11. Modeling Communication in Maude

개요

- Chapter 11은 **동시 객체들 간의 다양한 통신 형태를 Maude의 리라이팅 논리(Rewriting Logic)**로 어떻게 모델링할 수 있는지를 설명한다.
- Chapter 10의 객체 기반 동시 모델을 기반으로, 이번 장에서는 통신 방식에 대한 추상적/일반적 모델링 방법을 다룬다.

통신 모델링이 필요한 이유

1. **다양한 장치의 통신 능력 차이**
   - 예: TCP/IP 컴퓨터, TV 위성 방송, 무선 센서 노드 등.
2. **추상화 수준의 필요성**
   - 불필요한 네트워크 세부 사항(패킷 구조, 헤더 등)을 생략하고 고수준에서 시스템을 모델링하기 위함.
3. **일반성(generality)의 확보**
   - 특정 통신 가정을 만족하는 다양한 시스템에 재사용 가능한 모델링 제공.

통신의 주요 유형

- **동기(Synchronous) 통신**
  - 통신 주체들이 **이벤트 시점에 동기화**하여 메시지를 주고받는 방식.
  - 예: 사람들이 직접 대화하는 경우.
- **비동기(Asynchronous) 통신**
  - 통신 주체들이 **동기화하지 않고** 메시지를 송수신.
  - 예: 우편, 이메일, 음성메시지, 게시판.
- **Ordered vs Unordered**
  - **Ordered:** 메시지가 발신 순서대로 도착/처리됨.
    - 예: 같은 케이블을 통한 메시지 전송.
  - **Unordered:** 순서가 보장되지 않음.
    - 예: 이메일, 우편.
- **Unicast / Multicast / Broadcast**
  - **Unicast:** 1:1 통신
  - **Multicast:** 1:N 통신 (선택된 다수)
  - **Broadcast:** 1:모든 사용자 통신
- **Reliable vs Unreliable 통신**
  - **Unreliable:** 메시지 손실/오류/지연 가능
  - **Reliable:** 높은 신뢰의 전달 보장
    - 예: 비행기 내부 통신(이론적으로)

리라이팅 논리(Rewriting Logic)의 장점

- **고정된 통신 기본 연산 없음**
  - 다른 분산 시스템 모델링 언어들과 달리,
  - Maude는 **특정 통신 모델에 얽매이지 않고**, **원하는 형태의 통신을 직접 정의**하여 모델링할 수 있음.
  - 즉, **높은 유연성** 제공.

Chapter 11의 구성

- **11.1 Synchronous Communication**
  - 동기적 통신 모델링 방법 설명.
- **11.2 Asynchronous Communication**
  - 순서 없는(unordered) unicast 메시지 모델 제시
  - 일상적 상황(예: 부부가 메시지를 통해 소통할 때)의 복잡성을 통해 **비동기 시스템 분석의 중요성**을 강조
  - 이후 **multicast**, **broadcast**, **unreliable communication**으로 확장
  - **11.2.4:** 무선 브로드캐스트 모델 소개
- **11.3 Ordered Asynchronous Communication**
  - **link 객체**를 이용하여
    - 메시지 순서 보장
    - 링크 용량 제한
    - 신뢰/비신뢰 통신 모두 모델링
  - 분산 네트워크의 링크 중심 모델을 설명
- **11.4 Shared Variables 기반 모델**
  - 변수 공유를 통한 비동기 통신 모델
  - 공유 자원 경쟁 문제 소개
    - 예: 공유 은행 계좌, 비행기 좌석과 같은 한정된 자원

11.1 Synchronous Communication

개념 정의

- 동기적 통신(synchronous communication)은 통신에 참여하는 객체들이 **동시에 만나(synchronize)** 하나의 통신 이벤트를 수행하는 방식이다.
- Maude에서는 이러한 동기화를 **하나의 rewrite rule 내에 모든 관련 객체를 명시**함으로써 모델링한다.

핵심 메커니즘

- 객체들은 시스템 상태라는 “수프(soup)” 안에서 **동시에 존재하는(multiset of objects)** 구조로 표현된다.
- 동기적 통신은 규칙 좌변(LHS)에 **모든 참가 객체가 함께 나타나는 방식**으로 정의된다.
- 규칙이 적용될 때, 객체들은 함께 참여하여 상태가 동시에 변한다.

예시: Engagement Rule

두 사람이 서로 결혼 의사를 교환하는 예를 통해 동기적 통신을 설명한다.

- 조건: 두 사람 모두 나이가 15세 이상이고, 상태가 `single`
- 결과: 두 사람의 `status`가 서로를 가리키는 `engaged(...)`로 변경됨
- 이는 **두 객체가 동시에 참여해야만 발생할 수 있는 이벤트**를 모델링한 것이다.

특징

- 두 객체뿐 아니라 **여러 객체**도 하나의 동기적 통신 이벤트에 참여할 수 있다.
- 동기적 통신은 동시에 일어나야 하는 상호작용을 모델링할 때 적합하다. (예: 결혼, 계약 체결, 핸드셰이크 프로토콜 등.)

11.2 Unordered Asynchronous Communication by Message Passing

개념

- 비동기 통신은 **메시지를 보낸 시점과 받는 시점이 일치하지 않는** 통신 모델이다.
- 송신자는 메시지를 보내고 곧바로 다른 일을 할 수 있으며, 수신자는 나중에 메시지를 받을 수 있다.
- 현대 분산 시스템의 기본 모델이며, 실제 인터넷/이메일/네트워크 프로토콜 등이 모두 비동기로 동작한다.
- Maude에서는 비동기 통신을:
  - 메시지를 **rewriting system의 구성 요소(terms)**로 취급하고
  - 메시지의 **소비(consumption)**는 rewrite rule의 좌변에 등장할 때 발생하며
  - 송신은 단순히 메시지를 전체 구성에 **추가**하는 방식

  으로 표현한다.

11.2.1 Unordered Unicast Communication

- **개념**

  무순서 비동기 단일 전송(unordered asynchronous unicast)은 메시지가:
  - 언제 도착할지 예측할 수 없고,
  - 순서가 보장되지 않으며,
  - 늦게 도착하거나,
  - 심지어 손실될 수도 있는

  통신 모델이다.

  즉, 송신자는 특정 수신자에게 하나의 메시지를 보내지만, 그 메시지가 **도착 순서를 보장하지 않는다**는 점이 핵심이다.
- **특징**
  - 비동기: 송신/수신 시간이 다름
  - 무순서: 송신 순서 ≠ 수신 순서
  - 단일 전송: 메시지 대상은 1명

11.2.1.1 Example: Separating Using Messages

이 예시는 "부부가 메시지로 별거를 선언하는 상황"을 통해 **무순서 비동기 메시지가 얼마나 복잡한 문제를 야기하는지**를 설명한다.

기본 메시지 & 규칙 구성

- 메시지: `separate(X)` → "X의 배우자가 별거를 원한다"
- 두 개의 규칙:
  - `initiateSeparation` (별거 개시 및 separate 메시지 발송)
  - `acceptSeparation` (결혼 상태에서 separate 수신 시 separated로 전환)

첫 모델은 단순해 보이지만, 실제로는 매우 취약하다.

문제 1 - 오래된 메시지 문제 (stale message)

- 이미 **separated** 상태에서는 메시지가 소비되지 않음
- 버려지지 않은 메시지가 **다른 결혼을 파괴**할 수 있음
- 예: 과거 파트너가 보낸 `separate(msg)`가 늦게 도착해 현재의 새로운 결혼을 파괴함

→ 이 현상은 **통신 지연이 있는 모든 비동기 분산 시스템의 근본적 문제**로 연결된다.

문제 2 - 수년 후 도착하는 메시지의 파괴력

비동기 메시지는 언제든 도착할 수 있기 때문에:

- 오래전에 끝난 관계(결혼)의 메시지가
- 현재의 완전히 새로운 관계(새 결혼)를 파괴할 수 있다.

부분적 해결책 (불완전)

- separated 상태라면 separate 메시지를 **즉시 삭제하는 규칙**을 추가할 수 있음

  → 그러나 이것만으로는 **오래된 separate 메시지**가 결혼 상태에서 우연히 소비되는 문제를 해결할 수 없음.

완전한 해결책: waitSep 프로토콜

안전한 별거를 위해 **양측이 별거 절차를 명확히 동기화하는 메시지 기반 프로토콜**을 도입한다.

- 핵심 구성요소
  - `waitSep(p)` 상태: 해당 객체가 p에게 separate 메시지를 보냈으며, 그에 대한 응답(separate 메시지)을 p로부터 받기를 기다리는 상태이다.
  - 별거 절차를 3단계로 분리:
    1. 별거 요청 전송 (initSep)
    2. 상대가 married 상태일 때 별거 승인 (acceptSep)
    3. 상대가 waitSep 상태일 때 최종 별거 승인 (acceptSep2)
- 결과
  - 각 참여자는 정확히 **한 번만** separate 메시지를 보내고
  - 정확히 **한 번만** 메시지를 소비하는 절차가 보장됨

- 오래된 메시지가 새로운 결혼을 파괴하는 문제 해결

11.2.1.2 Message Wrappers

메시지는 실제 편지와 비슷하게 **봉투(envelope)** + **메시지 내용(content)** 구조로 되어 있다고 보고, Maude에서 이를 **wrapper 형태**로 모델링한다.

- 핵심 요점
  1. **메시지는 '내용(content)'과 '보낸 사람/받는 사람' 정보로 구성된다.**
  2. 전역 구성에서 유니캐스트 메시지는 다음과 같이 표현된다:

     ```
     msg content from sender to receiver
     ```

  3. 메시지 내용(content)은 `MsgContent`라는 별도 sort로 정의한다.
  4. 메시지 wrapper는 `msg_from_to_` 연산자를 사용해:
     - 메시지 내용
     - 송신자 Oid
     - 수신자 Oid

     를 합쳐 하나의 Msg 객체로 만든다.
- 목적
  - **메시지와 객체를 명확히 구분**하기 위함
  - 분산 시스템 모델링 시 **메시지 전달을 구조화**하기 위함
  - 이후 규칙에서 메시지를 쉽게 패턴 매칭하고 처리할 수 있도록 하기 위함

11.2.2 Multicast

- **개념**
  - **Multicast**는 한 발신자가 **여러 수신자 그룹**에 **동시에** 메시지를 전송하는 방식이다.
  - 주식 시세 알림, 회의 공지 등 **구독자 집단에 대한 일괄 전송**에 적합하다.
- **모델링 아이디어**
  - **수신자 집단**을 `OidSet`(객체 식별자 집합)으로 표현:
    - `subsort Oid < OidSet`
    - 집합 결합 연산: `_;_` (결합·교환, 항등원 `none`)
  - **Multicast 메시지 래퍼** 도입:
    - 표기: `multicast content from sender to rcv1 ; ... ; rcvn`
- **의미론(방정식 기반 전개)**
  - Multicast 한 건은 **각 수신자에 대한 unicast들의 멀티셋**으로 **환원**됨:
    - `multicast ... to none = none`

        - `multicast ... to a ; S = (msg ... to a) (multicast ... to S)`
    - 즉, **구현 관점에서는 "N개의 unicast"로 동등**하게 취급 가능.
- 규칙 형태(전송 트리거)
    - 발신자 객체가 보유한 `multicast-group`에 대해 전송:

```
rl [multicast] :
  < a : Sender | multicast-group : receivers, ... >
=>
  < a : Sender | ... >
  multicast content from a to receivers .
```

    - 결과적으로 시스템 구성에 **개별 unicast 메시지들이 전개되어 추가**된다.
- 정리
    - Multicast = **모델 수준에서는 간결한 한 문장**, **실행 의미론에서는 여러 unicast로 확장**.
    - 셋 표현(`OidSet`)과 방정식 전개로 **명확한 의미 부여**와 **규칙 단순화**를 동시에 달성.

11.2.3 Broadcast

Broadcast의 의미

- **broadcast = 한 노드가 시스템의 '모든 다른 노드'에게 메시지를 보내는 것**
- TV 방송처럼, 보낼 때 "누가 받는지" 모르는 방식.

어떻게 모델링하는가?

전체 시스템은 `{ ... }` 형태로 감싼 **GlobalSystem**으로 표현한다.

예:

```
{ < node1 > < node2 > < node3 > }
```

Broadcast 메시지의 형태

```
broadcast MC from O
```

- `MC` : 메시지 내용 (MsgContent)
- `O` : 메시지를 보낸 노드의 Oid

Broadcast를 실제로 처리하는 핵심 규칙

- 핵심 equation (정말 이것 하나가 전부임):

```
eq { < O : Node | > (broadcast MC from O) REST }
 = { < O : Node | > (multicast MC from O to objectIds(REST)) REST } .
```

  이 식이 하는 일은 딱 **한 가지**임:

- Broadcast MC from O

  → 이를 받는 모든 노드의 ID 목록 `objectIds(REST)`를 구해서 **multicast 메시지로 변환**함.

  즉, broadcast = (자신 제외한 모든 노드 목록)을 구해서 multicast로 바꿔 보내라

REST는 단순히 "나머지 전체 시스템"

- 현재 노드 `<O>`를 제외한 시스템의 나머지를 통째로 REST라고 부름.

  예:

  ```
  { < O > < A > < B > < C > }
  ```

  여기서 `<A> <B> <C>`가 REST이다.

---

objectIds(REST) 함수

```
op objectIds : Configuration -> OidSet [frozen (1)] .
eq objectIds(< O : Node | >  REST) = O ; objectIds(REST) .
eq objectIds(MSG REST) = objectIds(REST) .
eq objectIds((broadcast MC from O)  REST) = objectIds(REST) .
eq objectIds(none) = none .
```

- 목적: REST 안에 들어 있는 **모든 Node 객체의 ID 집합**을 반환.

  즉, REST = `<A> <B> <C>` 라면:

  ```
  objectIds(REST) = {A, B, C}
  ```

- 왜 필요한가?

  broadcast 메시지를 "multicast to {A,B,C}" 로 바꾸기 위해서임.

그 뒤에 나오는 정리 코드들

- objectIds 함수 정의:
  - `< O : Node | > REST`가 나오면 ID = `O ; objectIds(REST)`
  - 메시지는 노드가 아니므로 무시: `objectIds(MSG REST) = objectIds(REST)`
  - broadcast 메시지도 노드 아니므로 무시
  - none이면 none

  즉, objectIds는 단순히 시스템 안에서 노드만 골라서 ID를 모아주는 함수

최종 broadcast 실행 규칙

```
rl [broadcast] :
  < o : ... >  ⇒  < o : ... > broadcast content from o .
```

- 노드가 어떤 행동으로 인해 "broadcast content from o" 메시지를 발생시키면, 위 equation에 의해 자동으로 multicast로 변환됨.

11.2.4 Wireless Broadcast

무선 브로드캐스트란?

- **송신자의 전파 범위(transmission range)** 안에 있는 노드들에게만 메시지를 보내는 방식.
- 즉, 일반 broadcast(모든 노드에게 전송)과 다르게, **닿을 수 있는 노드들에게만 전송**함.

전파 범위를 표현하기 위한 준비

모든 노드는 위치(location)를 가진다:

```
class Node | location : Location .
```

그리고 함수 `withinTransRangeOf`가 있다:

- `L withinTransRangeOf L'` : 위치 L의 송신자가 L'의 노드에 신호를 닿게 할 수 있다.

무선 브로드캐스트 메시지 정의

메시지는 다음 형태로 선언됨:

```
wl-broadcast content from sender
```

동작 원리 (핵심 equation)

무선 브로드캐스트는 결국 "범위 내 노드들에 대한 multicast" 로 변환된다.

무선 broadcast 메시지를 받으면 다음으로 rewriting 됨:

```
{ <O : Node | location : L> (wl-broadcast MC from O) REST }
=
{ <O : Node | > (multicast MC from O to nodesInRange(L, REST)) REST } .
```

즉:

- 송신자 O의 위치 = L
- REST 안의 모든 노드 중 → L과 통신 가능한 노드들 = nodesInRange(L, REST)
- 그래서 **무선 broadcast = 해당 노드들에 대한 multicast**

---

nodesInRange 함수

`nodesInRange(L, REST)` : REST 안에서 송신자 위치 L로부터 신호가 닿는 노드들의 ID 목록을 생성함.

작동 방식:

- REST 안의 각 Node를 확인해:
  - 위치가 범위 안이면 → ID 포함
  - 아니면 포함하지 않음
- 메시지 혹은 wl-broadcast 같은 구조물은 무시하고 계속 탐색.

11.2.5 Modeling Unreliable Communication

현실의 네트워크에서는 메시지가 전송 중에 유실(loss)되거나 손상(corruption)되며, 이는 일반적으로 통신 인프라에서 감지된다. 이러한 불안정한 통신을 Maude에서 모델링하기 위해 다음과 같은 방식이 사용된다.

메시지 유실(Message Loss) 모델링

메시지를 단순히 **없애는 규칙**을 사용해 추상화한다.

```
rl [lose-msg] : msg MC from O to O' ⇒ none .
```

- 송신자(O)에서 수신자(O')로 가는 메시지 `msg MC`가 **아무 흔적 없이 사라지는(unobservable)** 현상을 표현함.

메시지 중복(Message Duplication) 모델링

현실의 재전송(resending) 또는 네트워크 오류로 인해 메시지가 **여러 번 생성**되는 상황을 추상화한다.

```
rl [duplicate-msg] :
   msg MC from O to O'
 ⇒
   (msg MC from O to O') (msg MC from O to O') .
```

- 동일한 메시지가 둘(또는 여러 개)로 복제되는 효과를 모델링함.

메시지 유실 + 중복 조합

두 현상을 동시에 모델링할 수도 있으며,

Maude에서는 다음과 같이 **모듈 합성**으로 표현한다.

```
omod MESSAGE-LOSS-DUPLICATION is
   including MESSAGE-LOSS + MESSAGE-DUPLICATION .
endom
```

Shark 객체를 사용하는 대안적 모델

보다 미세하게 제어하기 위해 `Shark` 객체를 구성할 수 있다.

- 메시지 "포식(devour)" 모델

  ```
  rl [devour-msg] :
    (msg MC from O to O') < O'' : Shark | >
   ⇒
    < O'' : Shark | > .
  ```

- 메시지 "복제(duplicate)" 모델

  ```
  rl [duplicate-msg] :
    (msg MC from O to O') < O'' : Shark | >
   ⇒
    < O'' : Shark | > (msg MC from O to O') (msg MC from O to O') .
  ```

- Shark 객체는 시스템 구성(configuration)을 돌아다니며 메시지를 **잡아먹거나(devour) 복제**하는 역할을 수행.
- 메시지 유실/중복 횟수를 제한하거나 특정 조건에서만 일어나게 할 때 유용하다.

평가

모델링 방식	장점	단점
메시지 유실/중복 규칙 직접 정의	간단하고 선언적	유실/중복 횟수 제어가 어려움
Shark 객체 방식	제어력 높음 (예: 최대 n개만 유실/중복)	병렬성 하락, 모델이 덜 우아함

11.3 Ordered Asynchronous Communication using Links

Ordered asynchronous communication에서는 노드 간 메시지 전달 순서를 보장하기 위해, 각 연결된 노드 쌍마다 **Link 객체**를 사용한다. Link는 메시지를 저장하는 **FIFO(Message Content List)** 구조를 가지며, 이를 통해 송신된 메시지가 수신 측에 도착하는 순서를 유지한다.

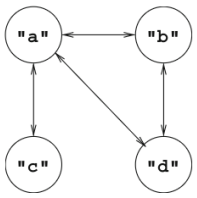
Link 객체의 역할

- Link는 두 노드 사이의 **단방향 통신 채널**을 모델링한다.
- 양방향 통신이 필요할 경우 두 개의 Link 객체가 존재한다.
- Link 객체는 다음과 같은 형태로 표현된다:

```
<a to b : Link | content : mc1 :: mc2 :: ... :: mck >
```

- `content`는 메시지 리스트이며, `mc1`이 가장 먼저 전달될 메시지, `mck`가 가장 나중에 전달될 메시지이다.

전체 시스템 상태(Global Configuration)에서의 표현



그림의 노드와 링크는 전역 상태에서 다음과 같이 나타난다:

```
<a : Node | ... >
<b : Node | ... >
<c : Node | ... >
<d : Node | ... >

<a to b : Link | ... >
<b to a : Link | ... >
<a to c : Link | ... >
<c to a : Link | ... >
<a to d : Link | ... >
```

```
<d to a : Link | ... >
<b to d : Link | ... >
<d to b : Link | ... >
```

각 링크는 특정 방향의 메시지 전달을 담당하며, 노드 간의 실제 네트워크 구조를 반영한다.

메시지 전송(send)

메시지를 보낼 때는 해당 링크의 메시지 리스트 **뒤에** 새로운 메시지를 추가한다. (= FIFO 큐의 enqueue)

규칙은 다음과 같이 정의된다:

```
rl [send-mc] :
  < a : ... >
  < a to b : Link | content : MCL >
⇒
  < a : ... >
  < a to b : Link | content : MCL :: mc > .
```

- `mc` : 새로 전송되는 메시지
- `MCL :: mc` : 기존 메시지 뒤에 새로운 메시지를 추가한 형태

이는 메시지를 보낸 순서대로 링크의 뒤에 축적되도록 한다.

메시지 수신(read)

수신 노드는 링크의 메시지 리스트에서 **첫 번째 요소**를 꺼낸다.

(= FIFO 큐의 dequeue)

규칙은 다음과 같다:

```
rl [read-mc] :
  < b : ... >
  < a to b : Link | content : mc :: MCL >
⇒
  < b : ... >
  < a to b : Link | content : MCL > .
```

- `mc` : 가장 먼저 도착한 메시지
- 수신 후 링크의 content는 `MCL`로 갱신된다.

11.3.1 Unreliable Links

11.3.1 절은 "신뢰할 수 없는 통신 링크"를 O-O Maude에서 어떻게 모델링하는지 설명한다.

핵심은 **메시지가 전송 중 유실되거나 중복될 수 있는 링크를 클래스 구조와 rewrite rule로 표현**하는 것이다.

LossyLink - 메시지 유실이 발생하는 링크

- `LossyLink`는 상위 클래스 `Link`의 서브클래스이다.
- rewrite rule `lose-msg`는 링크의 content 리스트에서 **임의의 메시지를 삭제**하여 유실을 모델링한다.

- 행동

  ```
  MCL :: MC :: MCL'  →  MCL :: MCL'
  ```

  즉, **메시지 MC 하나가 사라진 것처럼 행동**한다.

DuplLink — 메시지가 중복되는 링크

- `DuplLink` 역시 `Link`의 서브클래스이다.
- rewrite rule `duplMsg`는 content 리스트에서 **메시지를 하나 더 추가(복제)**한다.
- 행동

  ```
  MCL :: MC :: MCL'  →  MCL :: MC :: MCL' :: MC
  ```

  즉, **MC를 끝에 하나 더 붙여 중복 전송을 모델링**한다.

UnrelLink — 유실 + 중복이 모두 가능한 링크

- `UnrelLink`는 `LossyLink`와 `DuplLink`를 **다중 상속**한다.
- 따라서 메시지 유실과 중복이 **모두** 발생할 수 있는 "완전 비신뢰적 링크(unreliable link)"를 표현한다.

```
subclass UnrelLink < LossyLink DuplLink .
```

설계상의 핵심 원칙

- 메시지 송수신 규칙은 **반드시 상위 클래스 `Link` 기준으로 작성**해야 한다.
  - 이유: `LossyLink`, `DuplLink`, `UnrelLink`가 모두 `Link`를 상속하므로, 동일한 송수신 규칙을 모든 링크 유형에서 재사용하기 위함이다.
- 초기 상태에서 각 링크가 어떤 종류인지(`Link`, `LossyLink`, `DuplLink`, `UnrelLink`)를 명시하면 해당 링크 특성에 따라 자동으로 메시지가 유실/중복됨.

11.3.2 Links with Limited Capacity

핵심 개념

링크에 저장할 수 있는 메시지 수가 제한되어 있을 때, 링크가 가득 차면 메시지가 **전송 중에 버려질 수 있다(drop)**. 이를 모델링하기 위해 **BoundedLink** 클래스를 사용한다.

BoundedLink 구조

```
class BoundedLink |
   content : MsgContentList,   -- 현재 저장된 메시지 목록
   capacity : NzNat,           -- 최대 저장 가능 메시지 수 (N > 0)
   currentSize : Nat .         -- 현재 메시지 개수
```

- **capacity**: 링크가 저장할 수 있는 최대 메시지 수
- **currentSize**: content 리스트의 실제 크기 → currentSize < capacity일 때만 메시지를 추가할 수 있다.

메시지 전송 규칙

- 링크에 빈 공간이 있어 전송 성공 — `send-OK`

  조건: `currentSize < capacity`

  동작: 메시지 *m* 을 content 리스트의 뒤에 추가하고, currentSize를 1 증가시킨다.

  ```
  < a to b : BoundedLink | content : MCL, capacity : NZ, currentSize : N >
  →
  < a to b : BoundedLink | content : MCL :: m, currentSize : s N >
  ```

- 링크가 가득 차서 전송 실패 — `send-full`

  조건: `currentSize == capacity`

  동작: 링크는 변하지 않으며, 메시지는 버려진다(drop). (즉, 전송이 무시된다.)

  ```
  < a to b : BoundedLink | capacity : NZ, currentSize : NZ >
  →
  < a to b : BoundedLink | >
  ```

11.4 Asynchronous Communication Using Shared Variables

비동기 통신을 메시지 전달(send/receive) 대신, **공유 변수(shared variables)**를 통해 모델링하는 방법을 설명한다. 메시지를 보내는 대신, 여러 컴포넌트가 **공유된 메모리 위치에 값을 기록하고 읽음으로써 통신**을 수행한다.

공유 변수의 기본 개념

- 각 공유 변수 x는 다음 형태의 객체로 모델링된다:

  ```
  < x : SharedVar | value : v >
  ```

  - **x**: 공유 변수의 식별자
  - **v**: 현재 저장된 값
- 공유 변수가 가질 수 있는 값의 타입이 s일 경우, 아래처럼 클래스 정의:

  ```
  class SharedVar | value : s .
  ```

여러 종류의 공유 변수(s_1, …, s_n)를 다루는 세 가지 방법

시스템 내에서 서로 다른 sort의 값을 갖는 여러 공유 변수가 존재할 수 있다. 이 경우 value 속성의 타입을 어떻게 통합할지에 대한 세 가지 접근 방식이 제시된다.

- **방법 1. 각 sort s_i마다 별도의 SharedVar 클래스를 선언**

  ```
  class SharedVar_s1 | value : s1 .
  class SharedVar_s2 | value : s2 .
  ...
  ```

장점: 타입이 명확히 분리됨

단점: 공유 변수 종류가 많을수록 클래스가 증가함

- **방법 2. 모든 sort s_i를 super sort Data의 subsort로 포함시키기**

```
sort Data .
subsort s1 s2 ... sn < Data .

class SharedVar | value : Data .
```

  - 모든 s_i가 Data의 하위 sort가 되어 value에 직접 저장 가능

장점: 단일 SharedVar 클래스 사용 가능

단점: 값들이 모두 Data로 통합되어 **타입 구분이 약해짐**

- **방법 3. wrapping operator를 이용해 s_i 값을 Data로 매핑**

```
sort Data .
op [_] : s1 → Data .
op [_] : s2 → Data .
...
```

SharedVar 정의:

```
class SharedVar | value : Data .
```

값 표현:

```
< x : SharedVar | value : [v] >
```

장점: 타입 안정성 좋음, 모델링 엄밀함

단점: 표현이 다소 복잡해짐