병행성 명세를 위한 Z의 확장

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

An Extension of Z for the Concurrency Specification

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

요 약

소프트웨어 프로젝트의 비용의 대부분이 구현(implementation) 단계나 테스트 단계에서의 에러 수정에 소모되고 있으며, 이러한 에러들의 대부분은 프로젝트 개발 초기 단계에서의 부정확(imprecision)에 기인한 것이다. 정형 명세 기법은 명세 단계에서 기인하는 에러들을 줄이기 위해 Z 나 VDM 과 같은 정형 표기법에 의해 쓰여진다. 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 연구에서는 Z 와 같은 범용 목적의 명세 언어가 병행성(concurrency)을 표현할 수 있도록 하기 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.

1. 서론

소프트웨어 프로젝트의 비용의 대부분이 구현(implementation) 단계나 테스트 단계에서의 에러 수정에 소모되고 있으며, 이러한 에러들의 대부분은 프로젝트 개발 초기 단계에서의 부정확(imprecision)에 기인한 것이다.[1] 따라서, 개발 초기 단계에서의 에러들을 줄이기 위한 노력이나 기술이 절실하다.[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]

일반적인 목적의 명세에는 Z 가 널리 사용되고 있지만, 병행성을 Z 로 표현하려는 시도들은 병렬 시스템의 명세에만 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 범용 목적의 Z 와 함께 CSP 와 같은 특수 목적의 명세 언어를 함께 사용하는 경우가 생긴다. 그러나, 두 가지 다른 목적을 가진 명세 언어의 사용은 명세자의 폭을 감소시킬 우려가 있으며[3], 각 단계에 대한 적절한 언어 선택의 실수로 초기 단계로의 피드백으로 인한 손실로 연결될 수 있다. 그리고 명세가 끝난 후에는 두 명세의 통합 과정에서 생길 수 있는 오류에 대한 검증의 노력과 시간이 요구된다.

본 연구에서는 Z 와 같은 보편적인 목적의 명세 언어가 병행성을 표현할 수 있도록 하기 위해서, 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)로 정의된 스키마를 사용하기로 한다.

2. 연구동향

병행성을 명세할 때에는 보편적으로 CSP 와 같은 Process algebra 나 LOTOS(Language Of Temporal Ordering Specification)와 같은 algebraic approach 등 병행성 명세를 위해 특별히 고안된 언어가 사용된다.[5] Process algebra 의 경우, 이들은 서로간의 커뮤니케이션이 가능한 프로세스들의 집합으로 시스템을 모델링 한다.[3][1] 경우에 따라서는 Z 와 같은 모델 기반의 언어(model-based language)와 CSP 같은 process algebra 가 명세에 함께 사용되는데, 이들을 어떤 방법으로, 어떤 순서로 사용할 것인가에 대한 판단은 많은 이슈가 되기 때문에, 전적으로 명세자의 판단에 맡겨진다. [1]

최근에는 Process algebra 와 Z 의 특징들을 결합하기 위한 연구들이 진행되고 있다.[6][7]

3. 확장된 Z의 병행성 명세

기존의 Z에서는 프로세스 개념이 존재하지 않기 때문에 프로세스 개념과 관련된 모든 행위를 명세할 수가 없다. 따라서, 확장된 Z는 병행 프로세스 개념을 도입하고, 이에 따른 프로세스 기술(description)과 프로시듀어 기술을 사용한다. 그리고 모니터(monitor) 메커니즘을 이용하여 공유 자원에 대한 프로세스의 접근을 제어한다.

3.1 확장된 Z의 명세

먼저, 기본 타입(basic type)을 정의(definition)한다.

[*Process*]

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

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

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

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

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, 이를 ADT 형태로 제공한다. 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 명세한다. 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"의 명세

확장된 Z의 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 프로세스의 선언

먼저, 기본 타입을 정의하고, 병행 프로세스들을 선언한다. 병행 프로세스 개념을 도입하고 이를 나타내기 위해서 '□' 표기를 사용하여 선언한다.

[*Process*]

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

선언된 병행 프로세스들에 대해 n 개의 프로세스 변수들을 선언한다.

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

선언된 생산자(*producer*)와 소비자(*consumer*) 프로세스 스키마에 대해서 공유 자원과 연관되지 않는 *producing*과 *consuming* 스키마가 명세되고, 모니터 내부 공유 자원에 대한 접근 행위를 명세하는 프로시듀어에 해당하는 *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 선언 한다. 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

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 수행하는 *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 Z를 사용하여 병행성의 명세와 제어가 가능하다는 것을 보였다. 그러나, 완전한 명세를 위해서 확장된 Z의 명세가 모호성을 포함하지 않는지 검증 되어야 한다.

4. 결론

Z와 같은 보편적인 목적의 명세 언어가 병행성을 포함한 모든 명세에 적용 가능하게 된다면, 두 개의 언어 사용에서 오는 많은 불이익들, 즉 비용 및 시간, 노력을 감소할 수 있을 것으로 예상된다.

본 연구에서는 기존의 Z로 병행성을 명세하기 어렵다는 것에 초점을 두고, 병행성을 명세하기 위한 확장된 Z를 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 Z를 사용하여 병행성을 명세할 수 있다는 것을 보였다. 앞으로 이러한 연구 결과를 바탕으로, 실제 어플리케이션들의 명세에 확장된 Z를 사용하여 그 유효성(validity)을 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 Z로 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