2000년 한국정보처리학회 추계 학술발표논문집 제 7 권 제 2 호

병행성 명세를 위한 확장된 Z의 연구

남성욱* 조영석
동국대학교 전자계산학과*
동국대학교 컴퓨터학과
namvmc@wonhyo.dongguk.ac.kr jys@dongguk.ac.kr

A Study on Extended Z for the Concurrency Specification

Seong Uk Nam* Young Suck Cho
Dept. of Computer Science, Dongguk University

요 약

소프트웨어 개발 초기 단계에서의 부정확에 기인한 에러들을 줄이기 위한 노력이나 기술이 절실하다.[1][2] 정형 명세 기법은 명세 단계에서 기인하는 에러들을 줄이기 위해 Z 나 VDM 과 같은 정형 표기법(formal notation)에 의해 쓰여지며, 정형성(formality)과 추상화(abstraction)의 제공 등 두 가지 사항에 대한 요구사항을 충족시켜 준다.[3][4] 그러나, Z 표기법의 병행성 표현 능력의 부족으로 병행성을 요구하는 시스템의 명세에서 사용할 수 없거나, Process Algebra 의 CSP (Communicating Sequence Processes) 등과 같은 다른 정형 언어와 함께 명세해야 하는 단점이 있다. 본 논문은 이를 보완하기 위해 범용 목적의 명세 언어인 기존의 Z 를 확장하여 병행성을 명세 가능하도록 하고자 한다. 이를 위해서 병행 프로세스(concurrent process) 개념을 도입하며, 이를 나타내는 표기를 정의하고 사용한다. 또한, 병행성의 제어를 위해서 프로시듀어 기술부(procedure description)의 도입 및 관련 스키마(schema)들을 정의한다. 아울러, 확장된 Z 로 작성된 명세서를 목적 언어로 자동 변환(translate)하기 위한 변환기(translator)를 Lex 와 Yacc 을 이용하여 구현하고, 변환된 목적 언어 파일을 실행하여 확장된 Z 가 모호성을 포함하지 않는지 시뮬레이션을 통해 검증 한다.

1. 서론

소프트웨어 개발 초기 단계에서의 부정확에 기인한 에러들을 줄이기 위한 노력이나 기술이 절실하다.[1][2] 정형 명세 기법은 명세 단계에서 기인하는 에러들을 줄이기 위해 Z 나 VDM 과 같은 정형 표기법(formal notation)에 의해 쓰여지며, 정형성(formality)과 추상화(abstraction)의 제공 등 두 가지 사항에 대한 요구사항을 충족시켜 준다.[3][4] 범용 목적의 명세에는 Z 가 널리 사용되고 있지만, 병행성을 Z 로 표현하려는 시도들은 병렬 시스템의 명세에 주로 쓰이는 CSP(Communicating Sequential Processes)와 같은 Process algebra 의 사용과 비교할 때 많은 모호성들을 불러일으키기에 충분한 명세를 생성해 내게 된다.[3] 따라서 병행성의 명세가 포함되어야 하는 시스템에서는 Z 와 함께 CSP 와 같은 명세 언어를 함께 사용하는 경우가 있다. 그러나, 다른 목적을 가진 명세 언어의 사용은 명세자의 폭을 감소시킬 우려가 있으며[3], 각 단계에서의 적절하지 못한 언어 선택이 초기 단계로의 피드백으로 이어질 수 있다. 그리고 명세가 끝난 후 두 명세의 통합 과정에서 생길 수 있는 오류에 대한 검증의 노력과 시간이 요구된다.

본 연구에서는 보편적인 목적의 명세 언어인 Z 가 병행성을 표현할 수 있도록, 병행 프로세스(concurrent process) 개념을 도입하며, 이를 나타내는 표기를 정의하고 사용한다. 또한, 병행성의 제어를 위해서 프로시듀어 기술부의 도입 및 관련 스키마(schema)들을 정의하고, 특히 프로세스들의 공유 자원(shared resource)에 대한 접근의 제어는 ADT(Abstract Data Type) 형태로 정의된 스키마를 사용하기로 한다. 그리고, 확장된 Z 로 작성된 명세서를 목적 언어로 자동 변환하는 변환기(translator)를 Lex 와 Yacc 로 구현하며, 변환된 목적 언어 파일을 실행하여 확장된 Z 의 모호성 포함 여부를 시뮬레이션을 통해 검증한다.

2. 관련 연구

보편적으로 병행성 명세를 위해 특별히 고안된 언어인 Process algebra 나 algebraic approach 등이 병행성 명세를 위해 사용된다.[5] Process algebra 는 서로간의 커뮤니케이션이 가능한 프로세스들의 집합으로 시스템을 모델링 한다.[1][3] 경우에 따라서는 Z 와 같은 모델 기반의 언어(model-based language)와 CSP 같은 process algebra 가 명세에 함께 사용된다. 그러나, 이들의 사용 방법, 사용 순서에 대한 판단은 아직 이슈로 남아 있기 때문에, 전적으로 명세자의 판단에 맡겨진다.[1] 최근 보고된 논문들에서는 Process algebra 와 Z 의 특징들을 결합하기 위한 연구들이 진행되고 있다.[6][7]

3. 확장된 Z

확장된 Z 에서는 기존 Z 에는 존재하지 않는 프로세스 개념을 도입하여, 이와 관련된 모든 행위를 명세 가능하도록 확장한다. 또한, 프로세스 기술(description)과 프로시듀어 기술을 사용한다. 그리고 모니터 메커니즘(monitor mechanism)을 이용하여 공유 자원에 대한 프로세스의 접근을 제어한다.

3.1 병행성 명세

확장된 Z 의 명세는 다음과 같다. 우선, 기본 타입(basic type)을 정의(definition)한다.

[*Process*]

그리고, 병행 프로세스 개념을 도입하고, ' □' 표기를 사용하여 이를 나타내어 병행 프로세스들을 기술한다.

processes: □ *Process*
processes = {□ $Process_1$, □ $Process_2$, ⋯ , □ $Process_k$}

앞에서 선언된 병행 프로세스들에 대해 프로세스 변수(variable)를 선언한다.

$P_{11}, P_{12}, \cdots, P_{1l}$: □ $Process_1$
$P_{21}, P_{22}, \cdots, P_{2m}$: □ $Process_2$
⋯ ⋯
$P_{k1}, P_{k2}, \cdots, P_{kn}$: □ $Process_k$

다음으로 선언된 각 프로세스들의 스키마들을 명세한다. <statement>에는 프로세스가 실제로 할 일을 명세하고, 공유 자원에 대한 접근 행위를 명세하는 프로시듀어에 해당하는 행위(operation) 스키마인 *monitor_procedure*를 명세한다.

□ $Process_A$ □ <statement> □ $monitor\_procedure_A$
□ $Process_B$ □ <statement> □ $monitor\_procedure_B$
. . .
□ $Process_Z$ □ <statement> □ $monitor\_procedure_Z$

모니터 메커니즘을 응용한 모니터 스키마를 선언하고, 내부에는 세마포어와 공유 자원 등 병행성 제어를 위해 필요한 변수들이 선언된다. 모니터에 대한 접근은 기술된 프로시듀어 스키마 만이 허용된다.

```
Monitor
mutex, full, empty: Semaphore
buf: bufsize □ item
item : Z
bufsize: 1 .. n
```

마지막으로, 각 프로세스 스키마에서 정의된 프로시듀어들을 기술하고, 프로시듀어에 관련된 스키마들을 기술한다. 따라서 여기에서는 *monitor_procedure* 스키마를 명세 한다. 이 *monitor_procedure* 스키마의 구성은 각각 모니터 내부의 상호 배타 접근에 대한 제어의 역할을 담당하는 부분과 생산자나 소비자가 각각 꽉 찬 버퍼나 빈 버퍼에 대해 접근하는 것을 제어하는 부분 그리고 생산자나 소비자 프로세스가 공유 자원에 접근하여 해야 할 일을 사용자가 직접 기술하는 세 부분으로 크게 나누어져 있다. 사용자가 *access* 스키마만 명세하고, 프로시듀어의 나머지 스키마 들은 수정하지 않고 가져다 쓸 수 있게 하였으므로, 명세자에게 편의를 제공한다.

monitor_procedure □ wait(empty?)∧wait(mutex?)
∧ *access* ∧ signal(mutex?)
∧ signal(full?)

3.2 " Bounded Buffer' s Problem"

확장된 Z 의 병행성 명세를 " Bounded Buffer' s Problem" 을 예로 설명 한다. 생산자 프로세스는 공유 자원에 접근하여 생산된 아이템을 삽입하고, 소비자 프로세스는 생산자가 생산한 아이템을 공유 자원에서 꺼내는 작업을 하게 된다. 생산자나 소비자 각각의 타입을 가지는 프로세스들이 생성되면 정의된 프로세스 타입에 따라서 기능을 하도록 병행하여 실행이 되고, 이들이 공유 자원에 접근하는 일을 수행할 때에는 병행성에 따른 모호성이 없다는 것이 보증 되어야 한다.

3.2.1 병행 프로세스 선언

먼저, 기본 타입을 정의하고, 병행 프로세스들을 선언한다. 병행 프로세스 개념을 도입하고 이를 나타내기 위해서 □ *producer* 와 □ *consumer* 를 선언한다.

[*Process*]
processes: □ *Process*
processes = { □ *producer*, □ *consumer* }

선언된 병행 프로세스들에 대해 n 개의 프로세스 변수들을 선언한다.

$p_1, p_2, \cdots, p_n$: □ *producer*
$c_1, c_2, \cdots, c_n$: □ *consumer*

□ *producer* 와 □ *consumer* 프로세스 스키마에 대해서 공유 자원과 연관되지 않는 *producing* 과 *consuming* 스키마가 명세되고, 모니터 내 공유 자원에 대한 접근 행위를 명세하는 *insert* 와 *remove* 스키마를 명세한다.

□ *producer* □ *producing* ∧ *insert*
insert □ wait(empty?) ∧ wait(mutex?)
∧ *ins_access* ∧ signal(mutex?) ∧ signal(full?)
□ *consumer* □ *remove* ∧ *consuming*
remove □ wait(full?) ∧ wait(mutex?)
∧ *rm_access* ∧ signal(mutex?) ∧ signal(empty?)

3.2.2 모니터 메커니즘

모니터 스키마를 선언한다. 또한, 모니터 내의 공유 자원에 대한 프로세스의 접근은 모니터 내에서 정의된 프로시듀어 들에 의해서만 가능하다.

```
Monitor
mutex, empty, full: Semaphore
item, next_in, next_out : Z
buf: bufsize □ item
bufsize: 1 .. 100
```

다음으로 선언된 모니터 스키마를 초기화하는 스키마와 모니터 내에서 사용될 세마포어(semaphore)를 명세한다. 세마포어 스키마는 모니터 내부 접근에 대한 상호 배타 접근을 보증하기 위해 사용된다.

```
init_Monitor
Monitor
next_in = 1
next_out = 1
mutex.count = 1
full.count = 0
empty.count = bufsize
semaphore
```

```
count: Z
queue: seq processes
```

3.2.3 병행성 제어 기술

이미 앞에서 보았던 모니터 내 프로시듀어 스키마들, 즉 *insert* 스키마와 *remove* 스키마는 각각 모니터 내부의 상호 배타 접근에 대한 제어의 역할을 담당하는 부분과 생산자나 소비자가 각각 꽉 찬 버퍼나 빈 버퍼에 대해 접근하는 것을 제어하는 부분, 그리고 생산자나 소비자 프로세스가 공유 자원에 접근하여 해야 할 일을 사용자가 직접 기술하는 최소 세 부분으로 나누어져 있다. 이 때 사용자는 공유 자원에 접근하여 필요한 기능을 수

행하는 *ins_access* 스키마만 명세하고 프로시듀어의 나머지 세부 스키마들은 본 명세를 수정하지 않고 문제 해결에 사용할 수 있다.

insert □ wait(empty?) ∧ wait(mutex?) ∧ *ins_access*
∧ signal(mutex?) ∧ signal(full?)
remove □ wait(full?) ∧ wait(mutex?) ∧ *rm_access*
∧ signal(mutex?) ∧ signal(empty?)

wait
s?: semaphore
p: *processes*
s?' .count = *s*?.count - 1
s?' .count < 0 ⇒
s?' .queue = s?.queue □<p>
block(p)

signal
s?: semaphore
s?' .count = *s*?.count + 1
s?' .count ≤ 0 ⇒
s?' .queue = *s*?.queue ₩ {<head *s*?.queue>}
wakeup(head *s*?.queue)
ins_access

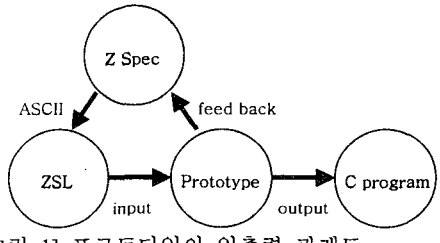
□ Monitor
item?: Z
buf' = *buf* □ {*next_in* □ *item*?}
next_in' = mod(*next_in* + 1, *bufsize*)
rm_access

□ Monitor
item!: Z
item! = *buf* □ *next_out* □
next_out' = mod(*next_out* + 1, *bufsize*)

이상의 " Bounded Buffer' s Problem" 명세 예에서 보는 바와 같이 확장된 Z 를 사용하여 병행성의 명세와 제어가 가능하다는 것을 보였다. 그러나, 완전한 명세를 위해서 확장된 Z 의 명세가 모호성을 포함하지 않는지 검증 되어야 한다.

4. 프로토타입의 구현

구현의 첫 과정으로서, 먼저 Z 명세를 모든 시스템에 적용할 수 있도록 Z 의 특수 문자들을 일반 문자로 변환하는 단계가 필요하며, 이를 위해서 Z 의 ASCII 버전인 ZSL(Z Specification Language)[8]을 사용하여 프로토타입의 입력 파일이 되게 한다. 프로토타입은 컴파일러 자동화 도구인 Lex 와 Yacc 을 이용하여 구현하며, C 프로그램을 파일로 출력한다.[그림 1]



[그림 1] 프로토타입의 입출력 관계도

4.1 구현 환경 및 방법

본 논문의 프로토타입은 Solaris 2.7 과 GNU gcc 2.95.2 환경에서 구현되었다. 작성된 Z 명세서는 ZSL 을 사용하여 텍스트로 변환하고, 이를 입력 파일로 사용하며, C 프로그램이 출력 파일이 된다.

전반적인 프로토타입의 구현 방법은 입력되는 확장된 Z 명세서의 키워드와 심볼 등을 지정하고, 이들을 ZSL 로 변환하여 각각의 규칙에 따라 정의된 공리(axiomatic) 기술과 모니터 스키마는 전역(global)으로, 세마포어 스키마는 구조체로, 각 프로세스 스키마와 프로시듀어 스키마, 그리고 이들의 구성 요소 스키마들은 프로그램의 메인과 필요한 각 함수로 선언 및 정의되며, 행위 스키마 내부에 선언된 동작은 해당하는 C 문장으로 변환된다.

4.2 구현 알고리즘

확장된 Z 로 기술된 Z 명세서는 "specification .. end specification"의 구조 내에서 5 가지 종류의 반복되는 단락, 즉 기본 타입 정의(basic type definition), 공리 기술(axiomatic description), 프로세스 기술(process description), 프로시듀어 기술(procedure description) 그리고 스키마 기술(schema description) 부분으로 이루어진다. 확장된 Z 로 작성된 명세서의 중요 변환 알고리즘은 다음과 같다.

(1)Basic Type declaration
/* [Process] 구문*/
(2)Axiomatic description
/* Global .. End Global 구문*/
(3)Process description
/*CP *process_name* IS *process_name* AND ... AND *process_name* CP ...;*/
(4)Procedure description
/* PROC *procedure_name* IS *procedure_name* AND ... AND *procedure_name* PROC ... ;*/
(5)Schema description
/* Schema *schema_name Definition_part*
Where *Predicate_part* End Schema */
(5-1)Schema Semaphore
/* typedef struct Semaphore{ }semaphore; */
(5-2)Schema Monitor
/* global 선언 */
(5-3)Schema Initialization
/* main( ) 내 초기화 */
(5-4)Schema Wait
/* 병행성 제어 위해 정의된 function */
(5-5)Schema Signal
/* 병행성 제어 위해 정의된 function */
(5-6)Schema Others
/* 외부 functions */

5. 검증

확장된 Z 로 작성된 Z 명세서가 C 프로그램으로 변환하는 프로토타입이 구현되었으므로, 확장된 Z 로 작성된 실제 명세서를 프로그램으로 변환하고, 이에 따라서 프로그램이 병행으로 동작할 수 있는가를 검증하게 된다.

검증에 사용할 명세서로서 "Bounded Buffer's Problem" 처럼 두 개의 병행 프로세스와 모니터 내부에 하나의 공유 버퍼를 가진 간단한 좌석 예약 시스템을 명세 한다. 좌석 예약 시스템은 고객의 요구가 있고, 빈 버퍼가 존재할 때 좌석 예약이 이루어지고, 이와는 반대로 버퍼의 내용이 채워져 있고, 고객이 예약을 취소하고자 할 때 채워진 버퍼의 내용을 삭제하므로서 예약의 취소가 이루

어 진다.

병행성이 모호하게 표현되지 않는가를 검증을 위해서는 병행성에서 필히 만족되어야 할 두 가지의 조건이 그 대상이 되어야 한다. 첫 째는 하나의 프로세스 타입을 가지는 두 개 이상의 복제 프로세스들이 공유 자원에 접근하고자 할 때 상호 배타 접근에 대한 모호성의 존재 여부를 살피고, 둘 째는 두 개 이상의 프로세스 타입을 각각 가지는 복제 프로세스들이 공유 자원에 접근하고자 할 때 프로세스 간의 협동(cooperation)에 대한 모호성의 존재 여부를 확인해야 한다.

5.1 확장된 Z로 작성된 명세서의 검증 방법

검증 방법으로는 여러 개의 프로세스가 실행이 되어야 하므로 만족되어야 할 다음의 각 항을 설정한다.

(1)2 개의 병행 프로세스에 대해
(2)N 개의 병행 프로세스에 대해
(3)N+1 개의 병행 프로세스에 대해

그리고, 이들 (1), (2), (3) 각 항에 대해서 아래 **(가)**, **(나)**의 각각의 요소가 충족되는지 확인을 한다.

(가)프로세스 하나는 이미 버퍼에 접근하여 좌석을 예약 중이고, 나머지 프로세스가 좌석 예약을 위해 버퍼에 접근을 시도할 때 다음의 각 요소에 대하여 조사

- 버퍼에 접근을 시도하는 프로세스의 접근 통제(대기와 봉쇄)
- 작업 중인 프로세스가 작업 종료시, 대기 중인 프로세스가 작업을 재기하거나, 새로운 프로세스가 버퍼에 접근하고자 할 때 접근 허용

(나)꽉 찬 버퍼에 접근하여 좌석을 예약하고자 하는 *$ 예약자* 프로세스의 접근을 통제하고, 역으로 빈 버퍼에 접근하여 예약을 취소 시키고자 하는 *$ 예약_취소자* 프로세스의 접근을 통제하는지 검사

- 꽉 찬 버퍼에 접근하여 좌석을 예약하고자 하는 *$ 예약자* 프로세스의 접근 통제
- 빈 버퍼에 접근하여 예약을 취소 시키고자 하는 *$ 예약_취소자* 프로세스의 접근을 통제

5.2 검증 결과

검증의 결과로서 모니터와 세마포어를 사용하여 단 하나의 프로세스 만이 버퍼에 접근할 수 있었으며, 이종의 프로세스간 작업이 제어가 되고, 아울러 빈 버퍼나 꽉 찬 버퍼에 접근하고자 하는 프로세스들을 통제하여 병행성을 제어할 수 있었다. 그림 2 는 병행성 명세 능력 비교 평가 결과이다.

표기법종류 / 검증기준	기존 Z	확장된 Z	CSP
프로세스 접근 통제	불가능	가능	가능
대기 프로세스 작업 재기	불가능	가능	가능
이종의 프로세스 간 작업 제어	불가능	가능	가능
꽉 찬 버퍼에 접근 통제	불가능	가능	가능
빈 버퍼에 접근 통제	불가능	가능	가능

[그림 2] 병행성 명세 능력 비교 평가 결과

6. 결론

본 연구에서는 기존의 Z 로 병행성을 명세하기 어렵다는 것에 초점을 두고, 병행성을 명세하기 위한 확장된 Z 를 제안하였다. 이를 위해서 병행 프로세스 개념의 도입과 이에 따른 프로세스 기술과 프로시듀어 기술의 사용, 그리고 모니터 메커니즘을 이용하여 공유 자원에 대한 프로세스의 접근을 제어하여 확장된 Z 를 사용하여 병행성을 명세할 수 있다는 것을 보였다. 앞으로 이러한 연구 결과를 바탕으로, 실제 어플리케이션들의 명세에 확장된 Z 를 사용하여 타당성과 유효성을 면밀히 검증하고, 확인해야 한다. 그러나, 명세의 목적이 구현의 방법인 어떻게(how)가 아니라 명세의 대상인 무엇(what)을 명세하는 것임에도 불구하고, 병행성을 제어함에 있어서 구현의 방법이 불가피하게 일부 포함되어 있어 명세의 목적면에서 확장된 Z 로 작성된 명세서는 명세서의 본질에 어긋난다고 말할 수 있다. 본 연구에서는 이러한 결점을 줄이기 위해 어떻게(how)에 해당하는 병행성의 명세 부분을 최소화 하는데 역점을 두었으나, 향후 이러한 결점이 보완 및 완전히 제거될 수 있는 연구가 계속되어야 할 것이다.

참고문헌

[1]Potter, Ben, Sinclair, Jane and Till, David, ***An Introduction to Formal Specification and Z***, Prentice Hall, 2nd edition, 1996

[2]Sommerville, Ian, ***Software Engineering***, Addison-Wesley, 5th edition, 1995

[3]Bowen, Jonathan, ***Formal Specification & Documentation Using Z***, Thomson, 1996

[4]Illingworth, V., ***Dictionary of Computing***, Oxford University, 3rd edition, 1990

[5]Hoare, C. A. R., " Communicating Sequential Processes," ***Communications of the ACM***, Volume 21, Number 8, pp.666-677

[6]Fischer, Clemens, " How to Combine Z with a Process Algebra," ***ZUM ' 98: The Z Formal Specification Notation***, Springer, September 1998

[7]Benjamin, M, " A message passing system: An example of combining CSP and Z," ***Z User Workshop***, oxford 1989, Workshop in computing, Springer-Verlag, 1990, pp.221-228

[8]Xiaoping, Jia, ***ZTC: A Type Checker for Z Notation User's Guide Version 2.03***, August 1998, http://saturn.cs.depaul.edu/~fm