

10. Concurrent Objects in Maude

이 장에서는 분산 시스템을 객체들의 집합으로 모델링하는 방법을 설명한다. 각 구성 요소는 객체로 표현되며, 메시지 송수신을 통해 상호작용한다. 시스템의 상태는 객체와 메시지의 다중집합(multiset)으로 나타난다.

주요 내용은 다음과 같다 :

1. **Concurrent Objects의 모델링** : 재작성 논리(rewriting logic)를 이용해 동시 객체 시스템을 표현하는 방법을 소개한다.
2. **Full Maude (10.2절)** : Maude에 객체지향 명세를 쉽게 작성할 수 있도록 문법적 확장을 제공하는 인터페이스를 설명한다.
3. **Dining Philosophers 문제 (10.3절)** —: 고전적 동시성 문제를 객체지향적으로 Maude에서 구현하는 예제를 다룬다.
4. **Blackjack 시뮬레이션 (10.4절)** : 무작위 시뮬레이션을 통해 다양한 전략을 실험하는 예시를 제시한다.

마지막으로, 객체지향 개념을 재작성 논리에서 이론적으로 다루는 내용은 참고문헌 [81]에서 더 깊이 설명된다.

10.1 Modeling Concurrent Objects in Maude

개요

- Maude에서 **객체(Object)**는 시스템의 각 구성 요소를 표현하기 위해 사용되며, 객체의 상태와 속성(attribute)은 **재작성 논리(Rewriting Logic)**로 모델링된다.
- 객체 간의 상호작용은 **메시지(Message)** 교환으로 이루어진다.

객체 정의 방식

Maude에서 객체는 다음과 같은 형태로 정의된다.

```
< o : C | att1 : val1, ..., attn : valn >
```

- **o**: 객체 식별자 (identifier)
- **C**: 클래스 이름
- **att₁ ... att_n**: 속성 이름
- **val₁ ... val_n**: 속성의 현재 값

예시:

```
< "Edward" : Person | age : 32, status : single >
```

클래스 선언 예시

클래스는 객체의 구조를 정의하며, 생성자(constructor) 형태로 선언된다.

```
op <_ : Person | age : _, status : _> : Oid Nat Status → Object [ctor] .
```

- **Oid**: 객체 식별자 타입

- **Nat, Status:** 속성의 타입
- 결과 정렬은 **Object**

메시지와 시스템 구성

시스템은 객체들뿐 아니라 **메시지(Msg)**도 포함할 수 있으며, 객체와 메시지의 **다중집합(multiset)**으로 시스템 전체 상태를 표현한다.

이를 위해 **Configuration** 정렬이 정의된다.

```
sorts Object Msg Configuration .
subsorts Object Msg < Configuration .
op none : → Configuration [ctor] .
op _ : Configuration Configuration → Configuration
    [ctor assoc comm id: none] .
```

- **none** : 빈 구성 (empty configuration)
- **_** : 구성 결합 연산 (결합적, 교환적, 항등원 존재)

예시 구성 (Configuration)

```
< "Edward" : Person | age : 32, status : single >
< "Mette" : Person | age : 47, status : married("Rich") >
< "Chrissie" : Person | age : 25, status : single >
```

→ 위와 같은 구성은 3명의 **Person** 객체로 이루어진 하나의 시스템 상태를 나타냄.

10.1.1 Rewrite Rules for Objects

개요

재작성 규칙(rewrite rule)은 **객체의 동작과 메시지 처리 방식**을 정의한다.

규칙의 양쪽은 객체와 메시지의 다중집합(multiset)으로, 객체나 메시지는 생성·삭제될 수도 있다.

객체의 지역 상태 변화

예시 규칙:

```
cr1 [birthday] :
  < X : Person | age: N, status: S >
  ⇒
  < X : Person | age: N + 1, status: S >
  if N < 999 .
```

→ 한 객체의 **지역 상태(local state)**를 변경하는 규칙으로, 객체의 나이를 1 증가시키는 동작을 나타낸다.

이 변화는 **합동 규칙(Congruence rule)** 덕분에 해당 객체가 포함된 **시스템 전체 구성(Configuration)** 내에서도 그대로 적용된다.

동시성 (Concurrency)

여러 객체는 **서로 독립적으로 동시에(concurrently)** 행동할 수 있다.

즉, 각 객체의 상태 변화가 병렬로 일어나며, 예를 들어 여러 사람의 생일이 한 번의 단계에서 동시에 처리될 수 있다.

동기적 통신 (Synchronous Communication)

두 개 이상의 객체가 동시에 상호작용하며 상태를 함께 바꾸는 규칙도 가능하다.

```
cr1 [engagement] :  
  < X : Person | age: N, status: single >  
  < X' : Person | age: N', status: single >  
  ⇒  
  < X : Person | age: N, status: engaged(X') >  
  < X' : Person | age: N', status: engaged(X) >  
  if N > 15 /\ N' > 15 .
```

→ 두 객체가 **동기적으로 통신(synchronous communication)** 하며 서로의 상태를 동시에 변경하는 예시.

10.1.1.1 Creation and Deletion of Objects

- Maude에서 **재작성 규칙(rewrite rule)**의 양쪽에 동일한 객체가 있을 필요는 없다.

→ 왼쪽에만 있는 객체는 **삭제**, 오른쪽에만 있는 객체는 **생성**으로 해석된다.

- 예시 **death1** 규칙:

```
rl [death1] : < X : Person | age: N, status: single > ⇒ none .
```

→ 단일 상태(single)인 **Person** 객체를 제거한다.

- 반대로, 규칙의 오른쪽에 새로운 객체를 추가하면 **객체 생성**을 표현할 수 있다.

예를 들어, “출생(birth)” 규칙은 이름 목록(**StringList**) 중 하나를 비결정적으로 선택해 새 사람을 만든다.

```
cr1 [birth] :  
  < X : Person | age: N, status: married(X') >  
  < X'' : Names | OKnames: L X''' L' >  
  ⇒  
  < X : Person | age: N, status: married(X') >  
  < X'' : Names | OKnames: L X''' L' >  
  < X''' : Person | age: 0, status: single >  
  if N < 60 .
```

- 결과적으로, 나이가 60 미만인 부부(**Kronos**, **Rhea**)의 상태에서 이름 후보 목록(**"Zeus" "Poseidon" "Hades"**) 중 하나가 선택되어 **"Zeus"** 라는 새로운 객체가 **탄생(born)** 한다.

10.1.1.2 Communication Through Message Passing

- 객체 간의 통신은 메시지를 주고받는 방식으로 표현된다.

→ 예: **separate(X)** 는 “X에게 이별을 통보하는 메시지”를 의미한다.

- `separationInit` 규칙: 한쪽(X)이 이별을 **시작**하면 메시지 `separate(X')` 가 생성된다.
- `acceptSeparation` 규칙: 상대방(X')이 이 메시지를 **받아들이면**, 이별이 완료된다.
- 시스템 전체는 결합적(associative)·교환적(commutative) 성질을 가진 **수프(soup)** 형태로, 객체와 메시지가 섞여 있다가 서로 "만나면" 상호작용이 일어난다.
- 예시:
 - Zeus가 Dione과의 결혼을 끝내기 위해 `separationInit` 을 보내면
 - 메시지 `separate("Dione")` 가 생성되고, Dione이 이를 읽으면
 - 두 사람의 상태가 각각 `separated` 로 갱신된다.
- 이 통신 방식은 **비동기(asynchronous)**로, `separationInit` 과 `acceptSeparation` 사이에는 시간 차(예: 생일 이벤트)가 있을 수 있다.

10.1.1.3 The Specification

- 실행가능한 Maude specification :

```

mod OO-POPULATION is protecting NAT + STRING-LIST .

  *** Objects, messages, object names, and configurations :
  sorts Oid Object Msg Configuration .
  subsorts Object Msg < Configuration .
  op none : → Configuration [ctor] .
  op _ : Configuration Configuration → Configuration [ctor assoc comm id: none] .

  subsort String < Oid . *** Object names are Strings

  *** Classes :
  op <_: Names | OKnames:_> : Oid StringList → Object [ctor] .
  op <_: Person | age:_ , status:_> : Oid Nat Status → Object [ctor] .

  *** Message for separating from spouse :
  op separate : Oid → Msg [ctor] .

  sort Status .
  op single : → Status [ctor] .
  ops engaged married separated : Oid → Status [ctor] .

  vars X X' X'' X''' : String .
  vars N N' : Nat .
  var S : Status .
  vars L L' : StringList .

  cri [birthday] :
    < X : Person | age: N, status: S >
  ⇒
    < X : Person | age: N + 1, status: S > if N < 999 .

```

```

crl [engagement] :
  < X : Person | age: N, status: single >
  < X' : Person | age: N', status: single >
⇒
  < X : Person | age: N, status: engaged(X') >
  < X' : Person | age: N', status: engaged(X) >
  if N > 15 /\ N' > 15 .

crl [birth] :
  < X : Person | age: N, status: married(X') >
  < X'' : Names | OKnames: L X''' L' >
⇒
  < X : Person | age: N, status: married(X') >
  < X'' : Names | OKnames: L X''' L' >
  < X''' : Person | age: 0, status: single > if N < 60 .

rl [separationInit] :
  < X : Person | age: N, status: married(X') >
⇒
  < X : Person | age: N, status: separated(X') >
  separate(X') .

rl [acceptSeparation] :
  separate(X)
  < X : Person | age: N, status: married(X') >
⇒
  < X : Person | age: N, status: separated(X') > .

*** Some rules are exercises and are therefore omitted

op greeks : → Configuration .
eq greeks =
  < "PossibleNames" : Names | OKnames: "Hera" "Zeus" "Hades" >
  < "Gaia" : Person | age: 999, status: separated("Uranus") >
  < "Kronos" : Person | age: 803, status: separated("Rhea") >
  < "Rhea" : Person | age: 23, status: separated("Kronos") >
  < "Uranus" : Person | age: 901, status: separated("Gaia") > .
endm

```

10.2 Concurrent Objects in Full Maude

핵심 개념

- Maude에서도 병행 객체(concurrent objects)를 정의할 수 있지만, **Full Maude**를 사용하면 클래스(class), 메시지(message) 등을 보다 **객체지향적으로** 정의할 수 있다.

Full Maude의 역할

- **Object-oriented modules**을 지원하여 다음을 가능하게 함:
 - 클래스(`class`), 서브클래스(`subclass`), 메시지(`message`) 선언
 - 규칙에서 영향 없는 속성을 생략 → 짧고 간결한 `rewrite` 규칙 작성
- Maude의 `search` 명령 확장:
 - **상속 관계(subclasses)**를 고려한 검색
 - 필요한 속성만 명시하여 검색 가능
- 내부적으로 객체 지향 모듈을 **일반 Maude 모듈로 변환** 후 실행
- Francisco Durán이 작성한 명세이며, Maude 배포판의 `full-maude.maude` 파일에 포함되어 있음.

10.2.1 Using Full Maude

실행 방법

- Full Maude는 `full-maude.maude` 파일로 제공됨.
- 일반 Maude 모듈처럼 실행 가능:

```
linux> maude full-maude.maude
```

또는 Maude 내부에서

```
Maude> load full-maude.maude
```

입력 방식

- Full Maude 명령은 반드시 **괄호 ()**로 감싸서 입력해야 함.
- 기본 Maude 명령 대부분을 지원하지만 일부 예외 존재.

예시

```
Maude> (fmod NAT-ADD is
  sort Nat .
  op 0 : → Nat .
  op s : Nat → Nat [ctor] .
  op _+_ : Nat Nat → Nat .
  vars M N : Nat .
  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm)
```

→ 모듈 `NAT-ADD` 생성

이후 연산 예시:

```
Maude> (red s(s(0)) + s(0) .)
```

결과:

```
s(s(s(0))) (즉, 2 + 1 = 3)
```

유용한 명령어

- `(show all .)`
→ Full Maude가 내부적으로 변환한 Maude 모듈을 보여줌.
- `trace exclude FULL-MAUDE .`
→ 괄호 없이 사용하며, `set trace on .` 뒤에 입력하여 **Full Maude 실행 추적**에 사용.

10.2.2 Object-Oriented Modules in Full Maude

객체 지향 모듈의 선언

- 객체 지향 모듈은 다음 구문으로 정의된다:

```
(omod M is ... endom)
```

- `Oid`, `Object`, `Msg`, `Configuration` 등의 정렬(sorts)은 `CONFIGURATION` 모듈에 정의되어 있으며, 이 모듈은 `prelude.maude` 파일에 포함되어 모든 객체 지향 모듈에 자동으로 import된다.

CONFIGURATION 모듈의 기본 구조

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : → AttributeSet [ctor] .
  op _,_ : AttributeSet AttributeSet → AttributeSet
    [ctor assoc comm id: none] .
endm
```

- `AttributeSet` 은 속성(attribute)의 집합을 나타내며, 순서가 중요하지 않다.
- `none` 은 빈 속성 집합을 의미한다.

객체 구성에 필요한 기본 정렬

```
sorts Oid Cid Object Msg Portal Configuration .
subsort Object Msg Portal < Configuration .
op <_:_|_> : Oid Cid AttributeSet → Object [ctor object] .
op none : → Configuration [ctor] .
op _ : Configuration Configuration → Configuration [ctor assoc comm id: none] .
op <> : → Portal [ctor] .
endm
```

- `Cid` : 클래스 식별자(class identifiers)
- `AttributeSet` : 속성-값(attribute-value) 쌍의 멀티집합(multiset)

- **Configuration** : 여러 객체 및 메시지로 구성된 전체 시스템 상태를 나타냄

클래스 정의 구문

```
class C | att1 : s1, ..., attn : sn .
```

예시:

```
class Person | age : Nat, status : Status .
```

- 객체 식별자(**Oid**)에는 미리 정의된 값이 없으며, 예를 들어 문자열을 사용할 수 있다:

```
subsort String < Oid .
```

객체 표기법

```
< "Edward" : Person | status : single, age : 32 > .
```

- 속성 순서는 중요하지 않으며, 콜론 앞에는 공백이 들어간다.
- 속성이 없는 객체는 다음과 같이 표현한다:

```
< o : EmptyClass | > .
```

Rewrite 규칙에서의 속성 사용

- 객체의 일부 속성만이 규칙의 적용에 영향을 미친다.
- 값이 변경되는 속성만 규칙의 오른쪽에, 규칙의 적용 여부에 영향을 주는 속성만 왼쪽에 포함해야 한다.

예시:

```
crI [birthday] :
  < X : Person | age : N >
  ⇒
  < X : Person | age : N + 1 > if N < 999 .
```

- **status** 는 나이 변화에 영향을 주지 않으므로 생략 가능.

약혼 규칙:

```
crI [engagement] :
  < X : Person | age : N, status : single >
  < X' : Person | age : N', status : single >
  ⇒
  < X : Person | status : engaged(X') >
  < X' : Person | status : engaged(X) >
  if N > 15 and N' > 15 .
```

- **age** 는 약혼 가능 여부에만 영향을 주므로, 결과에서는 생략됨.

전체 예제 모듈 (POPULATION)

```
load full-maude

(omod POPULATION is protecting NAT + STRING .
  sort Status .
  op single : → Status [ctor] .
  ops engaged married separated : Oid → Status [ctor] .
  subsort String < Oid .

  class Person | age : Nat, status : Status .
  vars N N' : Nat . vars X X' : String .

  crl [birthday] :
    < X : Person | age : N >
    ⇒
    < X : Person | age : N + 1 > if N < 999 .

  crl [engagement] :
    < X : Person | age : N, status : single >
    < X' : Person | age : N', status : single >
    ⇒
    < X : Person | status : engaged(X') >
    < X' : Person | status : engaged(X) >
    if N > 15 and N' > 15 .

  op greeks : → Configuration .
  eq greeks =
    < "Gaia" : Person | age : 999, status : married("Uranus") >
    < "Uranus" : Person | age : 900, status : married("Gaia") > .
  endom)
```

실행 예시

```
Maude> (frew [10] greeks .)
```

→ 모델을 10단계까지 시뮬레이션하여 객체들의 상호작용을 확인할 수 있다.

10.2.3 Subclasses

클래스 상속 (Subclass)

- `subclass B < C .`
→ 클래스 **B**는 **C**의 하위 클래스로, B는 C의 모든 속성과 기능을 상속받는다.
- 즉, **B** 객체는 자동으로 **C** 객체이기도 하다.
- Maude는 **다중 상속**도 지원한다.

```
subclass C < C1, C2, ..., Cn .
```

→ 클래스 C는 여러 상위 클래스의 속성과 규칙을 모두 물려받음.

예시: 사람(Person)과 종교(Christian, Muslim)

- Population model 확장 : some people are Christian, some are Muslim, and some are neither
- `Christian` 과 `Muslim` 은 `Person` 의 하위 클래스로 정의된다.

```
subclass Christian Muslim < Person .
```

- 각 클래스는 고유한 속성을 가짐:
 - `Christian` → `chrStatus` (세례, 확인 등)
 - `Muslim` → `hajji` (성지순례 여부)

개종 규칙 (Conversion Rules)

- 사람은 종교를 바꿀 수 있음, 즉 클래스가 변경됨.
- 예시:

```
cr1 [convertIslam] :  
  < X : Person | age : N, status : S >  
  ⇒  
  < X : Muslim | age : N, status : S, hajji : false >  
  if not (< X : Person | >) :: MuslimObject .
```

→ 아직 무슬림(이슬람을 믿는 사람)이 아닌 사람만 이슬람으로 개종할 수 있다.

- 조건부(`if not ... :: MuslimObject`)는 이미 무슬림인 객체를 다시 무슬림으로 변환하지 않도록 **중복 방지** 역할을 한다.

클래스 변경의 의미

- 클래스가 바뀌면 기존 객체가 삭제되고 **동일한 이름의 새 객체가 다른 클래스로** 생성된다.
- 따라서 새 객체의 모든 속성을 **오른쪽(RHS)**에 다시 명시해야 한다. (예: `age` , `status` , `hajji` 등)

10.2.4 Search in Full Maude

핵심 개념:

Full Maude의 `search` 명령은 **클래스 C** 및 그 하위 클래스(subclass)에 속한 객체들 중, 특정 속성 패턴(`att : pattern`)과 일치하는 객체를 찾는다.

주요 내용 정리

1. 하위 클래스도 자동 포함됨

`search` 는 클래스 C뿐 아니라 `subclass C` 객체도 자동으로 탐색하므로, 따로 하위 클래스를 지정할 필요가 없다.

2. 검색 패턴 구조

```
< o : C | att : pattern >
```

- `o` : 객체 이름
- `C` : 클래스 (혹은 그 하위 클래스)
- `att : pattern` : 속성과 그 값의 패턴

3. 검색 결과의 Echo

- Maude는 클래스 이름을 변수(`v#0`)로 바꿔 표시해, 실제 객체가 어떤 하위 클래스에 속하는지 유연하게 매칭함.
- 나머지 속성은 `AttributeSet` 변수(`v#1`)로 캡처됨.

4. 주의 사항

- 검색 시 관심 없는 속성이 있으면 `none` 을 반드시 명시해야 함. (빈 속성 자리로 두면 오류 발생)
- `such that` 조건을 사용할 때는 `var:sort` 형태로 작성해야 함.

```
< "Uranus" : Person | age : N:Nat >  
such that N:Nat > 902 .
```

10.2.4.1 Obtaining the Search Path

- Full Maude는 `show path` 명령을 지원하지 않음.
- 따라서 검색 결과로 도달한 경로를 보려면, Full Maude 명세를 (core) Maude 명세로 변환해야 함.

방법 요약

1. Full Maude에서 변환 명령 실행

```
(show all .)
```

→ 현재 모듈의 (core) Maude 버전을 출력함.

2. 출력 결과를 (core) Maude에 복사 후 검색 수행

```
Maude> search [1] greeks =>  
  C:Configuration  
  < "Uranus" : Person | age : 903, status : S:Status > .
```

3. 특정 상태의 경로 확인

```
Maude> show path labels 3 .  
birthday  
birthday  
birthday
```

자동 변환 방법 (파일 이용)

- Full Maude 모듈을 `file.mau` 로 저장하고, 끝부분에 다음 줄 추가:

```
(show all .)
q
```

- 리눅스 명령 실행:

```
maude file.mau > core-mau-file.mau
```

→ Full Maude 모듈이 (core) Maude 형식으로 변환되어 저장됨.

- 환영/종료 메시지를 제거한 후, 이 파일을 Maude에서 분석 가능.

10.2.5 Using Full Maude : Repetition

- **입력 규칙:**

- Full Maude 명령은 모두 괄호 `()` 로 감싸야 함.
- 괄호 없이 입력하면 (core) Maude로 인식됨.

- **모듈 규칙:**

- Full Maude 모듈도 괄호로 감싸야 함.
- 비객체지향 모듈은 core Maude에 먼저 정의 후, Full Maude에서 `import` 하는 것이 좋음.

- **명령어 규칙:**

- `red` , `rew` , `search` → 괄호 필요
- `in` , `load` → 괄호 금지 (core Maude용)

- **제한 사항:**

- 디버거(`debugger`)나 `show path` 등 일부 기능은 Full Maude에서 사용 불가.

- **활성화 방법:**

```
load full-mau.mau
```

10.3 Example : The Dining Philosophers

- 식사하는 철학자 문제는 Dijkstra가 제안한 **분산 시스템의 고전적 예제**이다.
- 여러 개의 컴포넌트가 **공유 자원(예: 프린터, 메모리)**을 사용할 때 발생하는 문제를 설명하기 위해 활용된다.

10.3.1 Problem Description — 요약

- 철학자 5명이 둥근 테이블 주변에 앉아 있으며, 중앙에는 음식(만두)이 있다.
- 철학자들은 **생각 → 배고픔 → 식사 → 생각**의 사이클을 반복한다.
- 테이블에는 철학자 수와 동일한 **5개의 젓가락**만 있고, 각 철학자 사이에 1개씩 놓여 있다.

- 철학자가 음식을 먹으려면 **양쪽 젓가락 2개 모두 필요하다**.
- 배고파진 철학자는 **왼쪽 또는 오른쪽 젓가락부터 하나 집고**, 다른 하나를 집을 때까지 waiting 한다.
- 일정 시간 동안 식사한 후에는 **두 젓가락을 모두 내려놓고** 다시 생각하기 시작한다.
- 문제의 핵심 질문:
 - 모든 철학자가 젓가락을 못 잡고 굶어 죽는 상황(deadlock)이 가능한가?
 - 한 철학자만 굶고 나머지 모두 식사하는 상황(starvation)도 가능한가?

10.3.2 Modeling the Dining Philosophers

젓가락 모델링 방식

- 젓가락은 객체가 아니라 **메시지(message)**로 모델링된다.
- `chopstick(i)` 메시지는 "i번 젓가락이 테이블 위에 있음(사용 가능)"을 의미한다.
- 철학자가 젓가락을 집으면 메시지는 소비되고, 식사를 마치면 두 개의 `chopstick` 메시지를 다시 내보내어 젓가락을 테이블에 돌려놓는다.

철학자 객체 구조

각 철학자는 다음 속성을 갖는 객체로 모델링된다.

```
< i : Philosopher | state : s, #sticks : j, #eats : k >
```

- `state` : 현재 상태 (`thinking` , `hungry` , `eating`)
- `#sticks` : 현재 손에 쥔 젓가락 수 (0,1,2)
- `#eats` : 지금까지 먹은 횟수

철학자는 번호(i)가 객체 이름으로 사용된다 (`Nat < Oid`).

규칙 정의

- hungry 규칙 – 철학자가 배고파짐 (`thinking` → `hungry`)

```
rl [hungry] :
  < I : Philosopher | state : thinking >
  ⇒
  < I : Philosopher | state : hungry > .
```

- grabFirst – 첫 번째 젓가락 집기
 - 철학자가 왼쪽 또는 오른쪽 젓가락 중 하나(J)를 집을 수 있을 때

```
crl [grabFirst] :
  chopstick(J)
  < I : Philosopher | state : hungry, #sticks : 0 >
  ⇒
  < I : Philosopher | state : hungry, #sticks : 1 >
  if I can use stick J .
```

- “사용할 수 있는 젓가락” 조건:

```
I can use stick J = (I == J) or (J == right(I))
right(I) = if I==5 then 1 else I+1 fi .
```

- 즉

- 왼쪽 젓가락: 번호가 I
- 오른쪽 젓가락: 번호는 right(I)

```
op right : Nat → Nat .
eq right(I) = if I == 5 then 1 else I + 1 fi .
```

- 철학자 I는 2개의 젓가락을 들 수 있음 (왼쪽 : I번 젓가락, 오른쪽 : I+1번 젓가락)
- 젓가락이 원형으로 배치되어 있기 때문에 마지막 철학자(5번)는 오른쪽 젓가락이 6번이 아니라 다시 1번으로 돌아감

- grabSecond – 두 번째 젓가락 집기

젓가락을 이미 1개 들고 있고, 두 번째를 집을 수 있을 때:

```
crI [grabSecond]:
  chopstick(J)
  < I : Philosopher | #sticks : 1, #eats : K >
⇒
  < I : Philosopher | state : eating, #sticks : 2, #eats : K+1 >
  if I can use stick J .
```

두 번째 젓가락을 집으면 state가 **eating**으로 바뀌고 #eats 증가.

- stopEating – 식사 끝, 젓가락 2개 반납

식사를 마치고 다시 생각 상태(thinking)로 전환:

```
rI [stopEating] :
  < I : Philosopher | state : eating >
⇒
  < I : Philosopher | state : thinking, #sticks : 0 >
  chopstick(I) chopstick(right(I)) .
```

젓가락 번호 두 개(I, right(I))를 다시 메시지로 배출.

초기 상태 정의

초기 상태는 5개의 젓가락 메시지와 생각 중인 5명의 철학자로 구성된다.

```
op initState : → Configuration .
eq initState =
  chopstick(1) chopstick(2) chopstick(3)
  chopstick(4) chopstick(5)
  <1: Philosopher | state:thinking, #sticks:0, #eats:0>
```

```
...
<5 : Philosopher | state:thinking, #sticks:0, #eats:0> .
```

10.3.3 Deadlock and Livelock

- 철학자들이 젓가락이라는 공유 자원을 필요로 하기 때문에 **교착상태(deadlock)**가 발생할 수 있다.
→ 예: 각 철학자가 젓가락 하나씩만 들고 더 이상 진행할 수 없는 상태.
- 라이블락(livelock, starvation)**은 한 철학자만 영원히 두 젓가락을 얻지 못해 **굶어 죽는 상황**을 의미한다.
→ 다른 철학자들은 정상적으로 계속 식사할 수 있기 때문에 더 복잡한 문제이다.

10.3.4 Fairness Issues

- 일반적인 공정성 가정:
 - 식사 중인 철학자는 **언젠가는** 식사를 멈춘다.
 - 생각 중인 철학자는 **언젠가는** 배고파진다.
 - 젓가락이 계속 사용 가능하면 철학자는 **언젠가는** 그 젓가락을 집는다.
- 이러한 공정성 조건은 명세에 직접적으로 포함되어 있지 않지만, 유한 실행(prefix)은 모두 공정한 무한 실행의 일부로 간주할 수 있어 **교착상태 분석에는 문제가 없다**.
- 그러나 starvation은 무한 실행과 관련되므로, 명세가 허용하는 일부 행동은 **공정성을 만족하지 않을 수 있다**.
- "언젠가는 eventually OO 한다"는 조건은 명확한 시간적 상한이 없으므로 **완전한 구현이 불가능하며**, 대신 "모든 공정한 계산에서 속성 X가 성립한다"는 형태로 분석한다.

10.3.5 Version 2: A Deadlock-Free Solution

- 교착상태를 제거하는 한 가지 방법은 **철학자가 두 젓가락을 동시에 집도록 강제하는 방식**이다.
- 즉, 한 개만 먼저 잡는 것을 막아 "젓가락 하나씩 들고 모두 대기하는" 교착상태가 발생하지 않도록 한다.

10.3.6 Version 3: A Deadlock-Free and Livelock-Free Solution

문제 상황

- 철학자가 오른쪽 젓가락 하나만 들고, 왼쪽 젓가락을 기다리며 무한히 대기하면 **교착상태(deadlock)** 발생.
- 이를 막기 위해 "양쪽 젓가락을 동시에 잡는 방식"을 쓰면 **deadlock은 사라짐**.
- 그러나 이 방식은 어떤 철학자가 계속 기회를 놓쳐 **영원히 먹지 못하는 livelock(=starvation)** 문제는 해결하지 못함.

Livelock까지 제거하는 새로운 해결책

- 핵심 아이디어**
 - 철학자들이 한 공간(식당)에 모두 몰려 있으면 서로 젓가락을 잡으려 하고, 기회가 특정 철학자에게 계속 오지 않을 수 있음 → livelock 발생 가능.

- 이를 방지하기 위해 철학자들에게 **식당(dining room)**과 **도서관(library)**을 분리함.
- **도어맨(입장 제한 장치)**을 두어 **항상 최대 4명만** 식당에 들어오도록 제한하여 livelock을 방지.

• 새로운 시스템 구조

```
< GlobalSystem : DinPhilHouse |
  diningRoom : philsAndSticks,
  #inDinRoom : n,
  library : philosophers >
```

- **diningRoom** : 현재 식당에 있는 철학자 + 젓가락들
- **#inDinRoom** : 식당 안의 철학자 수
- **library** : 식당에 들어가기 전 대기 중인 철학자들

주요 규칙

- *enterDinRoom*
 - 배고픈 철학자가 식당에 들어올 수 있음(단, 식당 인원이 4명 미만일 때만).
- 이전 규칙들(*grabFirst*, *grabSecond*) 그대로 적용
 - 젓가락을 1개, 2개 잡으면 먹기 시작.
- *enterLibrary*
 - 철학자가 먹기를 끝내고 **생각 상태로 전환되면 즉시 식당에서 나와 도서관으로 이동.**

초기 상태

- 철학자 5명 모두 **도서관에서 생각 중.**
- 젓가락 5개는 **식당 안에** 놓여 있음.
- 식당에는 아무도 없음(**#inDinRoom = 0**).

결과: 완전 자동 검증

- Maude 모델 검증 도구로 확인한 결과
 - **deadlock 없음**
 - **livelock 없음**

→ 즉, 이 모델은 dining philosophers 문제의 **교착상태도 없고, 기아(livelock)도 없는 완전한 해결책**임이 증명 됨.

10.4 Randomized Simulations : Winning in Vegas

10.4장은 블랙잭과 같은 도박 전략을 실제 카지노에서 실험하거나 복잡한 통계 계산을 수행하는 대신, **Maude의 의사 난수(random)** 기능을 이용해 게임을 반복 시뮬레이션하여 기대 수익을 분석하는 방법을 다룬다.

- 블랙잭은 21을 넘지 않는 범위에서 딜러보다 높은 값을 만드는 게임이다.
- 실제 게임에서는 여러 의사결정(히트, 더블 다운, 스플릿, 서렌더 등)이 필요하다.

- Maude의 무작위 시뮬레이션 기능을 사용하면 **전략별 반복 실험**을 통해 어떤 전략이 가장 이익을 주는지 평가할 수 있다.
- 시뮬레이션에서는 **남아 있는 카드 중 무작위로 다음 카드를 선택**하도록 모델링한다.

10.4.1 Blackjack

10.4.1은 시뮬레이션을 하기 위해 필요한 블랙잭 게임 규칙을 구체적으로 정리한다.

- **핵심 규칙 요약**
 - 그림 카드는 10, 에이스는 1 또는 11.
 - 플레이어나 딜러가 **두 장 합이 21이면 블랙잭**.
 - 플레이어는 카드 두 장을 받은 뒤 원하는 만큼 hit 가능.
 - 딜러는 **고정된 규칙(예: 17까지 반드시 hit)**에 따라 카드 획득.
- **플레이어 패배 조건**
 - 카드 합이 21 초과(버스트).
 - 딜러는 블랙잭인데 플레이어는 아닌 경우.
 - 딜러가 21 이하에서 플레이어보다 더 높은 합을 만든 경우.
- **무승부 조건**
 - 양측 모두 블랙잭.
 - 양측 모두 21 이하에서 합이 동일.
- **플레이어 승리 조건**
 - 플레이어만 블랙잭이면 **베팅금의 1.5배** 획득.
 - 그 외 일반 승리는 **베팅금만큼** 획득.
- **추가 행동**
 - **Double down**: 베팅 2배, 카드 1장 추가.
 - **Split**: 같은 값 두 카드 → 두 손으로 분리.
 - **Surrender**: 포기하고 베팅 절반을 돌려받음.
- **전략적 고려사항**
 - 딜러의 soft 17 규칙은 카지노마다 다름.
 - 딜러의 오픈 카드에 따른 전략 필요.
 - 카드 덱 개수에 따른 확률 변화.
 - 언제 더블 다운/스플릿/서렌더를 해야 하는지.

10.4.2 Modeling Blackjack Rounds

본 절에서는 **블랙잭 게임을 Maude에서 형식적으로 모델링**하는 과정을 단계별로 설명한다.

단일 플레이어와 딜러가 존재하는 단순한 규칙을 기반으로, 카드 생성 → 점수 계산 → 게임 진행 → 결과 산출 → 여러 라운드 반복까지 전체 시뮬레이션을 수행한다.

카드와 덱 모델링 (CARD module)

- 카드 구성 정의
 - **Suit**: spades, hearts, clubs, diamonds
 - **Value**: 2-10, J, Q, K, A
 - **Card 생성자**: `<Suit, Value>`
- Cards(카드 리스트)
 - `nil` 또는 `_ :: _` 형태의 리스트
 - `::` 는 Maude의 리스트 연결 연산자
- 덱 생성

`deck = generate(suits...)` 형태로 52장의 카드를 모두 생성한다.

`suit` 하나에 대해 `generate(S)` 는 다음 카드 리스트 생성 :

```
<S,2> :: <S,3> :: ... :: <S,K> :: <S,A>
```

손패 점수 계산 (RESULT module)

Ace(A)는 **1** 또는 **11**로 계산 가능하므로 다음 세 가지 점수를 계산한다.

- `leastValue` : Ace를 1로 계산한 값
- `largestValue` : Ace를 11로 계산한 값이 유효한 경우
- `bestValue` : 21 이하에서 가장 큰 값

이 값을 이용해 블랙잭 여부를 판단한다.

게임 결과 계산

result(player, dealer, bet) 함수는 플레이어의 손패, 딜러의 손패, 베팅 금액을 바탕으로 **수익 또는 손실**을 반환한다.

규칙:

1. 플레이어가 블랙잭이고 딜러는 아니면 → **5 * bet / 2**
2. 플레이어가 21 이하이고 딜러보다 높은 경우, 또는 딜러가 21 초과 → **이김 (2 * bet)**
3. 둘 다 블랙잭이거나 동점 → **push (bet 반환)**
4. 그 외 → **패배 (0)**

무작위 카드 선택 (RANDOM-CARD module)

- `getNthCard(CARDS, N)` : N번째 카드 가져오기
- `getRandomCard(CARDS, N)` : 난수 인덱스 기반 무작위 카드 선택
- `remove(card, deck)` : 덱에서 해당 카드를 제거
- 난수는 책 40쪽의 `random(n)` 함수로 결정됨

게임의 객체지향 모델링 (PLAY-BJ module)

- 클래스 구조

- **Table**
 - `shoe: Cards` (남은 카드 리스트)
 - `rndIndex: Nat`
 - `turn: Oid` (차례)
- **Player**
 - `hand: Cards`
 - `bet: Nat`
- **Dealer**
 - `hand: Cards`
- 게임 진행 `rewrite rules`
 1. **startGame**
 - 플레이어가 첫 카드 받음
 2. **dealerFirstCard**
 - 딜러가 첫 카드 받음
 3. **playerSecond**
 - 플레이어가 두 번째 카드 받음
 4. **playerHit / playerStand**
 - 전략 기반 Hit/Stand
 - 전략: `leastValue >= 15` 또는 `bestValue >= 18` 면 Stand
 5. **dealerTakesMore**
 - 딜러는 "모든 17에서 서 있음"

여러 라운드 모델 (PLAY-MANY-ROUNDS module)

- 멀티플레이어 클래스(Player subclass):
 - `gamesLeft` : 남은 게임 수
 - `money` : 현재 소지금
 - `eachBet` : 라운드마다 베팅 금액
- 규칙
 1. `reset`
 - 새 라운드 시작 전 초기화
 - 플레이어 자금이 충분하면 게임 지속
 2. `restart`
 - 한 라운드 종료 후 다음 라운드 준비

시뮬레이션 결과

초기 상태:

- 시작 자금: **\$1000**
- 각 게임 베팅: **\$100**
- 총 100 라운드 수행

Maude 결과:

```
result Configuration :
< p : MultiPlayer | gamesLeft : 0, money : 800, ... >
```

- **100 라운드 후 플레이어는 \$200 손해만 보았다.**
단순 전략임에도 예상보다 손실이 크지 않다는 점을 보여준다.

최종 핵심 요약

- Maude로 블랙잭 게임 전체를 형식적 모델로 구현하였다.
- 카드 생성 → 점수 계산 → 전략 적용 → 결과 계산 → 다중 라운드 시뮬레이션 모두 자동화된다.
- 단순 전략(≥ 15 또는 ≥ 18 에서 Stand)으로도 **100 게임 기준 -\$200**의 결과를 얻었다.
 - 본 절은 객체지향 Maude(Full Maude)의 사용법과 rewrite 규칙 기반 시뮬레이션 방법을 실질적인 예를 통해 보여주는 역할을 한다.

10.4.3 Further Guarantees

블랙잭 전략을 몇 번 시뮬레이션해 보는 것만으로는 충분하지 않기 때문에, 더 강한 확률적 보증이 필요할 수 있다. 이를 위해 다음과 같은 분석 기법들을 활용할 수 있다.

1. Probabilistic Model Checking (확률적 모델 검사)

- 시스템 전체의 확률적 동작을 분석하여 **\$1200 이상으로 끝날 확률이 60% 초과** 같은 성질을 수학적으로 증명할 수 있다.

2. Statistical Model Checking (통계적 모델 검사)

- 전체 확률적 분석은 비효율적이므로 반복 시뮬레이션을 통해 특정 신뢰도까지 통계적으로 추정하는 기법.
- 예: 신뢰도 0.9에서 "\$1200 이상일 확률이 60% 초과".

3. Value Estimation (가치 추정)

- 확률 자체보다 하루가 끝날 때 기대되는 평균 금액을 추정하는 데 더 관심이 있을 수 있다.