v1.1
BETA

Sistemi Concorrenti e di Rete LS

II Facoltà di Ingegneria - Cesena
a.a 2008/2009

[module 2.2]

VERIFICATION OF CONCURRENT PROGRAMS - BASICS

FORMAL METHODS

- Errors in concurrent programming and concurrent systems cannot be discovered by debugging and corrections cannot be checked by testing
  - > need of formal methods to specify and the verify rigorously the concurrent programs (systems)
- Two principal (class of) formal techniques:
  - **model checking**
    - where verification is done by generating one by one all the states of the systems and by checking the properties state by state
    - can be automated by model checkers tools
  - **inductive proofs of invariants**
    - invariant properties are proved by induction over the states of the system
    - can be automated by tools called deductive systems
- Both techniques rely on some kind of *formal language / calculus to specify correctness properties*

CORRECTNESS PROPERTIES IN PROPOSITION CALCULUS

- With propositional calculus, correctness properties are expressed as *logic formulae* that must be true in order to verify the property in some state of the system
  - formulae are assertions obtained by composing propositions through logic connectors
    - and, or, not, implications, equivalence
- In our case propositions are about *the values of the variables* and *of the control pointers* during an execution of a concurrent programming
  - e.g. given the boolean variable `wantp`, an atomic proposition (assertion) `wantp` is true in a certain state if and only if the value of the variable `wantp` is true in that state
- Each label of a statement of a process will be used as an atomic proposition whose interpretation is "the control pointer of that process is currently at that label"
  - e.g. p1 proposition asserts that the control pointer of the process p is at the label p1.

AN EXAMPLE: MUTUAL EXCLUSION

Third attempt	
boolean wantp ← false boolean wantq ← false	
p	q
loop forever p1: *non-critical section* p2: wantp ← true p3: await !wantq p4: *critical section* p5: wantp ← false	loop forever q1: *non-critical section* q2: wantq ← true q3: await !wantp q4: *critical section* q5: wantq ← false

- Formula $p_4 \wedge q_4$
  - is true if both control pointers of the processes are in the critical section
- if it exists some state in which this formula is true, then it means that *the mutual exclusion property is **not** satisfied*

> dually, a program *satisfies the mutual exclusion property* if the formula $\neg(p_4 \wedge q_4)$ is true for *every possible state of every scenario*

TEMPORAL LOGIC

- Processes and systems change their state over the time, and then also the interpretation of formulae about their state can change over the time.
  - > we need a formal language/calculus that would take this aspect into the account
  - > *temporal logic* is one of the most basic and popular one
- The **temporal logic** is a formal logic obtained by adding temporal operators to propositional or predicate logic
  - – **Linear Temporal Logic (LTL)**
    - to express properties that must be true (at a state) for *every possible scenario*
    - linear / discrete model of time
  - – Branching temporal logics
    - to express properties that must be true in some or all scenarios starting from a state
    - an example: CTL (computational tree logic)

LTL: TEMPORAL OPERATORS

- LTL is based on two basic temporal operators: *always* and *eventually*
  - **box** or **always** temporal operator: $\Box$A
    - the formula $\Box A$ is true in a state s_i of a computation if and only if the formula A is true in *all* states s_j with $j >= i$
      - synonim: $\Box$ p = G p (Globally p)
    - the always operator can be used then to specify *safety properties*, because it specifies what must be always be true
  - **diamond** or **eventually** temporal operator: $\Diamond$A
    - the formula $\Diamond A$ is true in a state s_i of a computation if and only if the formula A is true in *some* states s_j with $j >= i$
      - synonim: $\Diamond$p = F p (Finally p)
    - the eventually operator is used to specify *liveness properties*, because it specifies something that eventually be true

BASIC PROPERTIES

- Reflexivity: $\Box A \rightarrow A$

  $A \rightarrow \Diamond A$

- Duality: $\neg \Box A = \Diamond \neg A$

  $\neg \Diamond A = \Box \neg A$

- Sequences of operators: $\Diamond \Box A$

  $\Box \Diamond A$

DEDUCTION WITH TEMPORAL LOGICS

- Temporal logic is a formal system of deductive logic with its own axioms and rules of inference
  - it can be used to formalize the semantics of concurrent programs and used to rigorously prove correctness properties of programs
- An example of a theorems in TL:

$(\Diamond\Box A1 \land \Diamond\Box A2) \rightarrow \Diamond\Box (A1 \land A2)$ is true.

$(\Box\Diamond A1 \land \Box\Diamond A2) \rightarrow \Box\Diamond (A1 \land A2)$ is false

SPECIFYING SAFETY PROPERTIES

- Box operator can be used to specify safety properties
  - as properties that must be always true
- $\Box P$, where $P = \neg Q$ and Q is the description of a bad state
  - an example: mutual exclusion in CS problem

First attempt	
Integer turn ← 1	
p	q
loop forever p1: *non-critical section* p2: await turn = 1 p3: *critical section* p4: turn ← 2	loop forever q1: *non-critical section* q2: await turn = 2 q3: *critical section* q4: turn ← 1

  - mutal exclusion property: $\Box\neg(p_3 \wedge q_3)$

SPECIFYING LIVENESS PROPERTIES

- Diamond operator can be used to specify liveness properties
  - as conditions that eventually will be true
- $\Diamond P$, where P is the description of a good case
  - an example: *progress property (no starvation)* in CS problem

First attempt	
Integer turn ← 1	
p	q
loop forever p1: *non-critical section* p2: await turn = 1 p3: *critical section* p4: turn ← 2	loop forever q1: *non-critical section* q2: await turn = 2 q3: *critical section* q4: turn ← 1

  - progress property *for one shot* (no loops): $p_2 \rightarrow \Diamond p_3$
  - progress property *with loops*: $\Box(p_2 \rightarrow \Diamond p_3)$

BINARY OPERATORS

- Always and eventually are unary operators. An example of useful and frequently used binary operator is until
  - **Until** operator: A **U** B
    - A U B is true in a state Si if and only if B is true in some state Sj, j>=i and A is true in all state Sk, i<=k<j.
    - That is: eventually B becomes true and that A is true until that happens
  - **Weak-Until** operator: A **W** B
    - like Until operator, but formula B is not required to become true eventually. If it does not, A must remain true indefinitely
    - A W B = as long as A is false, B must be true

OVERTAKING

- Consider the following scenario in the CS problem

```
try-p,try-q,CSq,try-q,CSq,...,CSq,CSp
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

            1000 times
```

- It’s not an example of starvation...
  – it is true that ◇CSp
  > but it’s evident too that freedom from starvation can be a very weak property!
- in some cases we want to ensure that a process would enter its critical section within a reasonable amount of time

K-BOUND OVERTAKING PROPERTY

- ***k*-bounded-overtaking property**
  - from the time a process p attempts to enter its critical section, another process can enter at most *k* times before p does
  - Example: 3-overtaking
    - `try-p,try-q,CSq,try-q,CSq,try-q,CSq,CSp`

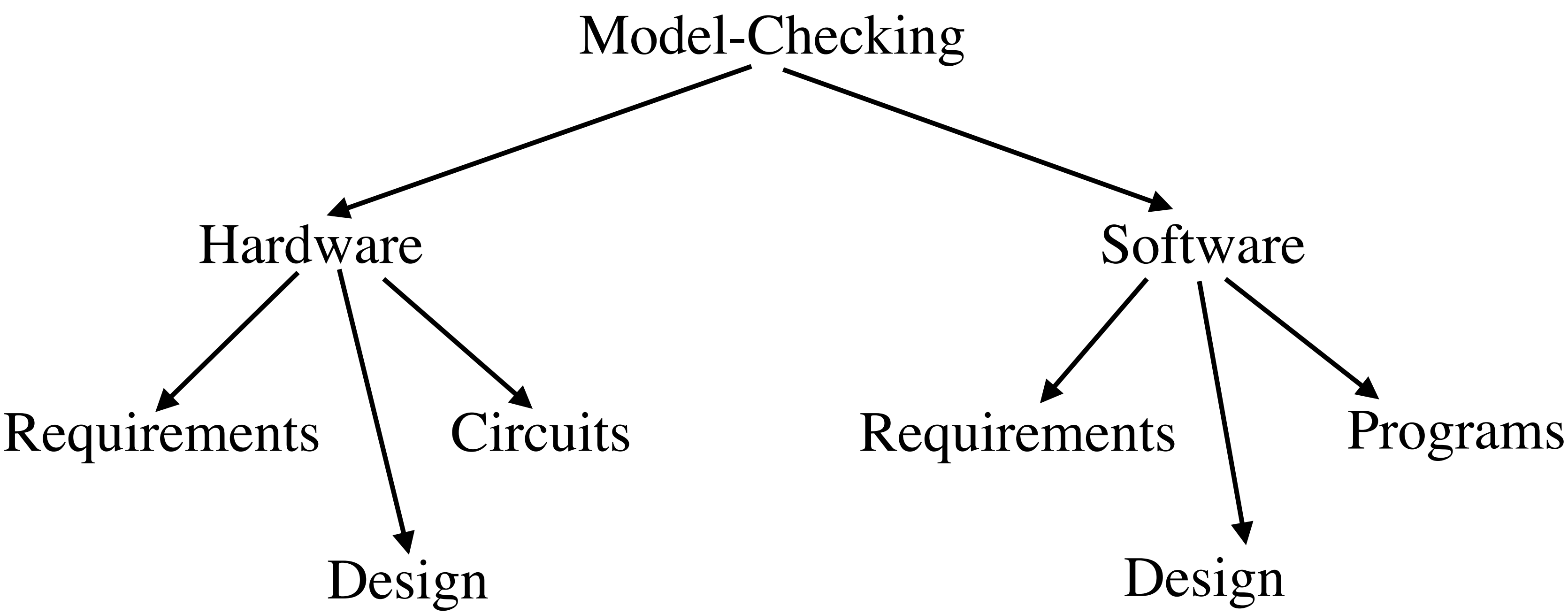
- The property can be expressed by the weak until operator W
  - example with 1-bounded-overtaking:

$$try_p \rightarrow \neg cs_q \text{ W } cs_q \text{ W } \neg cs_q \text{ W } cs_p$$

VERIFICATION TECHNIQUES (1/2): MODEL-CHECKING

- Model checking is the most important and used technique for automatically checking correctness properties of concurrent systems
  - invaluable conceptual and practical tool for software engineers
- Strategy based on exhaustively searching the entire state space of a system and verify if certain properties are satisfied
  - properties as predicates on a system state or states, expressed as a logical specification such as propositional temporal logic formula
  - if the system satisfies the property, the model checker generates a *confirmation* response
    - otherwise, it produces a *trace* (counterexample) => useful also to identify bugs, not only to prove correctness
- SW vs. HW model checking
  - can be applied also to hardware
  - e.g. Intel adopting Model-Checking after the Pentium Bug in 1994
  - used in mission critical software systems
    - e.g. NASA after Mars Polar Lander incident in 1999

MODEL-CHECKING APPLICATIONS



- Program model checking
  - application of the model-checking techniques to software systems
    - in particular to the final implementation
    - discovering software defects

DEALING WITH THE STATE-SPACE EXPLOSITION PROBLEM

- The big problem of model-checking technique is the size of the state space
  - how to manage graph of millions of states? Is it feasible ?
- State-of-the art techniques
  - applying rules to reduce the number of states
    - using variables that can be modeled by a limited number of values
  - incremental construction of the whole graph
    - exploring only reachable state of an execution.
    - checking the truth of a correctness specification as the incremental diagram is constructed, stopping the construction is a falsifying state is found
  - *symbolic* model checking
    - working with set of states

SPIN AND PROMELA

- **SPIN** is a widely used model-checker used in both academic research and industrial software development
  – extremely efficien
  – used in modeling and designing concurrent and distributed systems in general
- **PROMELA** is the language that is used in Spin to write concurrent programs modeling language
  – limited number of constructs intended to be used to build models of concurrent systems

AN EXAMPLE: DEKKER IN PROMELA

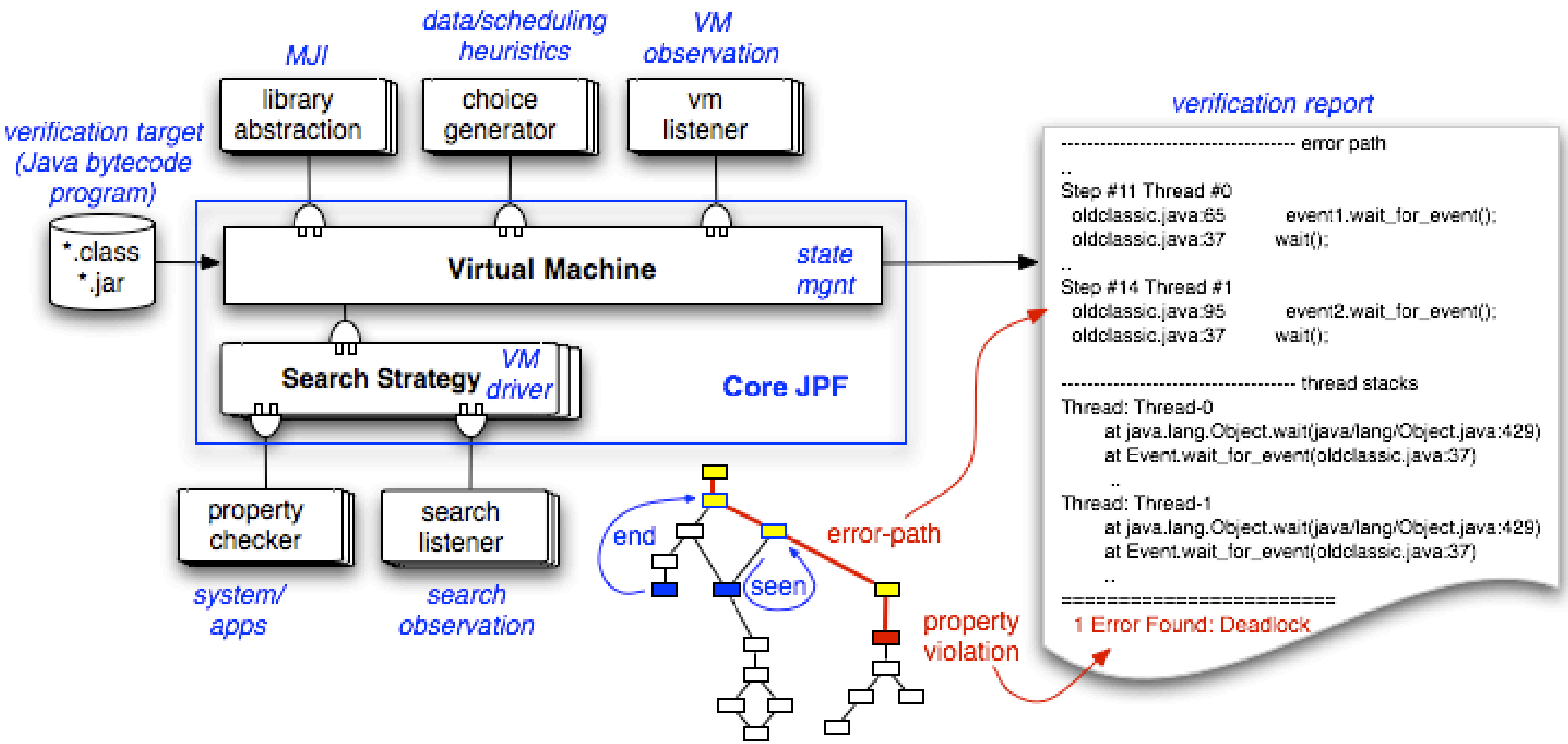
```
bool    wantp = false, wantq = false;
byte    turn = 1;

proctype p() {
    do ::
        wantp = true;
        do :: !wantq -> break;
           :: else ->
            if :: (turn == 1)
               :: (turn == 2) ->
                   wantp = false;
                   (turn == 1);
                   wantp = true
            fi
        od;
        printf("LOG: p in CS\n");
        turn = 2;
        wantp = false
    od
}

proctype q() { /* similar */}

init {
  atomic {
    run  p
    run  q
  }
}
```

JAVA PATH FINDER (JPF)

- JPF is a recent model-checker specialized for the verification of programs written in Java
  - developed by NASA, used for critical software
  - open-source project
    - http://javapathfinder.sourceforge.net/

- JPF is a special JVM executing programs theoretically along all possible scenarios (execution paths), checking for property violations
  - deadlocks, uncaught exceptions, etc
  - If it finds an error, JPF reports the whole execution that leads to it

JPF MODEL OF OPERATION



VERIFICATION TECHNIQUES (2/2): INDUCTIVE PROOF OF INVARIANTS

- **invariant**
  - a formula that must be invariably true at any point of any computation
    - e.g. $\neg(p_4 \wedge q_4)$
- Invariants can be proved using **induction** over the states of all the computations:
  - to prove that A is an invariant:
    - prove that A is true in the initial state (*the base case*)
    - assume that A is true in a generic state S *(inductive hypothesis)* and prove that A is true in all the possible state next to S *(inductive step)*
- Deductive systems
  - software systems for automated theorem proving

NOTE ABOUT SAFETY AND LIVENESS PROPERTY VERIFICATION

- safety property are easier to verify
  - a safety property must be true at all states
    - it is sufficient to find a state not veryfing the property to complete the verification
  - a liveness property claims that a state satisfying a property will inevitably occur
    - it is not sufficient to check states one by one, it is necessary to check all possible scenarios
    - > it requires more complex theory and software techniques