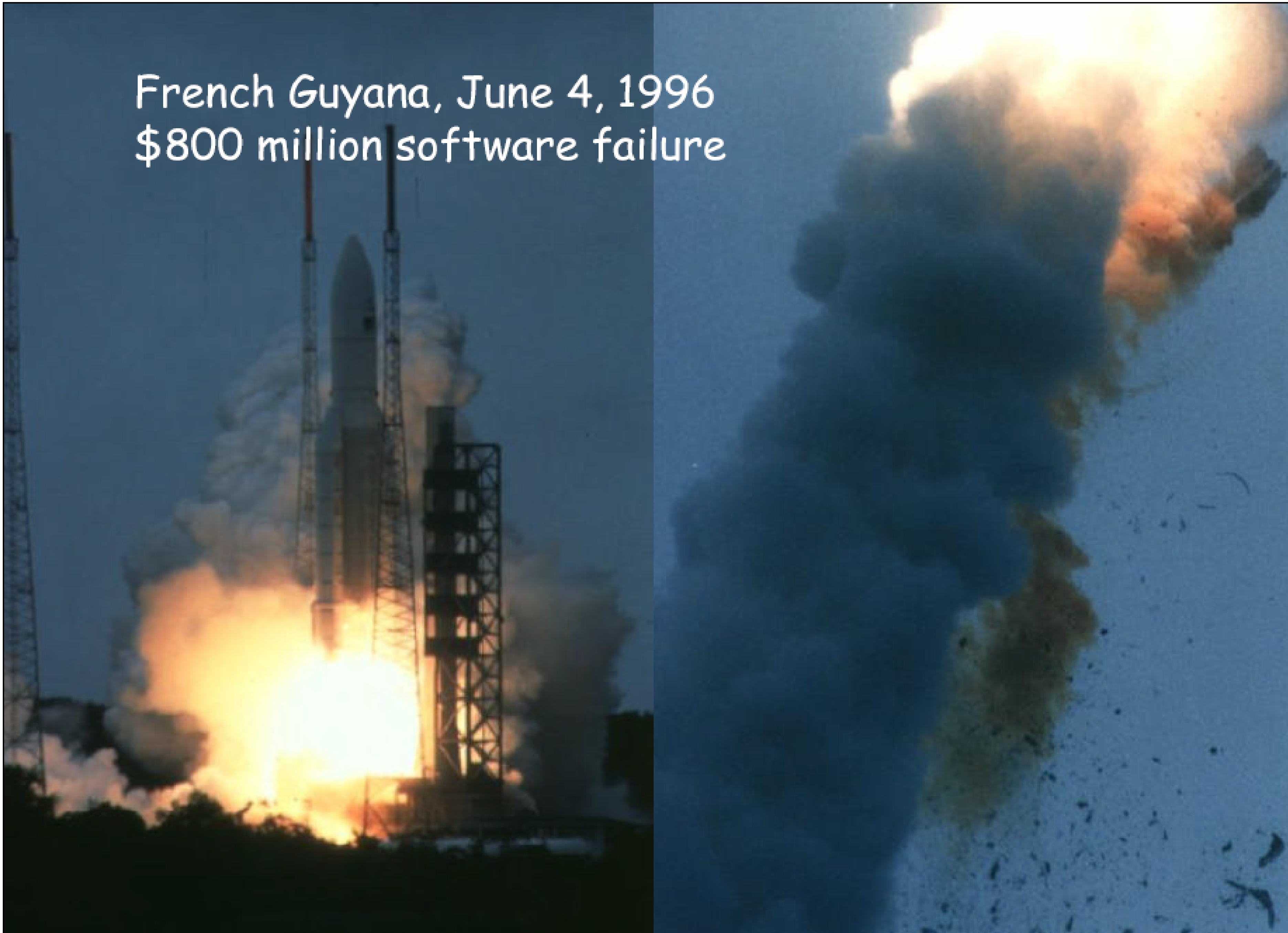
Computer-Aided Verification

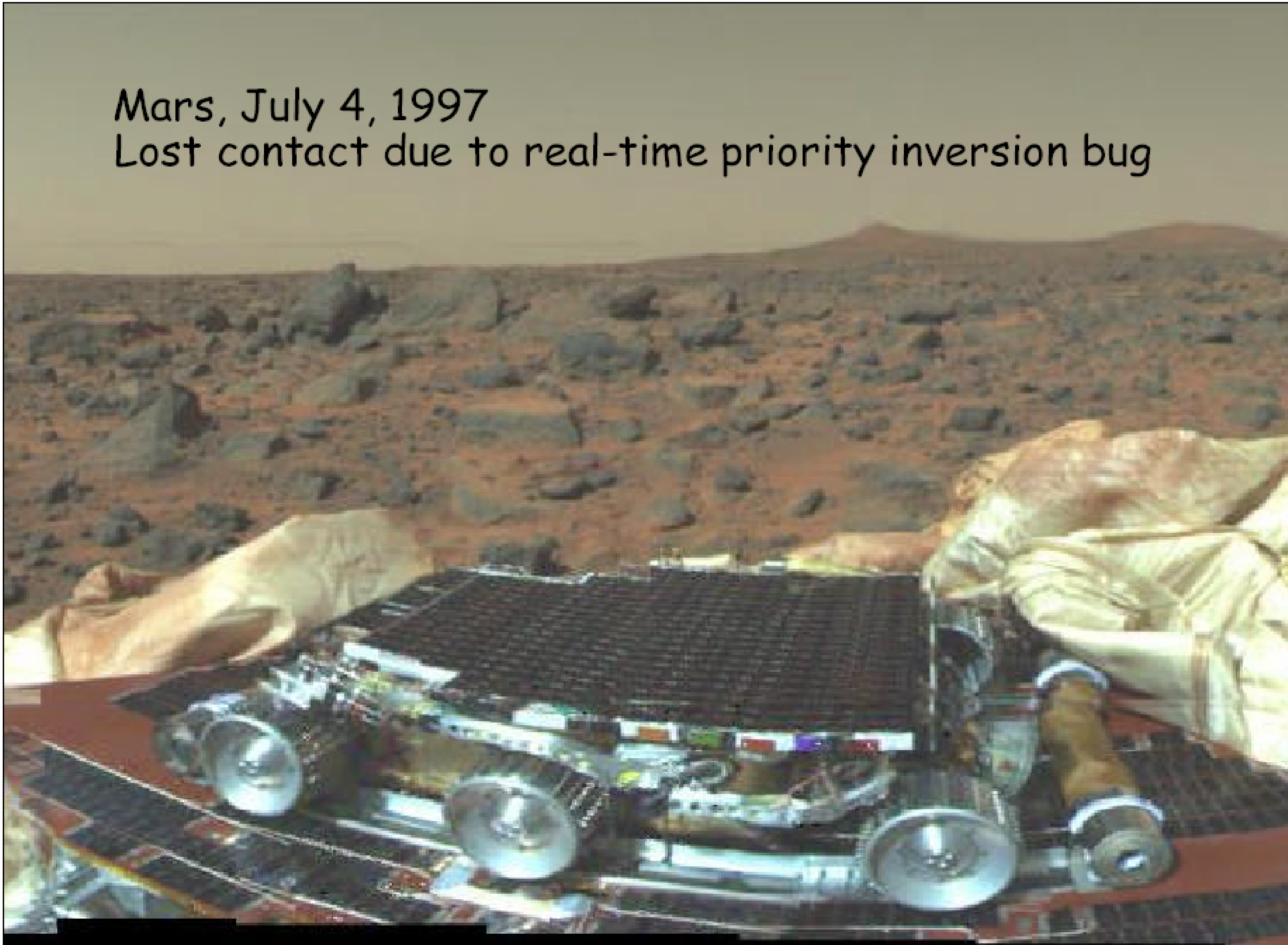
Topics

- Introduction
- Foundations
  - Logics
  - Transition System
- Methodologies
  - Temporal logics and Model checking
  - Theorem proving and Program verification
  - Process Algebra
  - Embedded Systems
  - VDM、Z、B、SDL
  - UML and State Chart

Topics

- Applications、Case study and Tools
  - HDL/PVS
  - Red and Uppall
  - CUDD / SMV / NuSMV
  - Statemate Rapsody


French Guyana, June 4, 1996
$800 million software failure


Mars, July 4, 1997
Lost contact due to real-time priority inversion bug


777
BOEING 777-300
$4 billion development effort
> 50% system integration & validation cost


400 horses
100 microprocessors
M NH 2519

```
*** STOP: 0x00000019 (0x00000000,0xC00E0FF0,0xFFFFEFD4,0xC0000000)
BAD_POOL_HEADER

CPUID:GenuineIntel 5.2.c irql:1f  SYSVER 0xf0000565

Dll Base DateStmp - Name                Dll Base DateStmp - Name
80100000 3202c07e - ntoskrnl.exe        80010000 31ee6c52 - hal.dll
80001000 31ed06b4 - atapi.sys           80006000 31ec6c74 - SCSIPORT.SYS
802c6000 31ed06bf - aic78xx.sys         802cd000 31ed237c - Disk.sys
802d1000 31ec6c7a - CLASS2.SYS          8037c000 31eed0a7 - Ntfs.sys
fc698000 31ec6c7d - Floppy.SYS          fc6a8000 31ec6ca1 - Cdrom.SYS
fc90a000 31ec6df7 - Fs_Rec.SYS          fc9c9000 31ec6c99 - Null.SYS
fc864000 31ed868b - KSecDD.SYS          fc9ca000 31ec6c78 - Beep.SYS
fc6d8000 31ec6c90 - i8042prt.sys        fc86c000 31ec6c97 - mouclass.sys
fc874000 31ec6c94 - kbdclass.sys        fc6f0000 31f50722 - VIDEOPORT.SYS
feffa000 31ec6c62 - mga_mil.sys         fc890000 31ec6c6d - vga.sys
fc708000 31ec6ccb - Msfs.SYS            fc4b0000 31ec6cc7 - Npfs.SYS
fefbc000 31eed262 - NDIS.SYS            a0000000 31f954f7 - win32k.sys
fefa4000 31f91a51 - mga.dll             fec31000 31eedd07 - Fastfat.SYS
feb8c000 31ec6e6c - TDI.SYS             feaf0000 31ed0754 - nbf.sys
feacf000 31f130a7 - tcpip.sys           feab3000 31f50a65 - netbt.sys
fc550000 31601a30 - el59x.sys           fc560000 31f8f864 - afd.sys
fc718000 31ec6e7a - netbios.sys         fc858000 31ec6c9b - Parport.sys
fc870000 31ec6c9b - Parallel.SYS        fc954000 31ec6c9d - ParVdm.SYS
fc5b0000 31ec6cb1 - Serial.SYS          fea4c000 31f5003b - rdr.sys
fea3b000 31f7a1ba - mup.sys             fe9da000 32031abe - srv.sys

Address  dword dump   Build [1381]                            - Name
fec32d84 80143e00 80143e00 80144000 ffdff000 00070b02           - KSecDD.SYS
801471c8 80144000 80144000 ffdff000 c03000b0 00000001           - ntoskrnl.exe
801471dc 80122000 f0003fe0 f030eee0 e133c4b4 e133cd40           - ntoskrnl.exe
80147304 803023f0 0000023c 00000034 00000000 00000000           - ntoskrnl.exe

Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option.
```

The Blue Screen of Death (BSOD)

More BSOD Embarrassments

The quest for correctness

"It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."

- Rapidly increasing *integration of IT* in different applications:
  - embedded systems
  - e-banking and e-shopping
  - transportation systems
- Reliability increasingly depends on hard- and software *integrity*
- Defects can be *fatal* and extremely *costly*
  - products subject to mass-production
  - safety-critical systems

What is system verification?

System verification amounts to check whether a system fulfills the qualitative requirements that have been identified.

Verification $\neq$ Validation:

Verification = “check that we are building the thing *right*”

Validation = “check that we are building the *right* thing”

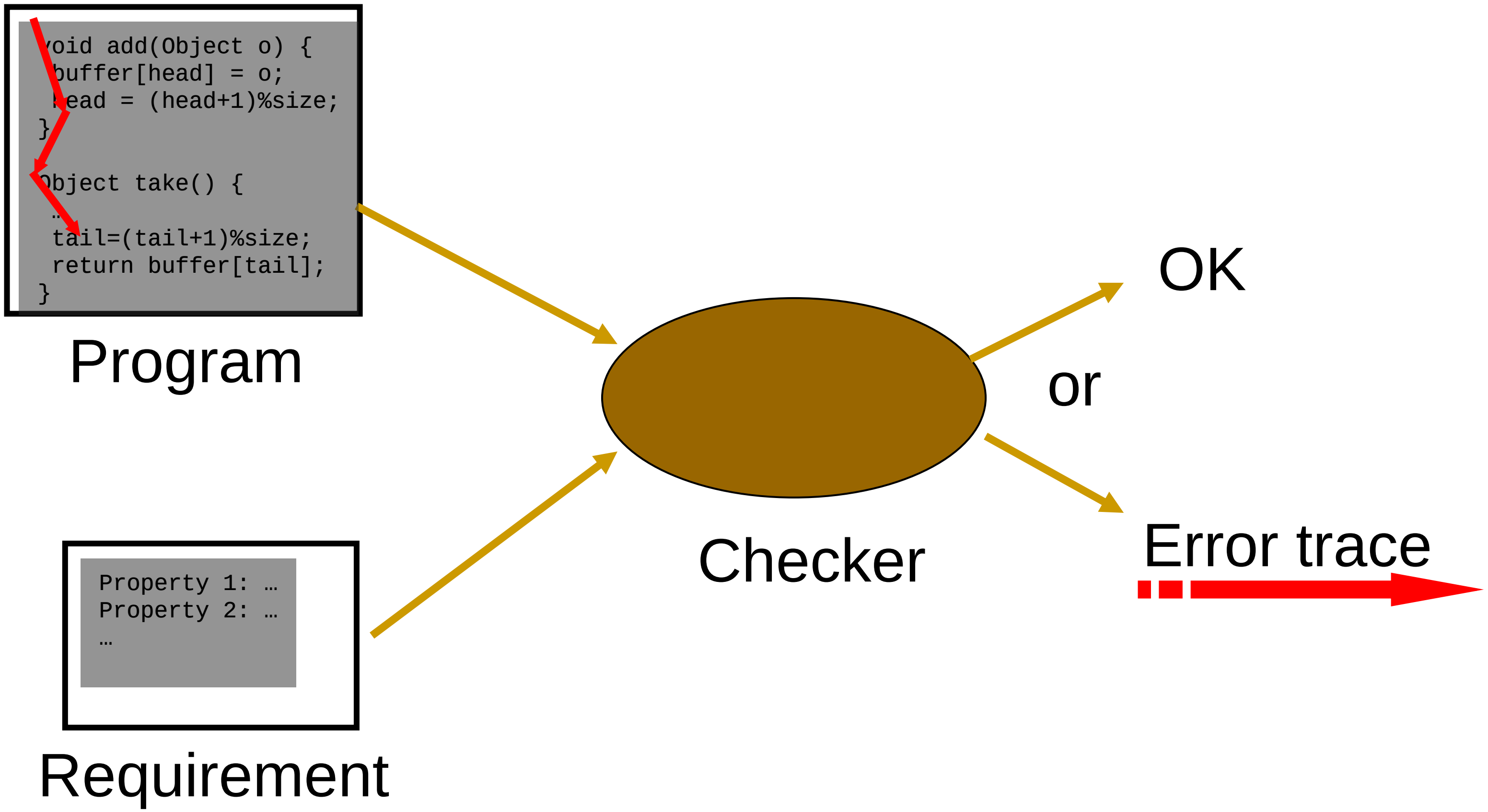
System verification = Model checking

Model checking:

Decision procedures for checking if a given Kripke structure is a model for a given formula of a modal logic.

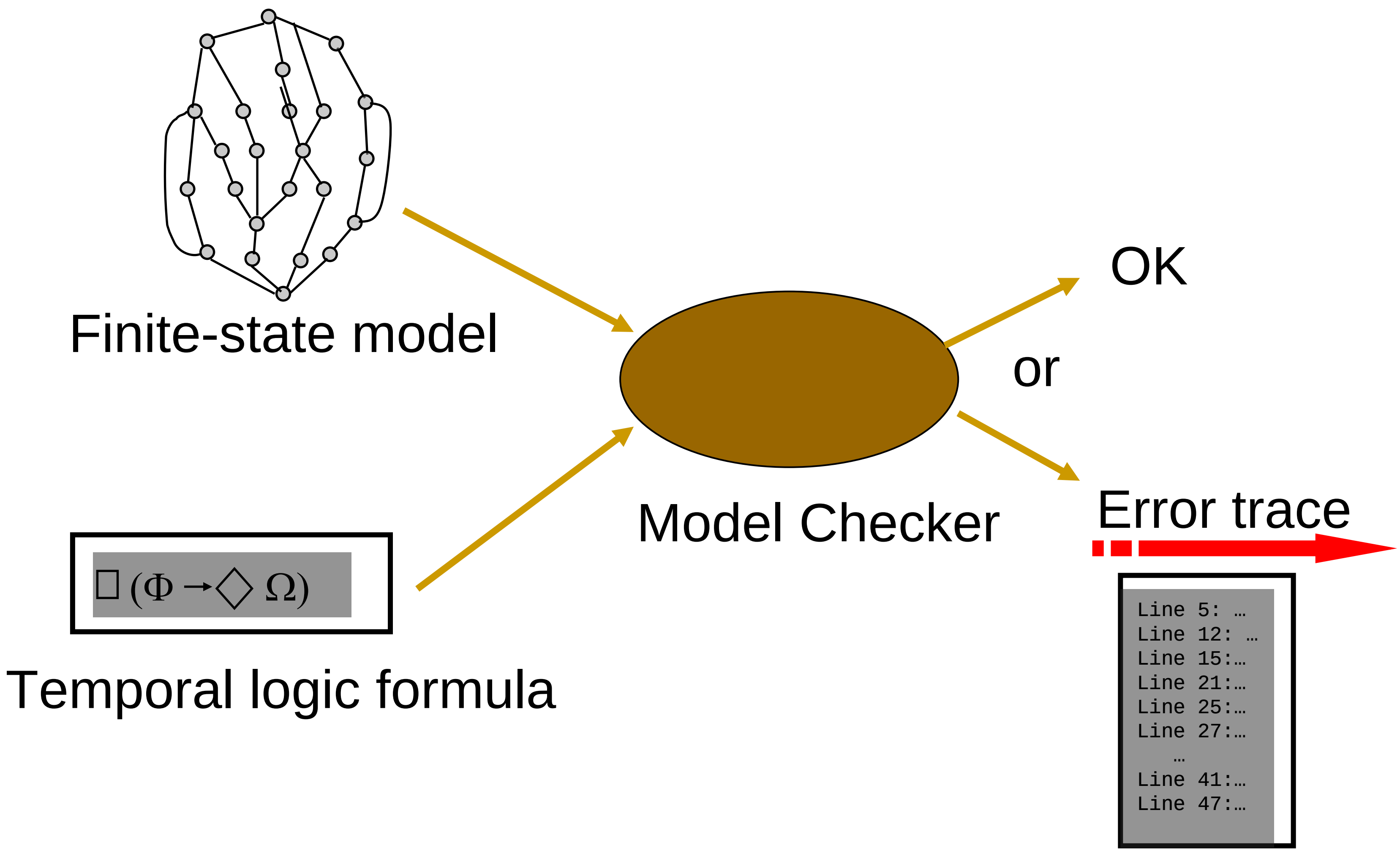
- Because the dynamics of a discrete system can be captured by a Kripke structure.
- Because some dynamic properties of a discrete system can be stated in modal logics.

The Dream



void add(Object o) {
buffer[head] = o;
head = (head+1)%size;
}
Object take() {
…
tail=(tail+1)%size;
return buffer[tail];
}
Program
Property 1: …
Property 2: …
…
Requirement
Checker
OK
or
Error trace

Model Checking



Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems.
- Developed independently by **Clarke and Emerson** and by **Queille and Sifakis** in early 1980's.
- **Specifications** are written in **propositional temporal logic.**
- Verification procedure is an **exhaustive search of the state space** of the design.

Why use Model Checking?

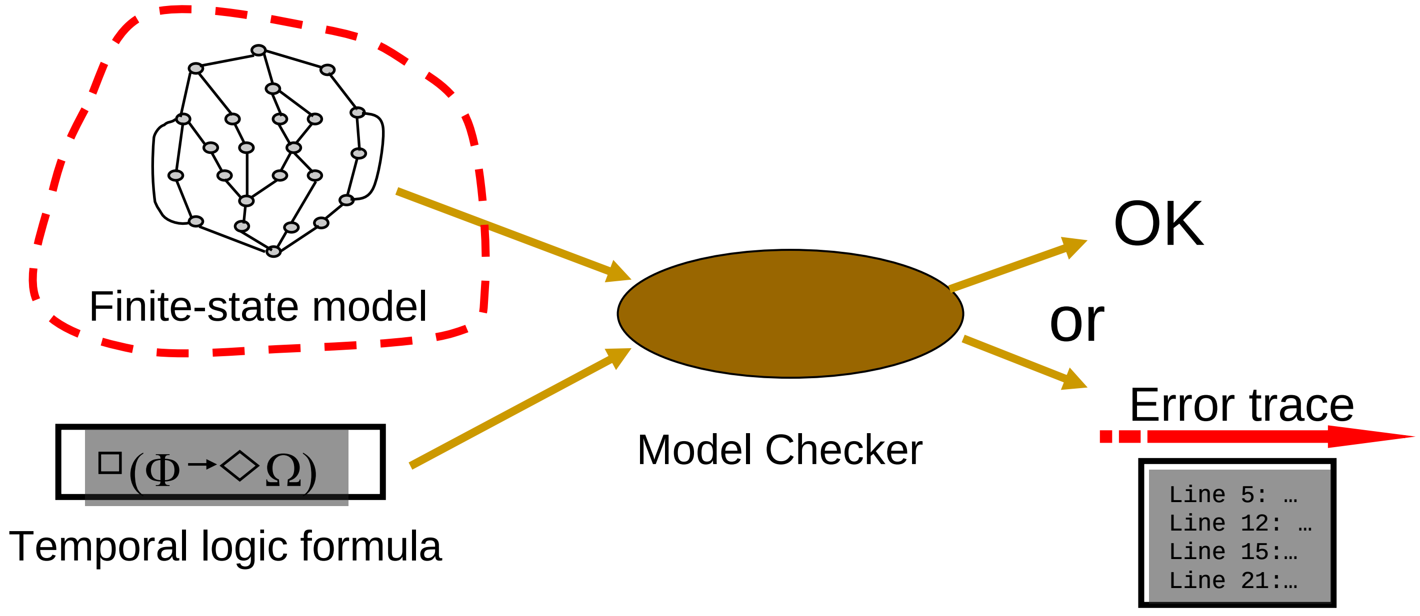
- **<u>Automatically check</u>, e.g.,**
  - **invariants, simple safety & liveness properties**
  - **absence of dead-lock and live-lock,**
  - **complex event sequencing properties,**

> "Between the window open and the window close, button X can be pushed at most twice."

- **In contrast to testing, <u>gives complete coverage</u> by exhaustively exploring all paths in system,**
- **It's been <u>used for years with good success</u> in hardware and protocol design**

This suggests that model-checking can <u>complement</u> existing software quality assurance techniques.

What makes model-checking software difficult?



Problems using existing checkers:

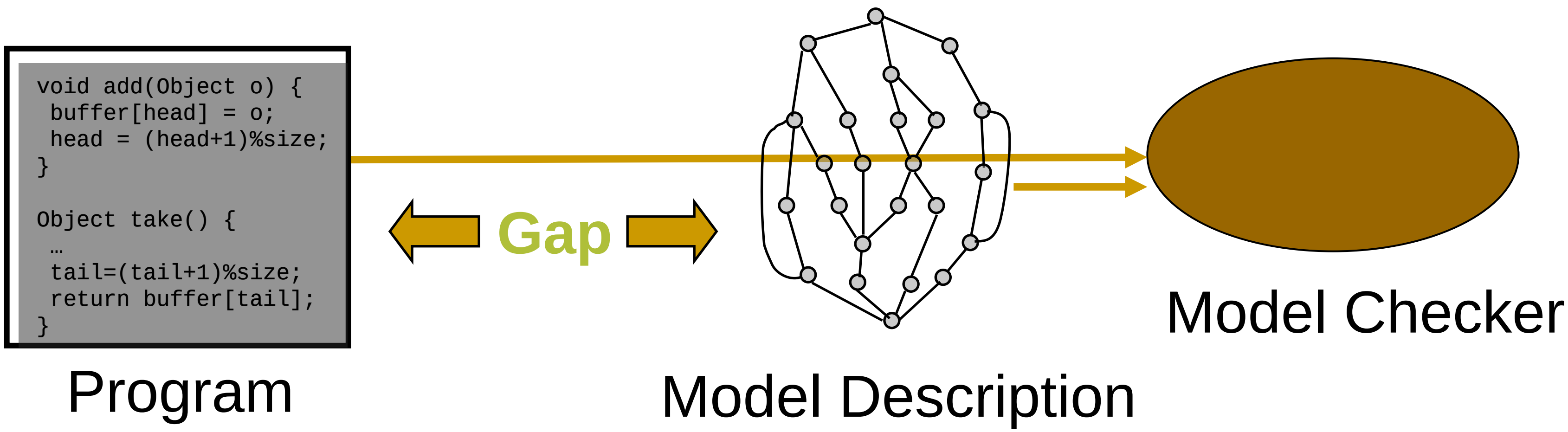
- **Model construction**

State explosion

Property specification

Output interpretation

Model Construction Problem



- **Semantic gap:**

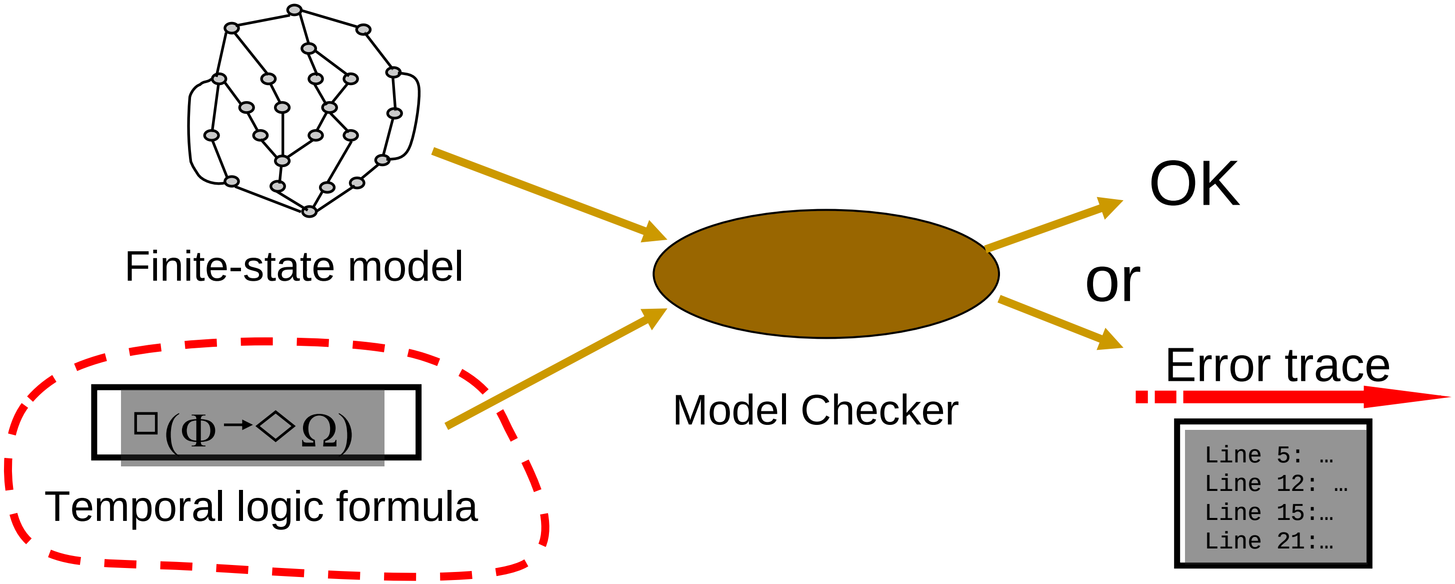
  Programming Languages

  methods, inheritance, dynamic creation, exceptions, etc.

  Model Description Languages

  automata

What makes model-checking software difficult?



Problems using existing checkers:

- **Model construction**
- **State explosion**
- **Property specification**
- **Output interpretation**

Property Specification Problem

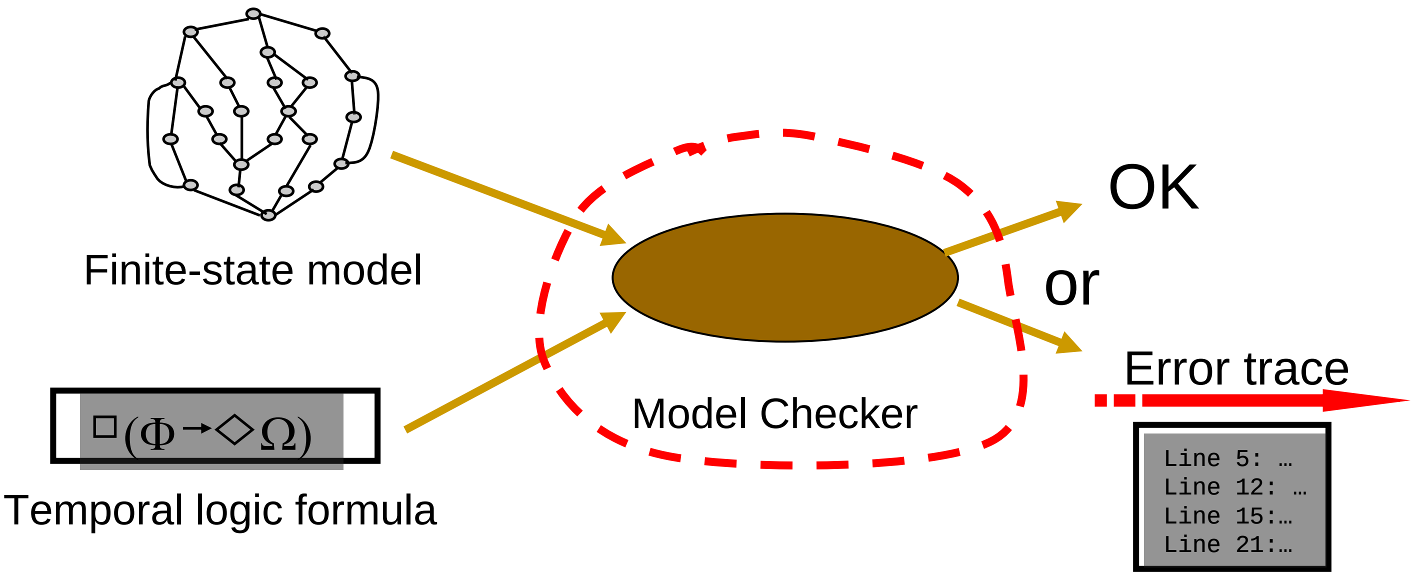
- **Difficult to formalize a requirement in temporal logic**

"Between the window open and the window close, button X can be pushed at most twice."

…is rendered in LTL as...

```
[]((open /\ <>close) ->
  ((!pushX /\ !close) U
    (close \/ ((pushX /\ !close) U
      (close \/ ((!pushX /\ !close) U
        (close \/ ((pushX /\ !close) U
          (close \/ (!pushX U close))))))))))
```

What makes model-checking software difficult?



Problems using existing checkers:

- **Model construction**
- **State explosion**
- **Property specification**
- **Output interpretation**

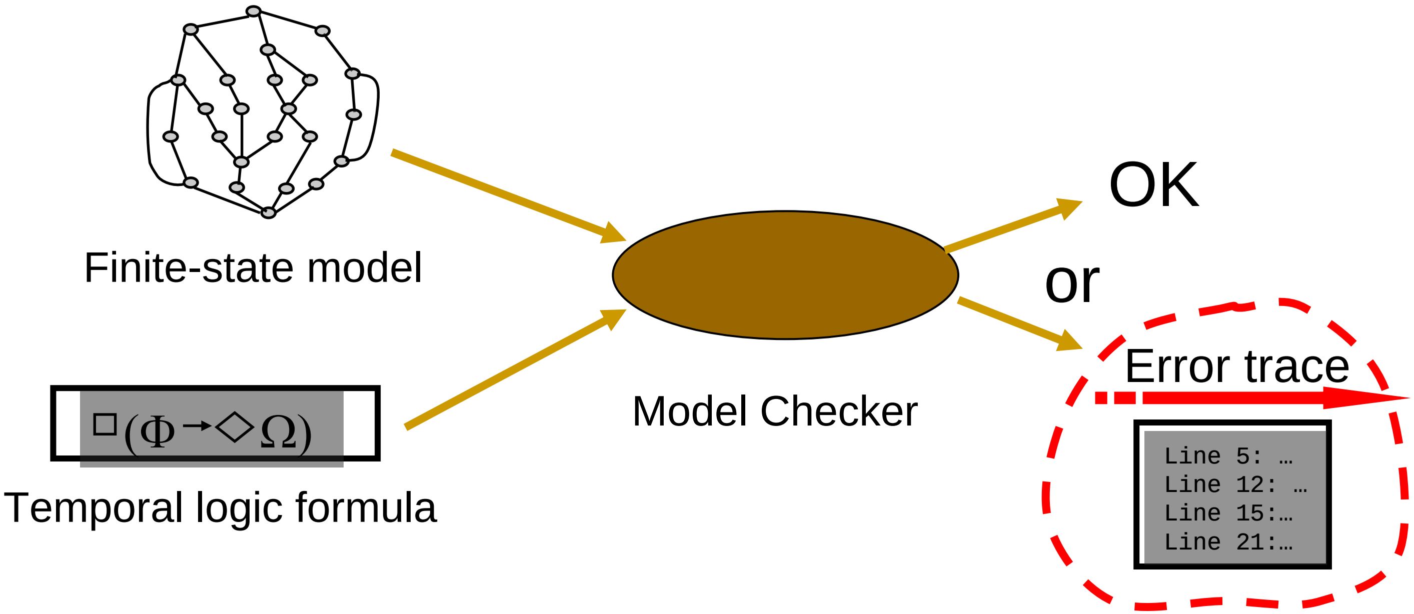
State Explosion Problem

Cost is exponential in the number of components

Bit x1,…,xN - - - ▸ 2^N states

- **Moore's law and algorithm advances can help**
  - **Holzmann: 7 days (1980) ==> 7 seconds (2000)**
- **Explosive state growth in software limits scalability**

What makes model-checking software difficult?



Problems using existing checkers:

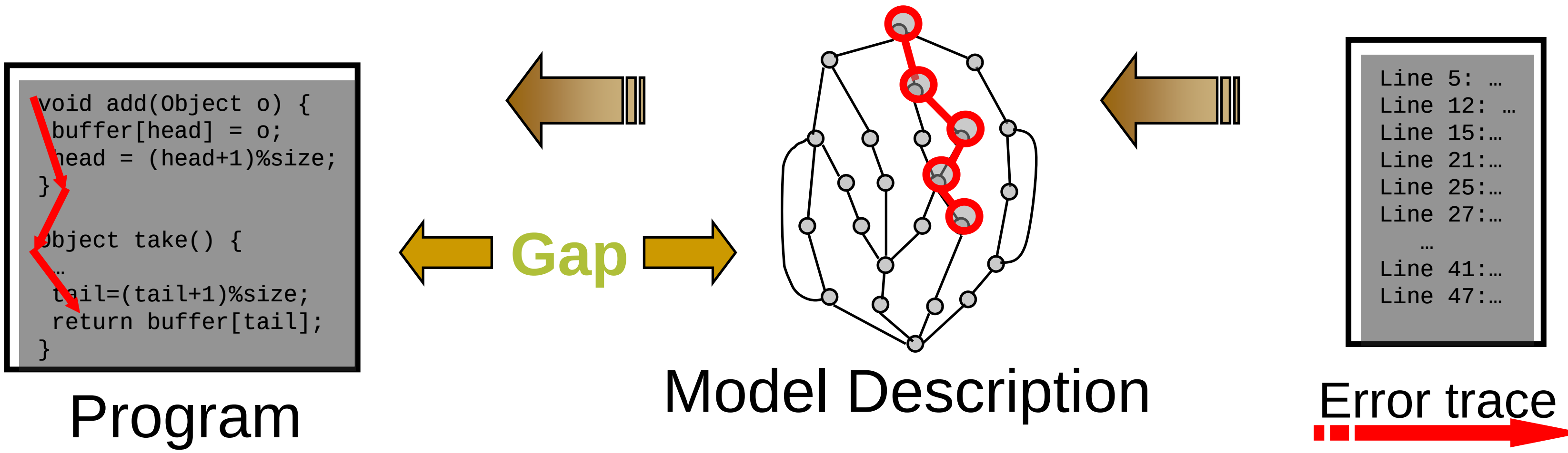
- **Model construction**

State explosion

Property specification

Output interpretation

Output Interpretation Problem



- **Raw error trace may be 1000's of steps long**

Must map line listing onto model description

Mapping to source is made difficult by

- **Semantic gap & clever encodings of complex features**
- **multiple optimizations and transformations**

Some Advantages of Model Checking

- No proofs!!!
- Fast
- Counterexamples
- No problem with partial specifications
- Logics can easily express many concurrency properties

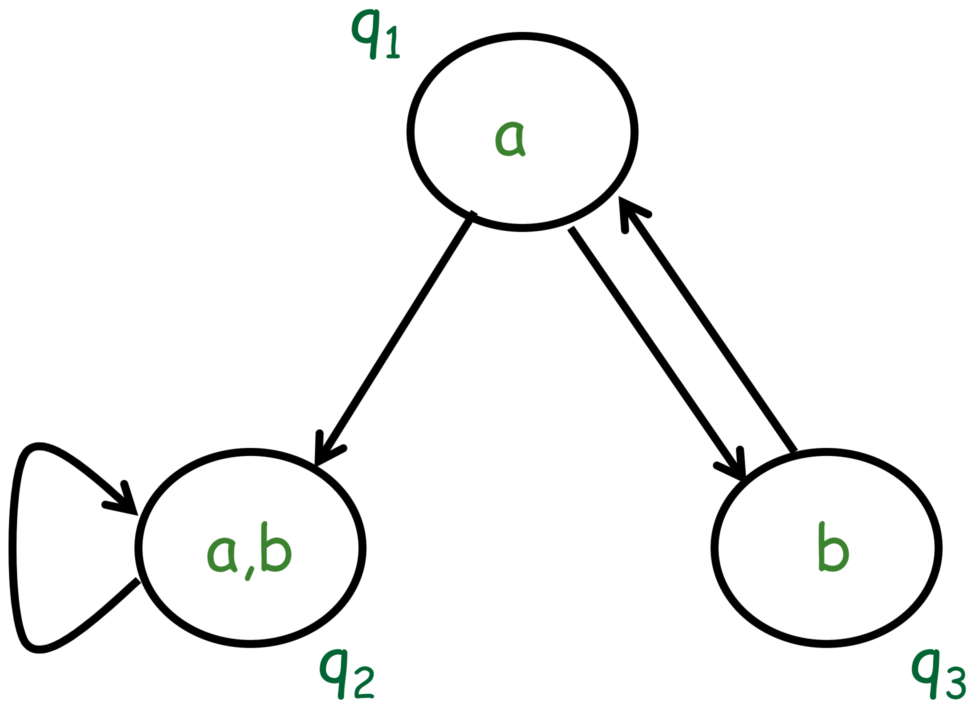
Main Disadvantage

- **State Explosion Problem:**
  - Too many processes
  - Data Paths
- **Much progress has been made on this problem recently!**

Model Checking Problem

- Let ***M*** be a **state-transition graph**.
- Let ***f*** be the **specification** *(system properties)* in temporal logic.
- Find all states ***s*** of ***M*** such that

$$M, s \models f.$$



q1
a
a,b
q2
b
q3

State-transition graph

Q	set of states	$\{q_1,q_2,q_3\}$
A	set of atomic observations	$\{a,b\}$
$\rightarrow \subseteq Q \times Q$	transition relation	$q_1 \rightarrow q_2$
$[\ ]: Q \rightarrow 2^A$	observation function	$[q_1] = \{a\}$

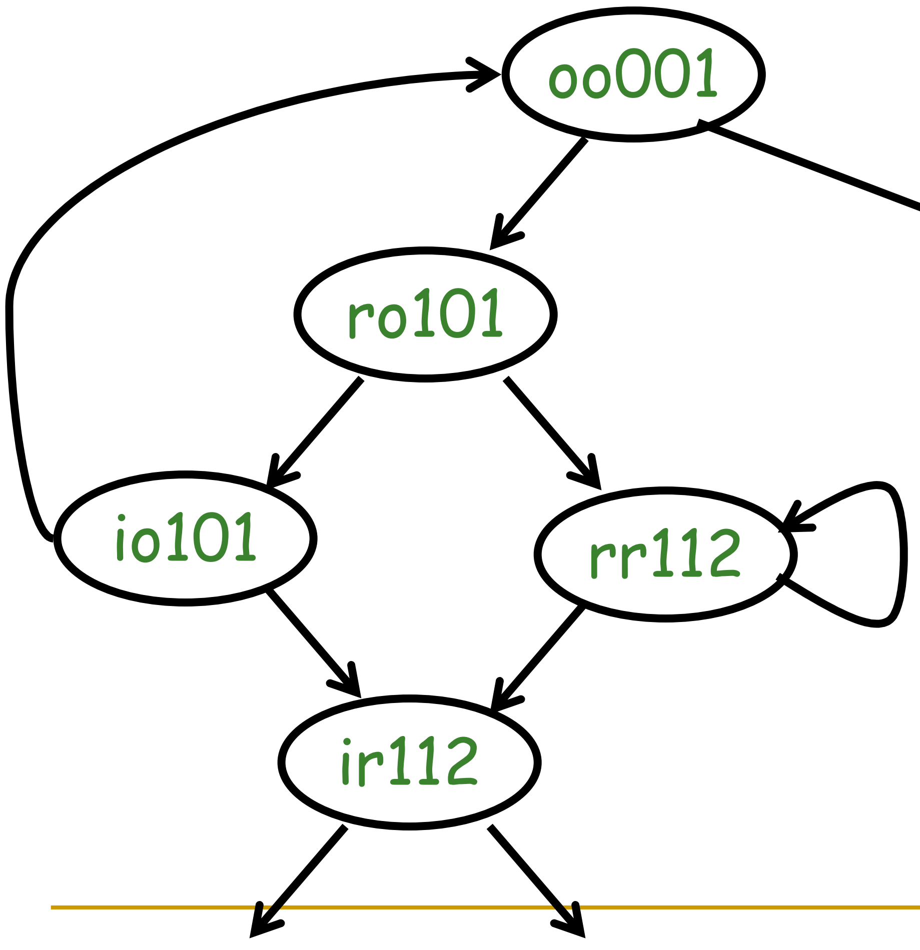
set of observations

Mutual-exclusion protocol

```
loop                                  ||   loop
   out:  x1 := 1; last := 1                   out:  x2 := 1; last := 2
   req:  await  x2 = 0  or  last = 2          req:  await  x1 = 0  or  last = 1
   in:    x1 := 0                             in:    x2 := 0
end loop.                                  end loop.

         P1                                          P2
```

The translation from a system description to a state-transition graph usually involves an exponential blow-up !!!



pc1: {o,r,i}
pc2: {o,r,i}
x1: {0,1}
x2: {0,1}
last: {1,2}

3·3·2·2·2 = 72 states

State-transition graphs are not necessarily finite-state, but they don't handle well:

- recursion (need pushdown models)
- environment interaction (need game models)
- process creation

Three important decisions when choosing system properties:

- prohibiting bad vs. desiring good behavior:
  safety vs. liveness
- may vs. must:
  branching vs. linear time
- operational vs. declarative:
  automata vs. logic

Safety vs. liveness

- Safety
  - something “bad” will never happen
- Liveness
  - something “good” will happen

(but we don’t know when)

Safety vs. liveness for sequential programs

induction on control flow

- Safety
  - the program will never produce a wrong result ("partial correctness")
- Liveness
  - the program will produce a result ("termination")

well-founded induction on data

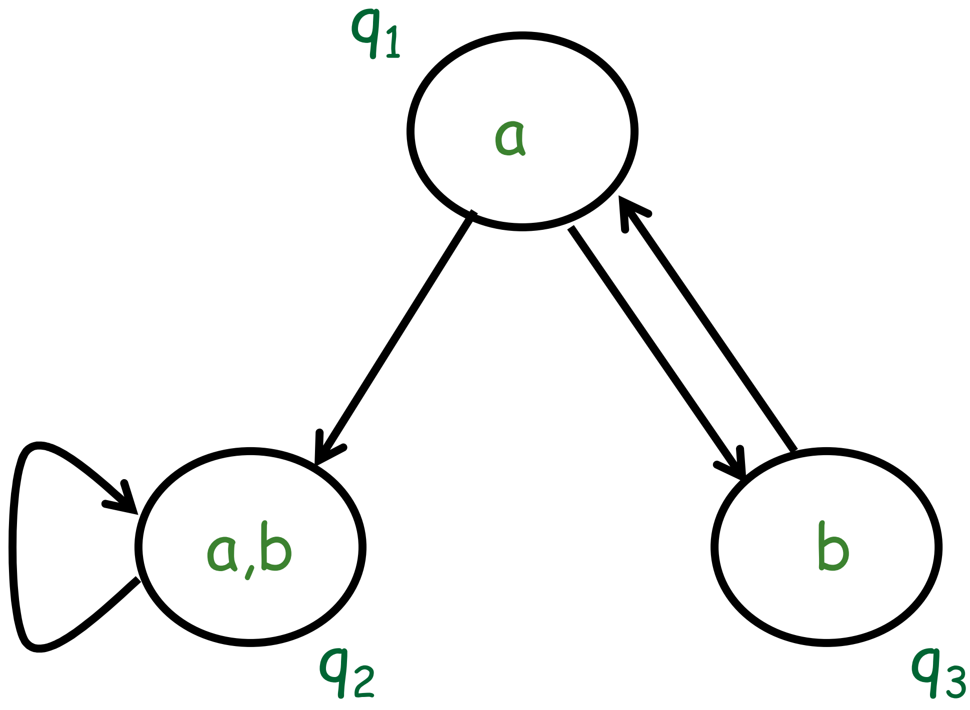
Safety vs. liveness for state-transition graphs

- Safety: those properties whose violation always has a finite witness

("if something bad happens on an infinite run, then it happens already on some finite prefix")

- Liveness: those properties whose violation never has a finite witness

("no matter what happens along a finite run, something good could still happen later")



Run: $q_1 \rightarrow q_3 \rightarrow q_1 \rightarrow q_3 \rightarrow q_1 \rightarrow q_2 \rightarrow q_2 \rightarrow$

Trace: a $\rightarrow$ b $\rightarrow$ a $\rightarrow$ b $\rightarrow$ a $\rightarrow$ a,b $\rightarrow$ a,b $\rightarrow$

State-transition graph $S = (Q, A, \rightarrow, [])$

Finite runs: finRuns(S) $\subseteq Q^*$

Infinite runs: infRuns(S) $\subseteq Q^\omega$

Finite traces: finTraces(S) $\subseteq (2^A)^*$

Infinite traces: infTraces(S) $\subseteq (2^A)^\omega$

This is much easier.

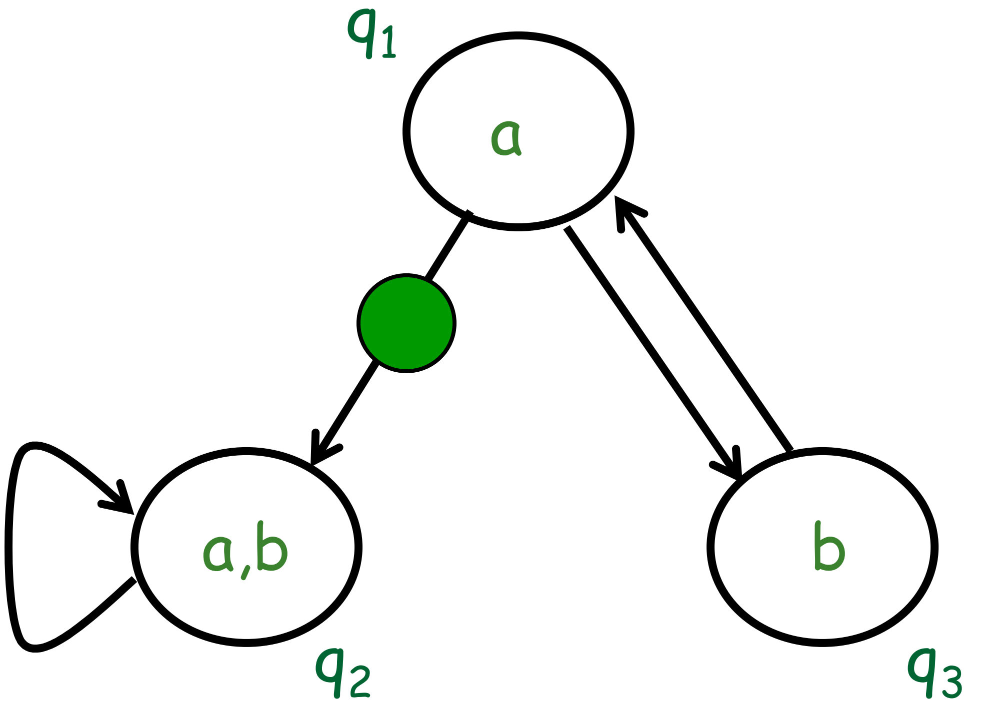
Safety: the properties that can be checked on finRuns

Liveness: the properties that cannot be checked on finRuns

(they need to be checked on infRuns)

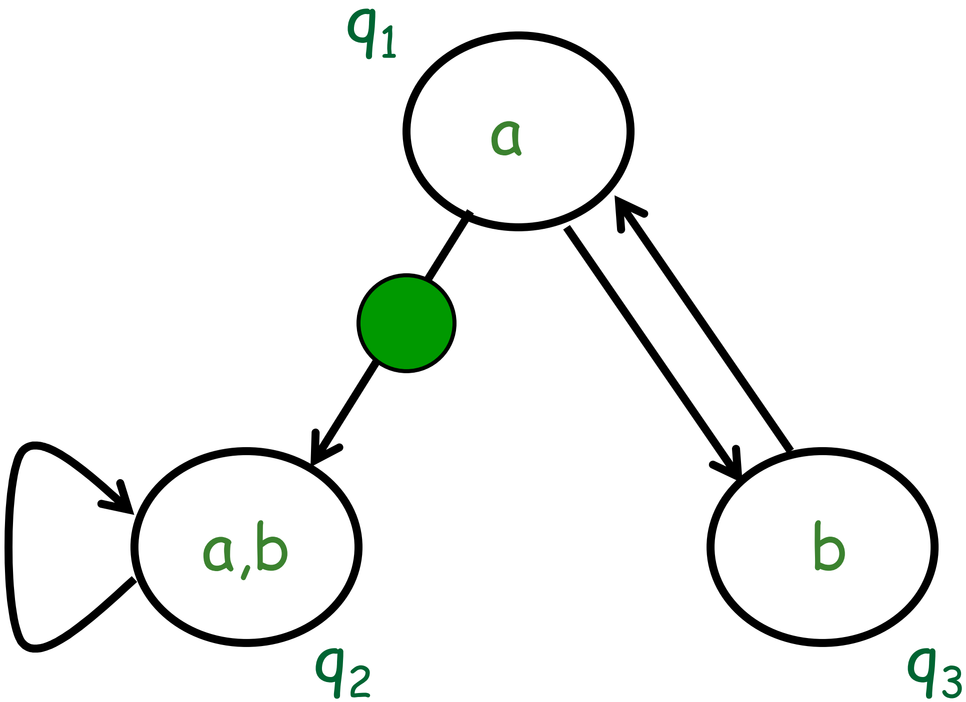
Example:

- Mutual exclusion
  - It cannot happen that both processes are in their critical sections simultaneously. Safety
- Bounded overtaking
  - Whenever process P1 wants to enter the critical section, then process P2 gets to enter at most once before process P1 gets to enter. Safety
- Starvation freedom
  - Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually. Liveness



Fairness constraint:

the green transition cannot be ignored forever

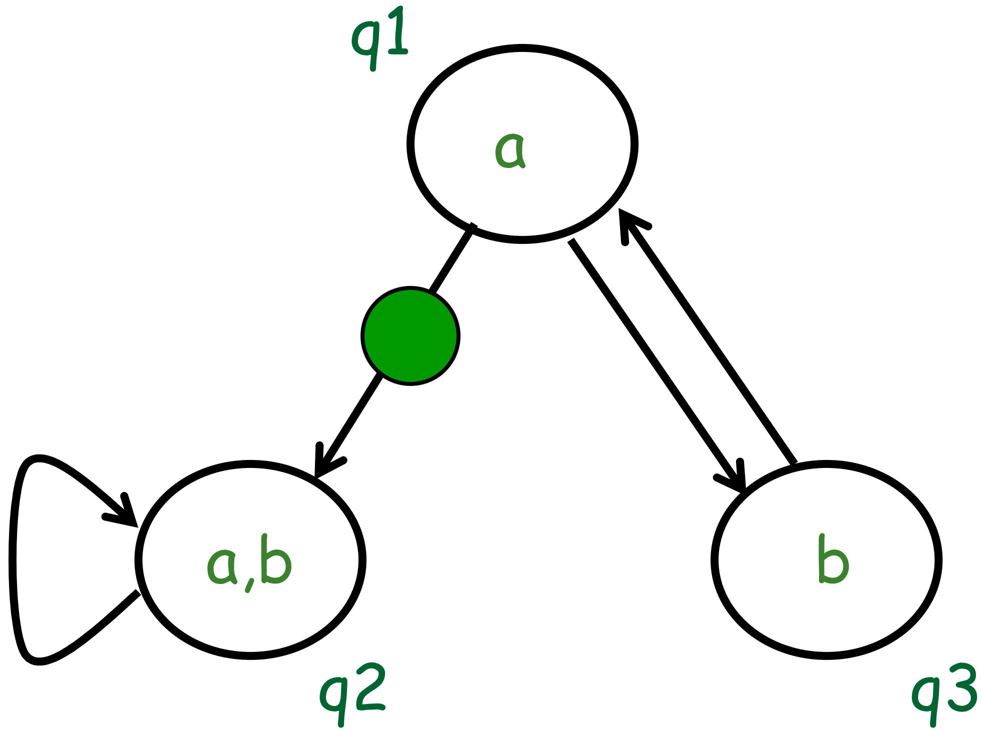


Without fairness: infRuns = $q_1 \, (q_3 \, q_1)^* \, q_2^\omega \cup (q_1 \, q_3)^\omega$

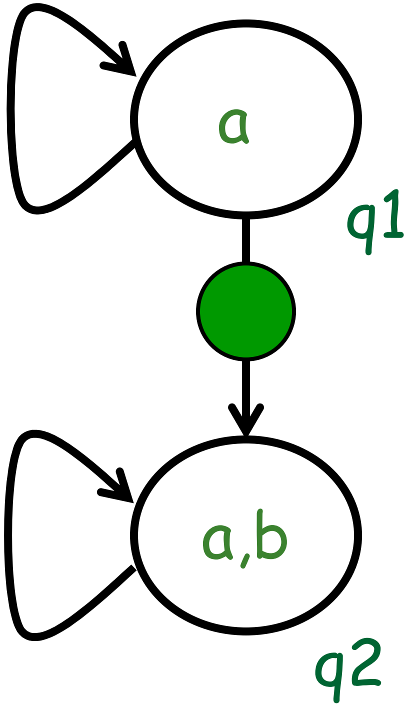
With fairness: infRuns = $q_1 \, (q_3 \, q_1)^* \, q_2^\omega$

Two important types of fairness

- Weak (Buchi) fairness
  - a specified set of transitions cannot be enabled forever without being taken
- Strong (Streett) fairness
  - a specified set of transitions cannot be enabled infinitely often without being taken



Strong fairness



Weak fairness

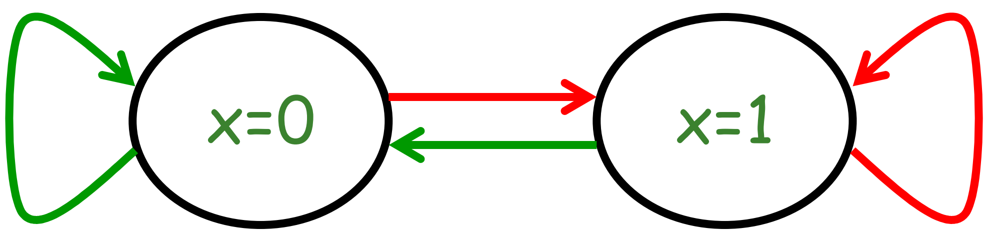
Fair state-transition graph $S = (Q, A, \rightarrow, [], WF, SF)$

- WF: set of weakly fair actions
- SF : set of strongly fair actions

where each action is a subset of $\rightarrow$

Weak fairness comes from modeling concurrency

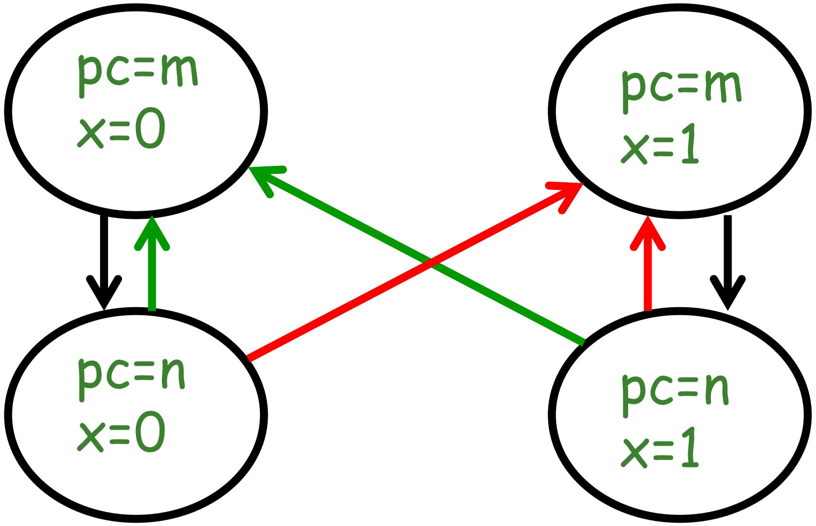
loop x:=0 end loop. || loop x:=1 end loop.



Weakly fair action
Weakly fair action

Strong fairness comes from modeling choice

```
loop    m:
        n:   x:=0 | x:=1
end loop.
```



Strongly fair action
Strongly fair action

Weak fairness vs. Strong fairness

- Weak fairness is sufficient for asynchronous models ("no process waits forever if it can move").
- Strong fairness is necessary for modeling synchronous interaction (rendezvous).
- Strong fairness makes model checking more difficult.

Fairness changes only infRuns, not finRuns.

$\Downarrow$

Fairness can be ignored for checking safety properties.

Two remarks

- The vast majority of properties to be verified are safety.
- While nobody will ever observe the violation of a true liveness property, fairness is a useful abstraction that turns complicated safety into simple liveness.

Branching vs. linear time

- Branching time
  - something may (or may not) happen
    (e.g., every req may be followed by grant)
- Linear time
  - something must (or must not) happen

    (e.g., every req must be followed by grant)

Fair state-transition graph $S = (Q, A, \rightarrow, [], WF, SF)$

Finite runs: finRuns(S) $\subseteq Q^*$

Infinite runs: infRuns(S) $\subseteq Q^\omega$

Finite traces: finTraces(S) $\subseteq (2^A)^*$

Infinite traces: infTraces(S) $\subseteq (2^A)^\omega$

Branching vs. linear time

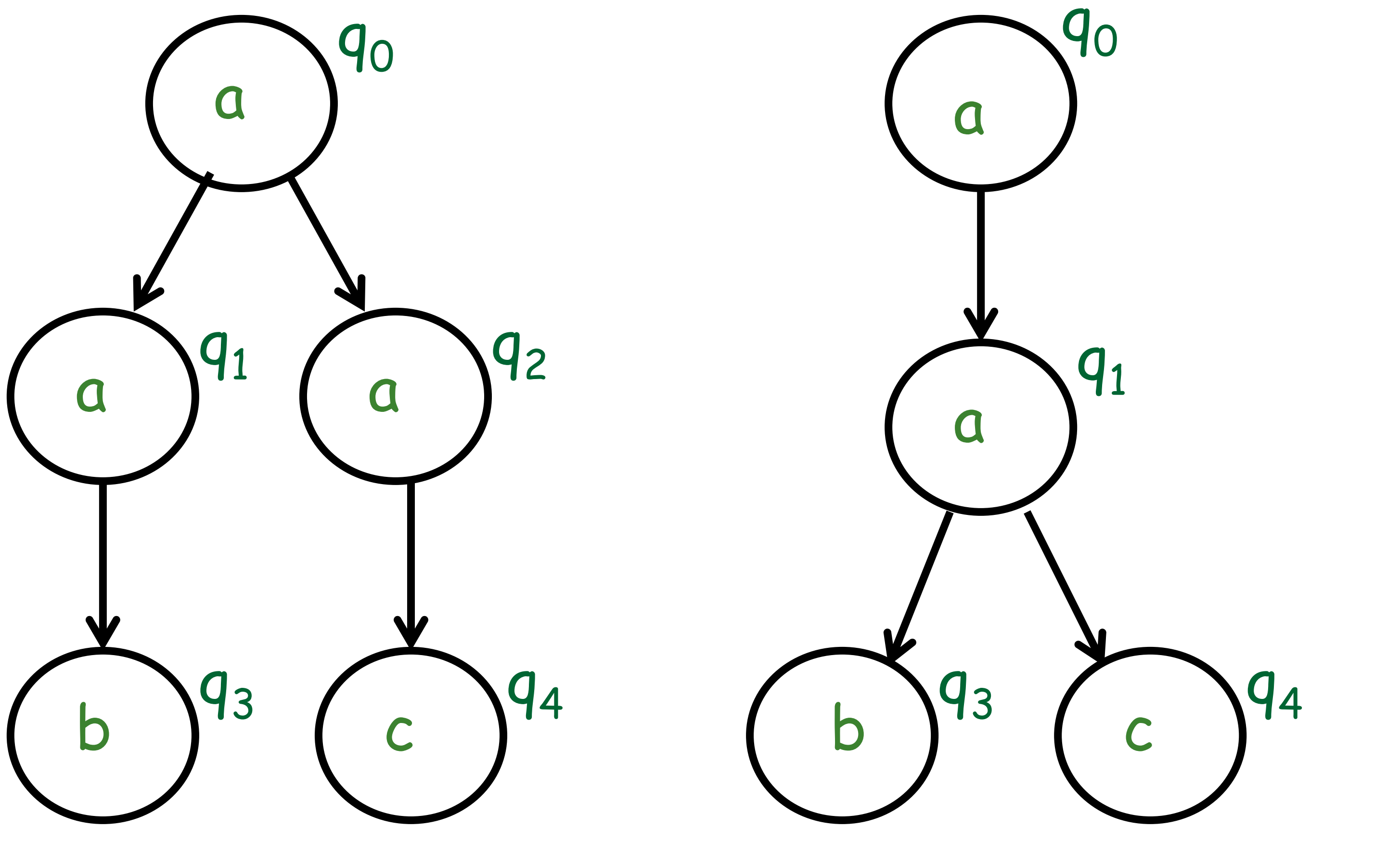
•Linear time:

the properties that can be checked on infTraces

•Branching time:

the properties that cannot be checked on infTraces

	Linear	Branching
Safety	finTraces	finRuns
Liveness	infTraces	infRuns



Same traces {aab, aac}

Different runs $\{q_0\ q_1\ q_3,\ q_0\ q_2\ q_4\}$, $\{q_0\ q_1\ q_3,\ q_0\ q_1\ q_4\}$

Linear

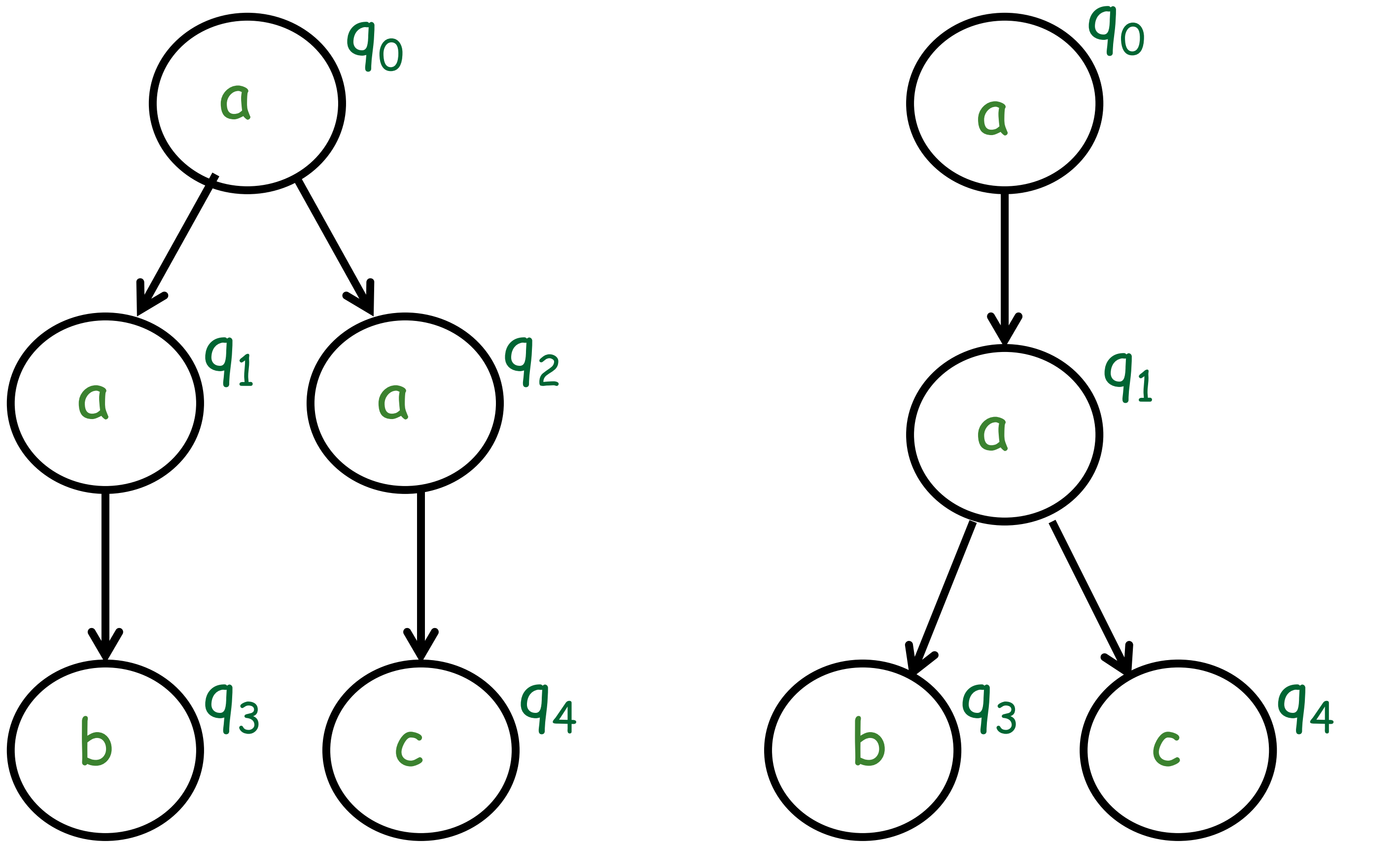
Observation a may occur.

||

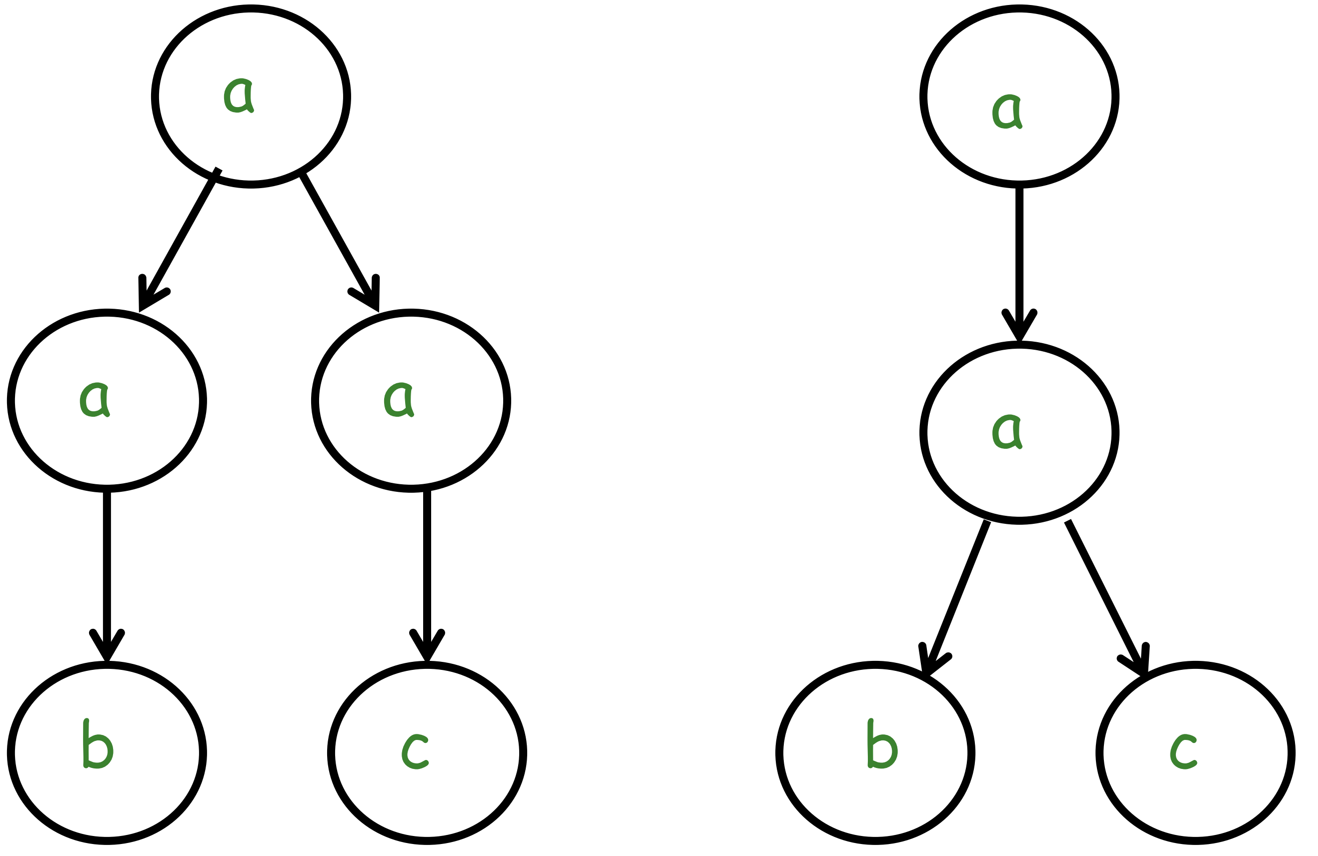
It is not the case that a must not occur.

Branching

We may reach an a from which we must not reach a b .



Same traces, different runs



Same traces, different runs (different trace trees)

Branching vs. linear time

- Linear time is conceptually simpler than branching time (words vs. trees).
- Branching time is often computationally more efficient.

(Because branching-time algorithms can work with given states,
whereas linear-time algorithms often need to "guess" sets of possible states.)

Logics

	Linear	Branching
Safety		STL
Liveness	LTL	CTL

Defining a logic

- Syntax:
  - What are the formulas?
- Semantics:
  - What are the models?
  - Does model M satisfy formula φ ? $M \models \varphi$

atomic observations

- Propositional logics:
  - boolean variables (a,b) & boolean operators ($\wedge$,$\neg$)
  - model = truth-value assignment for variables

- Propositional modal (e.g., temporal) logics:
  - ... & modal operators ($\Box$,$\Diamond$)
  - model = set of (e.g., temporally) related prop. models

observations

state-transition graph ("Kripke structure")

Model Checker Performance

- Model checkers today can routinely handle systems with between **100** and **300 state variables**.
- Systems with $\mathbf{10^{120}}$ **reachable states** have been checked.
- By using appropriate abstraction techniques, systems with an essentially **unlimited number of states** can be checked.

Notable Examples- IEEE Futurebus$^+$

- In 1992 Clarke and his students at CMU used SMV to verify the **IEEE Future+ cache coherence protocol.**
- They found a number of **previously undetected errors** in the design of the protocol.
- This was the first time that formal methods have been used to find errors in an **IEEE standard**.
- Although the development of the protocol began in **1988**, all previous attempts to validate it were based entirely on informal techniques.

Notable Examples-IEEE SCI

- In 1992 Dill and his students at Stanford used **Murphi** to verify the cache coherence protocol of the **IEEE Scalable Coherent Interface.**
- They found several errors, ranging from uninitialized variables to **subtle logical errors**.
- The errors also existed in the complete protocol, although it had been extensively **discussed**, **simulated**, and even **implemented**.

Notable Examples-PowerScale

- In 1995 researchers from Bull and Verimag used LOTOS to describe the **processors, memory controller, and bus arbiter** of the PowerScale multiprocessor architecture.
- They identified **four correctness requirements** for proper functioning of the arbiter.
- The properties were **formalized using bisimulation relations** between finite labeled transition systems.
- Correctness was established automatically in a **few minutes** using the CÆSAR/ ALDÉBARAN toolbox.

Notable Examples -HDLC

- A **High-level Data Link Controller** was being designed at AT&T in Madrid in 1996.
- Researchers at Bell Labs offered to check some properties of the design using the **FormalCheck verifier**.
- Within five hours, **six properties were specified and five were verified**.
- The sixth property failed, uncovering a **bug** that would have **reduced throughput** or caused **lost transmissions**!

Notable Examples
PowerPC 620 Microprocessor

- Richard Raimi used Motorola’s **Verdict** model checker to debug a hardware laboratory failure.
- Initial silicon of the PowerPC 620 microprocessor **crashed** during boot of an operating system.
- In a matter of seconds, Verdict found a BIU **deadlock** causing the failure.

Notable Examples-Analog Circuits

- In 1994 Bosscher, Polak, and Vaandrager won a best-paper award for proving manually the correctness of a control protocol used in **Philips stereo components**.
- In 1995 Ho and Wong-Toi **verified** an abstraction of this protocol **automatically** using HyTech.
- Later in 1995 Daws and Yovine used Kronos to check **all the properties** stated and hand proved by Bosscher, et al.

Notable Examples-ISDN/ISUP

- The NewCoRe Project (89-92) was the first application of formal verification in a software project within AT&T.
- A special purpose model checker was used in the development of the CCITT ISDN User Part Protocol.
- Five “verification engineers” analyzed 145 requirements.
- A total of 7,500 lines of SDL source code was verified.
- 112 errors were found; about 55% of the original design requirements were logically inconsistent.

Notable Examples-Building

- In 1995 the Concurrency Workbench was used to analyze an active structural control system to make **buildings more resistant to earthquakes.**
- The **control system** sampled the forces being applied to the structure and used hydraulic actuators to exert countervailing forces.
- A **timing error was discovered** that could have caused the controller to **worsen, rather than dampen**, the vibration experienced during earthquakes.

Model Checking Systems

- There are **many other successful examples** of the use of model checking in hardware and protocol verification.
- The fact that industry (**INTEL, IBM, MOTOROLA**) is starting to use model checking is encouraging.
- Below are some well-known model checkers, categorized by whether the specification is a formula or an automaton.

Temporal Logic Model Checkers

- The first two model checkers were **EMC** and **Caesar**.
- **SMV** is the first model checker to use **BDDs**.
- **Spin** uses the **partial order reduction** to reduce the state explosion problem.
- **Verus** and **Kronos** check properties of **real-time systems**.
- **HyTech** is designed for reasoning about **hybrid systems**.

Behavior Conformance Checkers

- The **Cospan/FormatCheck** system is based on showing inclusion between w-automata.
- **FDR** checks refinement between CSP programs; recently, used to debug security protocols.
- The **Concurrency Workbench** can be used to determine if two systems are observationally equivalent.

Combination Checkers

- Berkeley’s **HSIS** combines model checking with language inclusion.
- Stanford’s **STeP** system combines model checking with deductive methods.
- **VIS** integrates model checking with logic synthesis and simulation.
- The **PVS** theorem prover has a model checker for model mu-calculus.

Directions for Future Research

- Investigate the use of abstraction, compositional reasoning, and symmetry to reduce the state explosion problem.
- Develop methods for verifying parameterized designs.
- Develop practical tools for real-time and hybrid systems.
- Combine with deductive verification.
- Develop tool interfaces suitable for system designers.