# Verification Technology

# Winter/Spring 2010

# lecture 1

Bengt Jonsson

# Verification Techniques, WinterSpring 2010

Related Courses:

- Programming Theory (Parosh Abdulla):
  - principles for verifying and analyzing sequential programs
- Formal Program Development (Lars-Henrik Eriksson):
  - Systematic Development of correct programs
- Software Engineering (Roland Bol):
  - Organizing the development of software systems
- Operating Systems, Real Time Systems, Computer Networks
  - Principles and algorithms for coordinating parallel and distributed systems
- Logic, Automata theory
  - We will use some of the theory from these courses.

# Verification Techniques, WinterSpring 2010

Goal:

Modeling, Specifying, and Analyzing concurrent, parallel, and distributed algorithms, systems, and programs.

Contents:

- Modeling parallel systems (as transition systems)
- Specifying requirements and correctness properties
- Algorithms for automatically checking that a model satisfies a property
  - Model checking / state space exploration
- Application to algorithms encountered in operating systems/computer networks courses
- Analysis of concurrent software.
- Use of Software tool for all the above: SPIN

# Administrative

Instructors:

- Bengt Jonsson, room 1435 `bengt(at)it.uu.se`
- TBD

Course page
    http://www.it.uu.se/edu/course/homepage/verteknik/vt10/

Examination:

- 3 homework exercises (solved individually or in pairs)
- "mini-project": model, specify, and analyze a case study.
- Final exam on the topics covered in lectures.

SPIN

- You **must** use SPIN for the exercises.
    - You are encouraged to install SPIN on your own computer.
- On IT servers, installed at /it/sw/misc/bin/spin
- XSPIN at /it/sw/misc/bin/xspin
- Further material at `http:/spinroot.com/spin`
- `jSPIN at http://stwww.weizmann.ac.il/g-cs/benari/jspin`

# Course Material

You will **need**

- Lecture Handouts (slides)
- A few papers (will be distributed)
- SPIN documentation (on the WWW, and distributed)

Reference texts: Recommend to choose one/several from

- Mordechai Ben-Ari, *Principles of the Spin Model Checker*, Springer Verlag, 2008, pedagogic textbook covering Promela
- Gerard Holzmann: *SPIN MODEL CHECKER Primer and Reference Manual,* Addison-Wesley, 2003, detailed book on SPIN
- *Design and Validation of Computer Protocols*, G.J. Holzmann, Prentice Hall 1991, older book, can be downloaded from the net.
- Old notes prepared by me some years ago.

## Structure of the Material

The course is a close interplay between

- Concepts and techniques for modeling, specification, and verification
- Implementation in the tool SPIN
  - almost an exact realization of the theory
- Application to examples

# Examination

- Homeworks, to be solved individually or in pairs. **mandatory**
- "mini-project": model, specify, and analyze a case study. **Most important part of course.**
- Final  exam, covering lectures

Each counts for a third of your final grade.

HOW TO DO WELL:

- Do the homework seriously
- Make sure that you master the material to make a good mini-project
- Ask when things are not clear

**COURSE OVERVIEW:**
What problems can be solved?

# Verification

Verification = "building the system right"

System description | $\models$ "conforms" | Correctness properties
--- | --- | ---

Web server implementation

Protocol standard

Functional spec.

...........

Absence of

- Run-time errors
- deadlocks
- Memory leaks

Protocol service

.....

# Verification

- Testing consumes ~half of software development effort
- Several “expensive” accidents caused by bugs
  - Ariane 5 crash 1996
  - Pentium division bug
  - Mars pathfinder ceased to work 1997
  - Viruses, ….

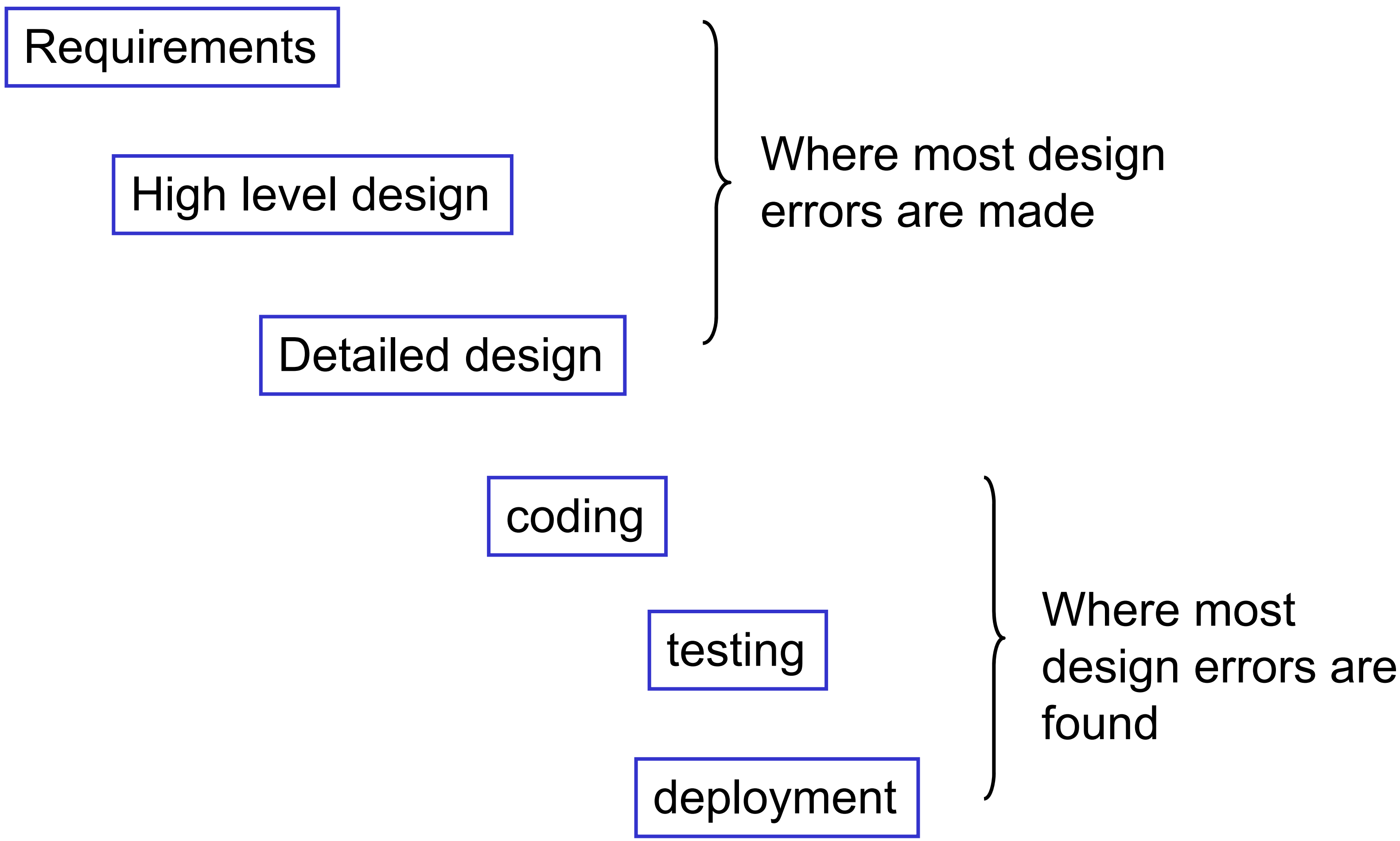
# Some of the Improvements needed

- Better Development tools
  - Programming languages
  - Development environments/Libraries
  - Software architectures
- Better Skilled People
  - Better designers
  - better programmers
  - Better testers and verifiers
- Better Processes
  - Better Collaboration between developers / with customers
  - Better Documentation
- Better Verification Techniques
  - Testing and verification: This (and other) courses

# Motivation: Idealized Design process

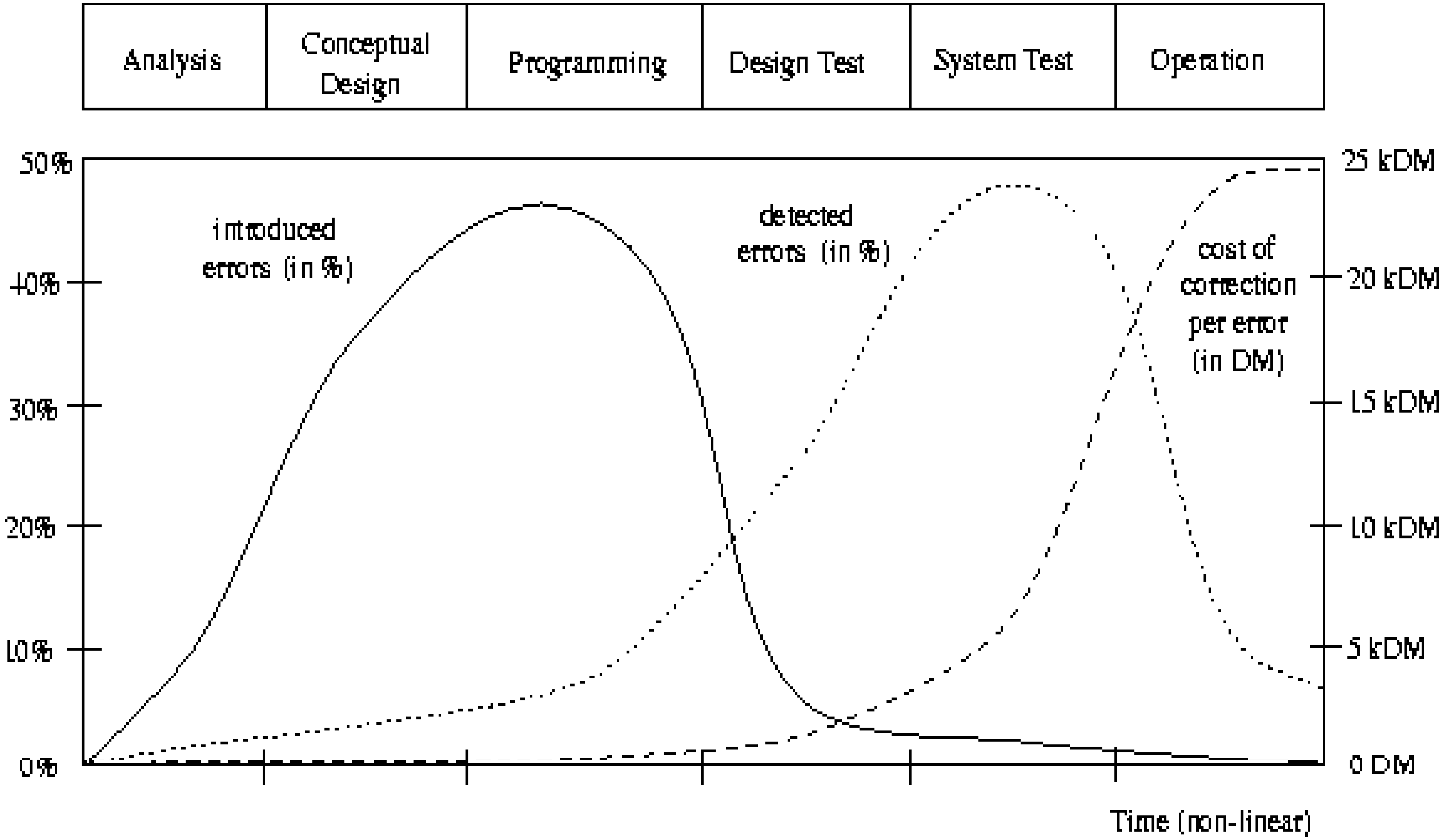
- Requirements
- High level design
- Detailed design
- coding
- testing
- deployment

# Motivation: Idealized Design process



# Introducing, Detecting and Correcting errors: cost

♦ Errors detected: the later the more expensive



# Verification Techniques: short overview

**Testing:** By far the most used technique
+ The most ”practical” technique
+ Can verify a wide range of properties
- Can only be used on implementation
- Difficult to make exhaustive
- Hard to make reproducible for concurrent/distributed programs
- Manual selection of test cases and input needs work.

**Prototyping and Simulation:**
+ Can be used on design level
- Difficult to make exhaustive
- Manual selection of test cases and input needs work

**Code and Design Reviews:**
+ Good at finding (some classes of) problems
- Needs organization and people

# Verification Techniques: short overview

**Static Program Analysis:** Analyzing the source code by tools

+ Completely automatic
- Can verify a limited set of properties (type-correctness, absence of some run-time errors)
- Tools available only for some languages and properties

**Model Checking:** Analyzing a prototype/model by tools

+ Can be done early in the design cycle, e.g., on design level.
+ Automated (provided tools available)
+ Can check many kinds of properties
- A model must be constructed (at a suitable level of abstraction)
- Model must be maintained when system evolves.
- Does not scale to very large models

# Motivation: Purpose of Model Verification

Requirements

High level design

Detailed design

Build model of the design.
Analyze it thoroughly

coding

testing

deployment

Implement your model, and spend effort to check conformance to the model

# Problems that can be addresed by Model Checking

Checking correctness of

- Communication protocols
- Distributed Algorithms
- Controllers
- Hardware circuits
- Embedded and real-time systems and software

e.g.,

Absence of race conditions, deadlocks, livelocks, priority inversions, proper synchronization, ....

> Model checking is the appropriate technique when there are many many different scenarios of interaction between components in a system

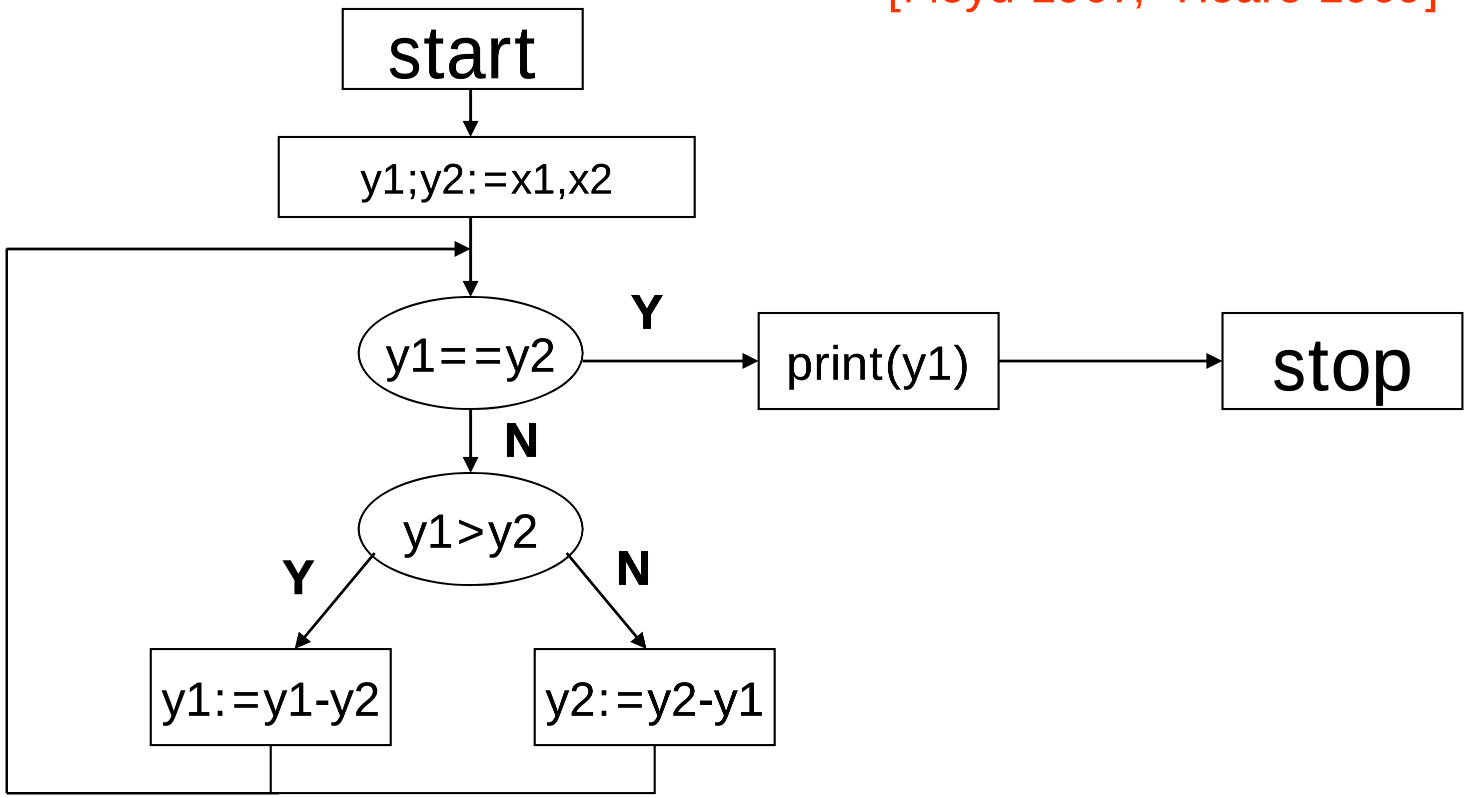
# Merits of model checking

- Checking simple properties (e.g. deadlock freeness) is already extremely useful!
- The goal is no longer seen as proving that a system is completely correct (bug-free)
- The objective is to have tools that can help a developer find errors and gain confidence in her/his design. That is achievable
- Now widely used in hardware design, protocol design, embedded systems, …

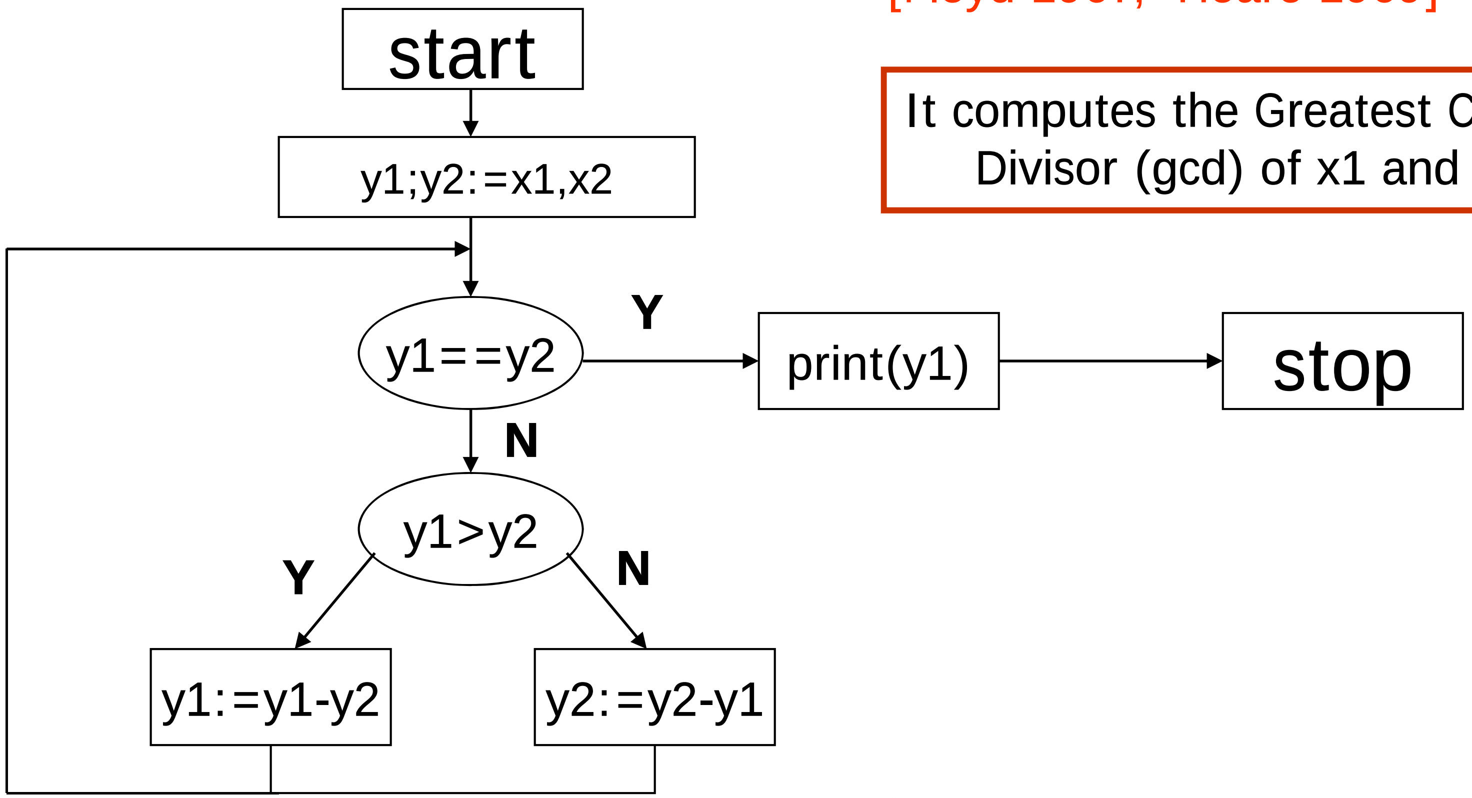
## CONTRAST: Assertional verification for ”data-centric” programs

What does this program do?

[Floyd 1967, Hoare 1969]



## CONTRAST: Assertional verification for ”data-centric” programs



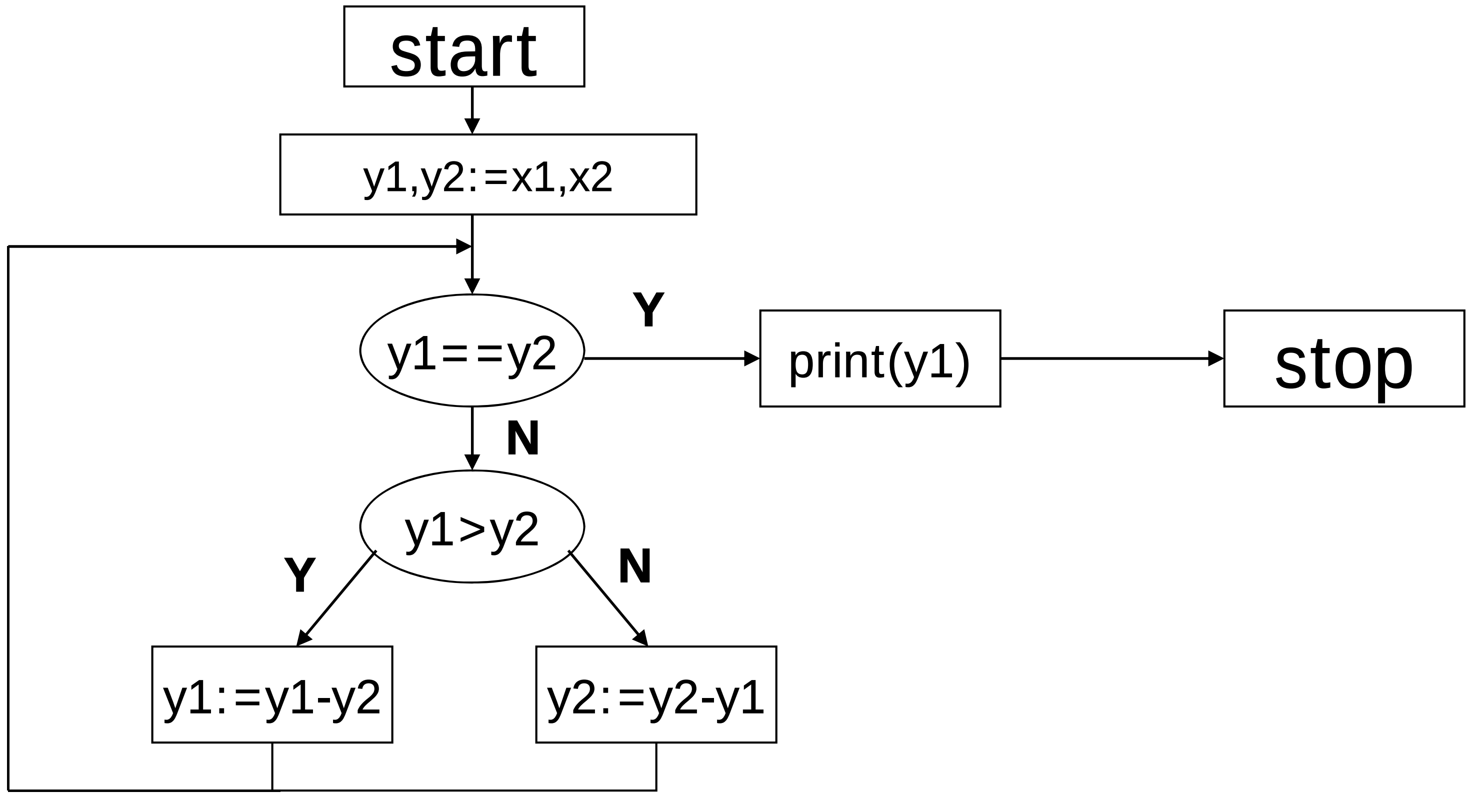

What does this program do?

[Floyd 1967, Hoare 1969]

It computes the Greatest Common Divisor (gcd) of x1 and x2

WY3

# How can a program check this fact?



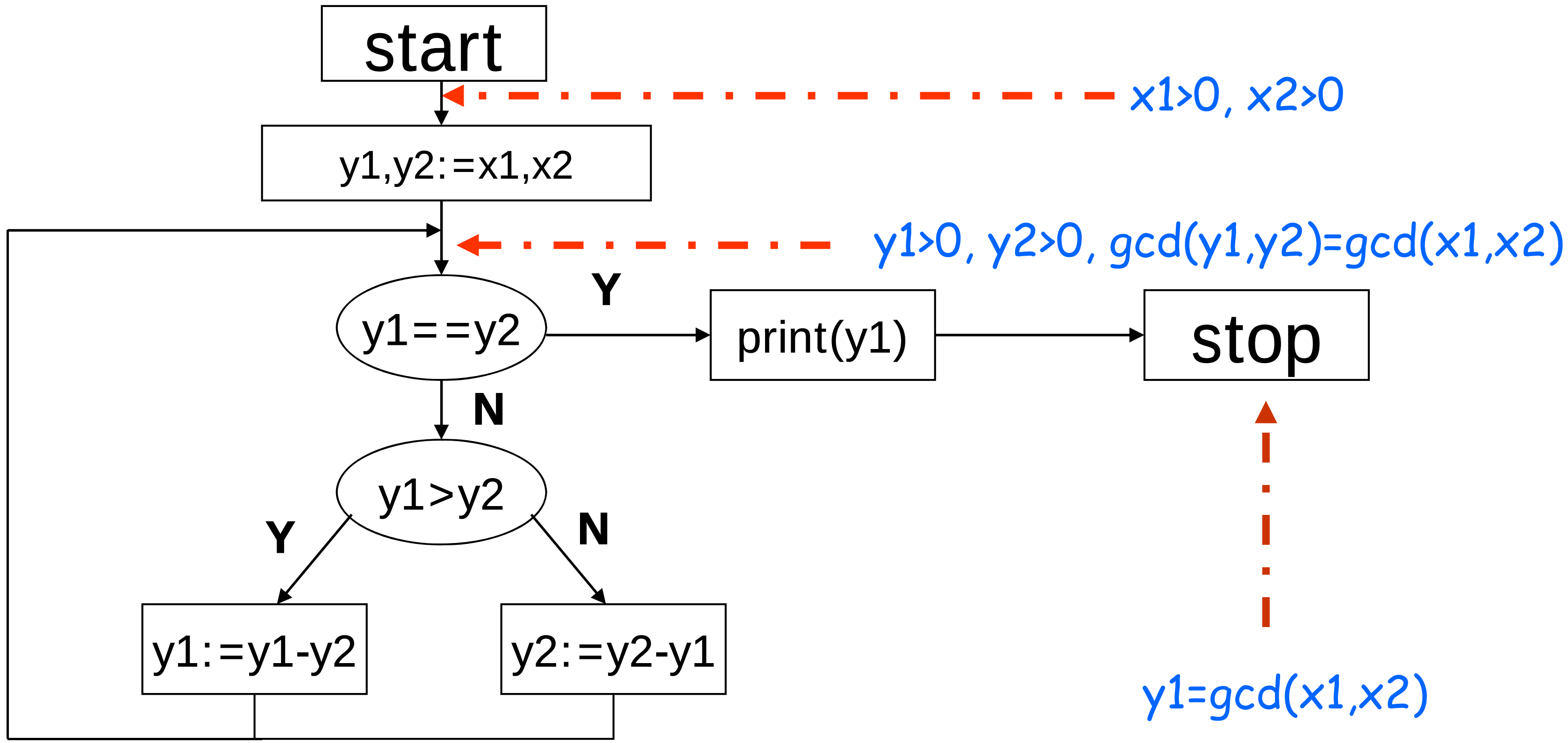
**Slide 22**

---

**WY3** Wang Yi; 2006-03-29

WY4

## How can a program check this fact?



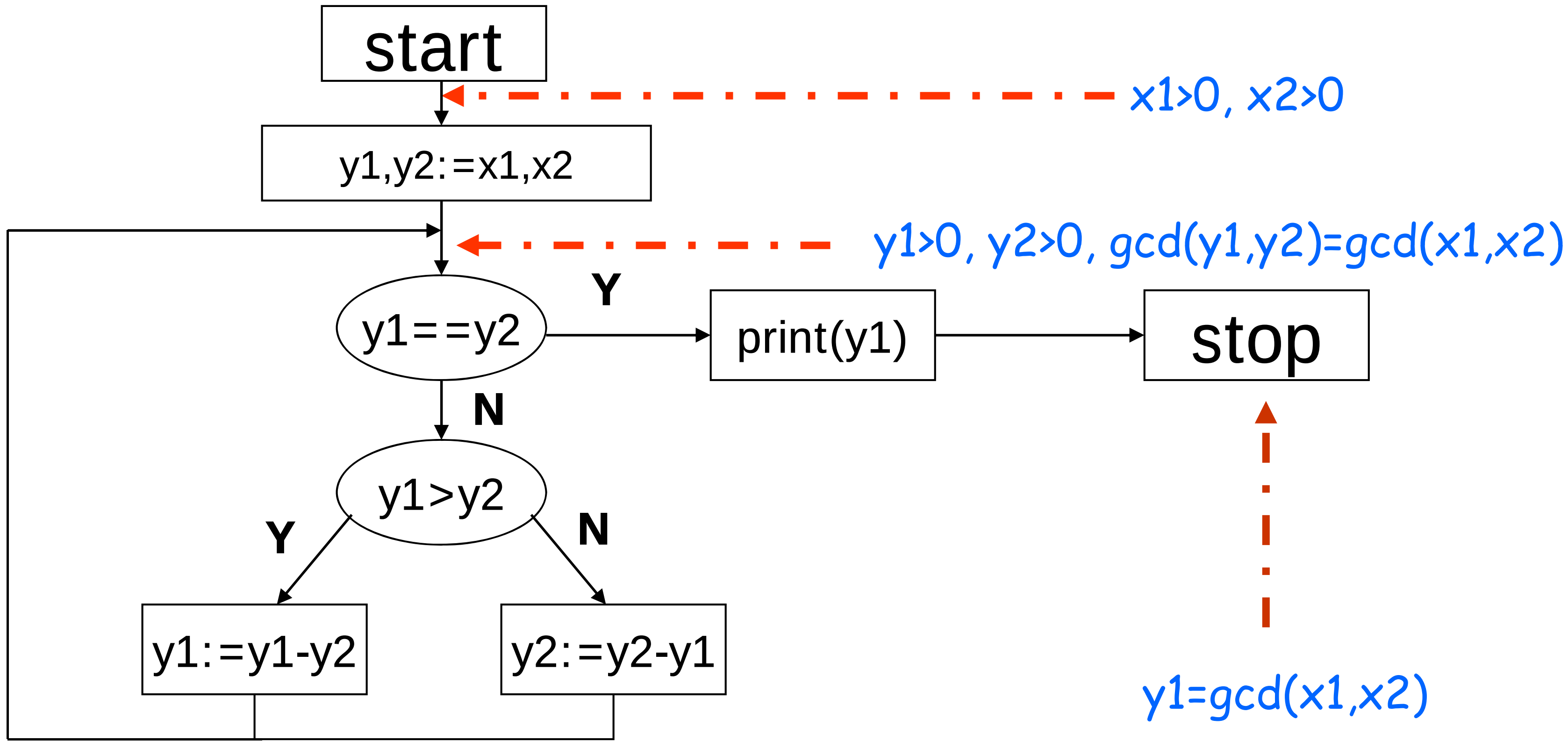
**Slide 23**

---

**WY4** Wang Yi; 2006-03-29

WY5

## How can a program check this fact?




Can this be checked by a computer?

**Slide 24**

---

**WY5** Wang Yi; 2006-03-29

# Static Analysis: Example (input)

```
n := n0;
i := n;
while (i <> 0 ) do
        j := 0;
        while (j <> i) do
                j := j + 1
        od;
i := i - 1
od
```

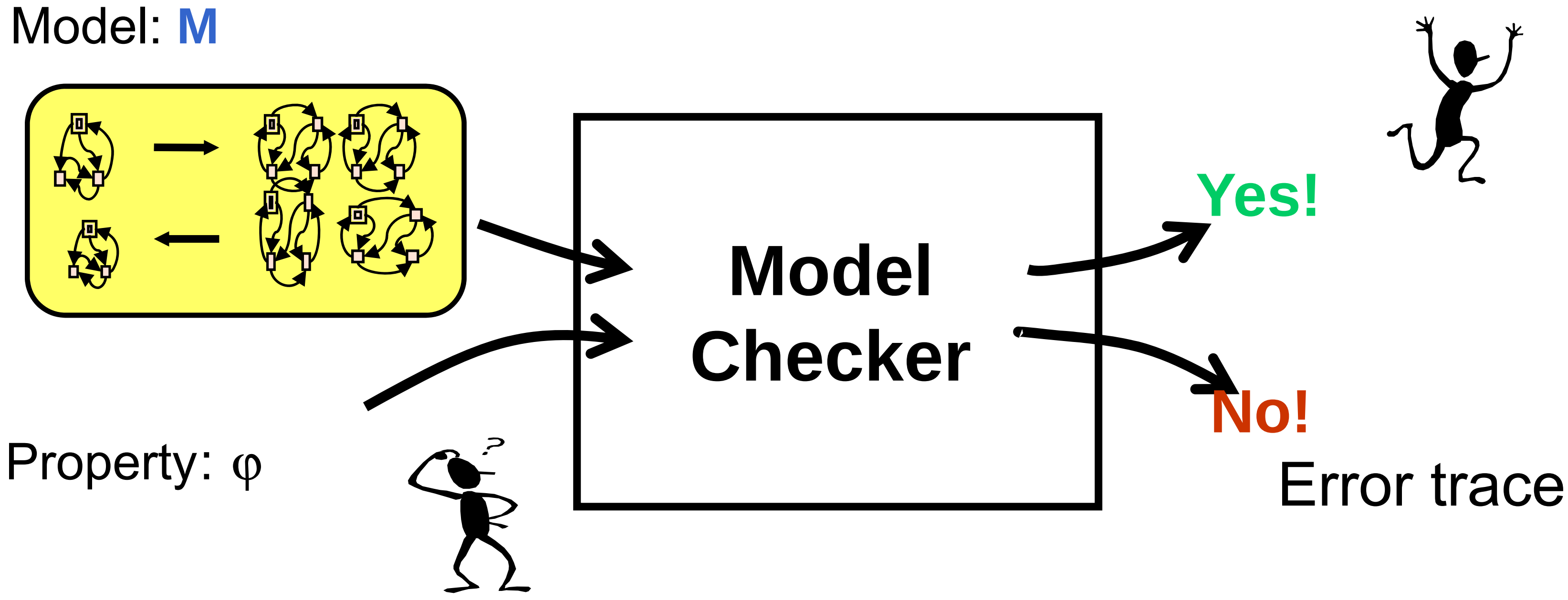
# Static Analysis: Example (output)

```
{n0>=0}
n := n0;
{n0=n , n0>=0}
i := n;
{n0=i , n0=n , n0>=0}
while (i <> 0 ) do
      {n0=n , i>=1 , n0>=i}
      j := 0;
      {n0=n , j=0 , i>=1 , n0>=i}
      while (j <> i) do
            {n0=n , j>=0 , i>=j+1 , n0>=i}
            j := j + 1
            {n0=n , j>=1 , i>=j , n0>=i}
      od;
      {n0=n , i=j , i>=1 , n0>=i}
i := i - 1
{i+1=j , n0=n , i>=0 , n0>=i+1}
od
{n0=n , i=0 , n0>=0}
```

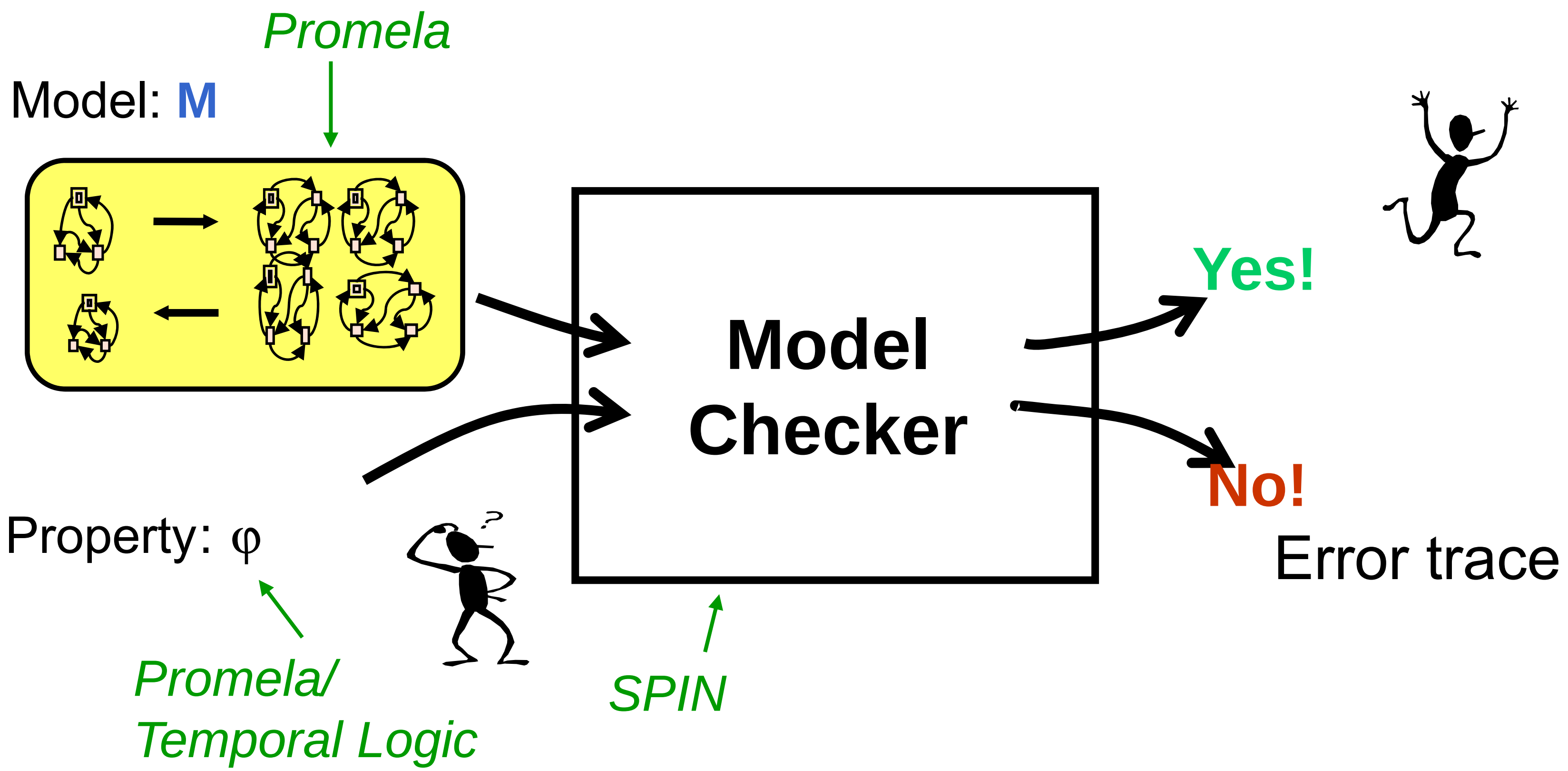
# Static Analysis: Example (output)

```
{n0>=0}
n := n0;
{n0=n , n0>=0}
i := n;
{n0=i , n0=n , n0>=0}
while (i <> 0 ) do
      {n0=n , i>=1 , n0>=i}
      j := 0;
      {n0=n , j=0 , i>=1 , n0>=i}
      while (j <> i) do
            {n0=n , j>=0 , i>=j+1 , n0>=i}        implies that j does not overflow
            j := j + 1
            {n0=n , j>=1 , i>=j , n0>=i}
      od;
      {n0=n , i=j , i>=1 , n0>=i}                 implies that i does not underflow
i := i - 1
{i+1=j , n0=n , i>=0 , n0>=i+1}
od
{n0=n , i=0 , n0>=0}
```

# Overview of Model Checking



# Overview of Model Checking



# Workflow for verification of e.g., software

Problem: Check whether all executions of program P work "correctly"

# Workflow for verification of e.g., software

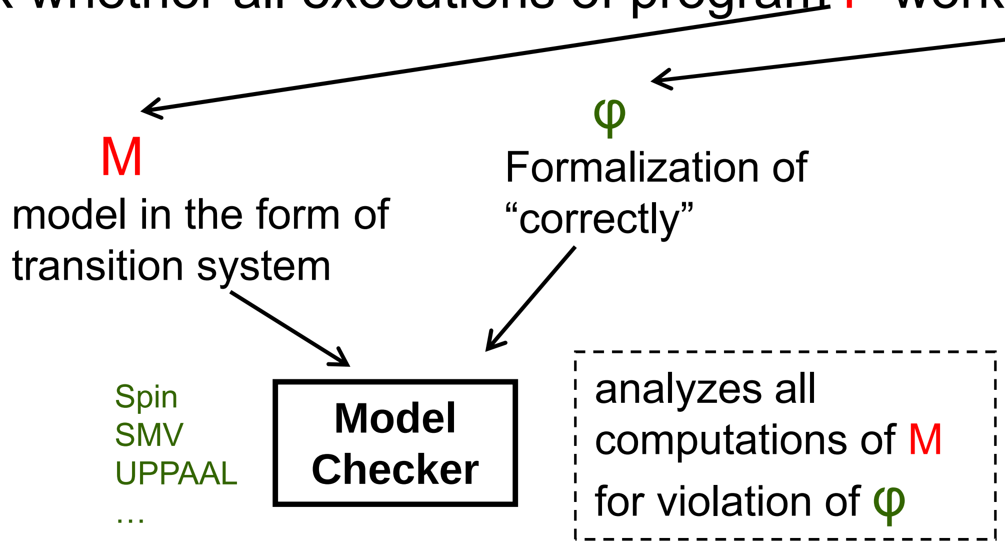
Problem: Check whether all executions of program P work "correctly"

M
model in the form of transition system

φ
Formalization of "correctly"

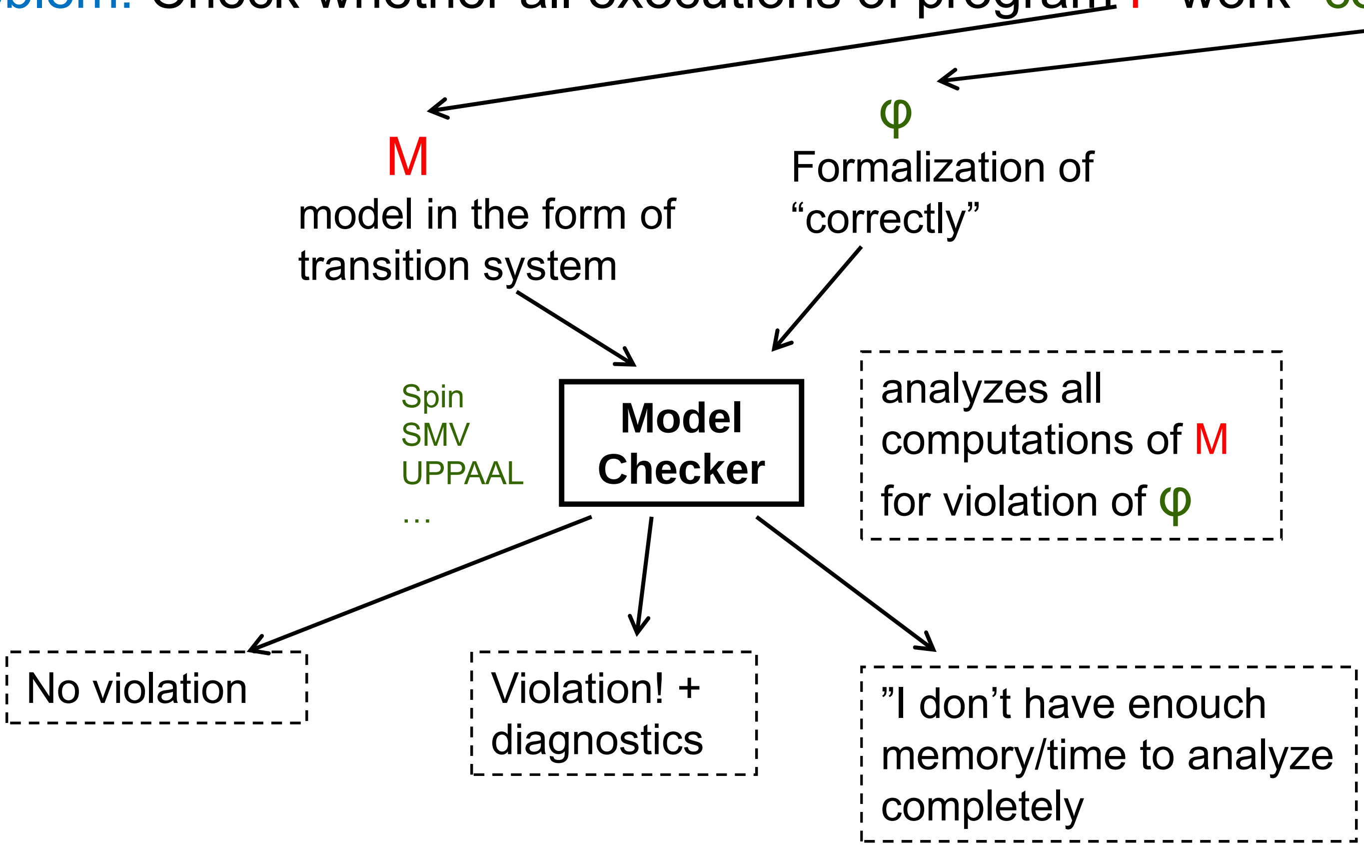
# Workflow for verification of e.g., software

Problem: Check whether all executions of program P work “correctly”



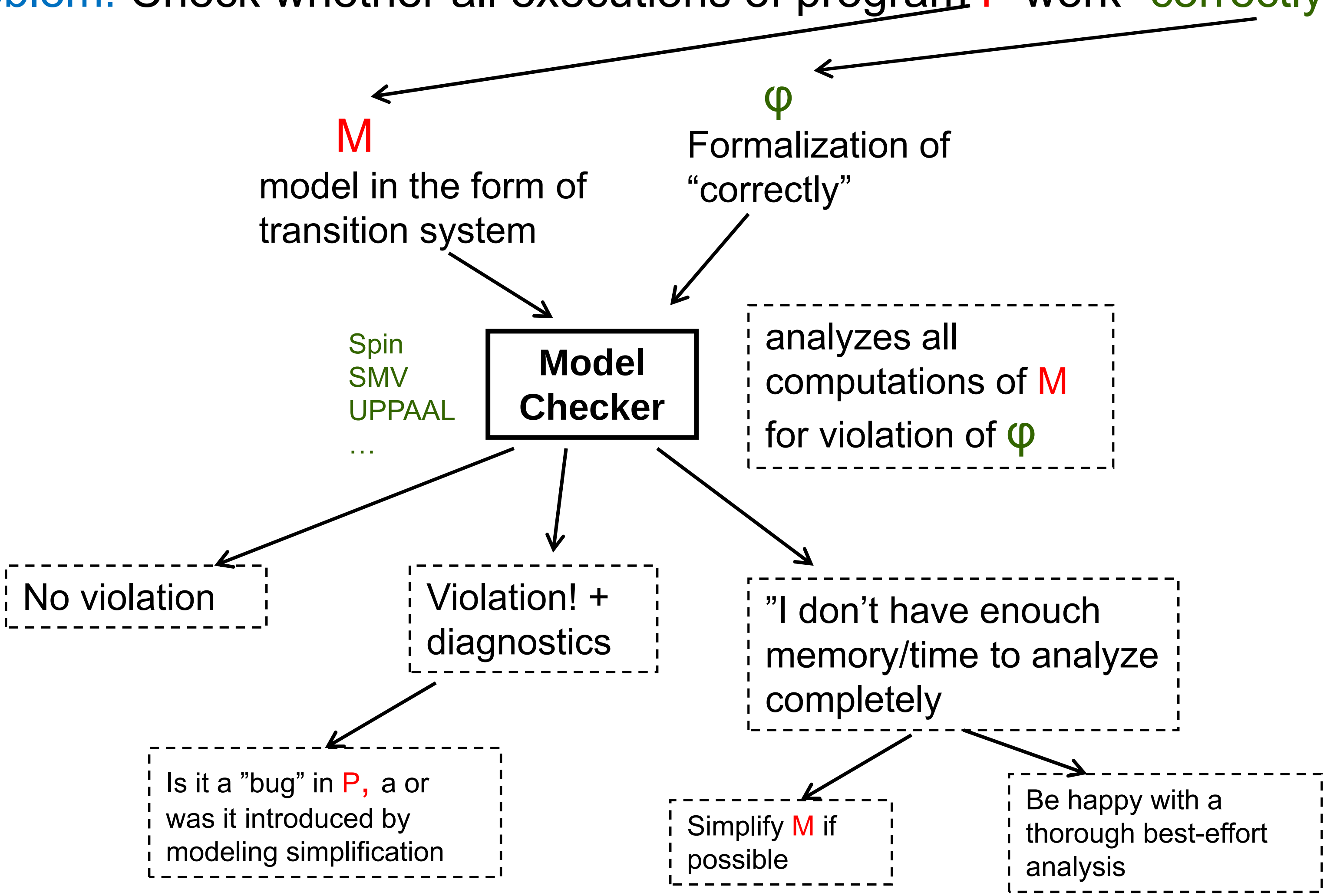
# Workflow for verification of e.g., software

Problem: Check whether all executions of program P work “correctly”



# Workflow for verification of e.g., software

Problem: Check whether all executions of program P work “correctly”



# Unveiling bad mutual exclusion algorithm

```
/* Bad Mutex Algorithm */

int x, y, z;

void lock(int Pid)
{
busywait:
    x = Pid;
    if (y != 0 && y != Pid)
          goto busywait;

    z = Pid;
    if (x != Pid)
          goto busywait;

    y = Pid;
    if (z != Pid)
          goto busywait;
}

void unlock()
{
    x = 0;
    y = 0;
    z = 0;
}
```

# Example: threaded software example

```
int main(void)
{    thread_t thread_id, main_id;
     main_id = thr_self();
     thr_setconcurrency(2);
     thr_create(NULL, 0 , thread_sub, (void *)main_id, THR_SUSPENDED, &thread_id);

     while(1) {
           printf("MAIN: continuing subroutine thread\n"); fflush(stdout);
           thr_continue(thread_id);
           printf("MAIN: suspending self\n"); fflush(stdout);
           thr_suspend(main_id);
     }
     return(0);
}
void *tread_sub(void *arg)
{    thread_t thread_id;
     thread_t main_id = (thread_t) arg;

     thread_id = thr_self();

     while(1) {
           printf("THREAD: continuing main thread\n"); fflush(stdout);
           thr_continue(main_id);
           printf("THREAD: suspending self\n"); fflush(stdout);
           thr_suspend(thread_id);
     }
     return((void *)0);
}
```

# Example: threaded software example

```
int main(void)
{    thread_t thread_id, main_id;
     main_id = thr_self();
     thr_setconcurrency(2);
     thr_create(NULL, 0 , thread_sub, (void *)main_id, THR_SUSPENDED, &thread_id);

     while(1) {
           printf("MAIN: continuing subroutine thread\n"); fflush(stdout);
           thr_continue(thread_id);
           printf("MAIN: suspending self\n"); fflush(stdout);
           thr_suspend(main_id);
     }
     return(0);
}
void *tread_sub(void *arg)
{    thread_t thread_id;
     thread_t main_id = (thread_t) arg;

     thread_id = thr_self();

     while(1) {
           printf("THREAD: continuing main thread\n"); fflush(stdout);
           thr_continue(main_id);
           printf("THREAD: suspending self\n"); fflush(stdout);
           thr_suspend(thread_id);
     }
     return((void *)0);
}
```

# Example: threaded software example

```
int main(void)
{   thread_t thread_id, main_id;
    main_id = thr_self();
    thr_setconcurrency(2);
    thr_create(thread_sub, &thread_id);

    while(1) {

          thr_continue(thread_id);

          thr_suspend(main_id);
    }
    return(0);
}
void *tread_sub(void *arg)
{   thread_t thread_id;
    thread_t main_id = (thread_t) arg;

    thread_id = thr_self();

    while(1) {

          thr_continue(main_id);

          thr_suspend(thread_id);
    }
    return((void *)0);
}
```

```
bool Suspend_main, Suspend_thread, arg;

active proctype main() provided (!Suspend_main) {
run thread();
L_0:
      do
      ::  Suspend_thread = 0;
          Suspend_main = 1;
      od;
      goto Return;
Return: skip
}

proctype thread() provided (!Suspend_thread) {
L_1:
      do
      :: Suspend_main = 0;
         Suspend_thread = 1;
      od;
      goto Return;
Return: skip
}
```

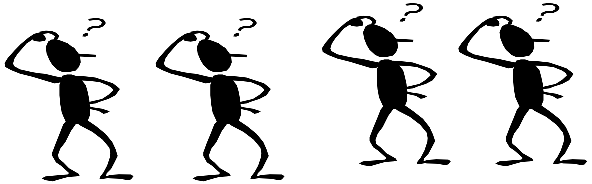
# Output from analysis by SPIN

```
THREAD: continuing main thread
THREAD: suspending self
MAIN: continuing subroutine thread
THREAD: continuing main thread
THREAD: suspending self
MAIN: suspending self

18: main(0):[Suspend_main = 1]
spin: trail ends after 18 steps
#processes 2:
 18:    proc 0 (main)  line   5 (state  7) (invalid end state)
         Printf("MAIN: continuing subroutine thread\n");
 18:    proc 1 (thread)  line  20 (state  7) (invalid end state)
         Printf("THREAD: continuing main thread\n");
global vars:
   bit    Suspend_main:       1
   bit    Suspend_thread:     1
   bit    arg: 0

...
```

# Hippies problem



5 10 20 25

Hippies must get across bridge.
Crossing needs torch. There is only one torch.
At most two people can cross together.
Can all cross in at most 60 minutes?

# Desiderata for good a model

- Captures essentials of behavior of system/program/algorithm
- Should be simple to understand, and well-structured
  - To validate that you model the correct thing
- Can be thoroughly analyzed (e.g., by SPIN)
  - Avoid unnecessary complications
  - Try to abstract/simplify necessary complicated aspects
  - Not ”too big”

# How to make models

- By hand from a description of algorithm/system
- As specification during system design

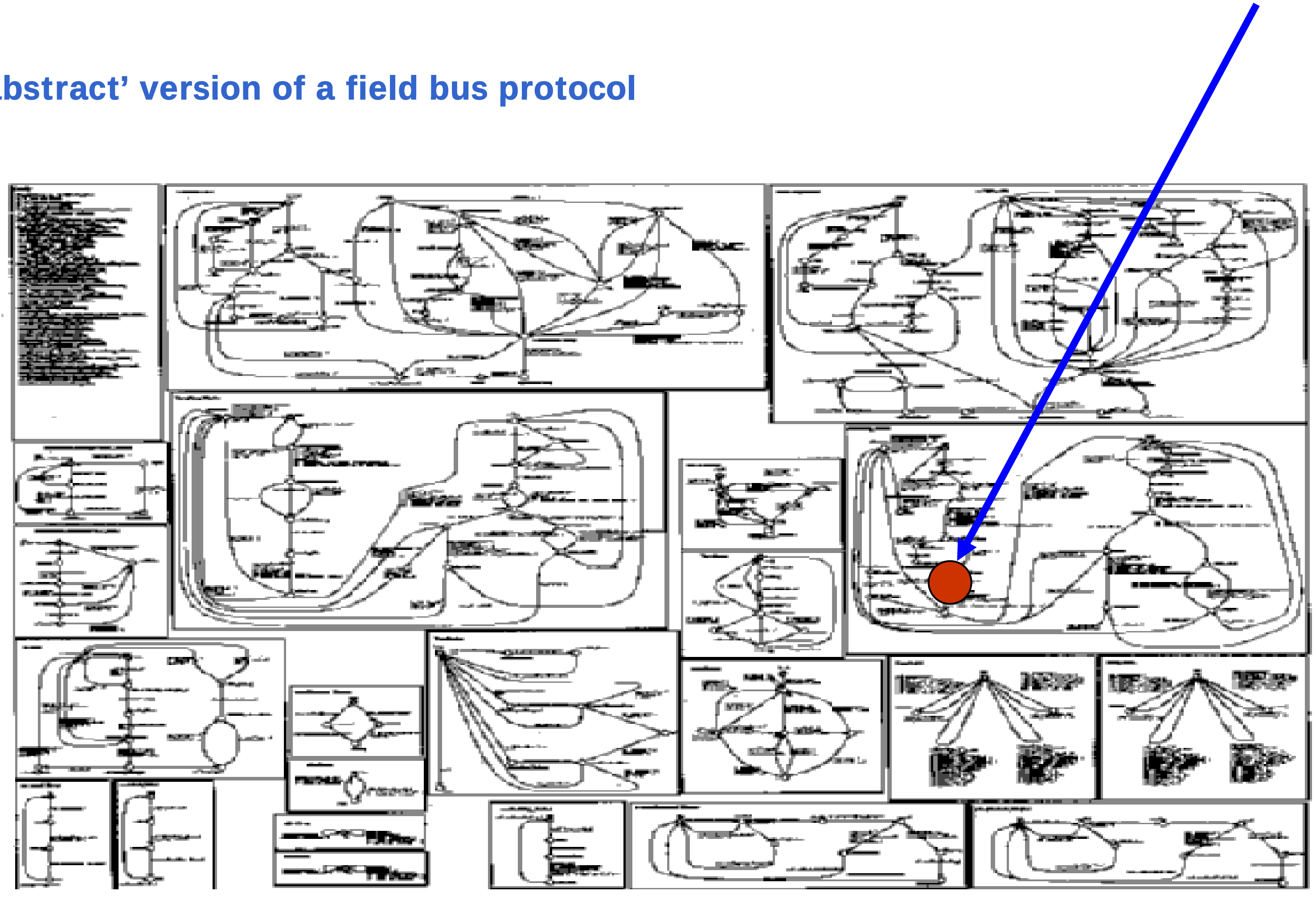
When analyzing existing programs

- By hand from code
- By automated extraction tools from program code
  - ModEx adapts ANSI-C code to SPIN
  - Hard problem: automated simplification
- Automatically from test suites

## Example of Model

***Reachable?***
***(bug?)***

**An 'abstract' version of a field bus protocol**



# Remaining Problems

Constructing a Model

- not so easy, this course will make you experts

Making absolutely sure that the actual system/software conforms to the model

- hard problem: there are several techniques:
  - Conformance testing
  - Static program analysis
  - Automated code generation

# Small Idealized Example of a “real” bug

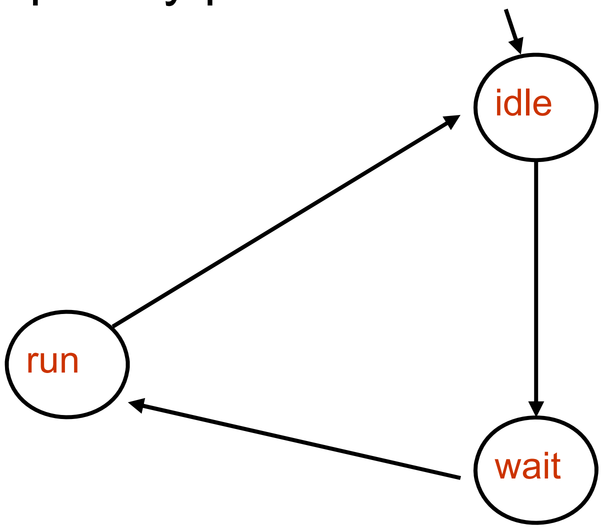
Small Example: Mars Pathfinder 1997

Typical properties of synchronization in real-time systems

- Mutual exclusion
  - A process cannot access the data-bus unless it owns a mutex-lock
- Scheduling priority
  - Saving data to memory has higher priority than processing data
  - Low priority process cannot execute when high priority process is ready to execute or executes

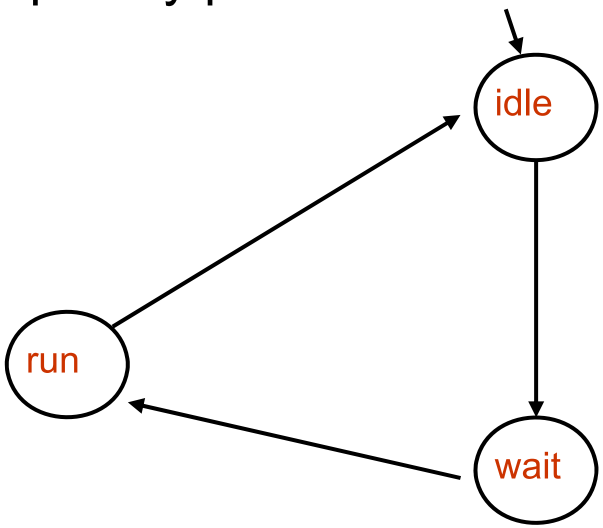
# Idealized model of processes

High priority process

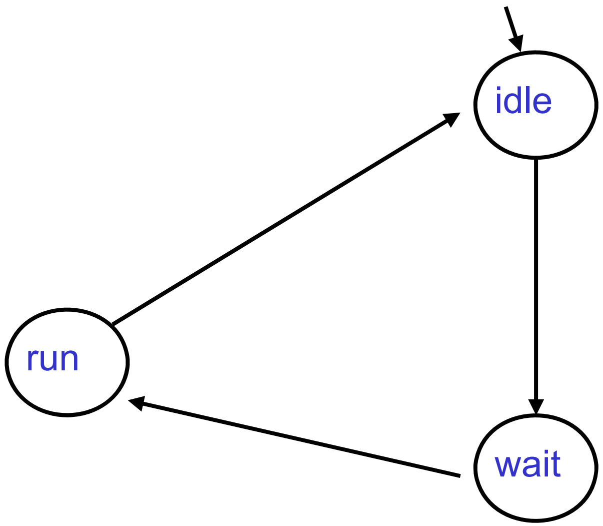


# Idealized model of processes

High priority process



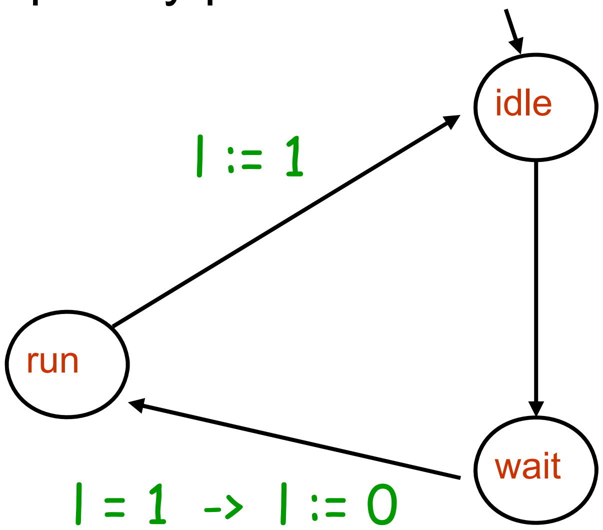

Low priority process



# Idealized model of processes

Variables: l: integer
Initially    l = 1

High priority process




Low priority process