는 동적인 메모리 할당이 금지되므로 사실상 포인터는 오로지 참조에 의한 호출을 구현하기 위해서만 필요하다. 따라서 참조에 의한 호출을 구현하기 위한 포인터 사용을 제외한 모든 포인터 관련 연산은 금지한다. 일반적으로 참조에 의한 호출은 다음과 같이 구현된다.

```
void swap(int *x, int *y) {
  int temp;
  temp = *x;
  *x = *y;
```

```
    *y = temp
};

swap(&x, &y);
```

참조에 의한 호출(call by reference)이 이루어지는 과정을 살펴보면, 값을 변화시키고자 하는 변수의 주소를 인자로 해서 참조에 의한 호출(call by reference)이 구현된 함수를 호출한다. 호출 받은 함수는 인자로 넘어온 주소가 가리키고 있는 변수의 값을 참조하거나 변경하기 위해 참조 연산자(*)를 사용한다. 따라서 어떤 로컬 변수도 포인터로 선언될 필요가 없다. 그리고 함수 본체 안에서, 인자로 넘어온 포인터 변수는 반드시 그 포인터가 가리키는 변수의 값을 참조하기 위해서만 사용되어야 한다.

포인터와 관련된 제한 사항들은 다음과 같다.

제한 12. 포인터 연산은 사용하지 않는다.

제한 13. 포인터 변수에 address 연산자(&)를 붙일 수 없다.

제한 14. 다중 포인터(포인터의 포인터)는 사용하지 않는다.

제한 15. 함수 포인터는 사용하지 않는다.

제한 16. 전역, 지역 변수를 포인터로 선언할 수 없다.

제한 17. address 연산자(&)는 대상 변수가 함수의 매개변수(function parameter)일 때만 사용할 수 있다.

제한 18. 포인터 변수는 함수의 매개변수(function parameter)로 사용되는 경우를 제외하고 항상 dereferencing operator(*), 또는 indirect member operator(->) 와 함께 사용되어야 한다.

이러한 제한 사항을 위반하는 코드의 예를 들면 다음과 같다.

```
void illegal_use_of_pointer(int *x)
{
1:      int *a;
2:      int b;
3:      a = a + 1;
4:      a = &b;
5:      x = temp;
}
```

1행에서는 포인터 변수 a가 로컬 변수로 선언되었으므로 제한 13을 위반하였다. 또한, 포인터 연산을 금지하는 제한 9에 의해 3행의 코드는 받아들여지지 않는다. 4행의 address(&)연산자의 사용도 함수의 인자로 넘기기 위한 것이 아니므로 제한 14에 의해 금지된다. 5행의 코드 역시 제한 15를 살펴보면 올바르지 않음을 알 수 있다. 함수 본체(body) 내에서 포인터 변수는 항상 '*', 또는 '->'와 같이 사용되어야 한다. 따라서 5행과 같은 행위를 갖게 하려면 다음과 같이 써 주어야 한다.

```
*x = *temp;
```

이 밖에도 타입 전환, 연산자 우선 순위, 사용 가능한 연산자의 종류, 배열이나 열거형의 사용과 관련된 제한 사항들 역시 정리하였다.

4. 변환 규칙의 정의

위에서 정리한 제한 사항 하에서 C프로그램을 SPARK으로 변환하기 위한 변환 규칙을 정의하였다. 몇 가지 예를 들면 다음과 같다.

```
* function declaration :
S ( return_type function_name ([argument] {, argument} ) ; )
  =
function function_name ( A ([argument] {; argument} ) )
return T(return_type) ;

* constant declaration :
S ( const type variable_name = initial_value;)
  =
variable_name : constant T (type) := initial_value;

* switch statement :
S ( switch ( expression ) {
    { case constant_expressioin :
       { case constant_expression : }
           statement { statement }
            break;
    }
   [ default : statement { statement } ]
 }
)
=
case E (expression ) is
  { when E (constant_expressioin ) { | E (constant_expression) } =>
      S (statement { statement } )
  }
  [ when others => S (statement { statement } )
end case;
```

S(), E()등의 함수가 인자로 C프로그램을 받아 대응하는 SPARK 코드를 생성하는 변환 함수이다. boldface 문자는 keyword이고 expression, statement등은 formal parameter이다.

예를 들어, 함수 선언에 대한 변환 규칙을 적용한 프로그램 변환은 다음과 같을 것이다.

```
S ( int sum ( int x, int y ); )
        =>
function sum (Integer x; Integer y) return Integer;
```

statement 변환 함수인 S( )는 전체 C 코드를 받아 대응되는 SPARK 코드로 변환한다. 위 예제에서 S( ) 의 인자들 중 return_type, function_name은 각각 'int', 'sum'이 될 것이고 type 변환 함수인 T( )에 의해 'int' 는 'Integer'로 변환된다.

5. 프로그램 분석을 위한 변환기의 구현

C 프로그램을 SPARK으로 변환하고 이를 Examiner로 검증하기는 과정을 자동화하기 위해 변환기를 구현하였다.

그림 5는 변환기를 사용해 C 프로그램을 읽어 SPARK으로 변환을 마친 상태의 화면을 보여 준다.

왼쪽 창의 코드가 입력된 C프로그램이며 오른쪽 창의 코드는 자동으로 변환된 SPARK프로그램이다.

이 도구를 사용해 프로그램을 검증하는 전체 과정을 살펴보기로 하자. 먼저 C 프로그래머는 검증하기 원하는 코드를 작성할 것이다.

```
void swap (int *x, int y) {
        int temp;

        temp = *x;
        *x = *y;
        *y = *temp;
}
```

이 코드를 SPARK으로 변환하기 전에 Examiner를 통한 분석을 위해 annotation을 삽입할 수 있다. SPARK코드상의 annotation위치에 상응하는 곳에 프로그램이 만족해야 할 특징을 기술한 annotation을 써 준다. SPARK의 annotation은 본질적으로 주석이며 컴파일러는 이를 무시한다. 따라서 C의 경우 역시 주석으로 처리하여 변환기에 의해서만 변환되도록 하였다.

```
void swap (int *x, int y)
//--# derive x from y&
//--#          y from x;
```

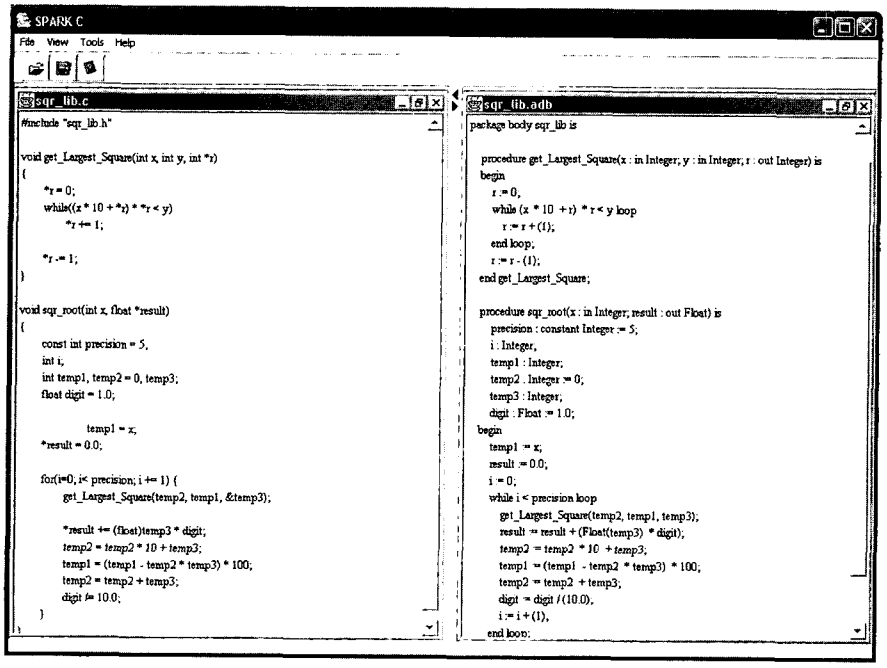

그림 5 SPARKC 변환기를 사용한 변환

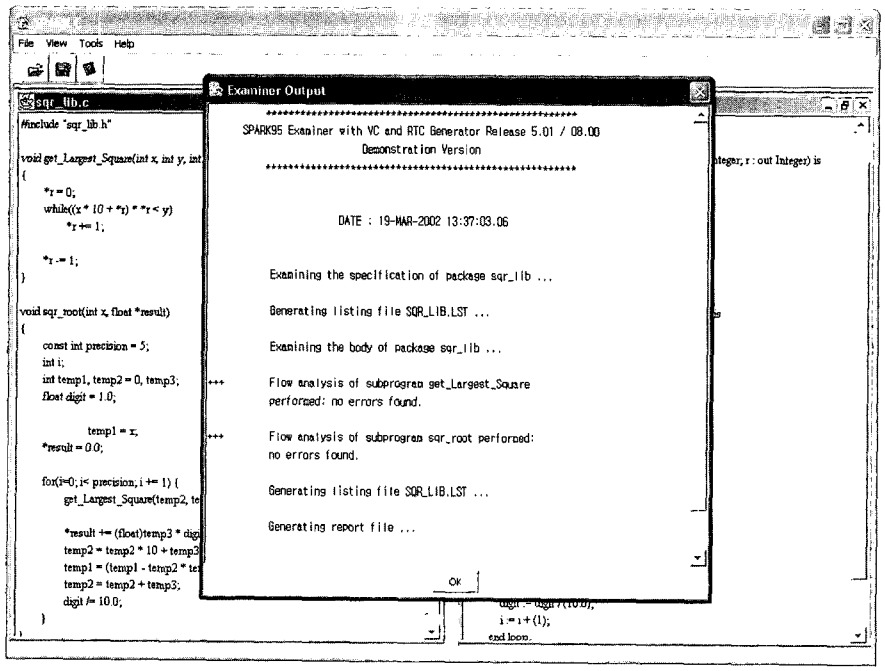

그림 6 Examiner의 분석 결과

```
//--# post x = ~y & y = ~x;
{
        int temp;

        temp = *x;
        *x = *y;
        *y = *temp;
}
```

위 프로그램은 논문에서 제안한 C 언어의 제한 사항들을 만족하므로 변환기를 통해 다음과 같이 성공적으로 변환된다.

```
procedure swap(x, y : in out integer)
--# derives x from y &&
--#          y from x;
--# post x = y~ && y = x~
is
    temp : integer;
begin
    temp := x;  x := y;  y := temp;
end swap;
```

변환된 SPARK프로그램을 검증하기 위한 도구인 Examiner를 변환기에서 직접 호출해 그 결과를 확인할 수 있도록 구현함으로 전체 검증 과정을 하나의 통합된 환경에서 수행할 수 있게 하였다. 그림 6에서는 변환된 SPARK프로그램을 Examiner로 분석한 결과가 어떻게 나타나는 지 보여준다.

위 예제를 Examiner를 사용해 분석한 결과는 다음과 같다.

```
+++ Flow analysis of subprogram swap performed:
    no errors found.
```

분석 결과 프로그램 코드가 제공된 annotation과 일치함을 확인할 수 있다.

6. 사례 연구

본 논문에서 제안한 제한 사항들이 지나치게 제한적이어서 비현실적인 면은 없는지 확인하기 위하여, subset에 관한 지식은 없지만 학부 과정 이상의 전산학 지식과 프로그래밍 기술을 가지고 있으며 C언어를 사용한 경험이 있는 프로그래머가 subset의 제한 사항 아래서 엘리베이터 컨트롤러 시스템[10]을 구현하였다.

두 명의 C 프로그래머가 이 실험에 참여했으며 최종 결과 코드는 약 100~200라인 정도 되었다. 실험 결과 subset의 표현력은 요구 사항을 만족하도록 시스템을 구현하는데 충분하였으며 제한 사항에 걸려 기술하지 못했던 요소는 없었다.

그러나 몇몇 요소들은 나름대로의 코딩 스타일을 가진 프로그래머에게 불편하게 느껴지기도 했으며, 제한 사항을 만족하도록 코드를 재구성하는 일이 쉽지만은 않은 경우도 있었다.

예를 들어 부수 효과(side-effect)를 갖기 때문에 금지한 증가/감소 연산자(++/--)는 C언어에서 제공하는 축약형 표현 가운데 가장 많이 사용되는 것 중 하나이므로 경험 있는 C프로그래머에게는 다소 불편하게 느껴질 수 도 있다.

또한, SPARK에서는 프로그램의 분석을 위해 모든 반복문은 단 하나의 출구(exit point)를 가져야 한다. 따라서 반복문 안에서 break문 등으로 여기저기서 빠져나갈 수 없으며 이러한 위험성을 지닌 break문의 사용을 금지하였다. 하지만 일부 프로그래머는 break문을 자주 사용하는 습관을 가지고 있을 지 모르며 break문을 사용하면 자연스러운 코드를 만들 수 있는 경우도 있다. 예를 들어 각 층의 버튼 상태를 차례로 조사해 나가면서 가장 처음 버튼이 눌려진 층을 알려고 할 때 대부분의 프로그래머는 직관적으로 다음과 같은 코드를 떠올릴 것이다.

```
for( i = 0; i < TOP_FLOOR; i++)
if (ButtonState[i] == ON)
        break;
```

그러나, 이 경우 아래와 같이 break를 사용하지 않는 형태로 반복문을 재구성해야 하며 실험의 참가자들도 공통적으로 이 과정을 거쳤다고 말했다.

```
i = 0;
while(i < TOP_FLOOR && ButtonState[i] != ON)
    i = i + 1;
```

위와 같은 간단한 예일 경우 반복문의 재구성은 비교적 간단해 보이나, break문의 개수가 많아지고 반복문이 커질수록 이 작업은 많은 사고력을 요하며 프로그래머가 실수를 할 가능성을 높일 수 있다

7. 결론 및 향후 연구 과제

본 논문에서는 C언어를 고신뢰도를 요하는 분야에 적용할 수 있도록 더 안전하게 프로그래밍하는 방법에 대해 제안했다. C언어에서 문제를 발생시킬 수 있는 잠재적인 위험 요소들을 제거한 subset을 정의하고 이 언어로 구현된 프로그램을 Ada의 안전한 subset인 SPARK으로 변환한다면 SPARK의 분석 도구인 Examiner로 변환된 코드를 검증할 수 있었다.

C언어가 안고 있는 위험한 요소들을 제거하고 SPARK으로의 변환을 가능하게 하기 위해 C언어의 모든 특징들을 분석하여 제한 사항들을 정의하였고 이 제한 사항들이 일반적인 C프로그래머가 익혀서 사용하기에 크게 불편하지 않으며 현실적인 시스템을 구현하는데 충분하다는 것을 실험을 통해 보였다.

또한 C의 subset를 SPARK으로 변환하기 위해 체계적인 변환 규칙을 정의하였다. 서로 다른 언어 특징들의 차이를 극복하고 같은 행위를 갖는 변환이 이루어지게 하기 위해 문제가 되는 논점들을 찾아 해결 방안과 그에 따른 변환 방법을 제안하였다. 그리고 제한 사항을 만족하도록 작성된 C 프로그램을 이 규칙들에 따라 SPARK코드로 바꿔 주는 변환기를 구현하여 변환 과정을 자동화하였다.

향후 연구 과제로는 실험 과정에서 프로그래머가 불편하게 느낀 몇몇 제한 사항들의 개선이 필요하다. Boolean 타입의 부재나 반복문 내에서 break문의 금지와 같은 특징들은 정의된 subset의 안전성이나 SPARK으로의 변환 가능성을 해치지 않는 범위 내에서 좀 더 확장할 수 있을 것이다.

참 고 문 헌

[1] W. J. Cullyer, S. J. Goodenough, B. A. Wichmann. "The choice of computer language for use in safety-critical systems". Software Engineering Journal 6(2):51-58, March 1991.

[2] Roger S. Rivett. "Emerging Software Best Practice and how to be Compliant". Proceedings of the 6th International EAEC Congress. July 1997.

[3] D. Evans. "Static detection of dynamic memory errors.". Proceedings of ACM SIGPLAN '96. Conf. On PLDI., SIGPLAN Notices, 31(5):44-53, 1996.

[4] George C. Necula, Scott McPeak, Westley Weimer. "CCured: Type-Safe Retrofitting of Legacy Code". Proceedings of Principles of Programming Languages, 2002.

[5] Trevor Jim. "Cyclone: A Safe Dialect of C", USENIX Annual Technical Conference, Monterey, CA, June 2002.

[6] Bernard Carre; Joanthan Garnsworthy. "SPARK-An annotated Ada Subset for Safety-Critical Programming". ACM Annual International Conference on Ada. Proceeding of the conference on TRI-ADA '90. 329-402, 3- Dec. 1990.
[7] R. Chapman. "Industrial Experience with SPARK", Praxis Critical System Limited, ACM SIGADA 2000 Conf.
[8] Les Hatton. "Safer C : Developing Software for High-integriry and Safety-critical Systems". McGraw-Hill, 1995.
[9] B. W. Kernighan, D. M. Ritchie. The C Programming Language. Second Edition, Prentice-Hall, Englewood, New Jersey, 1988.
[10] John Barnes. "High Integrity Ada: The Spark Approach". Addison-Wesley, chap. 13, 1996.

김 진 섭
2000년 2월 한국과학기술원 전산학과 졸업(학사). 2002년 8월 한국과학기술원 전자전산학과 전산학전공 졸업(석사). 2002년 9월~현재 (주)와이어리스 리퍼블릭 재직.

차 성 덕
1983년 University of California at Irvine 전산학 학사. 1986년 University of California at Irvine 전산학 석사. 1991년 University of California at Irvine 전산학 박사. 1990년~1991년 Hughes Aircraft Co. 연구원. 1991년~1994년 The Aerospace Corp. 연구원. 1994년 9월~현재 한국과학기술원 전자전산학과 부교수.