Formal Languages and Specs

Neil Ernst based on V Gervasi and D Damian

Validation and Verification

- Validation criteria:
  - Discover and understand all domain properties
  - Discover and understand all requirements
- Verification criteria:
  - The program satisfies the specification
  - The **specification**, given the **domain properties**, satisfies the **requirements**

Propositional logic in formal RE specs

- A ≡ a person comes close to the door
  B ≡ the door opens
  C ≡ motion sensor is triggered
  D ≡ OPEN signal to motor

- Verification:

- Reqs: { A → B }
  Dom: { A → C, D → B }
  Spec: { C → D }
  Prove that Dom ∧ Spec → Reqs

Definition and Overview of Formal Methods

- A broad view of formal methods includes all applications of (primarily) discrete mathematics to software engineering problems. This application usually involves modeling and analysis where the models and analysis procedures are derived from or derived by an underlying mathematically-precise foundation.
- A formal method in software development is a method that provides formal language for describing a software artifact (for instance, specifications, designs, or source code) such that formal proofs are possible in principle, about properties of the artifact so expressed.

Use of Formal Methods (FM)

- Reasoning about a formal description
  The requirements problem:
- Verification does not eliminate the need for validation! However, the discipline of producing a formal specification can
  - result in fewer specification errors.
- When to use formal methods
  - Mostly functional requirements
  - Critical components of the system

Formal Requirements Spec

- Formal requirements specifications are specifications that have formal semantics and syntax.
- Allows a specification to be precise, unambiguous, and verifiable.
- Can be verified automatically.
- Useful in reasoning about the relationships between domain properties, requirements and specifications
- Prove properties of requirements and specs
- Good news: very useful when applied properly

Example of FM use

- Paris metro line 14 entirely controlled by software formally developed by Matra Transport using the B abstract machine method

Z Spec

Account

$\upharpoonright (accountNumber, Deposit, Withdraw)$

$accountNumber : \mathbb{N}one$
$ownerID : \mathbb{N}one$
$balance : \mathbb{R}$

Deposit

$\Delta(balance)$
$amount? : \mathbb{R}$

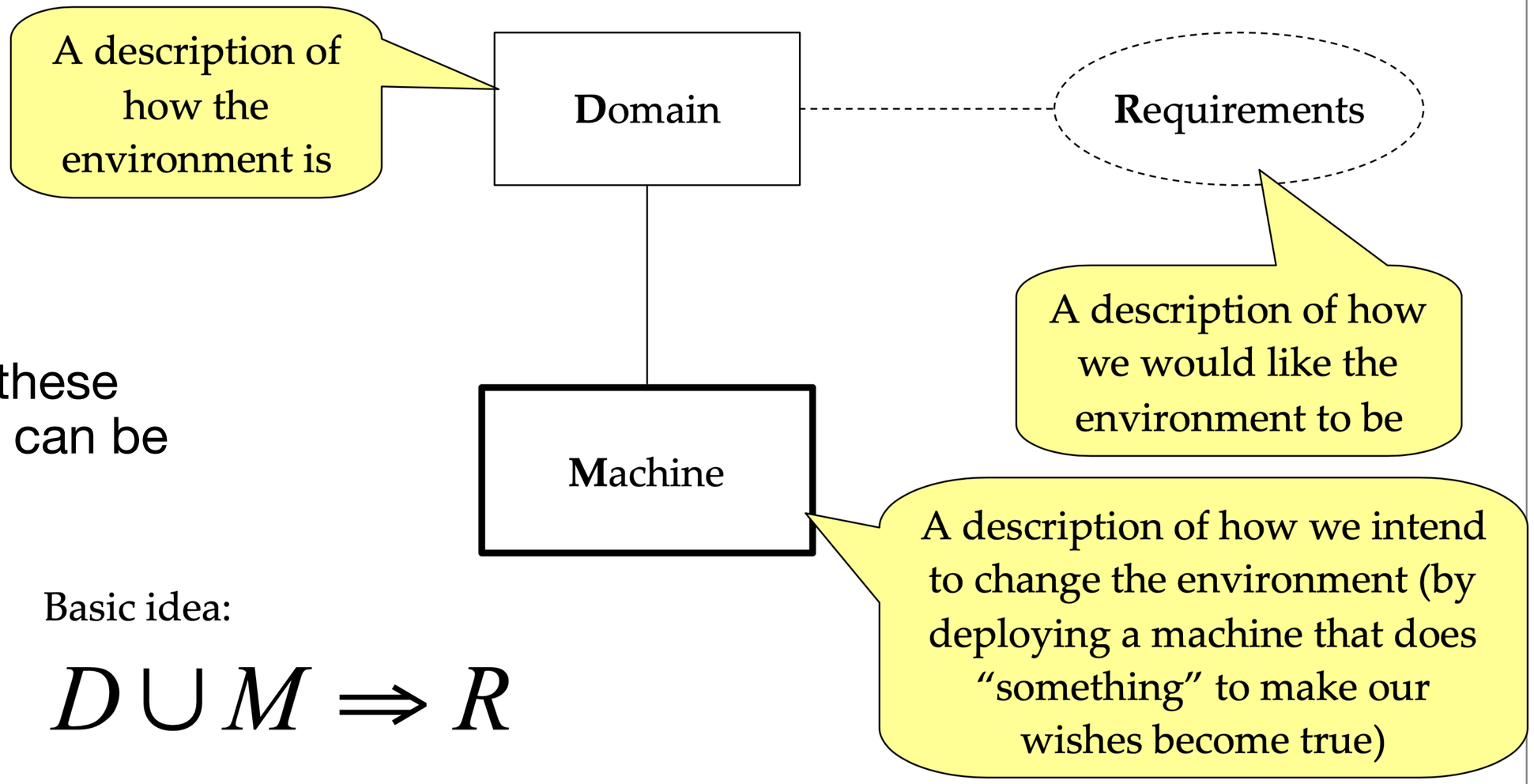
$amount? > 0.0$
$balance' = balance + amount?$

Withdraw

$\Delta(balance)$
$amount? : \mathbb{R}$

$amount? \geq 0.0$
$balance' = balance - amount?$

The role of FMs in RE



- All or some of these yellow callouts can be formalized

Basic idea:

$$D \cup M \Rightarrow R$$

An Example: Sliding Doors

- Design and implement the software for a sliding door controller (SDC) that we want to install in our restaurant (we specialize in stork soup)
- When someone comes close to the door, the door should open automatically
- Etc., e.g.: It should close after that person has entered the restaurant

Sliding doors

Requirements

When a person comes close to the door, the door should open

Domain

- My restaurant has a motor-controlled door
- A motion sensor is installed above the door
- Motion sensors are triggered by people coming close
- A motor-controlled door opens when the motor receives the OPEN signal

Specification

When the motion sensor is triggered, the system (SDC) should send the OPEN signal to the motor

A ≡ a person comes close to the door

B ≡ the door opens

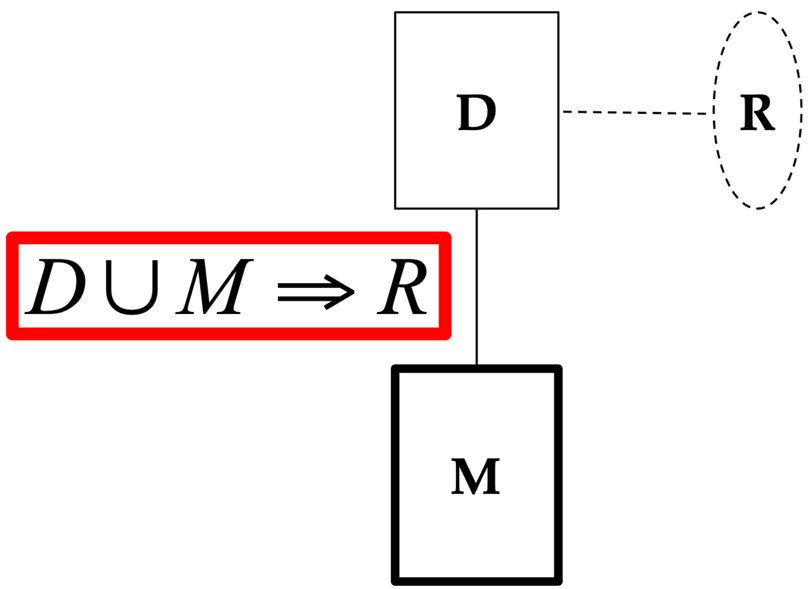
C ≡ motion sensor is triggered

D ≡ OPEN signal to motor

Formalize in Propositional Logic

The **specification**, given the **domain properties**, satisfies the **requirements**

Reqs: { A → B }
Dom: { A → C, D → B }
Spec: { C → D }
Prove that Dom ∧ Spec → Reqs



A ≡ a person comes close to the door
B ≡ the door opens
C ≡ motion sensor is triggered
D ≡ OPEN signal to motor

Types of Formal Specifications

- Property-oriented languages
  - System is described by a set of axiomatic properties
- Model-oriented languages
  - System is described by a **state**, and **operations**
  - An **operation** is a function which maps the value of the state and the value of input parameters to a new state

Automated Verification

- One major benefit for formal methods is automated verification.
- Two major categories of Verification that use Formal Methods:
- Theorem proving
- Model checking
-

Theorem Provers

- Coq example

NASA requirement

2.16.3.f) While acting as the bus controller, the C&C MDM CSCI shall set the e, c, w, indicator identified in Table 3.2.16-II for the corresponding RT to "failed" and set the failure status to "failed" for all RT's on the bus upon detection of transaction errors of selected messages to RTs whose 1553 FDIR is not inhibited in two consecutive processing frames within 100 msec of detection of the second transaction error if; a backup BC is available, the BC has been switched in the last 20 sec, the SPD card reset capability is inhibited, or the SPD card has been reset in the last 10 major (10 sec) frames, and either:

1. the transaction errors are from multiple RTs, the current channel has been reset within the last major frame, or
2. the transaction errors are from multiple RT's, the bus channel's reset capability is inhibited, and the current channel has not been reset within the last major frame.

[1] S. M. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," *TOSE*, vol. 24, pp. 4-14, 1998.

Verific

Cassini Requirement: If Spacecraft Safing is requested via a CDS (Command and Data Subsystem) internal request while the spacecraft is in a critical attitude, then no change is commanded to the AACS (Attitude and Articulation Control Subsystem) attitude. Otherwise, the AACS is commanded to the homebase attitude.

```
saf: THEORY
% Example is excerpted from saf theory.
% Spacecraft safing commands the AACS to homebase mode, thereby
% stopping delta-v's and desat's.
BEGIN

aacs_mode:   TYPE = {homebase, detumble}
attitude:    TYPE

cds_internal_request:   VAR bool
critical_attitude:    VAR bool
prev_aacs_mode:     VAR aacs_mode

aacs_stop_fnc (critical_attitude, cds_internal_request, prev_aacs_mode):
         aacs_mode =
         IF critical_attitude
           THEN IF cds_internal_request
                THEN prev_aacs_mode
                ELSE homebase
                ENDIF
           ELSE homebases
         ENDIF
```

```
aacs_safing_req_met_1:  LEMMA
       (critical_attitude AND cds_internal_request)
       OR (aacs_stop_fnc (critical_attitude, cds_internal_request,
          prev_aacs_mode) = homebase)

END saf
```

Model oriented languages

- The easiest way to consider a model-oriented language is as a state machine
- When the system is in a certain state, and is given a particular input, it will turn into another state
- Properties of the system are usually given using a temporal logic
- Want to verify that the model always holds the properties true
-

Challenges applying formal specification methods

- Limited scope (to functional requirements)
- Isolation from other software products and processes in organizations
- Cost (require high expertise in formal systems and mathematical logic)
-

Examples of well known formal methods and languages

- Language Z
- Communicating sequential processes (CSP)
- Vienna Development Method (VDM)
- Larch
- Formal development methodology (FDM)
- Software Cost Reduction (SCR)

Formal verification

- Prove correctness of a system using proof techniques and mathematical models
  - Theorem Proving (e.g. Coq)
  - Program derivation
  - Formal type-checking
  - **Model checking**
  - …
  - Widely used in hardware and real-time systems.

Motivation

- Problem
  - Want to **check** if system execution can guarantee **important properties** without extensive testing

- Solution
  - "**Query language**" to ask questions about execution
  - **Tool** which answers such queries

Overview

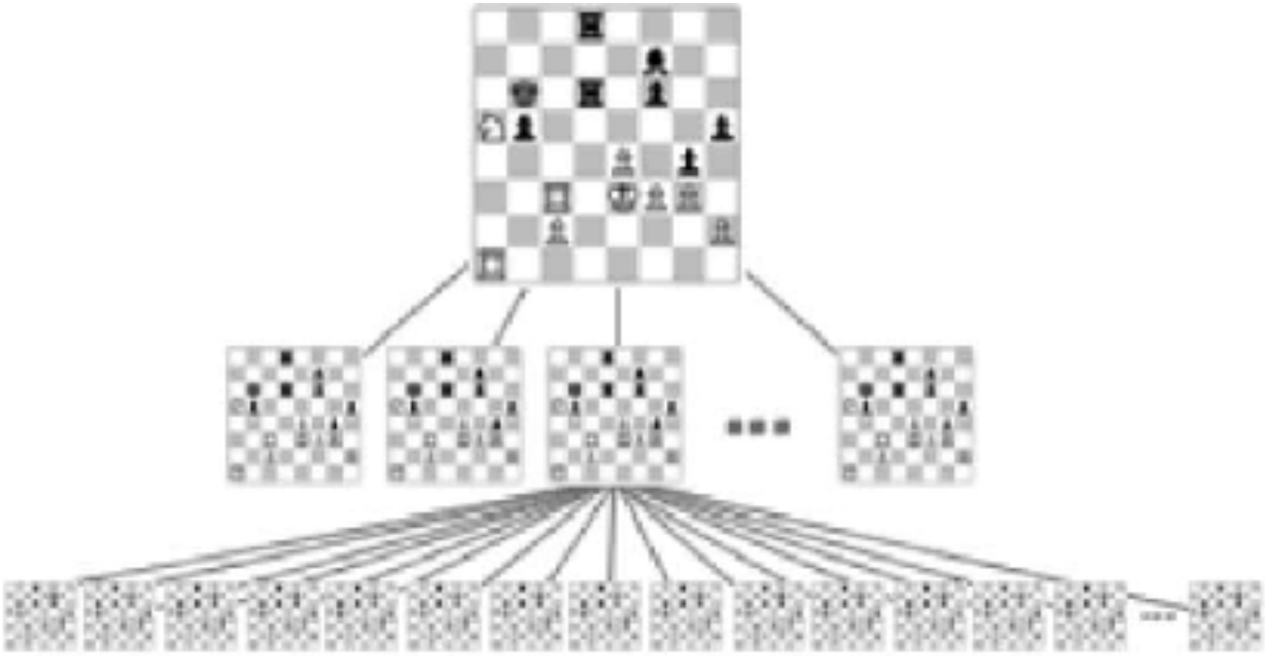
1. System
   - Statecharts
   - Source code
2. Properties
   - Safety
   - Liveness
   - Fairness
3. “Query language”
   - Computation Tree Logic (CTL)
4. Tool
   - Model Checker

Execution Properties

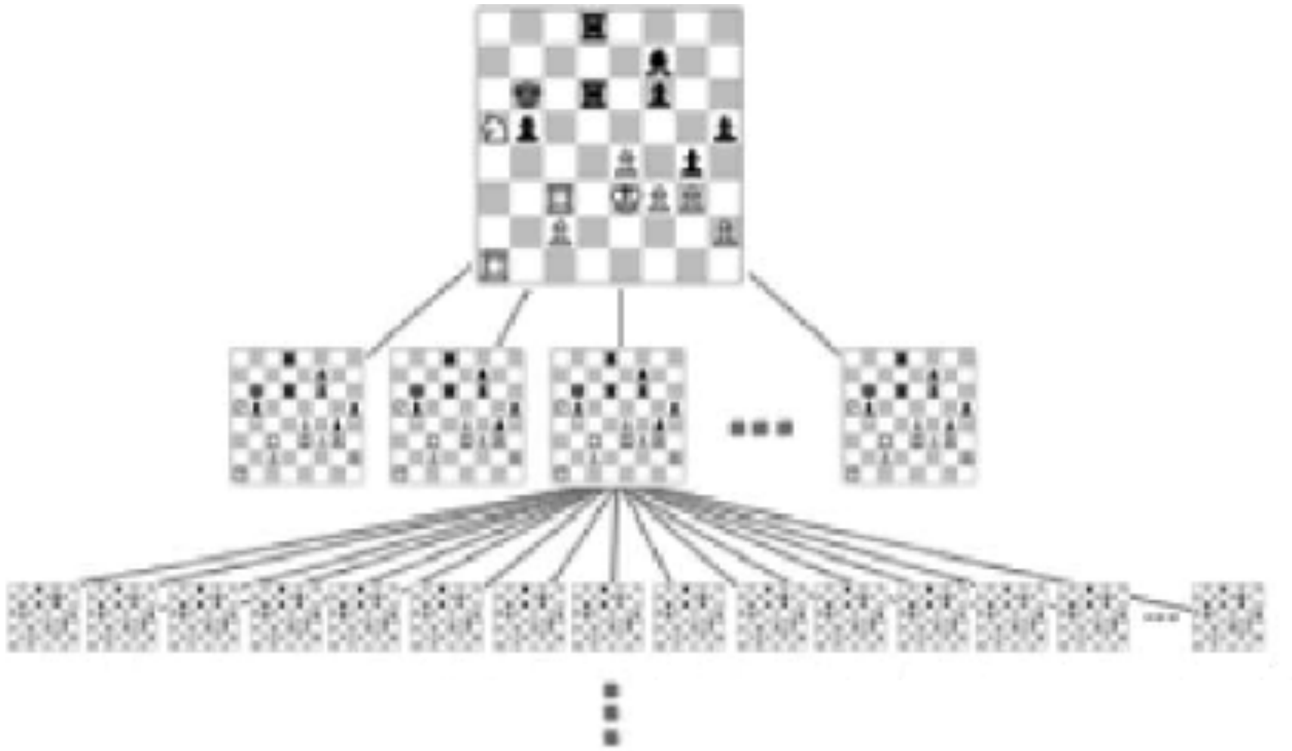
- **Safety**
  - “Something bad will not happen”
    - e.g. a null variable will never be referenced
- **Liveness**
  - “Something good will happen”
    - E.g. eventually the spinning rainbow will finish spinning
- **Fairness**
  - Regardless of other properties, something good will happen.
  - e.g. if I wait long enough, an elevator will arrive on my floor, regardless of when I arrive

What is execution?

- “A tree structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realized”



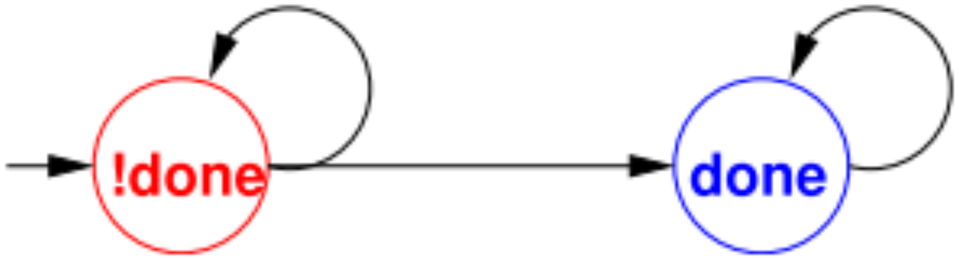
What about loops?

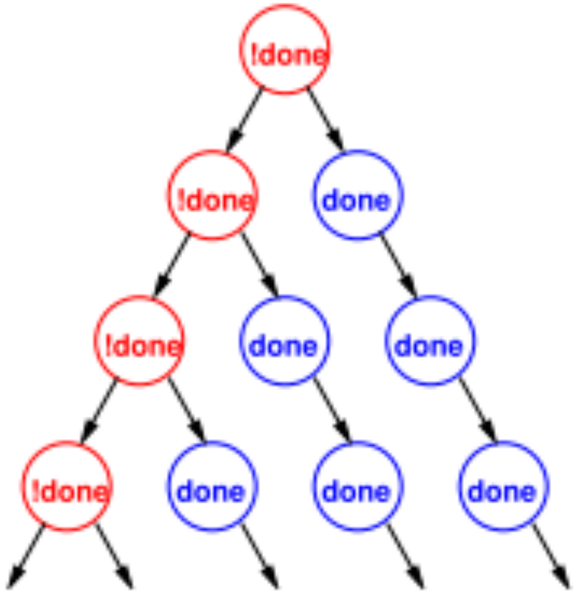


We assume tree can have infinite height

Call it a ***Computation Tree***

Computation Tree for Statecharts





- Shows all possible executions
- Each actual execution will follow a single path consisting of a sequence of discrete time steps

Computation Tree Logic

- Language for checking computation trees
- Extends propositional logic with
  temporal operators
- Quantifiers for time-sensitive expressions
- Propositional logic connectives

  Λ $\neg$ V $\rightarrow$

Unary Temporal Operators

- On all paths (A)

Next: Xp

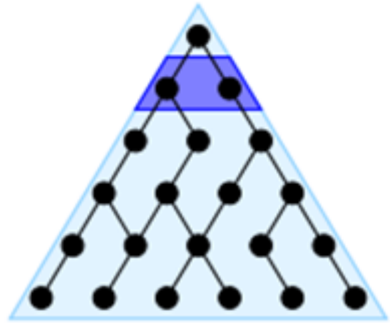
p is true in the next time step

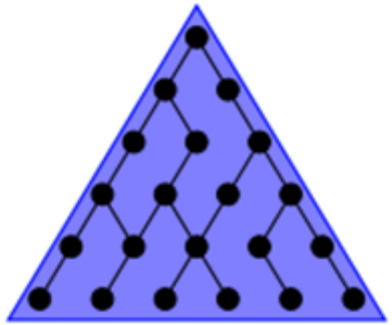
Globally: Gp

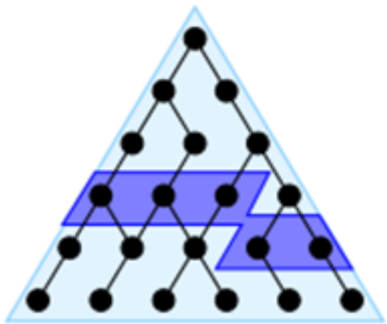
p is true and then forever

Finally: Fp

p will eventually become true

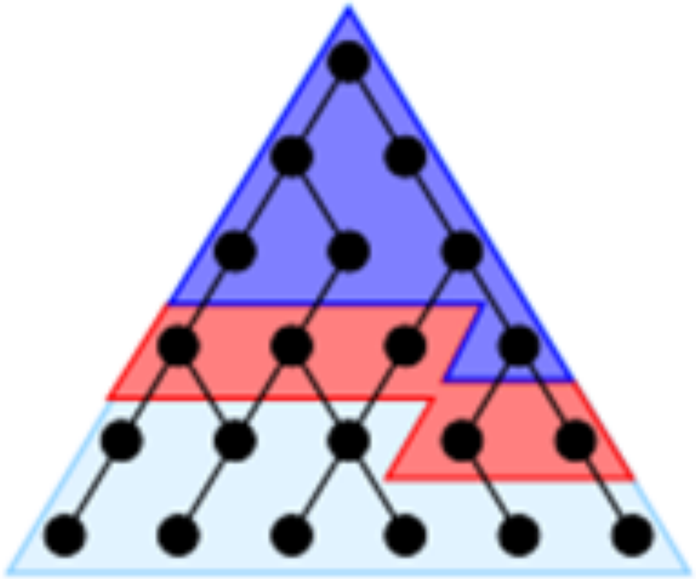






Until

- On all paths



Until: p U q

p is globally true until <u>at least</u> q is true

also q must eventually become true

Path Quantification

- Temporal operators just say what is true on a single execution(sequence)
- We want to talk about ***execution possibilities***

Forall paths: **A***p*

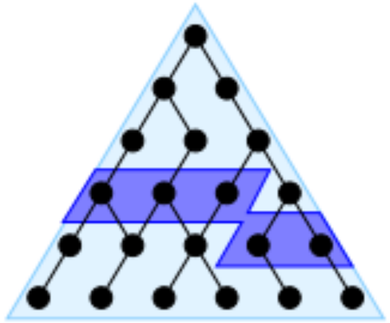
p is true for every possible execution path

There exists a path: **E***p*

p is true for at least one execution path

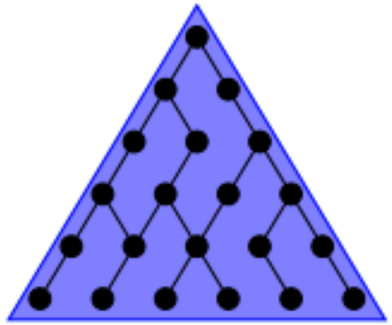
Computation Tree Logic

finally p



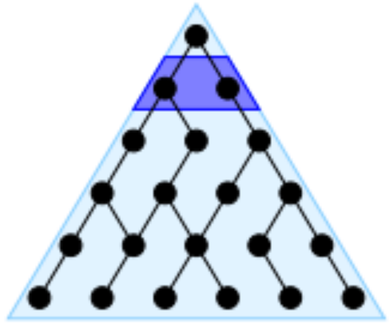
AF p

globally p



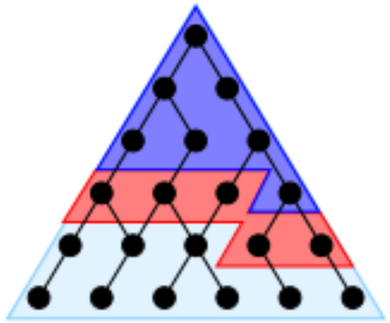
AG p

next p

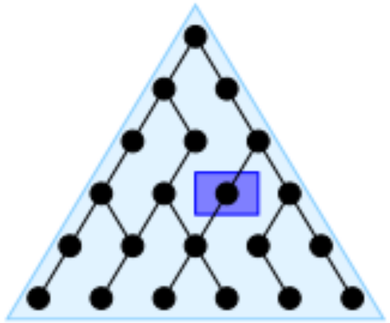


AX p

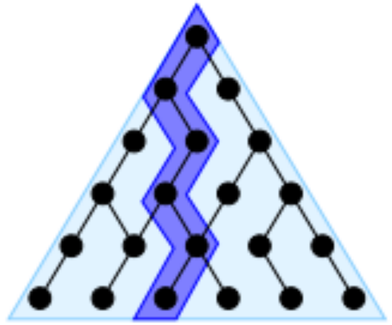
p until q



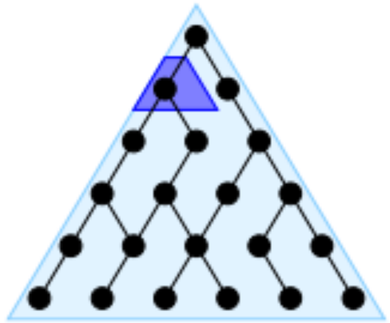
A [p U q]



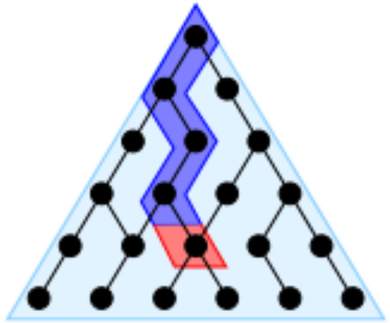
EF p



EG p



EX p



E [p U q]

Examples from Wikipedia

- Let "P" mean "I like chocolate" and Q mean "It's warm outside."
- $\mathbf{AG}.\mathrm{P}$
  - "I will like chocolate from now on, no matter what happens."
- $\mathbf{EF}.\mathrm{P}$
  - "It's possible I may like chocolate some day, at least for one day."
- $\mathbf{AF}.\mathbf{EG}.\mathrm{P}$
  - “It's always possible (AF) that I will suddenly start liking chocolate for the rest of time." (Note: not just the rest of my life, since my life is finite, while $\mathbf{G}$ is infinite).

Examples from Wikipedia

- $\mathbf{EG}.\mathbf{AF}.\mathrm{P}$
  - "This is a critical time in my life. Depending on what happens (E), it's possible that for the rest of time (G), there will always be some time in the future (AF) when I will like chocolate. However, if the wrong thing happens next, then all bets are off and there's no guarantee about whether I'll ever like chocolate."

- $\mathbf{A}(\mathrm{P}\mathbf{U}\mathrm{Q})$
  - "From now until it's warm outside, I will like chocolate every single day. Once it's warm outside, all bets are off as to whether I'll like chocolate anymore. Oh, and it's guaranteed to be warm outside eventually, even if only for a single day."

- $\mathbf{E}((\mathbf{EX}.\mathrm{P})\mathbf{U}(\mathbf{AG}.\mathrm{Q}))$
  - "It's possible that: there will eventually come a time when it will be warm forever (AG.Q) and that before that time there will always be *some* way to get me to like chocolate the next day (EX.P)."

Interesting Properties

- We would like to *verify* or check some relevant properties
  - Safety: Some proposition cannot be true (bad things cannot happen).
  - Liveness: Some proposition will eventually be true (good things will happen).
  - Fairness: Constrains the execution paths. E.g. every state is reached (Absolute fairness), every enabled state is executed (Strong fairness)

Safety Property Examples

AG ¬(temp > 1000)

AG ¬(NS_GREEN л **AX** EW_GREEN)

Liveness Property Examples

AF win_lottery

AG (requestQuit → AF quit)

Fairness Property Example

AG (**AF** skytrain_arrives)

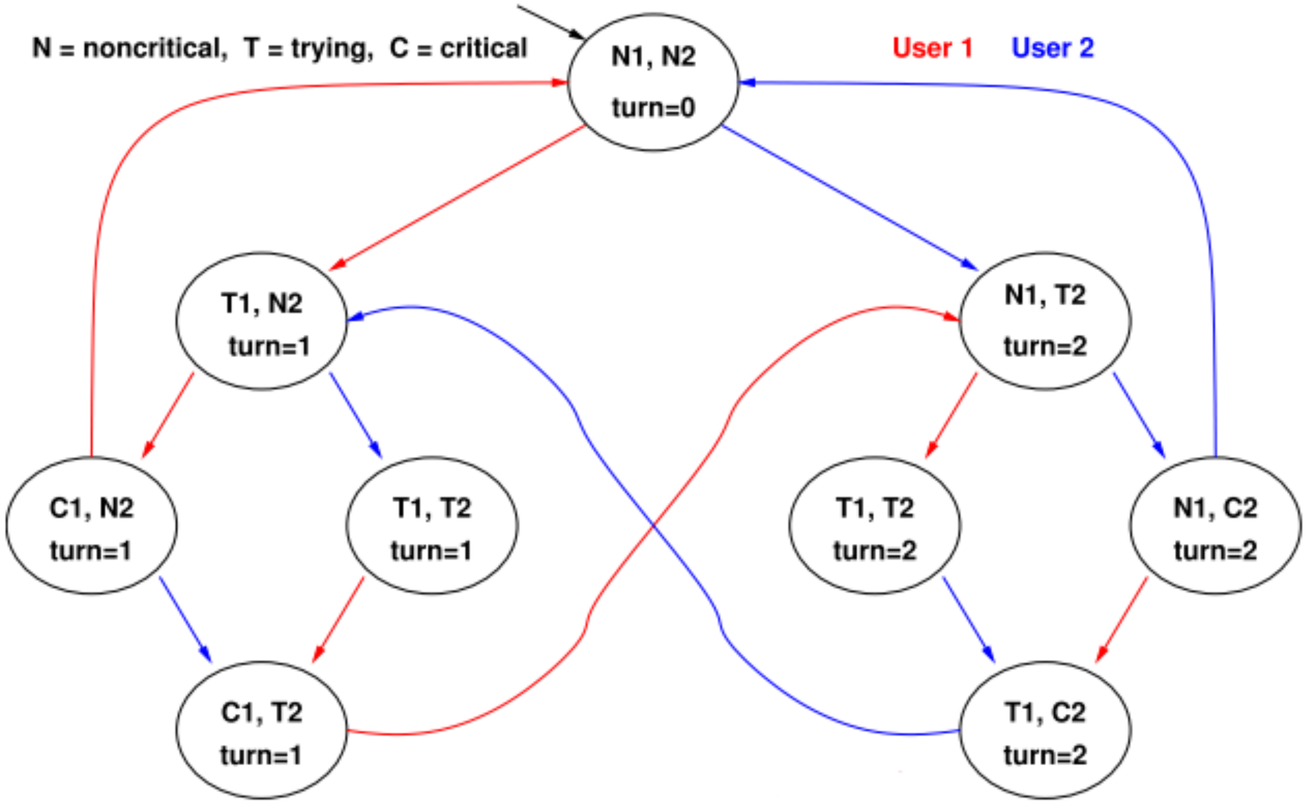
How is this different from:

AF skytrain_arrives

?

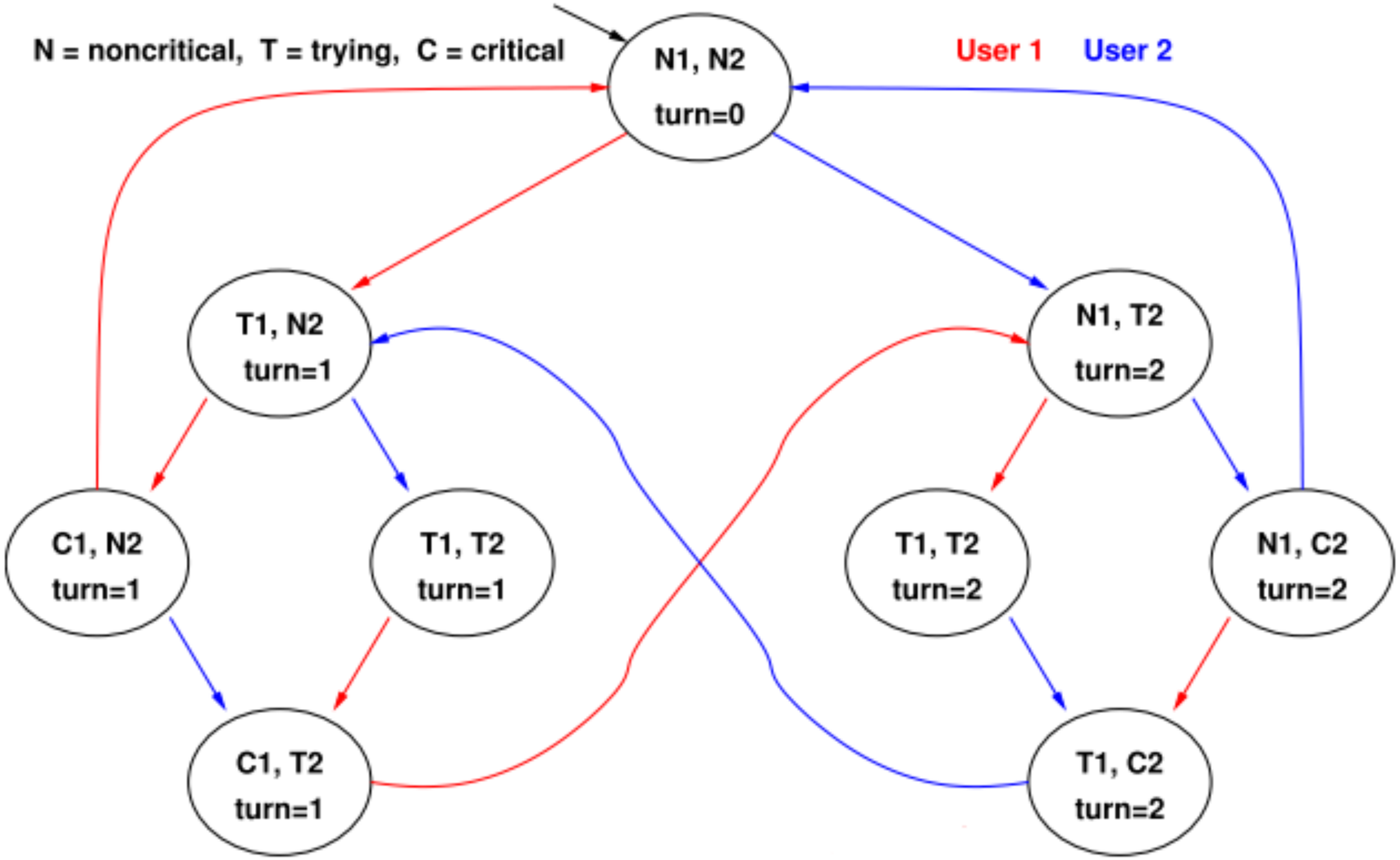
Code:

```
while(true) {
        nonCritical();
        obtainLock();
        criticalSection();
        releaseLock();
}
```

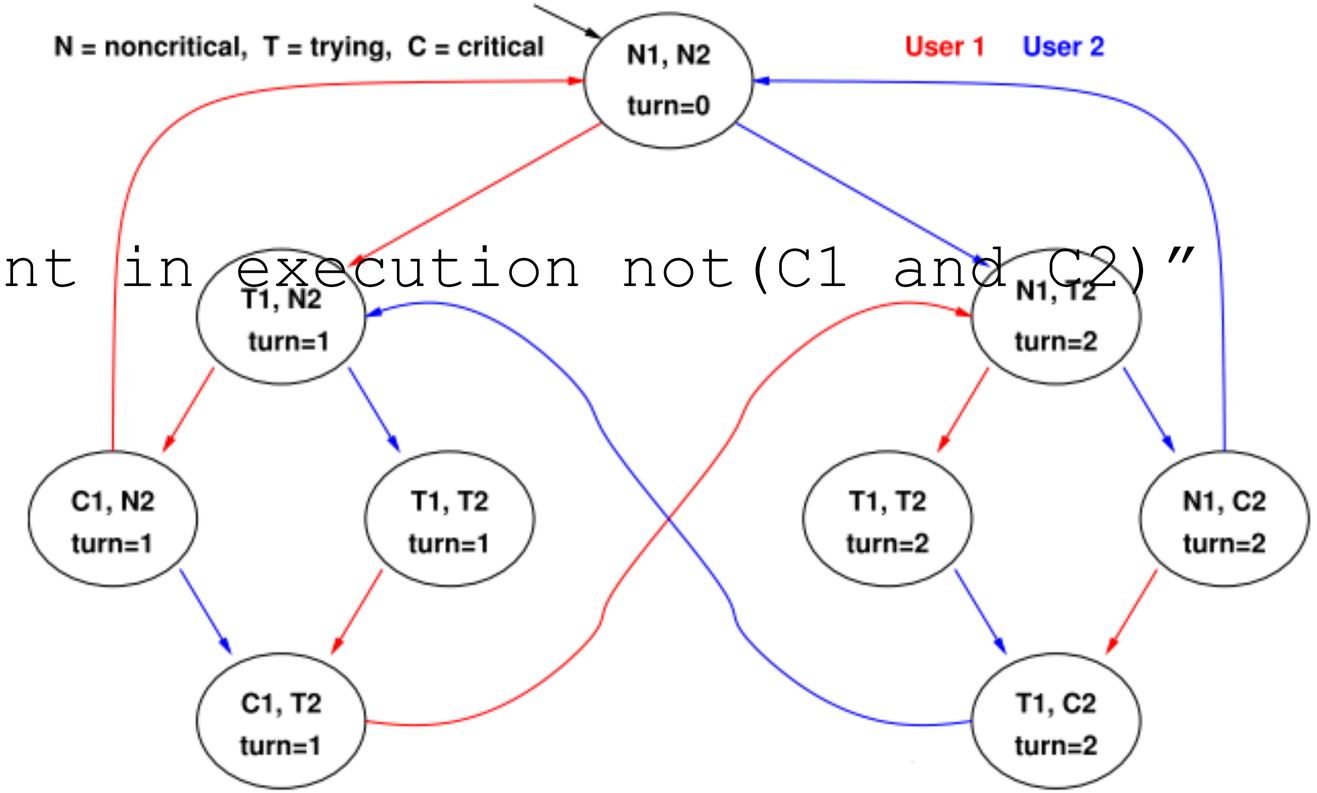


Model Checker: internal FSM representation assuming two processes

Modeling Concurrent States in a single machine

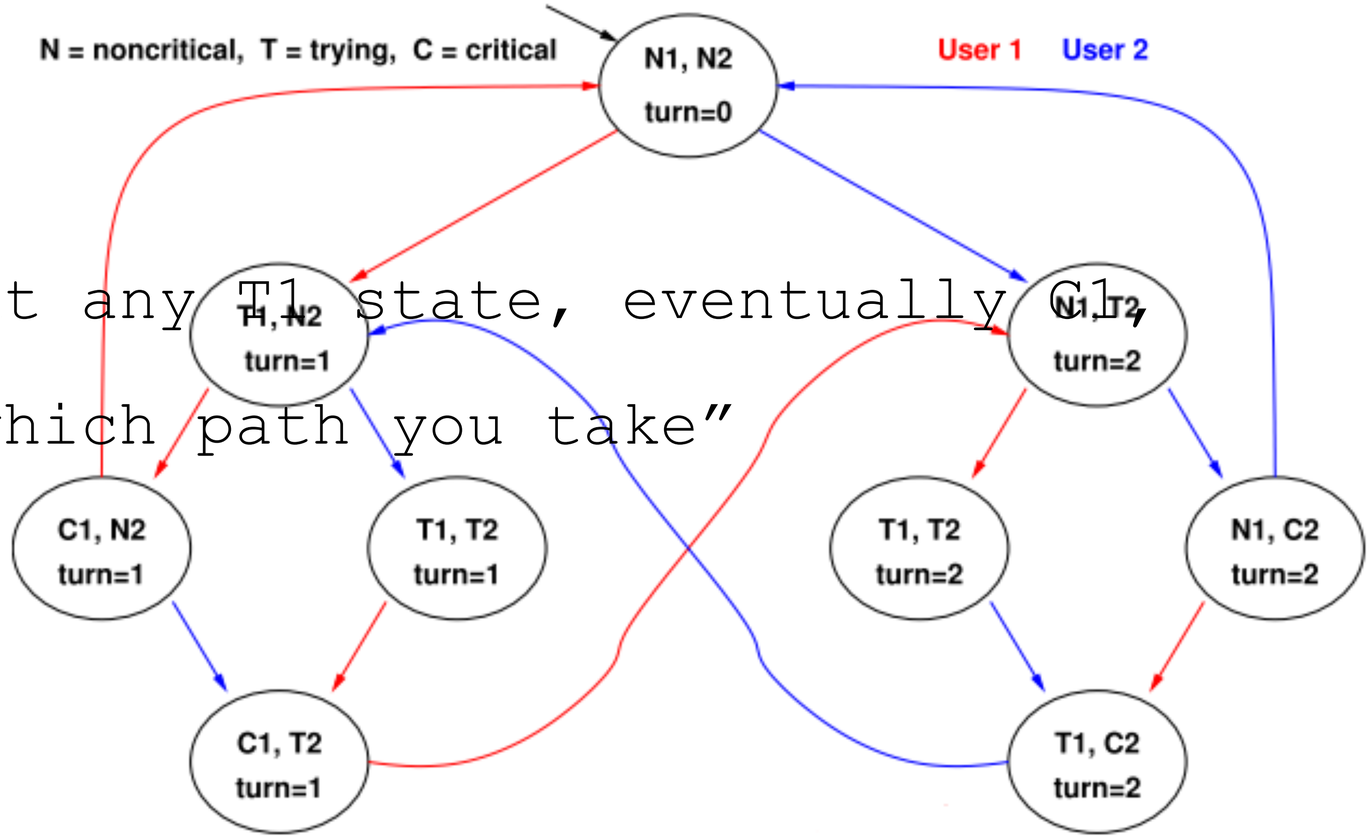


AG ¬(C1 ∧ C2) ?



"At every point in execution not(C1 and C2)"

AG (*T1* → AF *C1*) ?



"Starting at any T1 state, eventually C1

no matter which path you take"

AG (AF *C1*) ?

"Starting at any state, eventually C1, no matter which path you take"