15. System Requirements

System Spec vs Requirement Spec

- **System Specification**
  - “시스템이 어떻게 동작하는지” 모델로 정의한 것 (시스템의 실제 행동/구조를 설명)
- **Requirement Specification**
  - “시스템이 반드시 만족해야 하는 조건” (올바름을 판단하는 기준)

요구사항 예시

- 상호 배제: “두 프로세스는 동시에 임계구역에 있지 않는다.”
- 진행 보장: “각 프로세스는 결국 임계구역에 들어간다.”
- 순서 보장: “요청한 순서대로 임계구역을 실행한다.”
- 합의: “모든 노드는 결국 같은 리더를 뽑는다.”

핵심 질문

- 시스템 S가 어떤 초기 상태에서 시작하더라도 요구사항 R을 만족하는가?

  (= 잘못된 초기 상태면 만족 못할 수도 있으므로 initial state가 중요함)

이를 판단하려면 필요한 것

1. **정확한 시스템 모델**
2. **정확한 요구사항(temporal logic 등으로 표현)**
3. **검증 방법(model checking)**

Maude를 사용하는 이유

- 시스템을 **정확하게 모델링**할 수 있음
- 요구사항을 **temporal logic**으로 표현 가능
- **model checker가 자동으로 검증**
- 요구사항의 **모호함을 제거** (예: “항상 가능한가” vs “어떤 실행에서는 가능한가”)

동적 시스템의 correctness 개념

- 정적 시스템은 “정답(normal form)”이 있음
- 동적/분산 시스템은 **정답이라는 개념이 없고**, “실행 중 항상 참이어야 하는 속성”이 중요
  - **safety / liveness / invariant** 같은 속성으로 correctness를 정의함

Invariant (불변식)

- 시스템 실행 동안 **항상 참**이어야 하는 조건
- 예: “두 프로세스는 동시에 임계구역에 있을 수 없다.”

15.1 State-based and Action-based Properties

개요

시스템 요구사항(system requirements)은 두 가지 방식으로 표현될 수 있다:

1. **Action-based Properties (행동/이벤트 기반 속성)**

   → 시스템 실행 중 "어떤 행동이 언제/어떻게 발생할 수 있는가"를 조건으로 표현함.

2. **State-based Properties (상태 기반 속성)**

   → 시스템 실행 중 "어떤 상태가 허용되거나 금지되는가"를 조건으로 표현함.

대부분의 요구사항은 두 방식 모두로 표현할 수 있지만, 성격에 따라 더 자연스러운 표현 방식이 존재한다.

Action-based Properties (행동 기반 속성)

- 특징
  - 행동의 **순서**, **발생 조건**, **타이밍**이 중요한 경우 자연스러움.
  - "이 이벤트 직후 반드시 저 이벤트가 가능해야 한다" 같은 형태.
- 대표 예시
  - **예제 15.3 — 미식축구 규칙**
    - 한 팀은 **터치다운 직후에만** 추가점(extra point) 또는 2점 전환(two-point conversion)을 시도할 수 있다.
    - 필드골, 세이프티, 다른 팀의 터치다운 직후에는 2점 전환이 **불가능함**.
  - **기타 자연스러운 행동 기반 요구사항**
    - 한 사람은 두 번 세례(baptism)를 받을 수 없다.
    - 모든 결혼(wedding)은 반드시 약혼(engagement) 후에 이루어져야 한다.
    - 철학자는 무한히 자주 식사를 "시작(start eating)"할 수 있어야 한다.
    - 임계구역(CS)에 들어가는 행동 순서는 `requestAccessToCS` 호출 순서를 따라야 한다.

State-based Properties (상태 기반 속성)

- 특징
  - 시스템의 특정 상태가 **항상 또는 절대 나타나지 않아야 하는가**를 표현할 때 자연스러움.
  - 불변식(invariant) 형태로 자주 사용됨.
- 대표 예시
  - 어떤 상태에서도 두 노드는 동시에 `insideCS` 상태에 있을 수 없다.
  - 인구(population)는 항상 일관성을 유지해야 한다.
    - 예: Bridget이 Tom과 결혼했고 Tom이 Gisele과 결혼한 상태는 불가능.
  - 두 노드가 서로 다른 리더를 가져서는 안 된다.
  - 수신자의 `msgsRcvd`는 결국 발신자의 초기 `msgsToSend`와 같아져야 한다.
  - 스크루지는 은행과 신뢰 연결을 맺기 전에는 은행이 스크루지와 신뢰 상태가 되어서는 안 된다.
  - 인접한 철학자 둘은 절대 동시에 식사해서는 안 된다.

Action + State 조합이 필요한 요구사항

어떤 요구사항은 행동과 상태가 함께 표현되어야 자연스러움.

- 예시
  - `baptized` 상태의 사람은

    → 메카 순례(hajj)를 수행할 수 없고,

    → 다른 세례를 받을 수도 없다.
  - 50세 이상은 출산할 수 없다.

    → 상태(`age > 50`) + 행동(`give birth`) 조합.

행동 기반을 상태 기반으로 억지로 표현하기 어려운 이유

일부 행동 기반 속성은 상태만으로는 표현하기 부자연스럽다.

- 대표 사례 (예제 15.3)
  - **2점 전환(two-point conversion)**
  - **세이프티(safety)**

  둘 다 **2점**을 얻는 플레이이므로, 단순히 상태(= 점수)만 보면 **두 행동을 구분하는 것이 불가능**하다.

  즉, 행동 기반 속성은 **행동 자체를 명시하지 않으면 표현할 수 없는 경우가 존재함**.

본 책의 초점

- 행동 기반 속성도 표현할 수 있지만,
- **상태 기반 요구사항(state-based requirements)** 이 더 일반적이며,
- 모델 검사(model checking)에서도 상태 기반이 주로 사용됨. 따라서 **이 장은 상태 기반 요구사항을 중심으로 논의**함.

15.1.1 Actions/Events

- **핵심 개념**
  - 액션(action) 또는 이벤트(event)는 단순히 "특정 리라이트 규칙을 적용하는 것"으로 정의되지 않는다.
  - 실제 이벤트는 **리라이트 규칙 + (부분) 변수 치환(partial substitution)** 으로 구성된다.
- **왜 단순 규칙 적용과 다른가?**

  예:
  - "A와 B의 결혼" 이벤트
  - "A와 B의 약혼" 이벤트

  이 둘은 **wedding**, **engagement** 규칙을 그대로 적용하는 것이 아니라, 각각 규칙의 변수들을 다음과 같이 **치환해서 적용**하는 것이다:

  $$\{x \mapsto A, x' \mapsto B\}$$

  즉, 이벤트 = 리라이트 규칙에 변수 치환 σ를 적용해 실행되는 규칙 인스턴스
- **요약 문장**

- **이벤트는 리라이트 규칙의 특정 인스턴스를 실행하는 것**이며,
- **그 인스턴스는 규칙의 변수들에 대한 부분 치환(σ)을 통해 만들어진다.**

15.1.2 State Propositions

State Proposition의 정의

- **하나의 단일 상태(single state)**에 대해 참/거짓을 말하는 명제.
- **이전 상태, 이후 상태, 경로(path)에 대한 정보는 포함하지 않음.**
- 즉, 오직 현재 상태만 보고 판단할 수 있어야 한다.
- 형식적으로는 State → Bool 함수처럼 생각할 수 있다.

State Proposition의 예시

다음은 *현재 상태*만 보고 판단할 수 있으므로 State Proposition:

- 현재 상태에서 Seahawks의 점수가 Patriots보다 높다.
- 두 노드가 현재 `insideCS` 상태에 있다.
- 은행의 `respSessions` 속성에 `trustedConnection("Scrooge")`가 포함되어 있다.

State Proposition이 아닌 것

단일 상태뿐 아니라 **이전/미래 상태, 도달 가능성**을 묻는 경우는 State Proposition이 아님.

- "현재 점수가 이전 상태보다 6점 더 많다."
- "현재 상태에서 출발해, 은행이 Scrooge와 연결되는 상태에 도달할 수 있다."

→ 이런 것들은 여러 상태 간 관계를 포함하므로 **state proposition이 아니라 temporal property**에 해당.

정리

- **State Proposition:** 개별 상태의 속성을 판단하는 명제
- **State-based Property:** 여러 상태로 구성된 실행 전체(computation)를 다루는 속성
- State Proposition은 state-based property의 구성 요소로 사용됨.

15.2 Temporal Properties

핵심 개념

- **Temporal Properties(시간 속성)**은 시스템이 **초기 상태에서 시작해 수행할 수 있는 모든 가능한 동작 전체**에 대해 무엇이 참이어야 하는지를 정의한다.
- Temporal properties는 "상태 기반"으로 표현될 수 있으며, 시스템이 시간에 따라 어떻게 진행되더라도 특정 속성이 유지되어야 함을 세밀하게 기술한다.

등장 이유

- 단일 상태만 검사하는 **state proposition**으로는 "시간에 걸쳐 항상 만족되어야 하는 조건"을 표현할 수 없다.
- Temporal properties는 **전체 실행 경로 전체**를 대상으로 한다.

이후 내용과 연결

- Chapter 16에서 LTL, CTL 같은 시간 논리로 formal definition 제공.
- 15.2에서는 그중 가장 중요한 시간 속성 중 하나인 **Invariance(불변식)**을 소개한다.

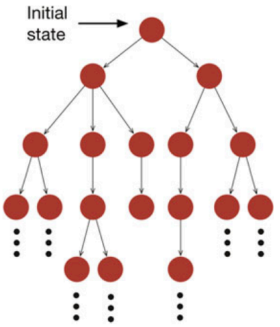
15.2.1 Invariance: "Nothing Bad Will Happen"

불변식(Invariant)의 정의

- 불변식은 "초기 상태에서 도달 가능한 **모든 상태에서** 항상 참이어야 하는 조건".
- 즉, **나쁜 일이 절대 발생하지 않는 것**을 보장하는 속성.
- 안전 속성(safety property)의 전형적인 형태.

→ 공식적으로, p가 invariant ⟺ 초기 상태 t_0에서 도달할 수 있는 모든 상태 t에 대해 **p(t)가 참이어야 한다.**

그림 15.1 설명



- 초기 상태에서 여러 경로로 분기된 모든 상태들(tree of behaviors)이 존재함.
- 불변식이란 이 전체 트리의 모든 상태가 "위반되지 않음"을 의미한다.
- 도중에 단 하나라도 p가 false인 상태가 생기면 더 이상 invariant가 아님.

예제 15.5 : Alternating Bit Protocol

- 수신한 메시지 목록(`msgsRcvd`)은 초기 송신 메시지 목록(`msgsToSend`)의 **접두(prefix)**여야 한다.

  → 메시지가 뒤죽박죽되거나 중간 삽입되는 일이 절대 없어야 한다는 불변식.

예제 15.6 : Mutual Exclusion

- "두 노드가 동시에 insideCS 상태일 수 없다."

  → 상호배제 알고리즘의 핵심 불변식.

예제 15.7 : Dining Philosophers

- "이웃한 철학자 둘은 동시에 eating 상태일 수 없다."

  → 데드락 및 충돌 방지를 위한 불변 조건.

예제 15.8 : Population Consistency

- 단순 조건 "A가 B와 결혼 → B가 A와 결혼"은 메시지가 지연 중인 상태에서는 깨질 수 있으므로 **불변식 아님**.

- 올바른 불변식: “A가 B와 결혼했다면 B가 A와 결혼했거나, B→A 메시지가 전달 중이어야 한다.”

예제 15.9 : Two-Phase Commit

- “모든 updated가 true 또는 false”는 불변식이 아님.
- 올바른 불변식은 다음 3가지 중 하나가 참이어야 함:
  1. 모두 updated = false
  2. 모두 updated = true
  3. commit 메시지가 존재하며, 그 노드가 updated = false
- 메시지가 소모된 뒤에도 데이터베이스가 일관되도록 보장.

15.2.2 Guarantee : “Something Good Must Eventually Happen”

Guarantee(보장) 또는 Liveness(활성) 속성의 개념

- **Invariant** = “나쁜 일이 절대 일어나지 않는다”
- **Guarantee** = “좋은 일이 결국 반드시 일어난다”
- 즉, 보장 속성은 **모든 실행 경로에서 언젠가는 p-상태에 도달해야 함**을 의미한다.

공식 정의

상태 명제 p가 guarantee ⟺ 초기 상태에서 시작하는 **모든 계산 경로(all computations)**에서

언젠가 p 상태로 도달 가능해야 한다.

→ 규칙이 어떤 순서로 적용되든, p는 “최종적으로” 일어나야 한다.

그림 15.2 의미

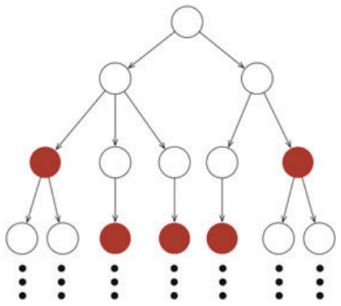


Fig. 15.2 “The state is red” is guaranteed

- 상태 공간이 트리 형태로 펼쳐질 때 어떤 경로를 선택하든 **빨간 상태(p 상태)**로 결국 도달할 수 있음.
- 즉, **p 상태로 가는 길이 무조건 존재한다.**

예제 15.10 : 중앙 서버 알고리즘

- “프로세스 node(4)가 CS 안에 들어간다”는 것은 보장되지 않음.

  → 프로세스들이 무한히 바깥/안쪽을 반복하며 starvation 가능.
- 하지만 **ring-based mutual exclusion algorithm**에서는 보장됨.

예제 15.11 : Dining Philosophers

- “철학자 3이 eating 상태가 된다”는 것은 보장되지 않음.

  → 식사 기회가 영원히 안 올 가능성(starvation)이 존재.

예제 15.12 : Leader Election

- “모두가 가장 큰 값을 가진 노드를 leader로 선택했다”는 것은 Section 13.3의 리더 선출 알고리즘 둘 다에서 **보장됨**.

예제 15.13 : Transport Protocol

- “원하는 문자열이 수신자 msgsRcvd에 저장된다”는 것은 우리의 전송 프로토콜에서 **보장되지 않음**.

  → 메시지 손실, 재전송 실패 등 가능.

15.2.2.1 Fairness

왜 공정성이 필요할까?

시스템 모델은 **비현실적인 극단적 실행**도 허용할 수 있음.

예를 들어:

- 어떤 철학자만 계속 실행되고, 다른 철학자는 평생 기회가 옴

  → 그러면 “철학자 2도 언젠가 식사해야 한다”는 보장을 못함

- 어떤 메시지 뒤에 다른 메시지가 계속 끼어들어서 특정 메시지가 **평생 전달되지 않을 수도 있음**

현실 시스템에서는 이런 일은 **공정하지 않음**

→ 하지만 모델은 이런 극단적 상황을 허용할 수 있음.

그래서 **불공정한 실행을 배제하기 위해 공정성(fairness)**을 가정하는 것.

공정성의 목적

공정성은 **“모든 요소에게 행동할 기회를 공평하게 주자”** 라는 원칙.

그래야:

- 어떤 프로세스도 영원히 무시되면 안 되고
- 어떤 메시지도 영원히 씹히면 안 됨
- 어떤 이벤트도 영원히 가능한데 실행이 안 되면 안 됨

즉, **공정성을 적용하면 liveness 보장 가능해짐.**

공정성의 두 종류

1. **Strong Fairness (강한 공정성, Compassion)**

   어떤 이벤트가 무한히 자주 가능하면, **무한히 자주 실행되어야 한다.**

   쉽게 말하면:

   - “계속 여러 번 기회가 오면 언젠가는 반드시 해라.”
   - 기회가 “자주” 있으면 무시하는 건 불공평.

2. **Weak Fairness (약한 공정성, Justice)**

어떤 이벤트가 계속 가능한 상태라면, 언젠가는 반드시 한 번 실행되어야 한다.

쉽게 말하면:

- 계속 실행할 수 있으면,
- 영원히 미루면 안 됨.

공정성이 필요한 실제 예시

- 예) Dining Philosophers
  - 철학자 3이 먹을 수 있는 기회가 있는데
  - 계속 옆 사람이 먼저 먹어서 *영원히 못 먹는 상황*이 생길 수 있음

    → 이건 불공정
  - 따라서, 공정성: 먹을 수 있는 상황이 계속 오면 결국 한 번은 먹어야 한다.
- 예) 메시지 전달
  - process P가 requestCS 메시지를 보내는데
  - 다른 메시지가 계속 그 앞을 가로채서
  - P의 메시지가 **영원히 읽히지 않을 수도 있음**

  → 현실에서는 이건 불공정

  → 그래서 "무한 overtaking 금지" 같은 공정성 조건이 필요.

공정성이 왜 중요하냐?

공정성을 가정하면 아래 같은 liveness가 가능해짐:

- 철학자는 **언젠가는 먹는다**
- 모든 노드는 **언젠가는 리더를 뽑는다**
- 메시지는 **언젠가는 전달된다**

공정성이 없으면 모델은 너무 비현실적 실행을 허용해서 liveness 속성이 깨져버린다.

15.2.3 Reachability

개념

- **Reachability(도달 가능성)** : "어떤 상태가 일어날 수 있는가?"를 묻는 속성
- 상태 명제 p가 **reachable** 이라는 것은:
  - 초기 상태에서 시작해
  - 어떤 실행 경로(some run)에서는
  - **p 상태에 도달할 수 있다**
- 즉, **나쁜 상태가 '일어날 수도 있음'을 확인하는 속성.**

Guarantee와의 차이

- **Guarantee:** 모든 실행(all runs)에서 반드시 p 상태에 도달해야 한다.
- **Reachability:** 어떤 실행(some run)에서 p 상태에 도달할 가능성만 있으면 된다.

속성	의미
Guarantee	좋은 일이 **반드시** 일어난다
Reachability	나쁜 일이 **일어날 수도 있다**

Invariant와의 관계 (쌍대 관계)

- p가 **불변(invariant)** ↔ **not p(¬p)가 reachable이 아니다.**
- 즉,
  - 불변 = 나쁜 상태는 절대 도달 불가
  - 도달 가능성 = 나쁜 상태가 도달 가능

활용

- Reachability는 시스템에서 **위험하거나 잘못된 상태가 발생할 가능성**을 분석하는 데 사용된다.

15.2.4 Response:“A Request Will Always be Answered”

개념

- **Response(응답) 속성**은 “어떤 상태 p_1이 발생하면, 반드시 p_2가 뒤따라야 한다.”라는 규칙을 의미한다.
- 반응형 시스템(reactive system)에서는 필수적인 속성.

공식 정의

상태 명제 쌍 (p_1, p_2)가 response property라는 것은:

- 초기 상태에서 가능한 **모든 실행(all runs)**에 대해
- p_1 상태가 나타나면
- **언젠가는 반드시** p_2 상태가 뒤따라와야 한다 (즉시일 필요는 없음)

예시

- 요청(request) → 반드시 그 후 acknowledgment(응답)가 와야 함
- 충돌 감지 → 반드시 에어백이 나와야 함
- 프로세스가 CS에 들어가고자 함 → 언젠가는 실제로 CS에 들어가야 함

즉, **어떤 이벤트가 → 반드시 그에 대한 반응이 온다**라는 시스템 동작 규칙.

15.2.5 Stability

개념

- **Stability(안정성) :** 어떤 상태 명제 p가 한 번 참이 되면, 그 이후의 모든 실행에서도 절대 거짓이 되지 않음.
- 즉, **한 번 만족되면 영원히 유지되는 성질**임.

왜 중요한가?

- 일부 프로토콜은 목표 상태에 도달한 후에도 계속 동작함.
- 이때 안정성이 없다면:

    - 이미 달성한 결과가
    - 이후 실행 단계에서 **깨져버릴** 수 있음.
- 안정성은 **"이미 달성한 좋은 상태를 보호하는 성질"**.

예시

- **Alternating Bit Protocol**

  "수신자가 원하는 문자열 s를 받았다"는 상태는 이후 메시지들이 이 결과를 망가뜨리면 안 됨 → 안정적이어야 한다.

- **Leader Election**

  "모든 노드가 최고값 노드를 리더로 선택했다"

  → 새로운 라운드가 돌더라도 리더가 바뀌면 안 됨

  → 안정적이어야 한다.

15.2.6 Other Requirements

Until (p_1 until p_2)

- p_2 상태가 나올 때까지, 모든 상태는 p_1을 만족해야 한다.
- 즉, **p_2가 나타나기 전까지 p_1이 계속 유지되어야 함.**

예시

- "메시지가 존재한다" until "모든 노드가 리더를 선택했다"
- "메시지가 존재한다" until "모든 DB가 동일한 updated 값을 가진다 (2PC)"

Weak Until (약한 Until)

- p_2가 **반드시** 일어날 필요는 없음.
- 대신 p_2가 일어나지 않으면 **p_1이 계속 참**이어야 함.
- 예: Paxos
    - "합의를 시도함" weak-until "합의가 일어남"
    - 합의가 실패해도 계속 시도 상태에 있으므로 조건 충족.

Strong Until (강한 Until)

- p_2가 **반드시 최종적으로 일어나야 한다.**

Termination

- 시스템이 무한히 실행만 하는 상황을 허용해서는 안 됨.
- 즉, **언젠가는 종료 가능한 상태가 존재해야 한다.**

Correct Final States

- 모든 최종 상태는 특정 조건을 만족해야 한다.
- 예

- 2PC → **모든 DB의 updated 값이 동일해야 함**
- Leader Election → **모든 최종 상태에서 동일한 리더**

No Deadlocks

- 더 이상 진행할 수 없는(deadlocked) 최종 상태가 존재하면 안 된다.
- 예:

  Dining Philosophers

  - 철학자들이 영원히 먹지/생각하지 못하는 상태 불가
  - rewritable하지 않은 상태(막힌 상태)는 데드락이므로 금지

15.3 Analyzing Invariants

이 파트가 말하고 싶은 핵심

- Maude의 search 기능을 이용해, 어떤 상태 명제가 invariant인지 자동으로 확인할 수 있다. (단, *하나의* 초기 상태에 대해서만)

기본 아이디어

- Invariant인지 확인하는 방법은 단 하나:
  - 초기 상태에서 출발해 p를 깨뜨리는(¬p) 상태가 reachable인지 검색(Search)
- reachable하면 → invariant가 아니다
- reachable하지 않으면 → invariant일 가능성이 있다
- 즉 **Maude search = "나쁜 상태 reachable인지 검사"** 도구.

실제 예시들 (책에서 반복적으로 했던 것들)

- Transport protocol:
  - 수신자 msgsRcvd가 msgsToSend의 prefix인지 확인하려고, prefix가 아닌 상태가 reachable인지 탐색
- Mutual exclusion:
  - 두 프로세스가 동시에 insideCS인지 찾음
- NSPK 프로토콜:
  - bank가 "trusted" connection을 갖는 나쁜 상태가 reachable인지 확인

→ 즉, 우리는 항상 "나쁜 상태"를 찾는 search를 사용함.

Maude search의 결과 유형

1. 나쁜 상태가 발견됨 → 그 속성은 invariant가 아님 → search 종료
2. 나쁜 상태가 없고, reachable 상태 집합이 유한함 → search는 "No solution"으로 끝남 → **invariant 맞음**
3. reachable 상태가 무한(infinite) → search가 *절대로 끝나지 않음* → **invariant인지 판단할 수 없음** (단지 "못 찾았을 뿐, 나쁜 상태가 미래에 있을 수도 있음")
   - 예: NSPK에서 30분 동안 search했는데 안 나오면 → 40분 후에 발견될 수도 있음

문제: 왜 infinite state space가 생기는가?

아래 경우에 **상태 공간이 무한**해져 search가 끝나지 않음:

- 값이 제한 없이 계속 증가할 수 있음 (football score, dining philosopher의 eats 카운터 등)
- 메시지 수가 무한히 증가할 수 있음
- 객체 수가 무한히 생성될 수 있음 (예: person을 birth rule로 무한 생성)

이런 경우 search로 invariant 분석 불가능.

Maude search의 두 가지 중요한 한계

1. Search로는 **infinite state space에서 invariance를 증명할 수 없다**
2. Search는 **단 하나의(initial state t_0)**에 대해서만 invariant를 증명할 수 있다

- 여러 초기 상태 집합에 대해 invariant인지 증명 불가.
- 예:
  - 철학자 5명에 대해서는 증명 못 함
  - 10명, 100명, 임의 개수에 대해서는 더더욱 불가
  - football 점수가 11점 이상인 *모든* 초기 상태에 대해 invariant인지 증명 불가

해결책: Inductive proof(귀납적 증명)

Search로 증명할 수 없다면, "손으로(by hand)" 다음 두 조건을 보이면 된다:

1. **모든 초기 상태 s_0가 p를 만족한다.**
2. **모든 rewrite rule r에 대해,**
   p-state가 r로 바뀌어도 p-state가 유지된다.

이 두 가지를 만족하면: p는 초기 상태 집합 I 전체에 대한 invariant이다.

예제 15.14 (귀납적 증명 예시)

증명하고 싶은 것: "점수 합(m+n)이 항상 10보다 크다" (최종 상태 포함 모든 reachable 상태에서)

귀납적 증명:

- 초기 상태들은 모두 m+n > 10
- rewrite rule(예: touchdown-home)은 점수를 (m+6, n)으로 바꿈 → m+6+n > m+n > 10
- 따라서 모든 rewrite rule이 p를 유지함 → **p는 모든 initial state에 대한 invariant**

강화(Strengthening)

- 가끔 p만으로는 inductive하지 않음
- 즉, p(t) ∧ (t→t') → p(t') 를 증명할 수 없음
- 이때는, p를 강화한 p′를 사용해 invariant를 증명한다. (p′가 invariant이고, p′ → p이면 p도 invariant)

최종 요약

- 15.3은 Maude search로 invariant를 분석하는 방법과 search의 한계를 설명하고, 그 한계를 넘기 위해 귀납적 증명(inductive invariant)을 사용하는 방법을 소개하는 파트이다.