

# 16. Formalizing and Checking Requirements

- 분산 시스템의 요구사항을 명확히 분석하려면, 요구사항을 **수학적으로 형식화해야 한다.**
- 가장 널리 사용되는 방법은 **선형 시간 논리(LTL)**를 이용한 **상태 기반 요구사항 표현**이다.
- LTL은 1977년 Pnueli가 제안했으며, 이 공로로 1996년 튜링상을 수상하였다.
- 초기 상태에서 도달 가능한 상태 공간이 유한하면, **LTL 모델 체커**를 통해 시스템이 LTL 공식  $\varphi$ 를 모든 가능한 실행 경로에서 만족하는지 자동으로 판단할 수 있다.
- Maude 시스템은 **명시적 상태 기반 LTL 모델 체커**를 제공하며, 요구사항이 만족되지 않으면 **구체적 반례 (trace)**를 제시한다.
- 본 장의 구성:
  - 16.1: LTL 기본 개념 소개
  - 16.2: 다양한 시스템 속성을 LTL로 표현하는 방법
  - 16.3: Maude 모델 체커로 요구사항 만족 여부 검사 및 예제 설명
  - 16.4: LTL의 확장과 변형 논의

## 16.1 Linear Temporal Logic

### LTL의 사용 목적

- LTL(Linear Temporal Logic)은 재작성 명세(rewrite specifications)의 속성을 형식화하기 위해 사용된다.

### 논리의 구성 요소

논리는 일반적으로 다음 세 요소로 이루어진다:

#### 1. Syntax

- 논리의 **공식(formulas)**을 정의한다.

#### 2. Semantics

- 공식이 주어진 명세에서 **성립한다는 것의 의미**를 정의한다.

#### 3. Proof system

- 공식을 **추론하거나 증명할 수 있는 규칙들의 집합**이다.

### 예제

#### 1. 예제 16.1 : 등식 논리(equational logic)

- 공식:  $t = u$
- 의미론:  $E \models t = u$  iff 모든 변수 할당  $\sigma$ 에 대해  $\sigma^*(t)$ 와  $\sigma^*(u)$ 의 해석이 각각의 모델에서 같은 원소일 때 성립 한다.
- 증명 체계: Section 6.1

## 2. 예제 16.2 — 재작성 논리(rewriting logic)

- 공식 형식:  $t \rightarrow u$
  - 모델: Section 8.6에서 간단히 언급됨
  - 증명 체계: Section 8.4
- 

### 본 절의 범위

- 본 절은 LTL의 구문과 의미론을 소개한다.
- 목표는 모델 체킹을 통해 속성 만족 여부를 자동으로 검사하는 것이다.
- LTL의 증명 체계는 존재하지만, 본 책에서는 제공하지 않는다.

### 16.1.1 Behaviors

#### 초기 상태에서의 동작 가정

- 본 장에서는 초기 상태  $t_0$ 에서의 모든 동작을 무한한 one-step sequential rewrites의 시퀀스로 가정한다.
  - 이 가정을 통해 유한 동작과 무한 동작을 따로 정의할 필요가 없어지도록 한다.
- 

#### 유한 동작의 확장

- 유한 동작:

$$t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$$

- $t_n$ 에서 더 이상 rewrite가 불가능해도, self-loop을 추가하여 다음과 같이 무한 경로(path)로 확장할 수 있다:

$$t_n \rightarrow t_n \rightarrow t_n \rightarrow \cdots$$

- Maude의 모델 체커는 이 확장을 자동으로 수행한다.

#### 정의 16.1 — Behavior

- 명세  $\mathcal{R}$ 에서 상태  $t_0$ 에서 시작하는 동작은 다음과 같은 무한 시퀀스이다:

$$t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots$$

- 이는 모든 one-step rewrite  $t_i \rightarrow t_{i+1}$ 들의 연속이다.

#### Notation

- 초기 상태  $t_0$ 에서 시작하는 모든 동작들의 집합:

$$\text{Paths}_{\mathcal{R}}(t_0)$$

- 경로  $\pi$ 에 대해:
  - $\pi(k) = t_k$  : 경로의  $(k + 1)$ -번째 상태
  - $\pi^k = t_k \rightarrow t_{k+1} \rightarrow t_{k+2} \rightarrow \cdots$  :
- 상태  $t_k$  이후의 나머지 동작 전체

## 16.1.2 The Syntax of LTL

### Atomic Propositions (원자 명제)

- LTL 공식의 기본 구성 요소는 **atomic propositions**(원자 명제).
- 상태 기반 논리에서 원자 명제  $p$ 는 **state proposition**이며, 특정 상태  $t$ 에서 참 또는 거짓이다.
- **state proposition**은 Section 15.1.2에서 정의됨.

---

### 예제

- **Example 16.3 (ONE-PERSON 명세)**
  - sort: Person
  - state propositions: alive, dead, teenager
- **Example 16.4 (Dining Philosophers)**
  - state propositions: noNeighborsEating, phil3Eating, phil2Eating, phil2hasOneStick
  - sort: Configuration
- **Example 16.5 (NSPK)**
  - state propositions: a\_hasTrustedConnectionWith\_b

---

### LTL의 논리 연산자

- **불리언 연산자**
  - $\neg$  (not)
  - $\wedge$  (and)
  - $\vee$  (or)
  - $\rightarrow$  (implies)
  - $\leftrightarrow$  (if and only if)
- **시간 연산자(Temporal Operators)**
  - $\Box \varphi$  : 경로 전체에서  $\varphi$ 가 참
  - $\Diamond \varphi$  : 경로 어딘가에서  $\varphi$ 가 참
  - $\bigcirc \varphi$  : 다음 상태에서  $\varphi$ 가 참
  - $\varphi \And \psi$  :  $\psi$ 가 등장하기 전까지 모든 위치에서  $\varphi$ 가 참이며,  $\psi$ 는 언젠가 참
  - $\varphi \And \psi$  :  $\varphi \And \psi$ 와 유사하나  $\psi$ 가 아예 나타나지 않아도 됨 ( $\psi$ 가 절대 참이 되지 않으면  $\varphi$ 는 계속 참이어야 함)

---

### 정의 16.2 : LTL 공식의 귀납적 정의

LTL 공식은 다음 규칙에 따라 정의된다:

1. **true, false**는 LTL 공식
2. AP(atomic propositions) 내의  $p$ 는 LTL 공식
3.  $\varphi, \psi$ 가 LTL 공식일 때 다음 역시 LTL 공식:
  - $\neg\varphi$

- $\varphi \wedge \psi$
  - $\varphi \vee \psi$
  - $\varphi \rightarrow \psi$
  - $\varphi \leftrightarrow \psi$
  - $\Box\varphi$
  - $\Diamond\varphi$
  - $\varphi \text{ } \text{u} \psi$
  - $\varphi \text{ } \text{W} \psi$
  - $\text{O}\varphi$
- 

#### Example 16.6 : Temporal Logic Formulas

1.  $\Box\text{alive}$  : 그 사람은 항상 alive
  2.  $\Diamond\text{dead}$  : 언젠가는 죽는다
  3.  $\text{alive } \text{u} \text{dead}$  : 죽기 전까지 계속 alive
  4.  $\text{alive } \text{W} \text{dead}$  : 영원히 alive일 수도 있음
  5.  $\text{Oteenager}$  : 다음 상태에서 teenager
  6.  $\Box \text{noNeighborsEating}$  : dining philosophers의 핵심 속성
  7.  $\Diamond\text{phil2Eating}$  : 철학자 2는 언젠가 먹게 됨
  8.  $\Box \neg \text{"Bank"} \text{hasTrustedConnectionWith} \text{"Scrooge"}$  : NSPK 핵심 속성
- 

#### 최소 연산자 집합

다음 연산자만 있으면 충분함:

- **true, p,  $\neg$ ,  $\wedge$ ,  $\text{u}$ , O**

다른 연산자들은 이들로 정의 가능:

- $\varphi \vee \psi = \neg((\neg\varphi) \wedge (\neg\psi))$
- $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$
- $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$

시간 연산자( $\Box$ ,  $\Diamond$ ,  $\text{W}$ ) 역시  $\text{u}$ 로 정의될 수 있음.

---

#### 중첩 가능성

- $\varphi$ 와  $\psi$ 는 다시 LTL 공식이므로 중첩 공식(nested formulas) 가능
- 예:  $\Box(p \rightarrow O\Diamond q)$

---

### 16.1.3 The Semantics of LTL

#### LTL이 뭘 하는 건가? (한 줄 요약)

- LTL은 “시간이 흐르면서 시스템이 어떻게 동작하는지”를 논리식으로 표현하고, 그 시스템이 특정 요구사항을 만족하는지 자동으로 검사하기 위한 논리이다.

- 즉,
  - 시스템이 “영원히 잘 돌아가야 한다”
  - “언젠가는 꼭 X가 일어나야 한다”
  - “Y가 일어나기 전까지는 반드시 Z여야 한다”

같은 **시간적 조건을 표현하는 논리다.**

→ 그래서 모델 체크(model checking)에서 많이 쓰임.

### 원자 명제(Atomic Proposition)

- “특정 상태가 어떤 특징을 만족하는가?”를 표현하는 가장 기본 단위.
- 예
  - alive
  - dead
  - phil2Eating
  - a\_hasTrustedConnectionWith\_b
- 즉, 시스템 상태에 “참/거짓” 값을 붙이는 태그 같은 것.

### 라벨링 함수(Labeling Function)

- “각 상태가 어떤 원자 명제를 만족하는지” 표시하는 함수.
- 예
  - Peter → {alive}
  - Joan of Arc → {dead, teenager}
- 즉, 상태 → 참인 명제들의 집합 매핑.

### 경로(Path)란?

- 시스템이 시간에 따라 어떻게 변하는지를 나타내는 상태들의 무한 시퀀스.
- 왜 무한?
  - 시스템이 끝나더라도 끝 상태에 **self-loop**을 달아서 끝없이 “같은 상태 → 같은 상태 → ...”로 이어지도록 정의하기 때문.
- 이렇게 하면 LTL 공식 의미를 일관되게 정의할 수 있음.

### 경로가 공식 $\varphi$ 를 만족한다는 것의 의미

- LTL 문장은 결국 경로 단위로 평가된다.
- 경로  $\pi$ 에 대해:

공식	의미
$\Box \varphi$	“경로 전체에서 $\varphi$ 가 항상 참”
$\Diamond \varphi$	“경로 어디선가 $\varphi$ 가 한 번이라도 참”
$\circlearrowleft \varphi$	“다음 상태(next state)에서 $\varphi$ 참”
$\varphi \sqcup \psi$	“ $\psi$ 가 나타날 때까지 $\varphi$ 가 계속 참이고, $\psi$ 는 언젠가 나타남”

공식	의미
$\varphi W \psi$	" $\psi$ 가 나타날 수도, 안 나타날 수도 있음. 안 나타나면 $\varphi$ 는 영원히 참이어야 함"

- 즉, 시간 흐름을 따라 명제가 어떻게 참이 되는지를 표현한다.

### 시스템이 LTL 공식을 만족한다는 것

- 초기 상태  $t_0$ 에서 출발하는 모든 가능한 경로가 공식을 만족해야 한다.
- 즉, "시스템을 여러 방식으로 실행해도 단 한 번도 규칙을 어기면 안 된다."라는 뜻.

### 왜 이렇게 복잡하게 정의할까?

- LTL의 본질적인 목적:
    - "시스템의 미래 행동"에 대해 자동으로 참/거짓을 판정하기 위해.
  - 이를 위해서는:
    - 상태들이 어떤 속성을 갖는지(=라벨링)
    - 상태가 시간에 따라 어떻게 이어지는지(=경로)
    - 경로가 시간적 조건을 충족하는지(=LTL 의미론)
- 을 정확하게 정의해야 한다.
- 그래야 모델 체커가 다음을 할 수 있음: "이 시스템이 절대 교착 상태에 빠지지 않는다는 것을 자동으로 증명해 줘."

### 예시로 이해해보자 (직관)

- $\Diamond \text{dead}$  : 언젠가는 죽어야 한다.
  - 경로 상에서  $\text{dead}$ 가 **한 번이라도** 나오면 True.
- $\Box \text{alive}$  : 항상 살아 있어야 한다.
  - 경로 전체에서  $\text{alive}$ 가 유지되지 않을 경우 False.
- $\text{alive} U \text{dead}$  : 죽을 때까지 살아 있어야 한다.
  - 즉,
    - 죽기 전까지는  $\text{alive}$ 여야 하고
    - 죽는 순간이 반드시 존재해야 한다.
- $\text{alive} W \text{dead}$ 
  - 죽을 수도 있고, 안 죽을 수도 있다. 죽기 전까지  $\text{alive}$ 여야 하며, 죽지 않으면 영원히  $\text{alive}$ 여야 한다.

### 왜 Until(U)와 Weak-Until(W)이 중요할까?

- LTL의 핵심 연산은 사실 **U (Until)**이다. 나머지는 대부분 U를 이용해 정의할 수 있다.
- 예
  - $\Diamond \varphi = \text{true} U \varphi$
  - $\varphi W \psi = (\varphi U \psi) \vee \Box \varphi$

### 전체 흐름 요약

1. 상태는 여러 속성(atomic proposition)을 가진다.
2. 라벨링 함수가 상태  $\rightarrow$  참인 명제 집합을 매핑한다.
3. 경로는 상태들의 무한 시퀀스.
4. LTL 공식은 "시간을 따라 명제들이 언제 참인지"를 표현한다.
5. 경로가 공식을 만족하는지 엄밀히 정의한다.
6. 시스템 전체는, 초기 상태에서 시작하는 모든 경로가 공식을 만족하면 "요구사항을 만족"한다고 한다.

### 16.1.4 Kripke Structures

왜 Kripke Structure가 필요한가?

- 지금까지 LTL 의미론을 **rewrite theory** 기반으로 정의했지만, LTL은 **Petri net, automata, process algebra** 등 다양한 모델에도 적용됨.
- 그래서 보다 일반적인 의미론 모델인 **Kripke Structure**가 사용됨.

**Kripke Structure의 정의 (Definition 16.7)**

Kripke Structure는 다음 세 요소로 이루어진 삼중항(**triple**):  $(S, \rightarrow, L)$

1. **S: 상태들의 집합**

- 시스템이 가질 수 있는 모든 상태들의 집합.

2.  **$\rightarrow$  : transition relation (전이 관계)**

$$\rightarrow \subseteq S \times S$$

- "총(total)"이어야 한다:

- 즉, 모든 상태  $s$ 는 적어도 하나의 후속 상태  $s'$ 을 가진다. (어떤 상태에서도 무조건 앞으로 갈 수 있어야 함)

3. **L: labeling function**

$$L : S \rightarrow \mathcal{P}(AP)$$

- 각 상태가 어떤 atomic proposition들을 만족하는지 할당.

**Rewrite Theory  $\rightarrow$  Kripke Structure로 변환**

Rewrite theory

$$\mathcal{R} = (\Sigma, E, R)$$

와 designated state sort **State**, 라벨링 함수  $L$ 이 주어지면

다음과 같이 Kripke Structure를 구성할 수 있다:

$$(T_{\Sigma, E, State}, \rightarrow^*, L)$$

여기서:

✓ 상태 집합 **S**

- sort **State**의 ground term들의 **E-equivalence class**.

✓ 전이 관계  $\rightarrow^*$

- one-step sequential rewrite relation
- deadlock 상태에는  $t \rightarrow t$  self-loop을 추가한 것.

✓ 라벨링 함수 **L**

- $E$ -equivalent한 상태들은 반드시 동일한 라벨 집합을 가져야 함.  
그래야  $L$ 이 “잘 정의된 함수(well-defined)”가 된다.

---

### 짧은 요약

- Kripke Structure = (상태 집합, 전이 관계, 라벨링 함수)
- 모든 상태는 적어도 하나의 다음 상태를 가져야 함
- Rewrite theory는 naturally Kripke Structure로 바뀔 수 있음
- 이 구조 위에서 LTL 의미론을 일반적인 방식으로 정의할 수 있음

## 16.2 Some LTL Formulas

이 절에서는 다양한 LTL 공식들을 논의하며, 15장에서 언급된 여러 종류의 속성(property)들과 공정성 가정(fairness assumptions)을 어떻게 형식화하는지를 다룬다.

### 16.2.1 Formalizing Classes of Requirements

이 절에서는 15장에서 언급된 속성들을 형식화하고, 그 외 추가적인 속성들도 논의한다.

---

## Invariance (불변성)

어떤 (상태) 공식  $\zeta$ 가 \*\*불변(invariant)\*\*인지, 즉 초기 상태  $t_0$ 로부터 도달 가능한 모든 상태에서 성립하는지를 검사하는 것은

다음 조건을 검사하는 것과 동일하다:

$$\mathcal{R}, t_0 \models \Box \zeta$$

---

## Guarantee (보장성)

어떤 속성  $\zeta$ 를 만족하는 상태가 초기 상태  $t_0$ 에서 가능한 모든 실행 경로에서 결국 도달됨이 보장되는지 확인하는 것은

다음 조건을 검사하는 것과 동일하다:

$$\mathcal{R}, t_0 \models \Diamond \zeta$$

---

## Reachability (도달 가능성)

LTL에는 “가능하다면 도달해야 한다(it must be possible to reach)”라는 의미의 도달 가능성을 직접 표현하는 공식이 존재하지 않는다.

왜냐하면:

- $\Diamond \zeta$  는 모든 경로에서 언젠가  $\zeta$ -상태가 도달되어야 함을 의미한다.
- 하지만 reachability는 “적어도 한 경로에서  $\zeta$  도달 가능”이라는 의미이므로 LTL로 표현할 수 없다.

그러나 LTL 모델 체킹을 이용하여 간접적으로 reachability를 검사할 수 있다:

- $\zeta$ -상태가 도달 가능함은

$$\mathcal{R}, t_0 \not\models \Box \neg \zeta$$

일 경우와 동일하다.

- 즉,  $\Box \neg \zeta$  를 검사했을 때 LTL 모델 체커가 반례(counterexample) 를 반환하면, 그 반례 경로 안에  $\zeta$ -상태가 포함되어 있기 때문에 도달 가능성이 증명된다.
- 

## Response (응답성)

“요청(request)”이 발생하면 반드시 “응답(response)”이 따라와야 한다는 요구사항은

아래 LTL 공식으로 형식화된다:

$$\Box(\varphi \rightarrow \Diamond \psi)$$

이 공식은 경로의 모든 위치에서:

- 만약  $\varphi$ 가 성립한다면,
- 나중에 어딘가에서 반드시  $\psi$ 가 성립해야 함을 의미한다.

응답이 요청 이후에 와야 하는 경우는 Exercise 229에서 다룬다.

---

## Stability (안정성)

어떤 속성이 일단 성립하기 시작하면 영원히 지속되어야 하는 경우,  
다음 공식으로 표현된다:

$$\square(\varphi \rightarrow \square\varphi)$$

즉,  $\varphi$ 가 성립하는 순간부터는 계속 성립해야 한다.

예:

$$\square(\text{dead} \rightarrow \square\text{dead})$$

---

## A Property that Cannot be Checked (검사할 수 없는 속성)

다음 요구사항은 LTL로 표현할 수 없다:

“ $\zeta$ -상태에서  $\phi$ -상태로 갈 가능성성이 존재해야 한다.”

예:

- “로또를 사면 언젠가 부자가 될 가능성이 있어야 한다”
- 하지만

$$\square(\text{hasValidLotteryTicket} \rightarrow \diamond\text{isMillionaire})$$

는 “로또를 사면 반드시 언젠가는 부자가 된다”를 뜻하므로 잘못된 표현이다.

- 즉 “may-lead-to” 속성은 LTL로 형식화할 수 없다.

또한,

$$\square(\text{hasValidLotteryTicket} \rightarrow \square\neg\text{isMillionaire})$$

에 대한 모델체킹 반례가 “원래 의도한 속성”이 틀렸다는 의미도 아니다.

---

## Infinitely Often (무한히 자주)

어떤 속성이 경로마다 무한히 자주 성립해야 하는 요구사항이 있다.

예:

"철학자 2는 무한히 자주 먹어야 한다."

이는 다음 LTL 공식으로 표현된다:

$$\square\lozenge\varphi$$

왜 이 공식이 "무한히 자주"임을 보장할까?

- 경로 첫 위치에서  $\square\lozenge\varphi$ 가 참이라고 하자.
- 만약  $\varphi$ 가 유한 번만 참이라고 가정하면 마지막으로  $\varphi$ 가 참이 되는 위치  $k$ 가 존재함.
- 그러나  $\square\lozenge\varphi$ 는 모든 위치에서 "언젠가  $\varphi$ 가 참"이어야 하므로 모순.

따라서  $\varphi$ 는 무한히 자주 참이어야 한다.

결론:

$$\square\lozenge phil2Eating$$

은 "철학자 2가 무한히 자주 먹는다"는 요구사항을 정확히 형식화한다.

---

## Holds Continuously (지속적으로 성립함)

유사한 속성으로 "어떤 시점 이후로는 계속  $\varphi$ 가 참이어야 한다"는 것이 있다.

이는 다음 공식으로 표현된다:

$$\lozenge\square\varphi$$

의미:

1. 경로 어딘가에서  $\varphi$ 가 처음으로 참이 되는 시점이 있고,
2. 그 이후로는  $\varphi$ 가 영원히 유지됨.

예:

$$\lozenge\square dead$$

(단, ONE-PERSON 모듈에서는 divorce 규칙 때문에 이 공식은 참이 아니다.)

## 16.2.2 Fairness Assumptions

### Fairness(공정성)의 목적

- 시스템이 "불공정한 행동" 때문에 진전(progress)을 못 하는 상황을 막기 위함.
- 예
  - 어떤 이벤트가 가능하지만 수행되지 않고 무시되기만 함

- 어떤 철학자는 계속 생각만 하고 영원히 hungry가 되지 못함
  - 메시지는 계속 생성되고 삭제만 됨
  - 이러한 비정상적, 불공정한 실행을 차단하기 위해 fairness 조건을 둔다.
- 

### Fairness의 두 종류

#### 1. Compassion(연민 공정성)

- “이벤트 e가 무한히 자주 가능하다면, e는 무한히 자주 실행돼야 한다.”
- LTL 공식:

$$(\square \Diamond e_{enabled}) \rightarrow \square \Diamond e_{taken}$$

#### 2. Justice(정의 공정성)

- “이벤트 e가 어떤 시점 이후 계속 가능하다면, e는 무한히 자주 실행돼야 한다.”
- LTL 공식:

$$(\Diamond \Box e_{enabled}) \rightarrow \square \Diamond e_{taken}$$

---

### Dining Philosophers 예제

- Compassion 예
  - 철학자 2가
    - 젓가락 하나를 이미 들고 있고
    - 다른 젓가락이 free인 상황이 무한히 자주 발생한다면
    - 철학자 2는 무한히 자주 먹어야 한다.
  - 공식:

$$(\square \Diamond (phil2hasOneStick \wedge (stick2free \vee stick3free))) \rightarrow \square \Diamond phil2eating$$

- Justice 예
  - Justice만으로는 철학자 2가 항상 hungry가 된다는 것을 보장하지 못함.
  - 불공정하게 계속 thinking만 할 수도 있기 때문.
  - Justice는 최소한 “배고파지는 상태”는 보장할 수 있다.
  - 공식:

$$(\Diamond \Box phil2thinking) \rightarrow \square \Diamond phil2hungry$$

---

### Fairness를 모델 체커가 직접 지원하지 않는 경우

- Fairness 조건 전체를 하나의 공식  $\psi$ 로 묶고, 검증하고 싶은 속성  $\varphi$ 에 대해:

$$\psi \rightarrow \varphi$$

를 모델체킹하면 된다.

- 즉, "fairness가 유지된다는 가정 아래  $\varphi$ 가 참인가?"

#### 이벤트 e\_taken을 직접 표현할 수 없는 이유

- 우리는 **state-based logic**을 쓰기 때문에 "이벤트 e가 실행됨(e\_taken)"을 직접 표현할 수 없다.
- 따라서 e\_taken은 "이벤트 e를 수행했을 때 어떤 상태 변화가 발생하는가?"를 기반으로 **간접적으로 정의해야 한다.**
- 예
  - 이벤트 = 철학자 2가 grabSecond 규칙 수행
  - 효과 = 철학자 2가 eating 상태가 됨
  - 이 상태(eating)를 통해 e\_taken을 표현

### 16.3 Model Checking in Maude

Maude의 모델 체커는:

- 초기 상태에서 시작하는 **모든 경로가 LTL 공식을 만족하는지** 검사
- 만족하지 않으면 **반례 경로(path)** 출력
- 파라미터형 atomic proposition** 사용 가능
- equation**을 활용한 복잡한 LTL 속성 정의 가능
- LTL 공식이:
  - satisfiable**(어떤 명세에서 참인지)
  - tautology**(모든 명세에서 참인지)
 여부도 검사 가능
- 따라서 Maude는 LTL 기반 검증을 **강력하고 유연하게 지원한다.**

#### 16.3.1 Getting Started

모델 체커 사용하기 위한 설정 절차:

- model-checker.maude 파일을 **직접 로드해야 한다.**
- 다음 두 모듈을 import:
  - MODEL-CHECKER**
  - 시스템을 정의한 사용자 모듈
- 시스템 상태를 LTL 평가에 포함하기 위해 다음 선언 필요:

```
subsort s < State .
```

(s : 시스템의 상태 sort)

#### 4. Full Maude 사용 시 모듈 전체를 괄호로 감싸서 선언

→ 즉, "Maude에서 모델체킹 준비하는 모듈 구조"를 구성하는 단계

### 16.3.2 Defining Atomic Propositions

- Atomic proposition = 속성이 어떤 상태에서 참인지 표현한 논리식
- Maude에서는 다음과 같이 정의:

```
ops alive dead teenager : → Prop .  
ops is_yearsOld olderThan : Nat → Prop .
```

- 의미 부여는 내장 함수 |= 로 정의:

```
op _|=_ : State Prop → Bool .
```

- 각 상태에서 proposition이 언제 true인지 equation으로 명시 (false인 경우는 따로 정의할 필요 없음)
- 예

```
eq person(N,S) |= alive = (S /= deceased) .  
eq person(N,S) |= dead = (S == deceased) .
```

- 즉, LTL에서 사용되는 atomic proposition을 "Maude 상태 모델과 연결"하는 단계이다.

### 16.3.3 Defining LTL Formulas

#### LTL 공식 정의 방식

- LTL 공식은 **sort Formula** 타입으로 정의함
- 기존 Prop(원자 명제)은 Formula의 하위 sort

```
subsort Prop < Formula .
```

- 주요 논리 및 시간 연산자 제공

Maude 표기	의미	LTL 기호
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">~ φ</span>	부정	$\neg \varphi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">φ ∧ ψ</span>	논리곱	$\varphi \wedge \psi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">φ ∨ ψ</span>	논리합	$\varphi \vee \psi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">φ → ψ</span>	함의	$\varphi \rightarrow \psi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">□ φ</span>	항상	$\Box \varphi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">◇ φ</span>	언젠가	$\Diamond \varphi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">φ U ψ</span>	until(강한)	$\varphi U \psi$
<span style="background-color: #e0e0e0; border: 1px solid black; padding: 2px;">φ W ψ</span>	weak-until	$\varphi W \psi$

Maude 표기	의미	LTL 기호
O φ	다음 상태에서	Oφ

```

ops True False : → Formula [ctor ...] .
op ~_ : Formula → Formula [ctor prec 53 ...] .
op _/\_ : Formula Formula → Formula [comm ctor prec 55 ...] .
op _\/_ : Formula Formula → Formula [comm ctor prec 59 ...] .
op O\_ : Formula → Formula [ctor prec 53 ...] .
op _U\_ : Formula Formula → Formula [ctor prec 63 ...] .
ops _→_ _↔_ : Formula Formula → Formula [prec 65 ...] .
op <>_ : Formula → Formula [prec 53 ...] .
op []_ : Formula → Formula [prec 53 ...] .
op _W\_ : Formula Formula → Formula [prec 63 ...] .

```

### 16.3.4 Performing Model Checking

- **Model Checking** 명령

```
red modelCheck(t0 , φ) .
```

- 초기 상태  $t_0$ 에서 시작하는 모든 경로가  $\varphi$ 를 만족하면 **true**
- 속성 불만족 시
  - Maude는 **counterexample path** 출력
  - deadlock 상태는 자동으로 **self-loop(deadlock)**이 추가된 경로로 표현됨
- Example 상황 요약

1. 검사 공식:

```
alive U dead
```

- “죽을 때까지 살아있어야 한다”
- 하지만 결혼/이혼 무한 루프가 존재 → 죽음에 도달 못할 수도 있음 → **반례(counterexample)** 출력

2. 좀 더 현실적으로 검사:

```
[] (dead → [] dead)
```

- 죽었다면 이후 항상 죽은 상태 유지 → **true**
- Fairness 적용 모델체킹 활용
  - 생일 이벤트가 무시되는 비공정 실행 제거를 위해 fairness 공식을 사용
  - 예:

```
fairBirthdays(46,55) → <> (is 55 yearsOld ∨ dead)
```

- 46세 사람은 공정하게 생일이 발생하면 반드시 55세 되거나 죽어야 함 → 결과 : true

### 16.3.5 Example : Analyzing Mutual Exclusion

이 절에서는 중앙 서버 기반 상호 배제(mutual exclusion) 알고리즘을 분석한다.

이때 각 프로세스는 무한히 반복 실행한다고 가정한다(Exercise 184 참조).

13.2절과의 유일한 차이는, 프로세서가 임계 구역을 빠져나올 때 afterCS 상태가 아니라 beforeCS 상태로 간다는 점이다.

이러한 분산 상호 배제는 아래 세 가지 요구사항을 만족해야 한다:

1. 두 프로세스가 동시에 임계 구역에 들어가면 안 된다.
2. 각 프로세스는 무한히 자주 임계 구역에 진입할 수 있어야 한다.
3. 프로세스들은 임계 구역 요청 순서대로 들어가야 한다.

요구사항 (1)은 search를 통해 13.2절에서 분석했다.

이 절에서는 요구사항 (2)와 (3)을 분석한다.

#### Requirement (ii) : 각 프로세스는 임계 구역을 무한히 자주 들어간다

우리는 우선 아래의 **parametric(매개변수화된)** 원자 명제를 정의한다:

명제	의미
beforeCS(o)	o 노드가 임계구역 밖에 있음
waiting(o)	o 노드가 임계구역 진입 대기 중
inCS(o)	o 노드가 임계구역 안에 있음

그러나 **공정성(fairness)**이 없다면, 예를 들어 node(3)가 무한히 임계 구역에 진입하지 못할 수도 있다.

실행 예:

```
red modelCheck(init(3), [] <> inCS(node(3))) .
```

→ 결과: counterexample (즉, 성립하지 않음)

그래서 우리는 **just path(공정한 실행)** 만 고려해야 한다. 이를 위해 rewrite rule들의 **공정한 적용 조건**을 정의한다.

#### Justice rule 1

노드 o가 beforeCS 상태에서 계속 활성(enabled) 상태라면 언젠가는 waiting 상태가 되어야 한다:

```
op justiceRule1 : Oid → Formula .
eq justiceRule1(O) = (<> [] beforeCS(O)) → <> waiting(O) .
```

#### Request message 공정성

서버가 운이 없는 노드의 요청을 계속 무시할 가능성이 있음 → 요청 메시지가 언젠가는 읽혀야 함을 명시해야 함.

요청 메시지가 지속적으로 상태에 남아있지 않도록:

```

op reqMsgFairness : Oid → Formula .
eq reqMsgFairness(O) = ~ (<> [] reqFrom O inState) .

```

이를 적용하려면, 먼저 같은 노드의 요청 메시지가 상태에 두 개 존재하지 않음을 증명:

```

search [1] init(4) ⇒* (msg requestCS from O:Oid to server)
                           (msg requestCS from O:Oid to server)
                           REST:Configuration .

```

→ No solution. (즉, 요청 메시지는 최대 하나만 존재함)

따라서 공정성 정의:

```

op justice : Oid → Formula .
eq justice(O) = justiceRule1(O) ∧ reqMsgFairness(O) .

```

### Requirement (ii) 검증 (node3)

```

red modelCheck(init(3),
               justice(node(3)) → [] <> inCS(node(3))) .

```

→ result Bool : true

모든 노드에 대해 동시에 검증:

```

red modelCheck(init(3),
               (justice(node(1)) ∧ justice(node(2))
                ∧ justice(node(3)))
               → ([] <> inCS(node(1))
                  ∧ [] <> inCS(node(2))
                  ∧ [] <> inCS(node(3))) .

```

→ result Bool : true

### Requirement (iii) : 임계구역 요청 순서대로 접근해야 함

두 노드  $o_1, o_2$ 에 대해:  $o_1$ 이 먼저 요청했고 아무도 임계구역에 없다면  $\rightarrow o_1$ 이  $o_2$ 보다 먼저 임계구역에 들어가야 한다.

이를 위해 새로운 연산자 before 정의:

```

op _before_ : Formula Formula → Formula .
eq P before Q = (~ Q) W (P ∧ ~ Q) .

```

그리고 정의:

```

op rightOrder : Oid Oid → Formula .
eq rightOrder(O1, O2) =
    [] ((~ inCS(O1)) ∧ (~ inCS(O2)))

```

```

    /\ (waiting(O1) before waiting(O2)))
→ (inCS(O1) before inCS(O2)) .

```

그러나 다음 실행을 통해 **rightOrder(node1,node2)** 는 성립하지 않음이 발견됨:

```
red modelCheck(init(4), rightOrder(node(1),node(2))) .
```

서버가 여러 요청 중 **임의로 읽기 때문** → 서버가 요청 메시지를 보낸 순서대로 읽어야 함

### Ordered request read fairness

```

op orderedReqRead : Oid Oid → Formula .
eq orderedReqRead(O1, O2) =
  ( reqFrom O1 inState /\ ~ reqFrom O2 inState )
  W (~ reqFrom O1 inState
    /\ (reqFrom O1 inState /\ reqFrom O2 inState)
    W (~ reqFrom O1 inState)) .

```

세 노드에 대해 모든 요청 읽기 순서 유지:

```

op allReqsReadInOrder : → Formula .
eq allReqsReadInOrder =
  [] (orderedReqRead(node(1),node(2))
    /\ orderedReqRead(node(2),node(1))
    /\ orderedReqRead(node(1),node(3))
    /\ orderedReqRead(node(3),node(1))
    /\ orderedReqRead(node(2),node(3))
    /\ orderedReqRead(node(3),node(2))) .

```

최종 검증:

```

red modelCheck(init(3),
  allReqsReadInOrder →
  ( rightOrder(node(1),node(2))
  /\ rightOrder(node(2),node(1))
  /\ rightOrder(node(1),node(3)) )) .

```

→ result Bool : true

## 16.4 Some More Temporal Logic

### Satisfiability and Tautology Checking (만족가능성과 모든 명제에서 참인지 검사)

모델 체커의 **SAT-SOLVER** 모듈은 다음을 검사할 수 있는 솔버를 제공한다:

- LTL 공식이 **satisfiable** 한지  
→ 어떤 Kripke 구조의 어떤 경로에서라도 성립하는지
- LTL 공식이 **tautology** 인지

→ 모든 Kripke 구조의 모든 경로에서 성립하는지

항진명제 검사(tautology checker)는 두 LTL 공식  $\varphi$ 와  $\psi$ 가 동치인지  $\varphi \leftrightarrow \psi$ 가 항진명제인지 검사하여 판정할 수 있다.

예: Exercise 231의 일부 공식을 비교할 수 있음

```
load model-checker
```

```
mod CHECK-TAUT is including SAT-SOLVER .
```

```
  ops P Q : → Formula .
```

```
endm
```

```
Maude> red tautCheck((([] [] P) ↔ ([] P))) .
```

```
result Bool: true
```

또 다른 예:

```
Maude> red tautCheck((([] P) → [] Q) ↔ [] (P → [] Q)) .
```

→ 결과: counterexample 출력 = 해당 공식이 항진명제가 아님을 보여주는 경로 존재

**Temporal Logic of Rewriting: Combining State and Action Propositions** (재작성 기반 시간 논리: 상태와 행동 명제 결합)

15.1절과 16.3.5절에서 보았듯이, 상태 기반 논리만으로는 **공정성(fairness)** 요구사항을 표현하기 어렵다.

- 공정성은 행동의 enable 상태(언제 실행될 수 있는지)와 행동이 실제 "실행됨"을 모두 포함하기 때문

이를 위해 **Temporal Logic of Rewriting(TLR)** 사용

- 상태 기반 원자 명제 + **행동 패턴(action patterns)** 사용 가능

행동 패턴:

- 규칙 이름과 부분 치환으로 구성 (rewrite rule의 적용 표현)
- path는 첫 재작성 단계가 패턴과 일치할 때 행동 패턴을 만족

TLR = LTL + state propositions + action patterns

예: 메시지 읽기 공정성 메시지  $m$ 에 대해:

```
◇ [] "message m from o is in the state"  
→ ◇ [] ("rule l1 적용" ∨ ... ∨ "rule lk 적용")
```

### TLR에서의 공정성 가정의 장점

Exercise 237에서 다루었듯이 LTL이나 LTLLR 공식  $\psi$ 도 커지면 모델체킹

- 상태 수에서는 선형
- **공식 크기에서는 지수적**

→ 큰 공식으로 구성된 공정성 조건은 **비효율** 발생

Maude의 TLR 모델체커는 다양한 공정성 조건을 보다 효과적으로 처리하도록 지원

### Branching Time Logics: CTL and CTL\* (분기 시간 논리)

- LTL은 하나의 경로(path)에 대해 속성 표현
- CTL은 계산 트리(computation tree) 전체에 대해 속성 표현

CTL 연산자는:

기호	의미
$\forall \Box \varphi$	모든 경로에서 항상 $\varphi$
$\exists \Box \varphi$	어떤 경로에서는 항상 $\varphi$
$\forall \Diamond \varphi$	모든 경로에서 언젠가 $\varphi$
$\exists \Diamond \varphi$	어떤 경로에서 언젠가 $\varphi$
$\forall \varphi U \psi$	모든 경로에서 $U$
$\exists \varphi U \psi$	어떤 경로에서는 $U$

예: "복권을 산다면, 언젠가 부자가 될 수 있는 가능성"  $\rightarrow$  LTL로는 표현 불가능하지만 CTL로는 가능:

$\forall \Box (\text{hasValidLotteryTicket} \rightarrow \exists \Diamond \text{isMillionaire})$

반면:

- CTL은 공정성(경로 기반 조건)을 표현하기 어렵다
- 따라서 LTL과 CTL은 표현력 면에서 상호 보완적

CTL\*은 LTL + CTL 확장으로,

두 논리의 장점을 모두 사용 가능

### Temporal Logic with Past Operators (과거 시간 연산자)

- 지금까지의 연산자  $\Box$ ,  $\Diamond$ ,  $U$ 는 미래 조건만 표현
- 과거 연산자:

기호	의미
$\triangleleft \varphi$	$\varphi$ 가 초기 상태부터 현재까지 계속 성립
$\Diamond^- \varphi$	과거의 어느 시점에서 $\varphi$ 가 성립
$\varphi S \psi$	과거에 $\psi$ 가 먼저 성립하고 이후 $\varphi$ 가 계속 성립 (Until의 과거 버전)

예: "모든 에어백 작동 전에 crash가 있었다"

$\Box (\text{airbagDepl} \rightarrow \Diamond^- \text{crash})$

그러나 과거 연산자는 필수는 아님  $\rightarrow$  과거 연산자를 쓴 논리는 항상 과거 연산자 없이도 표현 가능 (대신 공식 길이는 지수적으로 늘어날 수 있음)