A Loose-Synchronization System (1)

MOSIG Embedded Systems Exam, Jan 2011

```
// Thread T1         // Thread T2

x := 1 ;
while true loop      y := 1 ;
   tick1 ;           while true loop
   tick1 ;              tick2 ;
   x := x+2 ;           y := y+2 ;
end loop ;           end loop ;
```

A Loose-Synchronization System (2)

We consider parallel programs made of two threads `T1, T2`, managed by a special preemptive scheduler on a mono-processor. Each thread manipulates local variables, and may use a special instruction `tick` (`tick1` for `T1`, and `tick2` for `T2`), meaning that one unit of its local time passes. The two local times are not synchronized, and can pass independently. The scheduler keeps track of the time that has passed in each thread. Suppose that for `T1` the total amount of time since the beginning (i.e., the number of `tick` transitions it has taken) is n_1, and for `T2` it's n_2. Then, there are several cases: if $\mid n_1 - n_2 \mid < 2$ the two threads can execute; if $n_1 - n_2 \geq 2$, `T1` is blocked; if $n_2 - n_1 \geq 2$, `T2` is blocked. In other words, a thread cannot be "in advance" of more than two `ticks`.

Why Models?

We would like to study the potential behaviors of such a system and, for instance, the effect of the loose synchronization mechanism on the relative values of `x, y`.

The constraint on $\mid n_1 - n_2 \mid$ could be an abstract way to model the effect of a clock synchronization protocol.

This is an example of a constrained asynchronous system for which we can build a model in two ways: (1) using synchronous operators and several clock inputs, or (2) designing an ad hoc asynchronous product+encapsulation. We will do (2) in the sequel.

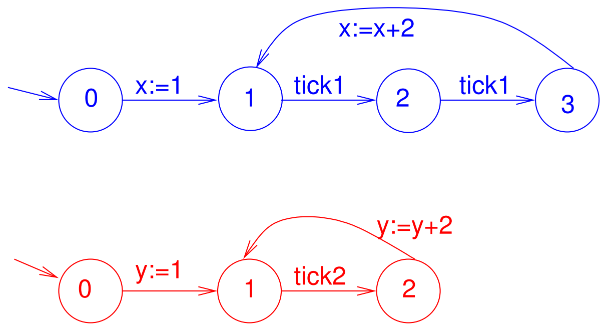
Building an Operational Model (1)

— Draw an automaton for each thread
— Explain how you build the asynchronous product of these two automata, representing the behaviour of the scheduler described above. Draw a significant part of this asynchronous product, such that we can observe the blocking effect. Since there is no effect of the scheduler related to variables, you should not expand the values of the variables in the product. You should keep the assignments as labels of transitions.
— What is the maximum number of states for this product?
— Generalization to n threads (the `tick` "distance" between any two threads is never greater than 2).

Building an Operational Model (2)

- Preemptive scheduler on a mono-processor: implies the use of a pure asynchronous product
- Atomicity hypothesis?: we assume it's "as usual"; simple actions on local variables are atomic, the `tick` instructions are also atomic.
- Additional state information in the product: the "shared memory" needed for the synchronization is only $n_1 - n_2$ (we do not need to recall n_1 and n_2 individually). For process i, doing `tick`i means incrementing its n_i.

Automata (Decision on Atomicity)



Computing the Asynchronous Product with the Constraint on the Number of `ticks` (1)

States of the Product:

- A state in the first one (among {0,1,2,3}) , a state in the second one (among {0,1,2})
- If we choose to "expand" the values of the local variables: the value of `x`, the value of `y`
- The difference between the numbers of ticks $n_1 - n_2$

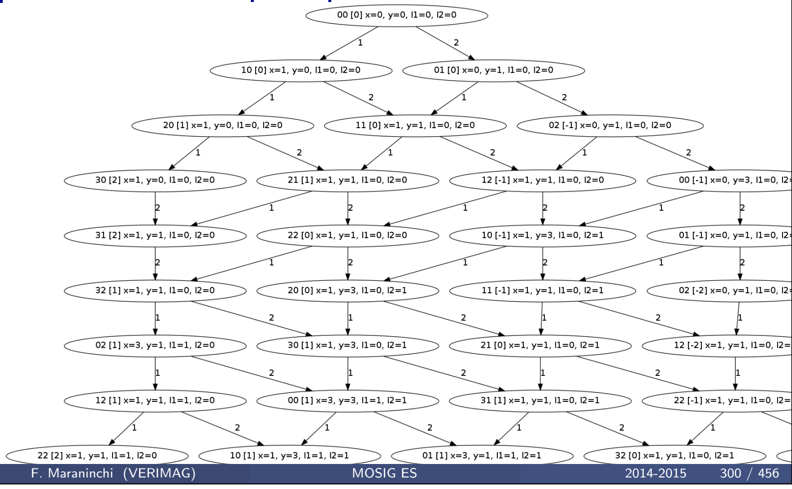
Invariant (easy to prove, and will be "discovered" when building the model): $n_1 - n_2 \in [-2, 2]$.

Computing the Asynchronous Product with the Constraint on the Number of `ticks` (2)

Transitions of the Asynchronous Product:

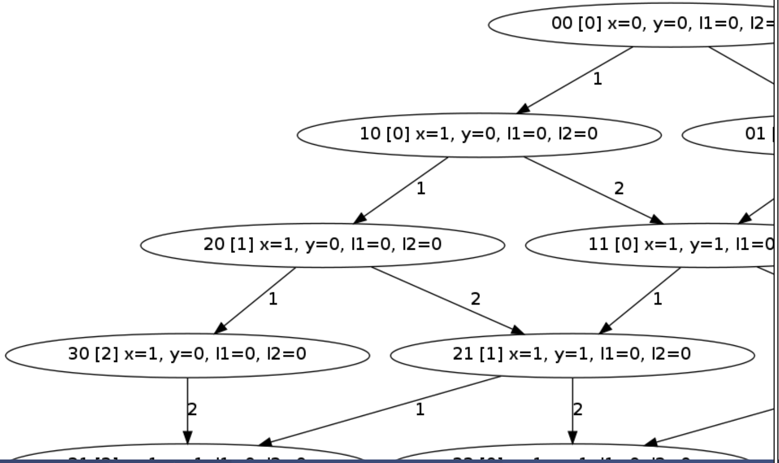
- From a state where $-2 < n_1 - n_2 < 2$:
  both processes can move, there are two transitions
- From a state where $-2 \not< n_1 - n_2 < 2$:
  only the first process can move
- From a state where $-2 < n_1 - n_2 \not< 2$:
  only the second process can move

Asynchronous Product - `x`, `y` and number of passes in the loops expanded



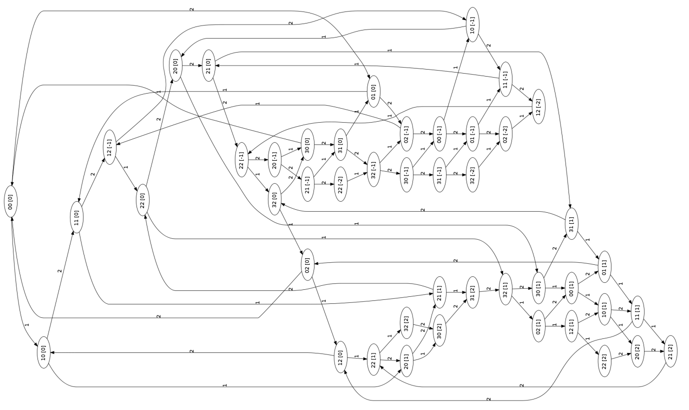
00 [0] x=0, y=0, I1=0, I2=0
10 [0] x=1, y=0, I1=0, I2=0
01 [0] x=0, y=1, I1=0, I2=0
20 [1] x=1, y=0, I1=0, I2=0
11 [0] x=1, y=1, I1=0, I2=0
02 [-1] x=0, y=1, I1=0, I2=0
30 [2] x=1, y=0, I1=0, I2=0
21 [1] x=1, y=1, I1=0, I2=0
12 [-1] x=1, y=1, I1=0, I2=0
00 [-1] x=0, y=3, I1=0, I2
31 [2] x=1, y=1, I1=0, I2=0
22 [0] x=1, y=1, I1=0, I2=0
10 [-1] x=1, y=3, I1=0, I2=1
01 [-1] x=0, y=1, I1=0, I2
32 [1] x=1, y=1, I1=0, I2=0
20 [0] x=1, y=3, I1=0, I2=1
11 [-1] x=1, y=1, I1=0, I2=1
02 [-2] x=0, y=1, I1=0, I2
02 [1] x=3, y=1, I1=1, I2=0
30 [1] x=1, y=3, I1=0, I2=1
21 [0] x=1, y=1, I1=0, I2=1
12 [-2] x=1, y=1, I1=0, I2=
12 [1] x=1, y=1, I1=1, I2=0
00 [1] x=3, y=3, I1=1, I2=1
31 [1] x=1, y=1, I1=0, I2=1
22 [-1] x=1, y=1, I1=0, I2=
22 [2] x=1, y=1, I1=1, I2=0
10 [1] x=1, y=3, I1=1, I2=1
01 [1] x=3, y=1, I1=1, I2=1
32 [0] x=1, y=1, I1=0, I2=1

Asynchronous Product - `x`, `y` and number of passes in the loops expanded



00 [0] x=0, y=0, I1=0, I2=
10 [0] x=1, y=0, I1=0, I2=0
01
20 [1] x=1, y=0, I1=0, I2=0
11 [0] x=1, y=1, I1=0
30 [2] x=1, y=0, I1=0, I2=0
21 [1] x=1, y=1, I1=0, I2=0

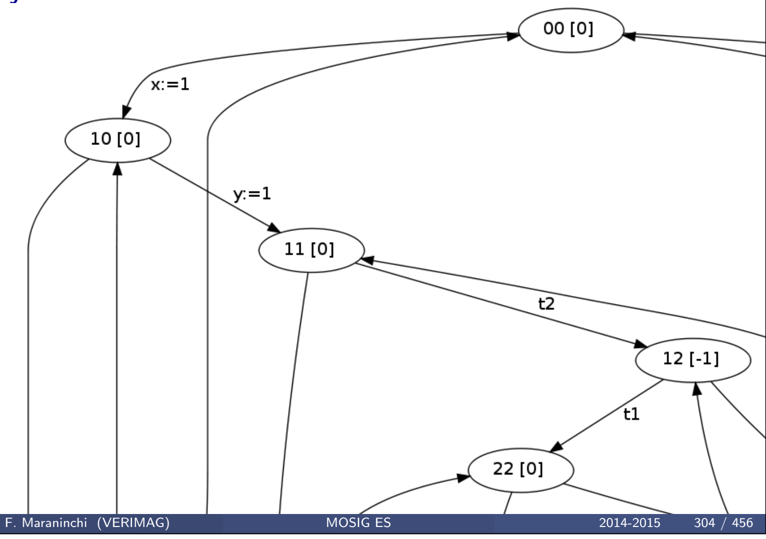
Asynchronous Product - `x`, `y` and number of passes in the loops not expanded



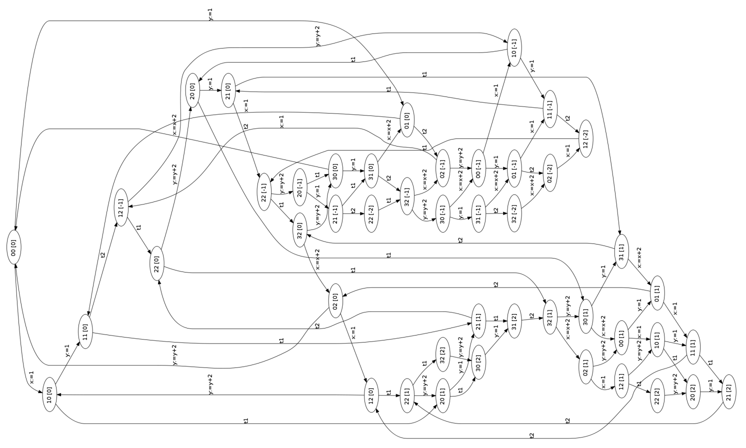
Asynchronous Product - Discussion

The relevant "control states" are made of: a state in process 1, a state in process 2, $n_1 - n_2$.
All the synchronization decisions can be taken based on these information.
What happens to the local variables `x, y` can be considered as a mere "decoration" for the moment.

Asynchronous Product - Decorations



Asynchronous Product - Decorations



Asynchronous Product - The Non-Expanded Graph is Finite

If we ignore the values of `x, y` and the number of passes in the two loops, the graph is finite:
It has 46 reachable states among
$|\{0,1,2,3\} \times \{0,1,2\} \times [-2,2]| = 4 \times 3 \times 5 = 60$ states of the complete Cartesian product.

Generalization to n threads $0, 1, \ldots n-1$

The thread i has Σ_i states and uses $\texttt{tick}_i$.

The generalized "synchronization" mechanism: a thread i is allowed to move only if its number of local ticks n_i is not too much in advance w.r.t. all the other numbers of local ticks, i.e.,
$\forall k \in [0, n-1]. \quad n_i - n_k < 2$.

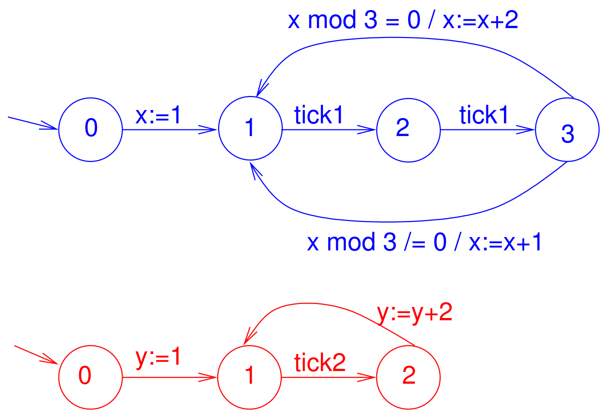
If we ignore local variables (and the number of passes in the loop), what's a global state in the product? How many such states are there?

Extension: Conditions on Local Variables

```
// Thread T1

x := 1 ;                                         // Thread T2
while true loop
  tick1 ;
  tick1 ;              ..........                y := 1 ;
   if x mod 3 = 0 then                           while true loop
     x := x+2 ;                                     tick2 ;
   else                                             y := y+2 ;
     x := X+1 ;                                  end loop ;
   end if ;
end loop ;
```

Automata



Product

If the value of `x` is expanded (the global state mentions the value of `x`), then the conditional transitions can be built exactly.
From a state where `x mod 3 = 0` build one transition, from a state where `x mod 3` $\neq$ `0`, build the other one.
We could also "remember" only `x mod 3` in the global state (finite set).

If the value of `x` is not expanded (and `x mod 3` is not expanded either), we build two transitions with `x mod 3 = 0 / x:=x+2` and `x mod 3` $\neq$ `0 / x:=x+1` as "decorations".

Granularity?

A transition `x mod 3 = 0 / x:=x+2` is acceptable in an asynchronous product only if this is atomic.

Otherwise? ... build the graph from the assembly-line code, not from the high-level programming language. But same remark on the expansion of variables.

Part III

System-Level Design and Modeling for Verification

Outline

Static or Dynamic (guarantee before use or monitor&repair)

static = everything that can be done by looking at the system description only (the program, the HW architecture), without executing it.

dynamic = everything that depends on one (or a set of) particular execution(s).

For embedded systems, "monitor&repair" is usually not an option, hence the need for static validation methods.

Exhaustive or Partial, positive or negative answer

Types of answers from a validation tool:

- In any operating conditions, for all possible inputs, my system will be correct
- There exists a sequence of inputs that make my system fail
- There *may* exist sequences of inputs that make my system fail

Exercice

compare with the potential answers of a testing tool...

Exhaustive or Partial. Java initialization detection: does this program compile?

```
void myfunction ( int a, int b ) {
    int i, j ;
    if ( (a+b)*(a+b) ==
        (a*a + 2*a*b + b*b) ) {
            i = 42 ;
    }
    j = i + 1 ;
}
```

ERROR: variable i might not have been initialized

A compiler cannot prove that $\forall a, b . a^2 + 2ab + b^2 = (a+b)^2$.
It gives a conservative answer.

Conservative analysis

Definition: an analysis that may *reject correct* systems, but that may never *accept incorrect* systems.

Exercice

Compare with 9-proof for arithmetic operations.

$$a \text{ op } b = c \implies$$
$$((a \mod 9) \text{ op } (b \mod 9)) \mod 9 = (c \mod 9)$$

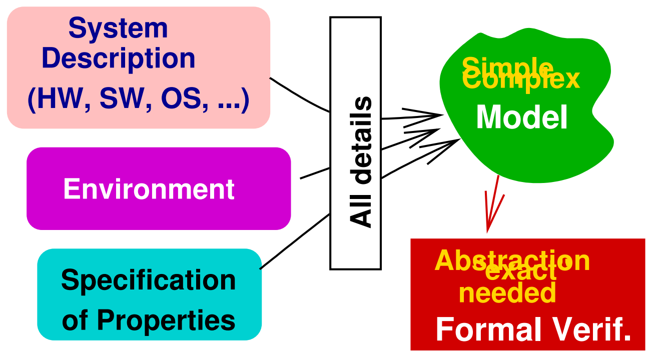
Automatic or Assisted

"push-button" tools: provide the program and a property to be checked on it (*e.g.: does my program execute a divide by zero for some executions?*), and the tool answers automatically.

Assisted tools are based on theorem-provers (PVS, Coq, ...) and require human help.

For general systems (i.e., including arithmetic computations):
— automatic implies partial.
— exhaustive implies assisted.
(because of decidability problems)

The General Picture and Abstractions



Model-checkingAbstract interpretation

2007 Turing Award: Model-Cheking

ACM Turing Award Honors Founders of Automatic Verification Technology
`www.acm.org/press-room/news-releases/turing-award-07/view`
ACM, the Association for Computing Machinery, has named Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis the winners of the 2007 A.M. Turing Award, widely considered the most prestigious award in computing, for their original and continuing research in a quality assurance process known as Model Checking. Their innovations transformed this approach from a theoretical technique to a highly effective verification technology that enables computer hardware and software engineers to find errors efficiently in complex system designs. This transformation has resulted in increased assurance that the systems perform as intended by the designers. (...) developed this fully automated approach that is now the most widely used verification method in the hardware and software industries.

Ariane 501 and Abstract Interpretation

`http://en.wikipedia.org/wiki/Ariane_5_Flight_501`

*Ariane 5 Flight 501
From Wikipedia, the free encyclopedia
Flight 501, which took place on June 4, 1996, was the first, and unsuccessful, test flight of the European Ariane 5 expendable launch system. Due to a malfunction in the control software the rocket veered off its flight path 37 seconds after launch and it was destroyed by its automated destruct system when high aerodynamic forces caused the core of the vehicle to disintegrate. It is one of the most infamous computer bugs in history.
The breakup caused the loss of the payload: four Cluster mission spacecraft, resulting in a loss of more than US$370 million.*

Ariane 501 and Abstract Interpretation

Flight 501's high profile disaster brought the high risks associated with complex computing systems to the attention of the general public, politicians, and executives, resulting in increased support for research on ensuring the reliability of safety-critical systems. The subsequent automated analysis of the Ariane code was the first example of large-scale static code analysis by abstract interpretation.

Three types of Models

- Model of the environment
  (physics, human beings, and other computer systems)
- Model of the computer system under study
  (HW, SW, OS, ...)
- The property to be checked

The distinction between safety and liveness properties

safety
Something wrong will never happen

liveness
Something good will eventually happen

Examples

- When a message is sent, it will be received (L)
- It is never the case that the two lights A and B are on at the same time (S)
- Whenever the temperature reaches the threshold T, an alarm is emitted within 10 ms (S, this case is also called *bounded liveness*)
- Whenever the temperature reaches the threshold T, an alarm will be emitted sometime in the future (L, often useless)

Safety property : when it is false, there exists a point in time where you know that the property is false (looking only at the past). You can write another program (called an *observer*) that observes the behavior of the system without modifying it and enters a special sink state ERROR when the property has become false.

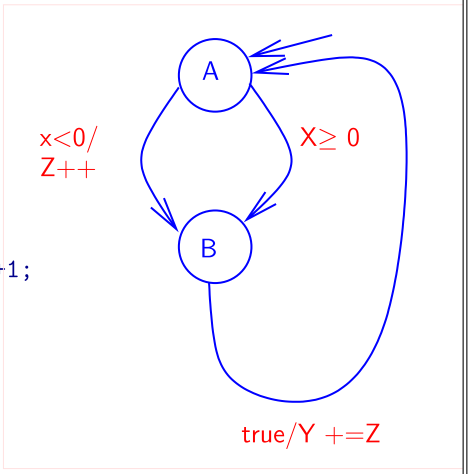
Expressing safety properties with observers

Idea: the program is an automaton.
The (safety) property is an automaton with an error state.
Their synchronous product is also an automaton with an error state.

The property is said to be true for the program if the error state of the product is unreachable from the initial state.

An example program

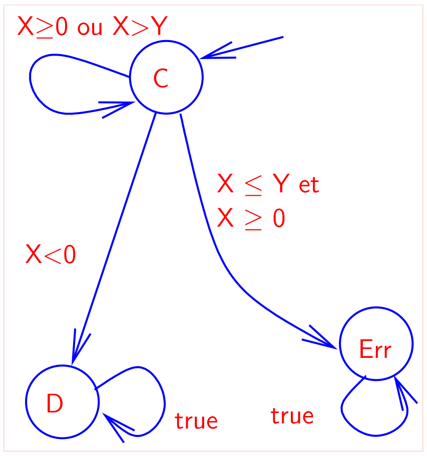
```
declare
  X : integer := -20 ;
  Y : integer := -30 ;
  Z : integer := 0 ;
begin
  A: while true loop
    if X < 0 then Z:=Z+1;
    end if ;
    B: Y := Y+Z ;
  end loop ;
end
```



An example property

At each program point (A, B) :

```
  (X > Y)
or else
  (X < 0)
  in the past
```



The synchronous product of the program and the property

Exercice: start the construction of the synchronous product, from the initial state. Discuss the labels of the transitions.

Remarks on the Analysis of the Product

All the states of the form xE are indeed unreachable from the initial one. How to prove it automatically?

— The tool needs to “know” facts like: $X \geq 0$ and $X < 0$ is not satisfiable. Is it possible?

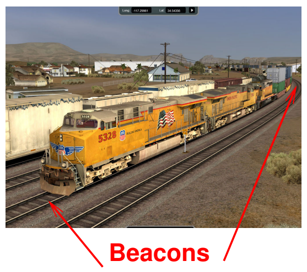
Yes, with this particular form of constraint (build a convex polyhedron, check for emptiness).

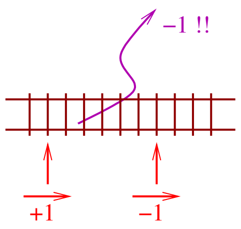
Modeling the environment, examples

Most of the properties we want to prove on computer systems are false if we do not express and take into account reasonable assumptions on the environment of the computer system. Examples:

- For a railway anti-collision system: trains do not land on tracks between beacons
- Same system: trains do stop at traffic lights
- For a temperature controller: when the heater is on, the temperature increases (physical model)
- For an ABS system: when the brake is activated, the car reacts according to some physical model

Modeling the environment: Trains do not Fly!





The number of trains in the section can be known exactly by counting the inputs +1, -1 in the train control SW.

Models for SW, HW, OS, Environments, Properties?

What kind of object can be used to represent:

- The behaviors of a piece of SW, or OS
- The behaviors of a piece of HW
- The (simplified) behavior of a physical environment
- The (safety) properties to be verified

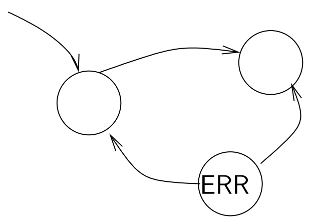
General Interpreted Automata can be used +
Synchronous or asynchronous products to model concurrency +
Synchronous product with observers for the safety properties.

Summary of the verification approach

A system (possibly concurrent) P + a safety property ϕ
$\longrightarrow$
Several automata for P + one automaton for ϕ with Error states
$\longrightarrow$
A single automaton A (obtained by computing products) with Error states

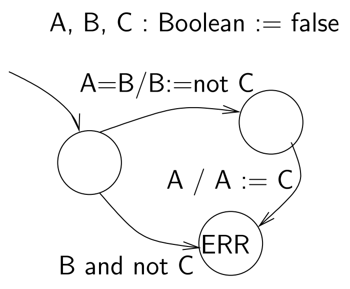
The property ϕ is true for P $\iff$
the Error states are unreachable in A

Reachability of States - 1) no labels



Interpreted automata where:
conditions are always true and
assignments have no effect
(X:=X).
Use classical graph algorithms.

Reachability of States - 2) Boolean labels



Programs with Boolean variables only:
— Expand all variables and get previous case or
— Use (Boolean) symbolic model-checking: sets of states are described by formulas like
`A = B and not C`

Reachability of States - 3) General I.A.

In a general interpreted automaton, reachability of a state is undecidable (i.e.: there is not algorithm that can compute it for all cases).

- If we limit the size of data, and expand all variables : reachability is decidable (easy: the graph is finite).
- But general I.A. = no limit on the size of data : the expanded graph is infinite. Moreover, the formulas that describe sets of states are too general for a computer.
- Approximation on the set of reachable states: always possible, sometimes not too rough.

Reachability of States and Conservative Analyzes

Believe that the ERROR state is reachable, while it is not =
Declare that the (safety) property is false, while it is true =
Conservative answer.

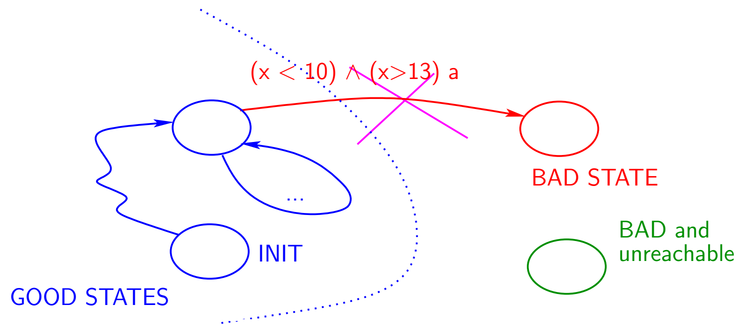
Interesting if the set of programs for which the property is true while it is declared false is not too big = precision of the approximation.

Reachability of States and Approximations of the Sets of States

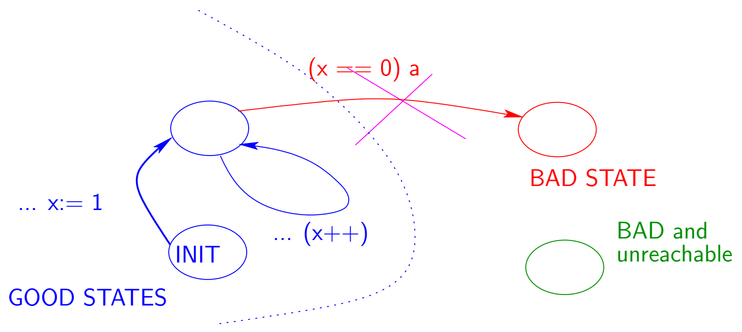
If we are able to compute a superset of the reachable states, we may declare a state to be reachable while it is not.

Enables approximate and conservative safety property verification.

Superset of reachable states - example 1



Superset of reachable states - example 2


(x == 0) a
BAD STATE
... x:= 1
... (x++)
INIT
BAD and unreachable
GOOD STATES