분산시스템의 정형명세 및 모델검증

배경민

2022년 2월 8일 (한국 소프트웨어공학 학술대회)

포항공과대학교 컴퓨터공학과

소프트웨어 검증이란?

- 소프트웨어가 원하는 대로 동작하는가?
  - 입·출력, 기능 명세, 안전, 신뢰, 보안, …
- 소프트웨어 검증의 목적
  - 소프트웨어의 오류를 발견하거나 오류가 없음을 증명
- 소프트웨어 검증의 비용?
  - 인력, 시간, 도구, …

소프트웨어 오류 사례

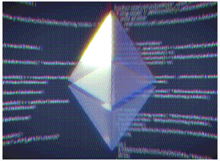
North America blackout, 2003
(> 10 deaths)

Toyota's ETCS bugs, 2009–11
(> 80 deaths)

OpenSSL Heartbleed Bug, 2014
($500 million loss)

Tesla/Uber Autopilot Crashes,
2016–19 (5 deaths)

Boeing 737 MAX crashs, 2018–19
(346 deaths)



Ethereum Blockchain Bugs, 2018
(> $600 million loss)

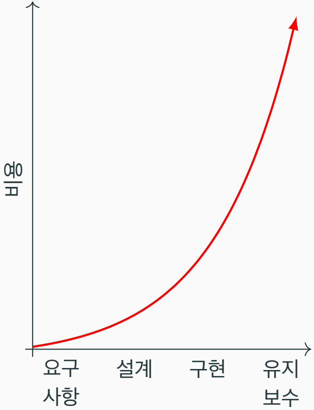
정형기법의 개념

소프트웨어 오류의 원인

- 구현 오류
  - (주로 개발자의 실수로) 코드 상에 존재하는 버그
- 설계 오류
  - 설계/알고리즘 수준의 오류
- 소프트웨어 취약점
  - 예상치 못한 방법(입력)으로 소프트웨어를 사용하여 발생

코드 수준 소프트웨어 분석

- 코드를 분석하여 구현/설계 오류 및 취약점 분석
    - 실행 가능한 산출물이 존재하여 직관적
- 다양한 종류의 코드 분석 기술 존재
    - systematic test, fuzzing, static analysis, ⋯
- 한계점
    - 구현 후에만 적용 가능
    - 실행 환경에 의존적 (예: 분산 시스템의 코드 수준 분석)

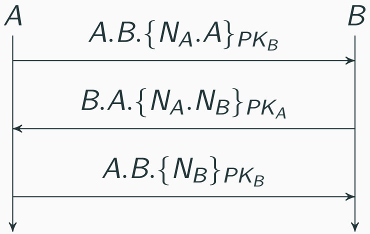


오류 수정 비용

설계 오류의 예: Needham-Schroeder Public-Key (NSPK) 인증 프로토콜 (1)

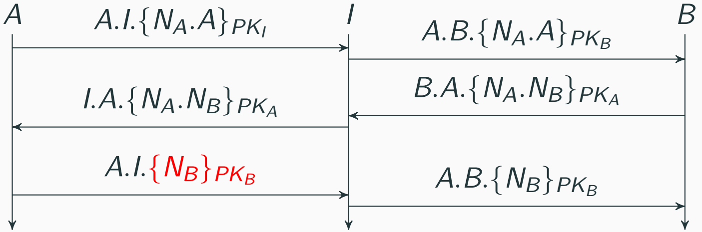
- 공개키 암호에 기반한 상호 인증 프로토콜
  - 각 노드 A는 공개키 PK_A, 비밀키 $PrvK_A$, 고유 정보 N_A를 가짐
  - 공개키로 암호화된 메세지 $\{msg\}_{PK_A}$는 비밀키 $PrvK_A$로 해독 가능
  - 두 노드는 각각 고유 정보를 본인의 비밀키로 암호화하여 상호교환하여 상호 인증

- NSPK 인증 프로토콜 (1978)



설계 오류의 예: Needham–Schroeder 프로토콜 (2)

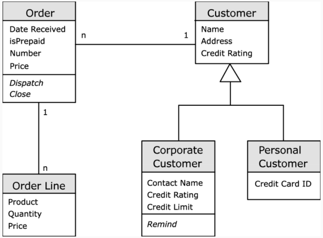
- Man-in-the-middle attack 취약점 (1995년에 모델검증으로 발견됨)

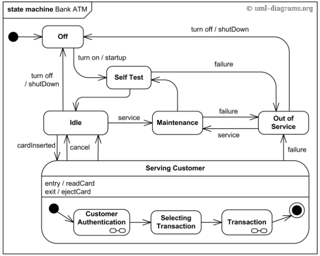


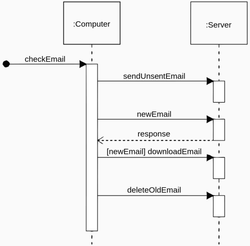
- 코드 수준 소프트웨어 분석의 어려움
  - 네트워크 환경 및 침입자의 다양한 행위 재현 어려움
  - 취약점/오류가 코드 수준이 원인이 아님

설계 수준 소프트웨어 분석

- 다양한 소프트웨어 구조 및 행위 설계 방법 존재







- 보통 소프트웨어 개발 단계에서 문서화 및 디자인 리뷰 과정에 사용됨
- 코드와 같이 설계 수준에서 실행 및 자동 분석이 가능한가?

필요 기술: 소프트웨어 설계에 대한 수학적 방법론

- 컴퓨터 시스템의 수학적 모델
- 이러한 모델을 분석할 수 있는 수학적 이론
- 이러한 이론에 근거한 분석/증명 기법

⇒ 공학에서의 일반적인 접근방법

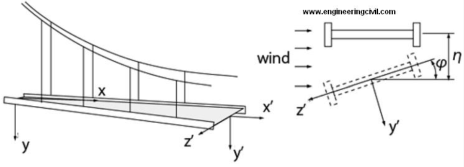


Figure 1. Theoretical model of suspension bridge.

Hirai's research on lateral torsional buckling of suspension bridge starts at the Equation 1.

$$EI\frac{d^4\eta}{dx^4} - 2H_\omega\frac{d^4\eta}{dx^4} - 2h_1\frac{d^2y}{dx^2} + \frac{d^2}{dx^2}(\mathrm{M}\varphi) - (S + (C_d))pb\varphi = 0$$

$$\mathrm{M}\frac{d^2\eta}{dx^2} - EC_\omega\frac{d^4\eta}{dx^4} - \left(GK + \frac{H_\omega b^2}{2}\right)\frac{d^2\eta}{dx^2} - bh_2\frac{d^2y}{dx^2} - S_t pb\varphi^2 = 0 \quad 1$$

Where, η and φ mean main girder's buckling displacement in vertical and torsional

정형기법의 목적

정형기법 $=$ 컴퓨터 시스템에 대한 수학적 방법론

정형기법(Formal Methods)이란?

수학적 방법론에 기반한 소프트웨어/하드웨어 개발 방법

- 정형명세(Formal Specification): 컴퓨터 시스템 설계에 대한 엄밀한 모델링
- 정형명세 기반 구현: 자동(코드 생성) 혹은 수동(설계 기반 구현)
- 정형분석(Formal Analysis): 정형명세의 성질 분석, 코드와 설계의 일치성 검증 등

정형기법 분야의 범위

	교량 건설	소프트웨어
기초	Calculus	Logic
이론	Mechanics	Computational M
공학	Structural Engineering	Software Enginee
개발	Bridge Construction	Software Develop

정형
기법

정형기법 적용 사례 (1)

- Model-based development
  - Simulink, AADL, Modelica 등 모델링 도구로 소프트웨어 개발, 분석 및 코드 자동 생성
  - 자동차, 항공기, 철도, 선박 등의 안전필수 소프트웨어 개발에 널리 사용
- 프로토콜 및 분산시스템 검증
  - TLS를 포함하여 다양한 종류의 보안 프로토콜 검증
  - Amazon Web Service, Microsoft Azure 등 클라우드에서 사용되는 분산알고리즘 검증
- 하드웨어 설계
  - Eectronic design automation (EDA)에서 주요 분석 기술로 널리 사용됨
  - 오류의 직접적인 파급효과가 크고 정형기법 적용이 소프트웨어 분야보다 용이함

정형기법 적용 사례 (2)

- 금융 관련 소프트웨어
  - 온라인 뱅킹, 모바일 결재 등을 위한 알고리즘 및 응용 프로그램 검증
  - 블록체인 스마트 컨트랙트의 오류 분석 및 검증된 실행 엔진 개발
- 코드 분석
  - 프로그래밍 언어의 수학적 의미를 기반으로 (사전에 정의된) 코드 오류 분석
  - Microsoft, Facebook, Google 등을 포함한 많은 회사에서 개발 프로세스에 포함
- 운영체제 및 컴파일러
  - seL4, CertiKOS 등 검증된 운영체제 커널 및 CompCert 등 검증된 컴파일러 개발
  - 자동차, 항공, 의료, 국방 분야 등에서 사용되고 있음

소프트웨어 안전/보안 국제 산업 표준

- IEEE 61508 전기/전자/임베디드 시스템 안전성 표준
- ISO 26262 자동차 기능 안전성 국제 표준
- DO-178C 항공 소프트웨어 안전성 인증 표준
- IEC 62304 의료기기 소프트웨어 프로세스 표준
- ISO/IEC 15408 정보보안 인증 평가 기준 표준
- ISO/IEC 29128 보안 프로토콜의 검증
- ...

산업 표준에 정의된 보안성 등급

- ISO/IEC 15408 Evaluation Assurance Levels

	보안 정책	기능 명세	구조 설계	상세 설계	구현
EAL 4	informal	informal	informal	informal	informal
EAL 5	formal	semi-formal	semi-formal	informal	informal
EAL 6	formal	semi-formal	semi-formal	semi-formal	informal
EAL 7	formal	formal	formal	semi-formal	informal
Verified	formal	formal	formal	formal	formal

⇒ 높은 안전성 혹은 보안성 등급 인증 시 엄밀한 검증 기술 적용 요구

정형기법 기술의 중요성 증가

- 과거 – 현재
  - 주로 안전 필수 시스템의 경우
  - 항공기, 우주선, 열차제어, 자동차, 의료기기, …
- 현재 – 가까운 미래
  - 소프트웨어의 중요성 증대: IoT, 인공지능, 무인자동차/항공기, …
  - 소프트웨어 취약점을 악용하는 보안 사고 증가
- 가까운 미래
  - 검증된 소프트웨어 vs. 검증되지 않은 소프트웨어
  - 검증된 운영체제, 검증된 컴파일러, 검증된 애플리케이션, …

(정형기법 기반) 소프트웨어 검증 기법의 종류

- 자동 테스팅
  - 가장 널리 사용
  - “어려운 오류”를 찾기 어려움
  - “오류 없음” 확인 불가
- 정적 분석
  - 적은 비용
  - 제한된 범위의 검증
  - “가짜 오류”
- 모델검증
  - “오류 없음” 확인 가능
  - “어려운 오류” 찾기 수월
  - 큰 규모에 적용 어려움
- 정리 증명
  - 가장 높은 수준의 보장
  - 매우 큰 비용
  - 소규모 소프트웨어에 적용

모델검증 기법 소개

분산/병행 시스템

- 안전필수시스템(Safety-critical systems)은 많은 경우 분산/병행 시스템






병행 시스템 예제

```
// Inc
while (true) {
  if (x < 200)
    x = x + 1;
}
```

```
// Dec
while (true) {
  if (x > 0)
    x = x - 1;
}
```

```
// Reset
while (true) {
  if (x == 200)
    x = 0;
}
```

- 질문: 변수 x는 항상 0과 200사이의 값을 가지는가?

$\Rightarrow$ 아니요! 이유는?

병행 시스템 검증의 어려움

```
// Inc
while (true) {
  if (x < 200)
    x = x + 1;
}
```

```
// Dec
while (true) {
  if (x > 0)
    x = x - 1;
}
```

```
// Reset
while (true) {
  if (x == 200)
    x = 0;
}
```

- 테스트의 어려움: 가능한 모든 경우의 수를 고려하는 테스트?
- 정리증명의 어려움: 가능한 모든 경우의 수에 대한 증명 작성 필요!

모델검증의 태동 (1981)

Edmund Clarke

E. Allen Emerson

"*The task of proof construction is in general quite tedious and a good deal of ingenuity may be required to organize the proof in a manageable fashion. We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic.*"

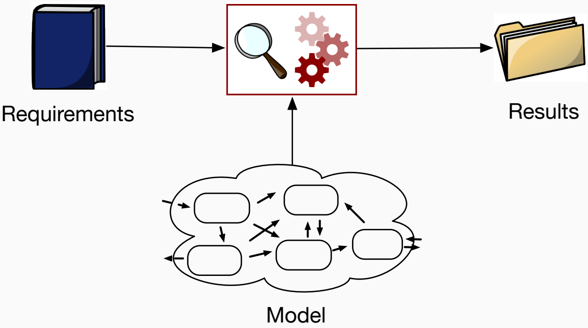
- 2007년도 ACM Turing Award : Edmund Clarke, E. Allen Emerson, Joseph Sifakis

---

Clarke, E. M., & Emerson, E. A. (1981, May). Design and synthesis of synchronization skeletons using branching time temporal logic. In Workshop on logic of programs (pp. 52-71). Springer.

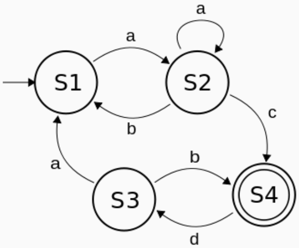
모델검증 (Model Checking)

- 시스템의 오류를 자동으로 찾는 기술
  - 소프트웨어/하드웨어 디자인,
    프로토콜 디자인, 소스 코드, …
  - 다양한 모델검증 도구 존재
- 특징
  - 시스템의 모든 가능한 상태를
    확인하여 "오류 없음" 증명 가능
  - 자동적으로 복잡한 성질을 검증 가능



모델검증의 원리 (1): 시스템 = 수학적 모델

- (병행) 시스템의 실행은 상태기계(State machine)로 표현될 수 있다



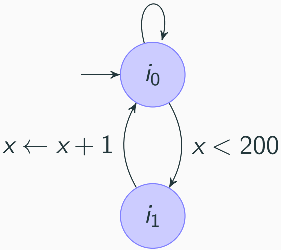
- 튜링머신: 상태기계의 일종

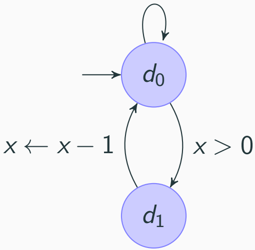
예제: 시스템 = 수학적 모델

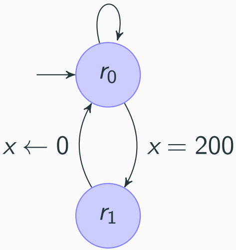
```
// Inc
while (true) {
  if (x < 200)
    x = x + 1;
}
```

```
// Dec
while (true) {
  if (x > 0)
    x = x - 1;
}
```

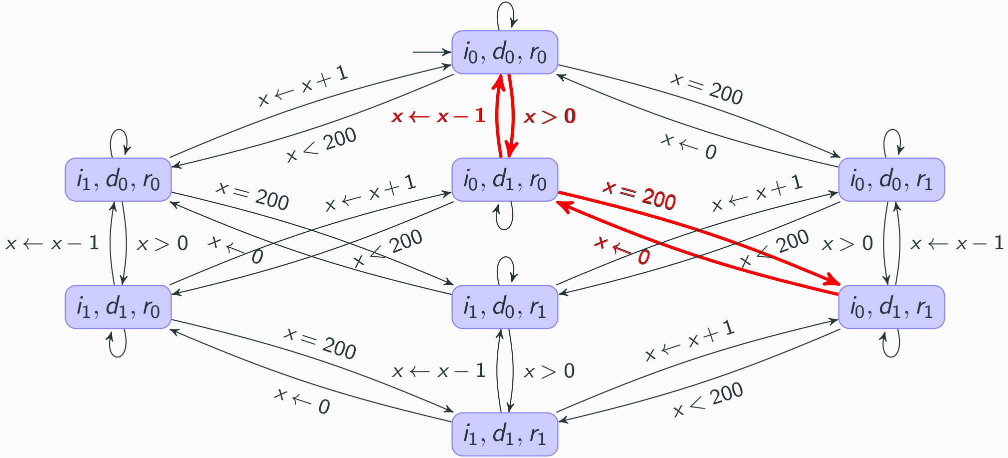
```
// Reset
while (true) {
  if (x == 200)
    x = 0;
}
```







- 질문: 변수 x는 항상 0과 200사이의 값을 가지는가?



- 질문: 변수 x는 항상 0과 200사이의 값을 가지는가? 아니오!

모델검증의 원리 (2): 성질 검증 = 상태공간 탐색

- 검증문제: (분산/병행) 시스템이 요구사항을 만족하는가?

$\Downarrow$

- 모델검증: 수학적 모델이 수학적 성질을 만족하는가?

$\Downarrow$

- 알고리즘: 상태공간 그래프가 특정한 부분 그래프를 포함하는가?

모델검증 단계

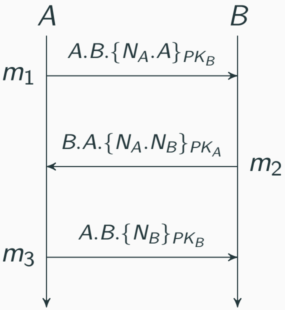
1. 시스템 명세 (system specification)
   - 모델링 언어 (Promela, Simulink, Verilog, ⋯)
   - 프로그래밍 언어 (C, Java, Haskell, ⋯)
2. 검증 성질 명세 (property specification)
   - functional correctness, safety, liveness, fault tolerance, ⋯
3. 모델 검증 도구
   - SPIN, CBMC, NuSMV, ⋯

모델검증 예제: NSPK 프로토콜 시스템 명세

- 객체와 메세지의 집합으로 네트워크 환경 모델링
  - 객체: 노드 (A, B 및 침입자)
  - 메세지: 노드 간에 주고 받는 통신 내용

- Initiator A 모델링
  1. m_1 보냄 (N_A 저장)
  2. m_2 받음 (복호화, N_A 확인)
  3. m_3 보냄 (B 인증)

- Responder B 모델링
  1. m_1 받음 (복호화)
  2. m_2 보냄 (N_B 저장)
  3. m_3 받음 (N_A 확인, A 인증)



모델 검증 예제: NSPK 프로토콜 성질 명세

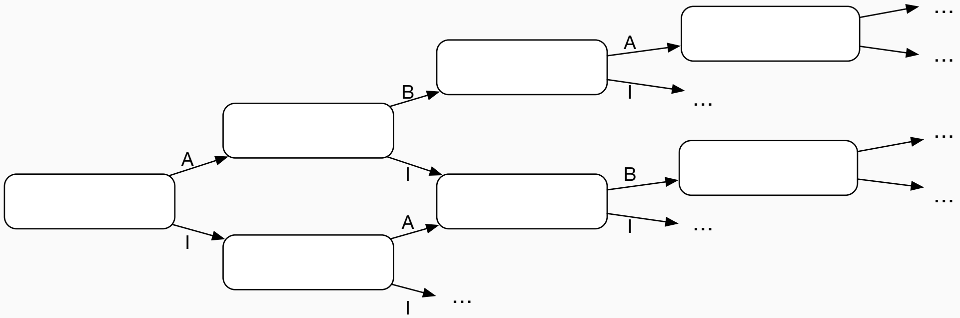
- 침입자 모델링: Dolev-Yao Model
  - 네트워크 상 모든 메세지의 도청 및 가로채기
  - 본인의 공개키로 암호화된 메세지 복호화
  - 지금까지 관측한 고유 정보로 메세지 생성
  - 지금까지 관측한 (암호화된) 메세지 송신
- 검증 성질
  - 침입자가 다른 노드를 대신해서 인증되지 않음

Danny Dolev

Andrew Yao

모델 검증 예제: 모델 검증 알고리즘

- 가능한 모든 interleaving을 검사하여 얻어지는 그래프



- 시스템 모델이 주어진 성질을 만족하는지 검사
  - 예: 침입자가 다른 노드를 대신해서 인증하는 상태에 도달 가능한가?

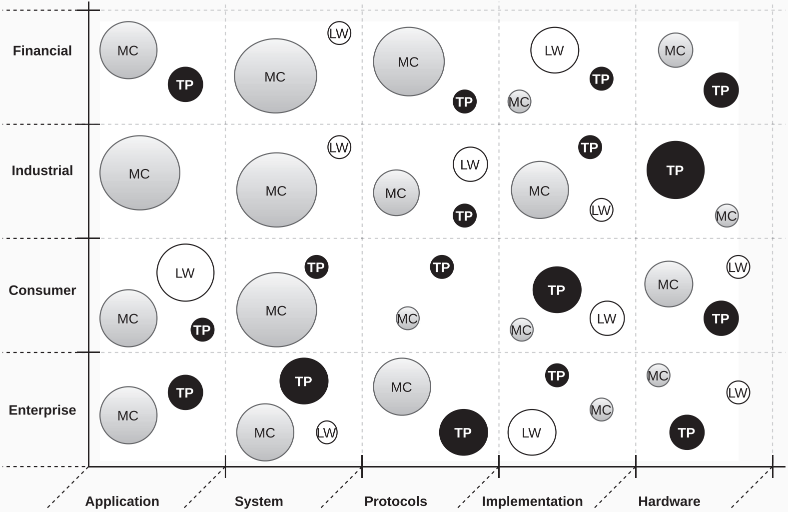
모델 검증 기법의 장점

- 자동화
  - 시스템 및 성질 명세 후 자동 실행
- 복잡한 성질 검사 가능
  - 동시성 오류, 실시간 요구조건 등
- 오류 재현 용이
  - 오류 발견 시 반례 생성
- 무결성 증명 가능
  - 시스템/성질 명세 수준에서 “오류 없음” 증명

산업적 파급효과 (Revisited)

- Model-based development
  - Simulink, AADL, Modelica 등 모델링 도구로 소프트웨어 개발, 분석 및 코드 자동 생성
  - 자동차, 항공기, 철도, 선박 등의 안전필수 소프트웨어 개발에 널리 사용
- 하드웨어 설계
  - Electronic design automation (EDA)에서 주요 분석 기술로 널리 사용됨
  - 오류의 직접적인 파급효과가 크고 정형기법 적용이 소프트웨어 분야보다 용이함
- 프로토콜 및 분산시스템
  - TLS를 포함하여 다양한 종류의 보안 프로토콜 검증
  - Amazon Web Service, Microsoft Azure 등 클라우드에서 사용되는 분산알고리즘 검증
- ...

산업적 파급효과: 보안 분야에서의 모델검증



MC: 모델검증

TP: 정리증명

LW: 정적분석 + 테스팅

T. Kulik, et al. A Survey of Practical Formal Methods for Security. Form. Asp. Comput. 34.1 (2022): 1-39.

Maude 소개

C 언어를 통한 정형명세?

- 장점
  - 개발 코드에 대한 직접적인 적용이 가능
  - “설계”없이 개발되는 대다수의 소프트웨어에 적용 가능
- 단점
  - 각각의 언어에 대해 별도의 모델검증 도구 개발 필요
  - 해당 언어에서 지원하지 않는 특성 표현 불가 (예: C 언어로 NSPK 설계 수준 명세?)

C 언어를 통한 정형명세?

- 치명적인 단점: 모든 경우에 대한 엄밀한 분석이 매우 어려움
  - 메모리, 포인터, 표준 라이브러리, 의미가 정의되지 않은 코드, ···
- C 언어에서 의미가 정의되지 않은 코드의 예 (컴파일러마다 다른 실행결과)

```
int main(void){
    int x = 0;
    return (x = 1) + (x = 2);
}
```

- 분산시스템의 정형명세 언어로는 부적합

분산시스템의 정형명세에 적합한 특성

- 수학적 의미구조의 정의 용이

  $\Rightarrow$ 단순한 논리적 규칙으로 의미가 정의가 되어야 함

- (대부분의) 분산시스템에 대한 모델링이 가능

  $\Rightarrow$ 다양한 동시성 개념의 의미가 정의 가능해야 함

- 높은 모델검증의 성능 달성

  $\Rightarrow$ 효과적인 모델검증 알고리즘 및 방법론이 적용 가능해야 함

Rewriting Logic 및 Maude

José Meseguer

Joseph Goguen

- Rewriting logic
  - term rewriting이라는 단순한 논리적 규칙을 통하여 동시성에 대한 수학적 모델 정의
- Maude (`http://maude.cs.illinois.edu`)
  - rewriting logic 기반의 시스템 명세 언어 및 계산 도구

Maude의 장점

- 높은 표현력을 가진 시스템 명세 언어
- 간단하지만 직관적인 수학적인 의미
- 모델의 직접적인 실행 및 시뮬레이션 가능
- 다양한 모델링 언어의 의미 정의 가능

  동시성 모델: actors, process calculi, Petri nets, ⋯

  프로그래밍 언어: C, Java, JavaScript, Scheme, Python, ⋯

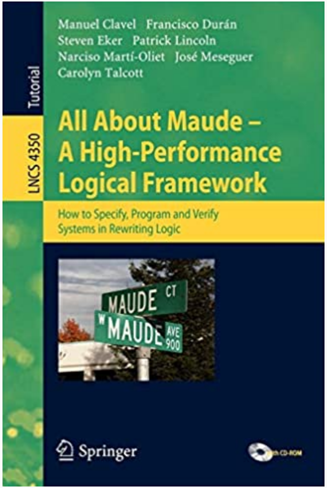
  디자인 언어: Verilog, ABEL, AADL, Ptolemy II, Orc, ⋯

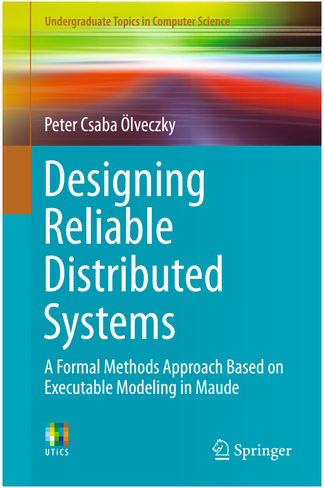
Rewriting Logic 및 Maude 사용 사례

- 분산시스템, 프로토콜, 및 알고리즘
  - IETF multicast protocols, wireless sensor network algorithms, ⋯
  - Cloud transaction systems: Apache Cassandra, Google's Megastore, ZooKeeper, ⋯
- 프로그래밍 언어
  - C, Java, JVM, Scheme, Ethereum smart contracts, ⋯
  - Verilog, NASA Plan Execution Language (Plexil), AADL, Ptolemy II, ⋯
- 보안
  - Internet Explorer에서 address/status bar spoof attacks 발견
  - 보안 프로토콜 검증 도구: Maude-NPA, Tamarin, ⋯
- 기타: neuroscience, biological reactions (e.g., Pathway Logic at SRI), ⋯

Maude 참고자료

- 메뉴얼: `http://maude.cs.illinois.edu/w/index.php/Maude_Manual_and_Examples`





Maude 기반 정형 명세

- 시스템 상태: 대수적 자료 구조
  - recursive data types and functions
  - lists, sets, multi-sets, $\cdots$
- 시스템의 상태 변화
  - rewrite rule $t \longrightarrow t'$
  - 패턴 t에서 패턴 t' 으로의 변화

Maude에서의 대수적 자료구조 명세

대수적 자료구조

- 데이터 값 및 연산자

Elements	Functions
$\mathbb{N}$	$+, <, *, \ldots$
$\mathbb{Z}$	$+, -, \ldots$
lists of numbers	add, first, concat, remove element, sort, ...
stacks	pop, push, top, empty?, ...
multisets	add, remove, in?, ...
strings	substring, concat, ...
binary trees	size, inorder, preorder, isSearchTree, ...
graphs	hasCycle?, newEdge, ...
...	...

- 데이터 값 및 연산자는 특정한 대수적 성질을 만족함

대수적 자료구조 예제: 자연수

- Operators
  - constant 0, unary function s, and binary function $+$
  - $0,\ s(0),\ s(s(0)),\ \cdots$
- Axioms
  - $\forall x\,(x + 0 = x)$
  - $\forall x \forall y\,(x + s(x) = s(x + y))$
- Computation
  - $s(0) + s(0) \longrightarrow s(s(0) + 0) \longrightarrow s(s(0))$

Maude Example: Natural Numbers

```
fmod PEANO-NAT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(0, M) = M .
  eq plus(s(N), M) = s(plus(N, M)) .
endfm
```

- Declarations are separated by periods (with white spaces before)
- Sorts (i.e., types) are declared with the keywords sort or sorts
- Variables are declared with the keywords var or vars
- Equations are declared with the keyword eq

Running Maude

```
$ maude
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
            Maude 3.2.2 built: Dec 22 2022 16:26:25
             Copyright 1997-2022 SRI International
                   Wed Feb  8 03:14:40 2023
Maude>
```

The `reduce` Command

```
Maude> reduce plus(s(s(0)), s(s(s(0)))) .
reduce in PEANO-NAT : plus(s(s(0)), s(s(s(0)))) .
rewrites: 3 in 0ms cpu (0ms real) (1500000 rewrites/second)
result Nat: s(s(s(s(s(0)))))
```

- The reduce (simply red) command performs equational rewriting
- All equations are applied from left to right

Module Importation

```
fmod TRUTH-VALUES is
  sort Truth .
  ops tt ff : -> Truth .
endfm
```

```
fmod PEANO-NAT-LESS is
  protecting PEANO-NAT .
  protecting TRUTH-VALUES .

  vars M N : Nat .

  op less : Nat Nat -> Truth .
  eq less(0, s(M)) = tt .
  eq less(M, 0) = ff .
  eq less(s(M), s(N)) = less(M, N) .
endfm
```

- Modules are imported with protecting, extending, or including.
- Variable declarations are not imported.

Mix-Fix Notation

```
fmod PEANO-NAT-MIX is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33] .
  vars N M : Nat .
  eq 0 + M = M .
  eq s(N) + M = s(N + M) .
endfm
```

```
fmod PEANO-NAT-LESS-MIX is
  protecting PEANO-NAT .
  protecting TRUTH-VALUES .
  op _<_ : Nat Nat -> Truth [prec 37] .
  vars M N : Nat .
  eq 0 < s(M) = tt .
  eq M < 0 = ff .
  eq s(M) < s(N) = M < N .
endfm
```

- Arguments of operators can occur anywhere (denoted by _)
- Operator priorities given by precedence attribute

Conditional Equations

```
fmod PEANO-NAT-MAX is
  protecting PEANO-NAT-LESS-MIX .

  op max : Nat Nat -> Nat .

  vars M N : Nat .

  ceq max(M, N) = N if M < N = tt .
  ceq max(M, N) = M if M < N = ff .
endfm
```

- Conditional equations are declared with the keyword ceq.
  (see the manual for more information).

Built-In Modules

- `BOOL`
  - sort `Bool` with constants `true` and `false`
  - Boolean operators, including `_and_`, `_or_`, `not_`
  - Logical operators, including `_==_` and `if_then_else_fi`
- `NAT`
  - sort `Nat` with symbols `0` and `s_`
  - numbers 0, 1, 2, ⋯, denote terms $s\,0$, $s\,s\,0$, $s\,s\,s\,0$, ⋯
  - usual natural number operations
- `INT`, `RAT`, `FLOAT`, `STRING`, ⋯

Constructors vs. Defined Operators

- Constructors
  - define an abstract data type itself
  - e.g., 0, *succ*, *none*, `__`, ⋯
- Defined operators
  - define operations of abstract data types
  - e.g., `_+_`, `_*_`, `_in_`, ⋯

Example: Binary Trees (1)

```
fmod TREE is
  protecting INT .

  sort Tree .
  op ___ : Tree Int Tree -> Tree [ctor] .
  op empty : -> Tree [ctor] .
endfm
```

- ctor attributes denote constructors

Example: Binary Trees (2)

```
fmod MIRROR is
  protecting TREE .

  op mirror : Tree -> Tree .

  vars L R : Tree .
  var I : Int .

  eq mirror(L I R) = mirror(R) I mirror(L) .
  eq mirror(empty) = empty .
endfm
```

Example: Binary Trees (3)

```
fmod SEARCH is
  protecting TREE .

  op search : Int Tree -> Bool .

  vars I J : Int .
  vars L R : Tree .

  eq search(I, L I R) = true .
  eq search(I, L J R) = search(I, L) or search(I, R) [owise] .
  eq search(I, empty) = false .
endfm
```

- equations with the attribute owise are applied “otherwise”

Subsort Declaration

- `subsort` s' `<` s `.`
  - sort s' is included in the sort s.
- Defines partially ordered set of sorts
  - each connected component of sort s is denoted by $[s]$.
- Subsort overloading

```
sorts Nat Int .
subsort Nat < Int .

op _+_ : Nat Nat -> Nat .
op _+_ : Int Int -> Int .
```

Example: List (1)

```
fmod INT-LIST is
  protecting INT .

  sorts IntList NeIntList .
  subsort NeIntList < IntList .

  op nil : -> IntList [ctor] .
  op __ : Int IntList -> IntList [ctor] .
  op __ : Int IntList -> NeIntList [ctor] .
endfm
```

- `subsort NeIntList < NeIntList`: a nonempty list is also a list
- ctor attributes denote constructors

Example: List (2)

```
fmod LENGTH is
  protecting INT-LIST .

  var I : Int .
  var L : IntList .

  op length : IntList -> Nat .
  eq length(I L) = 1 + length(L) .
  eq length(nil) = 0 .
endfm
```

Example: List (3)

```
fmod FIRST-LAST is
  protecting INT-LIST .

  var I : Int .
  var L : IntList .

  op first : NeIntList -> Nat .
  eq first(I L) = I .

  op last : NeIntList -> Nat .
  eq last(I nil) = I .
  eq last(I L) = last(L) .
endfm
```

Example: List (4)

```
fmod APPEND is
  protecting INT-LIST .

  var I : Int .
  vars L1 L2 : IntList .

  op append : IntList IntList -> IntList .
  eq append(I L1, L2) = I append(L1, L2) .
  eq append(nil, L2) = L2 .
endfm
```

Example: List (5)

```
fmod REV is
  protecting APPEND .

  var I : Int .
  var L : IntList .

  op rev : IntList -> IntList .
  eq rev(I L) = append(rev(L), I nil) .
  eq rev(nil) = nil .
endfm
```

Associativity, Commutativity and Identity Attributes

- Structural axioms
  - combinations of associativity (A), commutativity (C), identity (I)
  - can be defined by using attributes of operator declarations
- Example

```
op _+_ : Int Int -> Int [assoc comm id: 0 prec 33] .
op _*_ : Int Int -> Int [assoc comm id: 1 prec 31] .
```

- ACI attributes are logically equivalent to equations, e.g.,

```
eq A + (B + C) = (A + B) + C .
eq A + B = B + A .    --- not terminating
eq A + 0 = A .
```

Rewriting Modulo ACI Attributes (1)

```
fmod NAT-SET is
  protecting NAT .

  sort NatSet .
  subsort Nat < NatSet .
  op none : -> NatSet [ctor] .
  op __ : NatSet NatSet -> NatSet
             [ctor assoc comm id: none] .
```

```
  var N : Nat . vars S : NatSet .

  op _in_ : Nat NatSet -> Bool .
  eq N in N S = true .
  eq N in S = false [owise] .
endfm
```

- `subsort Nat < NatSet`: numbers are also sets of numbers
- constant `none` and concatenation operation `__` generate sets
- term `N S` can match any set, where `N` is any element in the set
- equations with the attribute owise are applied “otherwise”

Rewriting Modulo ACI Attributes (2)

```
Maude> red 0 none 1 none 2 none .
reduce in NAT-SET : 0 1 2 .
rewrites: 0 in 0ms cpu (0ms real) (0 rewrites/second)
result NatSet: 0 1 2
```

```
Maude> red 1 in 0 1 2 .
reduce in NAT-SET : 1 in 0 1 2 .
rewrites: 1 in 0ms cpu (0ms real) (1000000 rewrites/second)
result Bool: true
```

```
Maude> red 3 in 0 1 2 .
reduce in NAT-SET : 3 in 0 1 2 .
rewrites: 1 in 0ms cpu (0ms real) (1000000 rewrites/second)
result Bool: false
```

Example: Associative List (1)

```
fmod ASSOC-INT-LIST is
  protecting INT .

  sorts IntList NeIntList .
  subsort Int < NeIntList < IntList .

  op nil : -> IntList [ctor] .
  op __ : IntList IntList -> IntList [ctor assoc id: nil] .
  op __ : NeIntList NeIntList -> NeIntList [ctor assoc id: nil] .
endfm
```

Example: Associative List (2)

```
fmod ASSOC-FIRST-LAST is
  protecting ASSOC-INT-LIST .

  var I : Int .
  var L : IntList .

  op first : NeIntList -> Nat .
  eq first(I L) = I .

  op last : NeIntList -> Nat .
  eq last(L I) = I .
endfm
```

Example: Associative List (3)

```
fmod ASSOC-APPEND is
  protecting ASSOC-INT-LIST .

  var I : Int .
  vars L1 L2 : IntList .

  op append : IntList IntList -> IntList .
  eq append(L1, L2) = L1 L2 .
endfm
```

Example: Associative List (4)

```
fmod ISORT is
  protecting ASSOC-INT-LIST .

  vars I J : Int .
  var L : IntList .

  op insert : Int IntList -> IntList .
  eq insert(I, J L)
   = if I > J then J insert(I,L) else I J L fi .
  eq insert(I, nil) = I .

  op isort : IntList -> IntList .
  eq isort(I L) = insert(I, isort(L)) .
  eq isort(nil) = nil .
endfm
```

Example: Associative Lists and Trees

```
fmod FLATTEN is
  protecting ASSOC-INT-LIST .
  protecting TREE .

  vars L R : Tree .
  var I : Int .

  op flatten : Tree -> IntList .
  eq flatten(L I R) = flatten(L) I flatten(R) .
  eq flatten(empty) = nil .
endfm
```

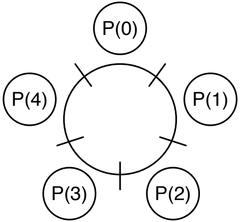
Maude 기반 정형 명세

- 시스템 상태: 대수적 자료 구조
  - recursive data types and functions
  - lists, sets, multi-sets, $\cdots$
- 시스템의 상태 변화
  - rewrite rule $t \longrightarrow t'$
  - 패턴 t에서 패턴 t' 으로의 변화

Maude에서의 시스템 상태변화 명세

Example: Dining Philosophers

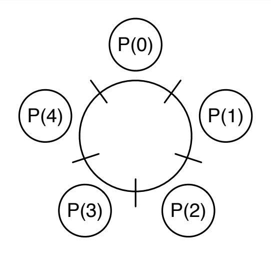
- Five philosophers, and five chopsticks on a circular table



- Philosophers are thinking, waiting, or eating
- Need two chopsticks for eating

Example: Dining Philosophers

- Operators: p, c, $\mathtt{think}$, $\mathtt{wait}$, $\mathtt{eat}$, $\mathtt{\_,\_}$, $\mathtt{none}, \ldots$



$$= \quad \begin{array}{l} \mathtt{p}(0, \mathtt{think}),\ \mathtt{c}(0), \\ \mathtt{p}(1, \mathtt{think}),\ \mathtt{c}(1), \\ \mathtt{p}(2, \mathtt{think}),\ \mathtt{c}(2), \\ \mathtt{p}(3, \mathtt{think}),\ \mathtt{c}(3), \\ \mathtt{p}(4, \mathtt{think}),\ \mathtt{c}(4) \end{array}$$

Example: Dining Philosophers in Maude (1)

```
fmod DINING-PHILOS-CONF is
  including NAT .
  sort Status .
  ops think eat : -> Status [ctor] .
  op wait : Nat -> Status [ctor] .

  sorts Philo Chopstick .
  op p : Nat Status -> Philo [ctor] .
  op c : Nat -> Chopstick [ctor] .

  sort Conf .
  subsorts Philo Chopstick < Conf .
  op none : -> Conf [ctor] .
  op _,_ : Conf Conf -> Conf [ctor comm assoc id: none] .

  eq s s s s s N:Nat = N:Nat .
endfm
```

Example: Dining Philosophers in Maude (2)

```
mod DINING-PHILOS is
  including DINING-PHILOS-CONF .
  vars I J : Nat .

  rl [wake]: p(I,think) => p(I,wait(0)) .

 crl [grabF]: p(I,wait(0)), c(J) => p(I,wait(1))
   if J == I or J == s(I) .

 crl [grabS]: p(I,wait(1)), c(J) => p(I,eat)
   if J == I or J == s(I) .

  rl [stop]: p(I,eat) => p(I,think), c(I), c(s(I)) .
endm
```

- Rules and equations can be conditional (with crl and ceq).

The `rewrite` Command

```
Maude> rew [11] p(0,think), c(0), p(1,think), c(1),
                p(2,think), c(2), p(3,think), c(3),
                p(4,think), c(4) .
rewrite [11] in DINING-PHILOS : p(0, think),c(0),p(1, think),c(
    1),p(2, think),c(2),p(3, think),c(3),c(4),p(4, think) .
rewrites: 47 in 0ms cpu (0ms real) (540229 rewrites/second)
result Conf: c(2),c(3),c(4),p(0, eat),p(1, think),p(2, think),
    p(3, think),p(4, think)
```

- The rewrite (or simply rew) command executes rewrite rules.
- Compute one possible behavior among many
- A number of rewrite steps can be bounded (e.g., 11).

The `frewrite` Command

```
Maude> frew [11] p(0,think), c(0), p(1,think), c(1),
                p(2,think), c(2), p(3,think), c(3),
                p(4,think), c(4) .
frewrite in DINING-PHILOS : p(0, think),c(0),p(1, think),c(1),
    p(2, think),c(2),p(3, think),c(3),c(4),p(4, think) .
rewrites: 76 in 0ms cpu (0ms real) (863636 rewrites/second)
result Conf: c(1),c(2),c(3),p(0, wait(0)),p(1, wait(0)),p(2,
    wait(0)),p(3, wait(0)),p(4, eat)
```

- The frewrite (or simply frew) command also executes rewrite rules.
- The frewrite command a depth-first position-fair strategy
  - whereas rewrite uses the left-most & outermost strategy

The `search` Command

- Search for n states from initial state t

  `search [`*n*`]` *t* `=>*` *pattern* `such that` *condition* `.`

  - match the search pattern and satisfy the search condition
  - explore all possible behaviors by using breadth-first search

- Search for states that cannot be further rewritten by rules

  `search [`*n*`]` *t* `=>!` *pattern* `such that` *condition* `.`

Example (1)

```
Maude> search [3] p(0,think), c(0), p(1,think), c(1),
                  p(2,think), c(2), p(3,think), c(3),
                  p(4,think), c(4)
              =>* p(0,eat), p(2,eat), C:Conf .

Solution 1 (state 418)
C:Conf --> c(4),p(1, think),p(3, think),p(4, think)

Solution 2 (state 694)
C:Conf --> c(4),p(1, wait(0)),p(3, think),p(4, think)

Solution 3 (state 707)
C:Conf --> c(4),p(1, think),p(3, wait(0)),p(4, think)
Maude>
```

Example (2)

```
Maude> search [1] p(0,think), c(0), p(1,think), c(1),
                  p(2,think), c(2), p(3,think), c(3), p(4,think), c(4)
              =>* p(I,eat), c(J), C:Conf
        such that I == s(J) .

Solution 1 (state 26)
C:Conf --> c(2),c(3),p(1, think),p(2, think),p(3, think),p(4,
    think)
J --> 4
I --> 0
```

```
Maude> search [1] p(0,think), c(0), p(1,think), c(1),
                  p(2,think), c(2), p(3,think), c(3), p(4,think), c(4)
              =>* p(I,eat), c(J), C:Conf
        such that J == s(I) .

No solution.
```

Example (3)

```
Maude> search p(0,think), c(0), p(1,think), c(1),
                p(2,think), c(2), p(3,think), c(3),
                p(4,think), c(4)
            =>! C:Conf .

Solution 1 (state 1347)
states: 1363  rewrites: 64926 in 24ms cpu (24ms real) (2689226
    rewrites/second)
C:Conf --> p(0, wait(1)),p(1, wait(1)),p(2, wait(1)),p(3, wait(
    1)),p(4, wait(1))

No more solutions.
states: 1363  rewrites: 64954 in 24ms cpu (24ms real) (2682608
    rewrites/second)
```

Example (4)

```
Maude> show path 1347 .
state 0, Conf: c(0),c(1),c(2),c(3),c(4),p(0, think),p(1,
    think),p(2, think),p(3, think),p(4, think)
===[ rl p(I, think) => p(I, wait(0)) [label wake] . ]===>
state 1, Conf: c(0),c(1),c(2),c(3),c(4),p(0, wait(0)),p(1,
    think),p(2, think),p(3, think),p(4, think)
===[ crl c(J),p(I, wait(0)) => p(I, wait(1)) if J == I or J ==
    s I = true [label grabF] . ]===>

...

state 1249, Conf: c(4),p(0, wait(1)),p(1, wait(1)),p(2, wait(
    1)),p(3, wait(1)),p(4, wait(0))
===[ crl c(J),p(I, wait(0)) => p(I, wait(1)) if J == I or J ==
    s I = true [label grabF] . ]===>
state 1347, Conf: p(0, wait(1)),p(1, wait(1)),p(2, wait(1)),p(
    3, wait(1)),p(4, wait(1))
```

예제: Maude에서의 분산시스템 명세

Concurrent Objects

- Distributed systems
  - networked components that collaborate to achieve a certain goal
  - WWW, P2P, IoT, cloud computing, blockchain, ⋯
- Distributed systems are often modeled using concurrent objects
  - each component is an object
  - communication by “message passing”

Concurrent Objects in Maude (1)

- An object of class C is represented as a term

$$< o : C \mid att_1 : val_1, \ldots, att_n : val_n >$$

  - o: the name (or identifier) of the object
  - $att_1, \ldots, att_n$: the names of the attributes (or fields)
  - $val_1, \ldots, val_n$: the values of the attributes

Concurrent Objects in Maude (2)

- A message is a term of sort Msg
- A configuration is a multiset made up of objects and messages.

```
subsort Object Msg < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
          [ctor assoc comm id: none] .
```

Example

- A `Person` object

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

- A configuration

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

The Module Configuration

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

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

Example

```
mod PERSON is
   protecting STRING .
   including CONFIGURATION .

   sort Person .
   subsort Person < Cid .    --- class sort
   op Person : -> Person [ctor] . --- representative constant

   subsort String < Oid .
   op age`:_ : Nat -> Attribute [ctor] .
   op status`:_ : Status -> Attribute [ctor] .

   sort Status .
   op single : -> Status [ctor] .
   ops married engaged : String -> Status [ctor] .
endm
```

- Each class correspond to a sort (with a representative constant).

Rewrite Rules for Objects

- The dynamics of objects is defined using rewrite rules.
- A rule may involve zero, one, or many objects and messages.
- Objects and messages can be created and/or deleted by a rule.

Example: Local State Change

- A rule that defines the local state change for a single object.

```
vars X X' : String . vars N N' : Nat . var S : Status . vars ATTS ATTS' : AttributeSet .

crl [birthday] : < X : C:Person | age : N, status : S, ATTS >
              => < X : C:Person | age : N + 1, status : S, ATTS > if N < 999 .
```

- Example

```
Maude> rew [3] < "A" : Person | age : 21, status : single >
             < "B" : Person | age : 12, status : single > .
result Configuration: < "A" : Person | age : 24,status : single
   > < "B" : Person | age : 12,status : single >

Maude> frew [3] < "A" : Person | age : 21, status : single >
             < "B" : Person | age : 12, status : single > .
result (sort not calculated): < "A" : Person | age : 23,status
   : single > < "B" : Person | age : 13,status : single >
```

Example: Synchronous Communication

- More than one object may be involved in a rewrite rule.

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

- Example

```
Maude> rew [1] < "A" : Person | age : 35, status : single >
            < "B" : Person | age : 36, status : single >
            < "C" : Person | age : 29, status : single > .
result Configuration: < "A" : Person | age : 35,status :
   engaged("B") > < "B" : Person | age : 36,status : engaged(
   "A") > < "C" : Person | age : 29,status : single >
```

Example: Creation and Deletion of Objects

- Objects may be "removed" or "created" in the right-hand side.

```
rl [death] : < X : C:Person | age : N, status : single, ATTS > => none .

rl [birth] :
    < X : C:Person | age : N, status : married(X'), ATTS >
    < X' : C':Person | age : N', status : married(X), ATTS' >
 =>
    < X : C:Person | age : N, status : married(X'), ATTS >
    < X' : C':Person | age : N', status : married(X), ATTS' >
    < X + X' : Person | age : 0, status : single > .
```

Example: Communication Through Message Passing

- One object can "send" and "receive" a message.

```
op separate : Oid -> Msg [ctor] .

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

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

예제: 보안 프로토콜의 모델검증

Public-Key Cryptography

- Each agent A has a public key PK_A and a private key $PrvK_A$
  - the public key is known to all agents (e.g., by a trusted key server)
  - the private key of A is only known by A
- Data m encrypted with key K is denoted by $\{m\}_K$
  - $\{m\}_{PK_A}$ can only be decrypted with $PrvK_A$
  - i.e., one can send a secrete message m to A using $\{m\}_{PK_A}$

The NSPK Authentication Protocol

$$\begin{aligned} &\textit{Message } 1. \quad A \rightarrow B: \quad A.B.\{N_a.A\}_{PK_B} \\ &\textit{Message } 2. \quad B \rightarrow A: \quad B.A.\{N_a.N_b\}_{PK_A} \\ &\textit{Message } 3. \quad A \rightarrow B: \quad A.B.\{N_b\}_{PK_B} \end{aligned}$$

1. A sends the string "$A.B.\{N_a.A\}_{PK_B}$" to B
   - B can decrypt the encrypted part using his private key to obtain N_a
2. B sends the string "$B.A.\{N_a.N_b\}_{PK_A}$" to A
   - A can decrypt the encrypted part using her private key to obtain N_b.
3. A sends the string "$A.B.\{N_b\}_{PK_B}$"
   - A "knows" that B knows N_a, and B "knows" that A knows N_b

Modeling NSPK in Maude: Nonces, Keys, and Messages

- $nonce(A, i)$: the i-th nonce generated by A (we abstract from the numerical value)
- $pubKey(A)$: the public key of A
- $msg(V, A, B)$: a message from A to B with content V
- $encrypt(T, K)$: text T encrypt with key K
- Example: $A.B.\{N_A.A\}_{PK_B}$
  - $A\ .\ B\ .\ encrypt(nonce(A, 2)\ .\ A, pubKey(B))$

Modeling NSPK in Maude: Alice (1)

Initiator
initSessions
nonceCtr
send
recv-and-send

- `Initiator`: an agent who can initiate a run of the protocol

$$< A : \mathit{Initiator} \mid \mathit{initSessions} : \mathit{SESSIONS}, \mathit{nonceCtr} : \mathit{COUNTER} >$$

- *COUNTER*: the index of the next nonce generated by the object.
- *SESSIONS*: the set of all "sessions" of the protocol that A participated in.
  - *notInitiated*(B): want to initiate contact with B but has not yet done so
  - *initiated*(B, N): sent Message 1 to B with nonce N and waiting for Message 2 from B
  - *trustedConnection*(B): established authenticated connection with B.

Modeling NSPK in Maude: Alice (2)

- Sending Message 1

```
rl [send-1] :
  < A : Initiator | initSessions : notInitiated(B) SESSIONS, nonceCtr : N >
=>
  < A : Initiator | initSessions : initiated(B,nonce(A,N)) SESSIONS, nonceCtr : N + 1 >
  msg(encrypt(nonce(A, N) . A, pubKey(B)), A, B) .
```

- Receiving Message 2 and sending Message 3

```
rl [read-2-send-3] :
  msg(encrypt(NONCE . NONCE', pubKey(A)), B, A)
  < A : Initiator | initSessions : initiated(B,NONCE) SESSIONS >
=>
  < A : Initiator | initSessions : trustedConnection(B) SESSIONS >
  msg(encrypt(NONCE', pubKey(B)), A, B) .
```

Modeling NSPK in Maude: Bob (1)

Responder
respSessions nonceCtr
recv-and-send recv

- `Responder`: an agent who responds to an initiator

$$< B : Responder \mid respSessions : SESSIONS, nonceCtr : COUNTER >$$

- *COUNTER*: the index of the next nonce generated by the object.
- *SESSIONS*: the set of all “sessions” of the protocol that B participated in.
  - *responded*(A, N): received Message 1 from A and has responded using its nonce N.
  - *trustedConnection*(A): established authenticated connection with A.

Modeling NSPK in Maude: Bob (2)

- Receiving Message 1 and Sending Message 2

```
crl [read-1-send-2] :
  msg(encrypt(NONCE . A), pubKey(B), A, B)
  < B : Responder | respSessions : SESSIONS, nonceCtr : N >
=>
  < B : Responder | respSessions : responded(A,nonce(B,N)) SESSIONS, nonceCtr : N + 1 >
  msg(encrypt(NONCE . nonce(B,N), pubKey(A)), B, A)   if not A in SESSIONS .
```

- Receiving Message 3

```
rl [read-3] :
  msg(encrypt(NONCE, pubKey(B)), A, B)
  < B : Responder | respSessions : responded(A, NONCE) SESSIONS >
=>
  < B : Responder | respSessions : trustedConnection(A) SESSIONS > .
```

Modeling NSPK in Maude

- `InitAndResp`: agents that are both initiators and responders

$$\begin{aligned} < B : InitAndResp \mid\ & initSessions : SESSIONS1, \\ & respSessions : SESSIONS2, \\ & nonceCtr : COUNTER > \end{aligned}$$

- Inherits the rules for initiators and responders

Executing the NSPK Specification in Maude

```
Maude> search < "a" : InitAndResp | initSessions : notInitiated("c"),
                                    respSessions : empty, nonceCtr : 1 >
              < "Bank" : Responder | respSessions : empty, nonceCtr : 1 >
              < "c" : InitAndResp | initSessions : notInitiated("Bank") notInitiated("a"),
                                    respSessions : empty, nonceCtr : 1 >
      =>! C:Configuration .

Solution 1 (state 442)
C:Configuration -->
< "Bank" : Responder | respSessions : trustedConnection("c"), nonceCtr : 2 >
< "a" : InitAndResp | initSessions : trustedConnection("c"),
                      respSessions : trustedConnection("c"), nonceCtr : 3 >
< "c" : InitAndResp | initSessions : (trustedConnection("Bank") trustedConnection("a")),
                      respSessions : trustedConnection("a"), nonceCtr : 4 >
No more solutions.
states: 443 rewrites: 1882 in 7ms cpu (7ms real) (255741 rewrites/second)
```

Modeling Intruders: Dolev-Yao Model

Danny Dolev

Andrew Yao

- Overhear and/or intercept any messages in the network
- Decrypt messages that are encrypted with its own public key
- Introduce new messages using nonces that the intruder knows
- Replay any (encrypted) message it has seen

Modeling Intruders in Maude: Intercepting Messages

- An intruder intercepts an encrypted message

```
crl [intercept-but-not-understand] :
   msg(ENCRMSG, O', O)
   < I : Intruder | objsSeen : OS, encrMsgsSeen : MSGS >
=> < I : Intruder | objsSeen : OS ; O ; O', encrMsgsSeen : ENCRMSG ; MSGS >
if O =/= I .
```

- An intruder receives a message that can be decrypted

```
rl [intercept-msg-and-understand] :
   msg(encrypt(MSG, pubKey(I)), O, I)
   < I : Intruder | objsSeen : OS, noncesSeen : NSET >
=> < I : Intruder | objsSeen : OS ; O ; getOids(MSG),
                    noncesSeen : NSET getNonces(MSG) > .
```

Modeling Intruders in Maude: Sending (Fake) Messages

- An intruder sends a message with known encrypted contents

```
crl [send-encrypted] :
   < I : Intruder | encrMsgsSeen : encrypt(MSG, pubKey(B)) ; MSGS, objsSeen : A ; OS >
=> < I : Intruder | encrMsgsSeen : encrypt(MSG, pubKey(B)) ; MSGS, objsSeen : A ; OS >
   msg(encrypt(MSG, pubKey(B)), A, B)     if A =/= B .
```

- An intruder may compose any Message 1, Message or Message 3

```
crl [send-1-fake] :
   < I : Intruder | objsSeen : A ; B ; OS, noncesSeen : NONCE NSET >
=> < I : Intruder | objsSeen : A ; B ; OS, noncesSeen : NONCE NSET >
   msg(encrypt(NONCE . A, pubKey(B)), A, B)    if A =/= B /\ B =/= I .
```

- Overhearing can be mimicked by intercepting and sending.

Analyzing NSPK with Intruders in Maude (1)

```
eq intruderInit
 = < "Scrooge" : Initiator | initSessions : notInitiated("BeagleBoys"), nonceCtr : 1 >
   < "Bank" : Responder | respSessions : empty, nonceCtr : 1 >
   < "BeagleBoys" : Intruder | initSessions : empty,
                               respSessions : empty,
                               nonceCtr : 1,
                               agentsSeen : "Bank" ; "BeagleBoys",
                               noncesSeen : empty,
                               encrMsgsSeen : empty > .
```

- The Beagle Boys do not know any other agent, except the bank.
- Scrooge wants to contact the Beagle Boys but not the bank.

Analyzing NSPK with Intruders in Maude (2)

```
Maude> search [1] intruderInit
      =>* C:Configuration
         < "Bank" : Responder | respSessions : trustedConnection("Scrooge") SESSIONS > .

Solution 1 (state 130449)
states: 130450 rewrites: 2750762 in 4482ms cpu (4500ms real)
C:Configuration -->
  < "BeagleBoys" : Intruder |
       initSessions : empty, respSessions : empty, nonceCtr : 1,
       agentsSeen : ("Bank" ; "BeagleBoys" ; "Scrooge"),
       noncesSeen : (nonce("Bank", 1) nonce("Scrooge", 1)),
       encrMsgsSeen : encrypt(nonce("Scrooge",1) . nonce("Bank",1), pubKey("Scrooge")) >
  < "Scrooge" : Initiator | initSessions : trustedConnection("BeagleBoys"), nonceCtr : 2 >
SESSIONS --> (empty).Sessions
```

- The Beagle Boys successfully fooled the bank and Scrooge!

Analyzing NSPK with Intruders in Maude (3)

```
Maude> show path labels 130449 .
send-1
intercept-msg-and-understand
send-1-fake
read-1-send-2
intercept-but-not-understand
send-encrypted
read-2-send-3
intercept-msg-and-understand
send-3-fake
read-3
```

$$
\begin{array}{lrl}
S1.M1. & A \to I: & A.I.\{N_A.A\}_{PK_I} \\
S2.M1. & I(A) \to B: & A.B.\{N_A.A\}_{PK_B} \\
S2.M2. & B \to I(A): & B.A.\{N_A.N_B\}_{PK_A} \\
S1.M2. & I \to A: & I.A.\{N_A.N_B\}_{PK_A} \\
S1.M3. & A \to I: & A.I.\{N_B\}_{PK_I} \\
S2.M3. & I(A) \to B: & A.B.\{N_B\}_{PK_B}
\end{array}
$$

- The corrected protocol (Needham–Schroeder–Lowe)

$$
\begin{array}{lrl}
\textit{Message } 1. & A \to B: & A.B.\{N_A.A\}_{PK_B} \\
\textit{Message } 2. & B \to A: & B.A.\{N_A.N_B.B\}_{PK_A} \\
\textit{Message } 3. & A \to B: & A.B.\{N_B\}_{PK_B}
\end{array}
$$

Analyzing NSPK with Intruders: Discussion

- The attack was first found by Gavin Lowe in 1995 by formal analysis.
  - not known for more than 17 years!
  - the same attack is found by our Maude analysis.
- The corrected protocol (Needham–Schroeder–Lowe)

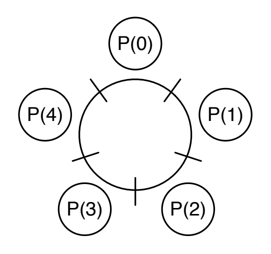
$$
\begin{aligned}
&\mathit{Message}\ 1. \quad A \rightarrow B: \quad A.B.\{N_a.A\}_{PK_B} \\
&\mathit{Message}\ 2. \quad B \rightarrow A: \quad B.A.\{N_a.N_b.B\}_{PK_A} \\
&\mathit{Message}\ 3. \quad A \rightarrow B: \quad A.B.\{N_b\}_{PK_B}
\end{aligned}
$$

- Many automated tools for cryptographic protocols developed.
  - Tamarin Prover, Scyther, ProVerif, Maude-NPA, $\cdots$

LTL 성질에 대한 모델검증

Example: Dining Philosophers (Revisited)

- Five philosophers, and five chopsticks on a circular table



Properties of Concurrent Systems

- Example
  - two adjacent philosophers cannot eat at the same time.
  - three philosophers cannot eat at the same time.
- Invariants
  - properties of single states
  - properties to be satisfied in all reachable states.

More Properties of Concurrent Systems

- Example
  - a philosopher will eventually eat.
  - whenever a philosopher is waiting, the philosopher will eat.
  - it is always possible that every philosopher thinks in the future.
- Are they invariants?
  - if not, why?

Safety and Liveness Properties

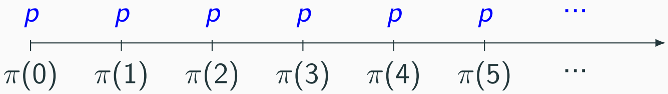
- Safety: nothing bad will happen
  - The system should not crash.
  - Three philosophers cannot eat at the same time.
- Liveness: something good must happen
  - Every packet sent must be received.
  - A philosopher will eventually eat.

Amir Pnueli

- Logic for specifying linear-time properties
- Propositional LTL extends propositional logic
- Temporal operators: $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), $\mathcal{U}$ (until)

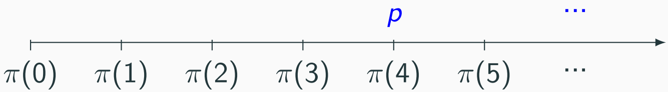
Temporal Operators: Always

- $\Box p$ (always p) is true iff p holds in all states along a path π



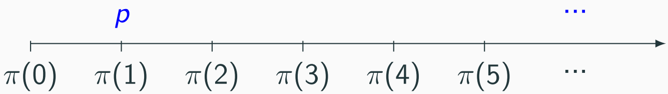
Temporal Operators: Eventually

- $\Diamond p$ (eventually p) is true iff p holds in some state along a path π



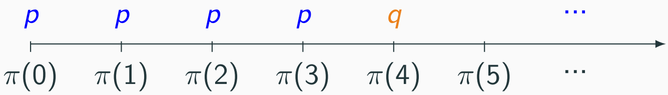
Temporal Operators: Next

- $\bigcirc p$ (next p) is true iff p holds in $\pi(1)$ (i.e., the next state of $\pi(0)$)



Temporal Operators: Until

- $p \mathcal{U} q$ (p until q) is true iff
  - q holds in some state s_i (i.e., eventually q), and
  - p holds in all states s_j for $0 \leq j < i$ between s_0 and s_i



Examples

- Philosopher 1 will eventually eat.

$$\Diamond \mathit{eating}(1)$$

- Whenever Philosopher 1 is waiting, the philosopher will eat.

$$\Box(\mathit{hungry}(1) \rightarrow \Diamond \mathit{eating}(1))$$

- No more than one thread (total 2 threads) can write a file.

$$\Box(\neg \mathit{write}(1) \vee \neg \mathit{write}(2))$$

- Every request signal must receive an acknowledge signal and the request should stay asserted until the acknowledge signal is received.

$$\Box(\mathit{req} \rightarrow \mathit{req}\,\mathcal{U}\,\mathit{ack})$$

Example

Consider the following sequence

$$\{p\} \rightarrow \emptyset \rightarrow \{p, q\} \rightarrow \{q\} \rightarrow \{q\} \rightarrow \cdots$$

- $\Box(p \rightarrow \Diamond q)$ *True*
- $\Box(q \rightarrow \Diamond p)$ *False*
- $\bigcirc(\neg q \mathcal{U} p)$ *True*
- $\neg q \mathcal{U} p$ *True*
- $p \mathcal{U} (p \wedge q)$ *False*

Maude LTL Model Checker

- Maude has efficient explicit-state LTL model checker
- Requires finite reachable state space from initial state
- Counterexample if some path does not satisfy a given LTL formula

LTL Model Checking in Maude (1)

- LTL formulas are declared in the module LTL.

```
sort Formula .
op ~_ : Formula -> Formula [...] .       op O_ : Formula -> Formula [...] .
op _/\_ : Formula Formula -> Formula [comm ...] .
op _\/_ : Formula Formula -> Formula [comm ...] .
op _U_ : Formula Formula -> Formula [...] .
...
```

- State labels are declared in the module SATISFACTION.

```
sorts State Prop .
op _|=_ : State Prop -> Bool [...] .
```

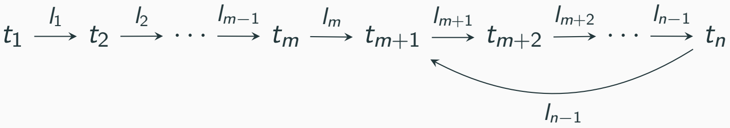
- The MODEL-CHECKER module includes:
  - LTL and SATISFACTION, and
  - signature for counterexamples, and the modelCheck operator

LTL Model Checking in Maude (2)

- Counterexamples are given by terms of the form

$$\texttt{counterexample}(\{t_1, l_1\}\{t_2, l_2\}\ldots\{t_m, l_m\},\ \{t_{m+1}, l_{m+1}\}\ldots\{t_n, l_n\})$$

  with rule labels $l_1, \ldots, l_n$, representing the “lasso-shape” path:



- The `modelCheck` function runs LTL model checking algorithm

```
op modelCheck : State Formula ~> ModelCheckResult [...] .
```

Example: Dining Philosophers (1)

```
mod DINING-PHILOS-CHECK is
  protecting DINING-PHILOS .
  including MODEL-CHECKER .
  subsort Conf < State .
  ops thinking eating : Nat -> Prop .
  op waiting : Nat Nat -> Prop .

  vars I J K : Nat .
  var REST : Conf .

  eq p(I,think), REST |= thinking(I) = true .
  eq p(I,wait(K)), REST |= waiting(I,K) = true .
  eq p(I,eat), REST |= eating(I) = true .
  eq REST |= P:Prop = false [owise] .
endm
```

Example: Dining Philosophers (2)

- Philosophers 1 and 2 cannot eat at the same time

```
Maude> red modelCheck(initial, []~ (eating(1) /\ eating(2))) .
rewrites: 66470 in 29ms cpu (30ms real) (2247734 rewrites/second)
result Bool: true
```

- Philosophers 0 and 2 cannot eat at the same time

```
Maude> red modelCheck(initial, []~ (eating(0) /\ eating(2))) .
rewrites: 1222 in 0ms cpu (0ms real) (1309753 rewrites/second)
result ModelCheckResult: counterexample(
  {c(0),c(1),c(2),c(3),c(4),p(0,think),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
  {c(0),c(1),c(2),c(3),c(4),p(0,wait(0)),p(1,think),p(2,think),p(3,think),p(4,think),'grabF}
  {c(1),c(2),c(3),c(4),p(0,wait(1)),p(1,think),p(2,think),p(3,think),p(4,think),'grabS}
  {c(2),c(3),c(4),p(0,eat),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
  {c(2),c(3),c(4),p(0,eat),p(1,wait(0)),p(2,think),p(3,think),p(4,think),'grabF}
  ...
  {c(3),c(4),p(0,wait(1)),p(1,eat),p(2,wait(0)),p(3,think),p(4,think),'grabF}
  {c(4),p(0,wait(1)),p(1,eat),p(2,wait(1)),p(3,think),p(4,think),'stop}
  {c(1),c(2),c(4),p(0,wait(1)),p(1,think),p(2,wait(1)),p(3,think),p(4,think),'grabS}
  {c(2),c(4),p(0,eat),p(1,think),p(2,wait(1)),p(3,think),p(4,think),'grabS}
  {c(4),p(0,eat),p(1,think),p(2,eat),p(3,think),p(4,think),'stop}
...)
```

Example: Dining Philosophers (3)

- Philosophers 1 will eventually eat.

```
Maude> red modelCheck(initial, <> eating(1)) .
rewrites: 420 in 0ms cpu (0ms real) (1944444 rewrites/second)
result ModelCheckResult: counterexample(
   {c(0),c(1),c(2),c(3),c(4),p(0,think),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
   {c(0),c(1),c(2),c(3),c(4),p(0,wait(0)),p(1,think),p(2,think),p(3,think),p(4,think),'grabF}
   {c(1),c(2),c(3),c(4),p(0,wait(1)),p(1,think),p(2,think),p(3,think),p(4,think),'grabS}
   {c(2),c(3),c(4),p(0,eat),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
   {c(2),c(3),c(4),p(0,eat),p(1,wait(0)),p(2,think),p(3,think),p(4,think),'grabF}
   ...
   {c(0),c(1),p(0,wait(0)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,think),'grabF}
   {c(1),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,think),'wake}
   {c(1),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabS}
   {p(0,eat),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'stop}
   {c(0),c(1),p(0,think),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabF}
   {c(1),p(0,think),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(1)),'wake}
   {c(1),p(0,wait(0)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(1)),'grabF}
   ,
   {p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(1)),deadlock})
```

Defining Deadlock as State Proposition

- Proposition enabled is true iff some rule can be applied.
  - by equations involving left-hand sides and conditions of rules
- Proposition deadlock is true iff enabled is false.

```
ops enabled deadlock : -> Prop .

eq p(I,think), REST |= enabled = true .
ceq p(I,wait(0)), c(J), REST |= enabled = true if J == I or J == s(I) .
ceq p(I,wait(1)), c(J), REST |= enabled = true if J == I or J == s(I) .
eq p(I,eat), REST |= enabled = true .

eq REST |= deadlock = not (REST |= enabled) .
```

Example: Dining Philosophers (4)

- Philosophers 1 will eventually eat, assuming no deadlock.

```
Maude> red modelCheck(initial, []~ deadlock -> <> eating(1)) .
rewrites: 737 in 0ms cpu (0ms real) (19394736 rewrites/second)
result ModelCheckResult: counterexample(
   {c(0),c(1),c(2),c(3),c(4),p(0,think),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
   {c(0),c(1),c(2),c(3),c(4),p(0,wait(0)),p(1,think),p(2,think),p(3,think),p(4,think),'grabF}
   {c(1),c(2),c(3),c(4),p(0,wait(1)),p(1,think),p(2,think),p(3,think),p(4,think),'grabS}
   {c(2),c(3),c(4),p(0,eat),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
   ...
   {c(0),c(1),p(0,think),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,think),'wake}
   {c(0),c(1),p(0,wait(0)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,think),'grabF}
   {c(1),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,think),'wake}
   {c(1),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabS}
   ,
   {p(0,eat),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'stop}
   {c(0),c(1),p(0,think),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'wake}
   {c(0),c(1),p(0,wait(0)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabF}
   {c(0),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabS})
```

- Philosopher 1 does nothing, while infinitely often able to eat.
  - not possible if we assume reasonable scheduler

Fairness Assumptions

- Rule out unrealistic infinite behaviors in concurrent systems
  - often necessary to establish liveness properties
  - often parameterized to specific system entities
- Weak fairness
  - if continuously enabled after certain point, infinitely often act.
  - $\Diamond\Box \mathit{enabled.action} \rightarrow \Box\Diamond \mathit{action}$
- Strong fairness
  - if enabled continuously often, infinitely often act.
  - $\Box\Diamond \mathit{enabled.action} \rightarrow \Box\Diamond \mathit{action}$
- Special case of liveness properties

Defining Fairness Constraints

- `enabled.action(I)` for each philosopher I

```
eq p(I,think), REST |= enabled.wake(I) = true .
ceq p(I,wait(0)), c(J), REST |= enabled.grabF(I) = true if J == I or J == s(I) .
ceq p(I,wait(1)), c(J), REST |= enabled.grabS(I) = true if J == I or J == s(I) .
eq p(I,eat), REST |= enabled.stop(I) = true .
```

- `action(I)` for each philosopher I
  - but which is not a state proposition
  - can be defined as a formula using $\bigcirc$ operator

```
eq wake(I) = thinking(I) /\ O waiting(I,0) .
eq grabF(I) = waiting(I,0) /\ O waiting(I,1) .
eq grabS(I) = waiting(I,1) /\ O eating(I) .
eq stop(I) = eating(I) /\ O thinking (I) .
```

  - generally, need to record last action taken in state

Example: Dining Philosophers (5)

- Philosophers 1 will eventually eat, assuming
  - no deadlock
  - strong fairness of `grabS` for philosopher 1.

```
Maude> red modelCheck(initial, (([] ~ deadlock) /\
                          ([]<> enabled.grabS(1) -> []<> grabS(1))) -> <> eating(1)) .
rewrites: 890 in 0ms cpu (0ms real) (1797979 rewrites/second)
result ModelCheckResult: counterexample(
   {c(0),c(1),c(2),c(3),c(4),p(0,think),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
   ....
   {p(0,eat),p(1,wait(1)),p(2,wait(0)),p(3,eat),p(4,think),'wake}
   ,
   {p(0,eat),p(1,wait(1)),p(2,wait(0)),p(3,eat),p(4,wait(0)),'stop}
   {c(3),c(4),p(0,eat),p(1,wait(1)),p(2,wait(0)),p(3,think),p(4,wait(0)),'wake}
   {c(3),c(4),p(0,eat),p(1,wait(1)),p(2,wait(0)),p(3,wait(0)),p(4,wait(0)),'grabF}
   {c(4),p(0,eat),p(1,wait(1)),p(2,wait(0)),p(3,wait(1)),p(4,wait(0)),'grabS})
```

- Philosopher 0 does nothing forever, while continuously enabled.
  - need weak fairness of `stop` for philosopher 0
  - same situation can also happen for philosopher 2

Example: Dining Philosophers (6)

- Philosophers 1 will eventually eat, assuming
  - no deadlock
  - strong fairness of `grabS` for philosopher 1
  - weak fairness of `stop` for philosophers 0 and 2

```
Maude> red modelCheck(initial, (([] ~ deadlock) /\ (([]<> enabled.grabS(1)) -> []<> grabS(1)) /\
                             ((<>[] enabled.stop(0)) -> []<> stop(0)) /\
                             ((<>[] enabled.stop(2)) -> []<> stop(2))) -> <> eating(1)) .
rewrites: 4209 in 3ms cpu (5ms real) (1105304 rewrites/second)
result ModelCheckResult: counterexample(
   {c(0),c(1),c(2),c(3),c(4),p(0,think),p(1,think),p(2,think),p(3,think),p(4,think),'wake}
   ...
   {c(4),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabS}
   ,
   {p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,eat),p(4,wait(0)),'stop}
   {c(3),c(4),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,think),p(4,wait(0)),'wake}
   {c(3),c(4),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(0)),p(4,wait(0)),'grabF}
   {c(3),p(0,wait(1)),p(1,wait(1)),p(2,wait(1)),p(3,wait(1)),p(4,wait(0)),'grabS})
```

- Philosopher 2 does nothing, while infinitely often able to eat
  - need strong fairness of `grabS` for philosopher 2

Example: Dining Philosophers (7)

- Philosophers 1 will eventually eat, assuming
  - no deadlock
  - weak fairness of `wake` and `grabF` for philosopher 1
  - strong fairness of `grabS` for philosophers 0, 1, 2, 3, 4
  - weak fairness of `stop` for philosophers 0 and 2

```
Maude> red modelCheck(initial, (([] ~ deadlock) /\
                    ((<>[] enabled.wake(1)) -> []<> wake(1)) /\
                    ((<>[] enabled.grabF(1)) -> []<> grabF(1)) /\
                    (([]<> enabled.grabS(0)) -> []<> grabS(0)) /\
                    (([]<> enabled.grabS(1)) -> []<> grabS(1)) /\
                    (([]<> enabled.grabS(2)) -> []<> grabS(2)) /\
                    (([]<> enabled.grabS(3)) -> []<> grabS(3)) /\
                    (([]<> enabled.grabS(4)) -> []<> grabS(4)) /\
                    ((<>[] enabled.stop(0)) -> []<> stop(0)) /\
                    ((<>[] enabled.stop(2)) -> []<> stop(2))) -> <> eating(1)) .
rewrites: 364 in 136527ms cpu (168454ms real) (2 rewrites/second)
result Bool: true
```

Example: Dining Philosophers (8)

- Philosopher 1 will eat whenever hungry, assuming
  - no deadlock
  - strong fairness of `grabF` for philosopher 1
  - strong fairness of `grabS` for philosophers 0, 1, 2, 3, 4
  - weak fairness of `stop` for philosophers 0 and 2

```
Maude> red modelCheck(initial, (([] ~ deadlock) /\
                     (([]<> enabled.grabF(1)) -> []<> grabF(1)) /\
                     (([]<> enabled.grabS(0)) -> []<> grabS(0)) /\
                     (([]<> enabled.grabS(1)) -> []<> grabS(1)) /\
                     (([]<> enabled.grabS(2)) -> []<> grabS(2)) /\
                     (([]<> enabled.grabS(3)) -> []<> grabS(3)) /\
                     (([]<> enabled.grabS(4)) -> []<> grabS(4)) /\
                     ((<>[] enabled.stop(0)) -> []<> stop(0)) /\
                     ((<>[] enabled.stop(2)) -> []<> stop(2))
                    ) -> [] (wake(1) -> <> eating(1))) .
rewrites: 122251 in 725655ms cpu (848663ms real) (168 rewrites/second)
result Bool: true
```

모델검증 연구동향 및 전망

모델 검증 기법 적용의 장애물

- Algorithmic challenge: 상태 폭발 문제 (state space explosion)
  - 가능한 상태의 숫자가 소프트웨어 규모에 따라 기하급수적으로 증가
- Modeling challenge: 다양한 소프트웨어 시스템의 정형명세
  - 상태전이그래프, 동시성, 객체지향설계, 통신 프로토콜, 실시간 시스템, …

기술적 발전 (1990 $\sim$ 2020): 모델검증 알고리즘

- 효과적인 자료구조: 그래프, BDD, SAT/SMT, Regular languages, logical terms, ⋯
- 효과적인 탐색: 그래프 알고리즘, 기호기반탐색, Inductions, Interpolation, IC3, ⋯
- 상태공간 축소/요약: Partial order reduction, predicate abstraction, CEGAR, ⋯

기술적 발전 (1990 $\sim$ 2020): 모델링

- 요구사항 명세: CTL, LTL, CTL*, μ-calculus, Hyper-CTL*, ⋯
- 시스템 명세: Transition system, Petri nets, Process calculus, Term rewriting, ⋯
- 도메인: 보안/통신 프로토콜, 실시간 시스템, 사이버물리시스템, 확률적 시스템, ⋯

기술적 발전 (1990 ∼ 2020): 다른 검증 기술과의 교류

- 모델검증 +.테스팅: concolic testing, 병행시스템 testing, ···
- 모델검증 +.정적분석: abstraction, infinite-state model checking, ···
- 모델검증 +.자동증명: deduction-based model checking,
- 모델검증 + AI Planning: Planning by model checking, temporal planing, ···
- 모델검증 + Machine learning: Invariant learning, model learning, ···
- ···

모델검증 기술의 전망: 중요성 증가 (Revisited)

- 과거 – 현재
  - 주로 안전 필수 시스템의 경우
  - 항공기, 우주선, 열차제어, 자동차, 의료기기, ⋯
- 현재 – 가까운 미래
  - 소프트웨어의 중요성 증대: IoT, 인공지능, 무인자동차/항공기, ⋯
  - 소프트웨어 취약점을 악용하는 보안 사고 증가
- 가까운 미래
  - 검증된 소프트웨어 vs. 검증되지 않은 소프트웨어
  - 검증된 운영체제, 검증된 컴파일러, 검증된 애플리케이션, ⋯

모델검증 기술의 전망: 적용범위 증가

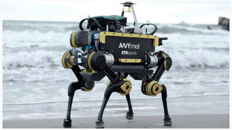
- 발전 방향
  - 보다 "일반적인" SW 개발 과정에 적용
  - 개발도구에 모델검증의 기술의 내재화
- 과제
  - 개발자의 정형명세의 직접 개발을 최소화하고, 일반적인 개발 산출물 활용
  - 전체 소프트웨어의 코드수준 분석은 상태폭발문제로 불가능
- 접근방향
  - 모델 기반 개발: 전통적인 안전필수 도메인 (개별 검증 $\rightarrow$ 통합 검증)
  - 모델 합성/학습: 코드, 실행 기록, 과거 검증 결과 등에서 모델 추출 (합성 $\rightarrow$ 학습)

모델검증 기술의 전망: 새로운 도메인

- 인공지능 기반 SW: 심층신경망(DNN) 등을 활용하여 제어

PilotNet (2016)

ACAS Xu DNN (2016)



ANYmal (2019)

- 인공지능 기만 SW의 오류: 테스트되지 않은 입력을 통하여 예상치 못한 제어 오류 가능
  - adversarial example, reward hacking, ⋯
- 모델검증 접근방법의 강점
  - 학습용 계산 모델이 존재하고, DNN 등 사람의 해석이 불가능한 모델에 대하여 적용 가능
- 접근방향
  - DNN 모델검증 알고리즘 (2018 ∼): Reluplex, DeepPoly, Neurify, ⋯

맺는말

- 정형기법
  - 소프트웨어/하드웨어 설계에 대한 수학적 방법론
  - 4차 산업혁명의 필수 기반 기술: 검증된 vs. 검증되지 않은 소프트웨어
- 모델검증
  - 정형명세 기반 시스템의 오류를 자동으로 찾는 기술
  - 장애물: 상태폭발문제 및 모델링/정형명세 문제
- Maude
  - 분산시스템 및 각종 프로토콜의 검증에 널리 사용되는 정형명세 및 모델검증 도구
  - 시스템의 상태: 대수적 자료구조 / 시스템의 상태변화: rewrite rule
- 모델검증 기술의 전망
  - 중요성/적용범위의 증가: 일반적인 SW 개발에 적용을 위한 연구
  - 새로운 도메인(AI 등)의 출연에 따른 연구

감사합니다!