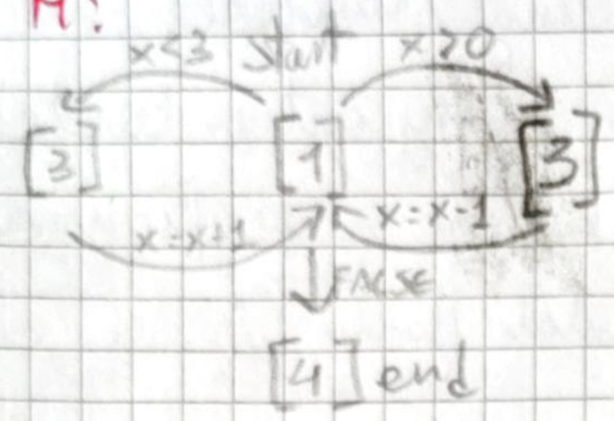
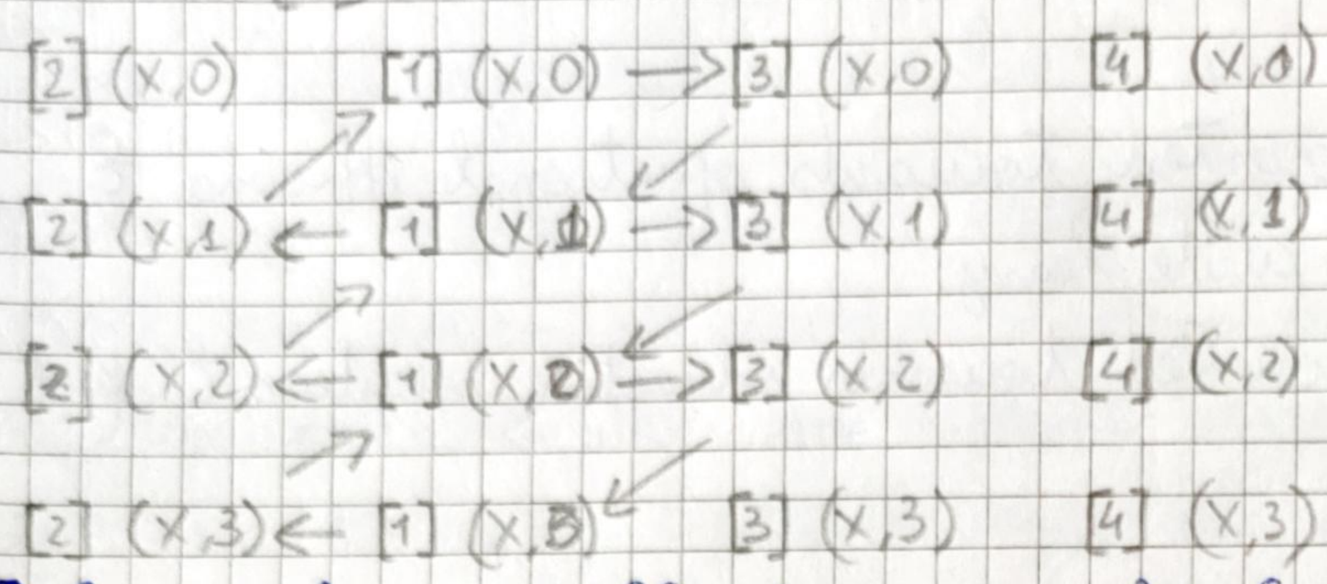
FM:



$L([1],(x,i)) = \{①1, ①(x,i), start\}, \dots, L([4],(x,i)) = \{①4,(x,i), end\}$

↳ set of all possible states built automatically.



PROPERTIES:

- for each initial state it is possible to terminate: start ⇒ EF end;
- for each initial state it is certain to terminate: start ⇒ AF end;

↑ if mem. has k variables taking in values $\{0, \dots, n-1\}$

↳ COMPLEXITY OF MODEL = n^k

MODEL CHECKER = program that can determine wether or not a CTL formula holds a transition system

Let $TS = (S, I \to AP, L)$ ← labelling function

Let $S_0 \subseteq S$

- Reachability in 1 step:

$Reach_1(S_0) =$ ← ASSUME EMPTY SET AS RESULT

$R := \{\}$ for each $\sigma \to \sigma'$ if $\sigma \in S_0$ then $R := R \cup \{\sigma'\}$

- Reachability in n steps:

$Reach(S_0) =$

$R := S_0$ while exists $\sigma \to \sigma'$ with $\sigma \in R \wedge \sigma' \notin R$ $R := R \cup \{\sigma'\}$

TO REDUCE MODEL CHECKERS' ALGORITHMS WE USE THE FOLLOWING ASSUMPTIONS:

- There are transitions leaving all states.
  ↳ To meet it we add self loops on stuck states.
- The set of states S is finite.

$Sat(tt) = S$ ← "TRUE" is satisfied by all states in S;

$Sat(ap) = \{\sigma \in S \mid ap \in L(\sigma)\}$ ← "ATOMIC PROPOSITION" is satisfied by all states in S that are labelled by an atomic proposition;

$Sat(\phi_1 \wedge \phi_2) = Sat(\phi_1) \cap (\phi_2)$;

$Sat(\phi) = \{\sigma \mid \sigma \models \phi\}$ ← set of states that satisfy the formula;

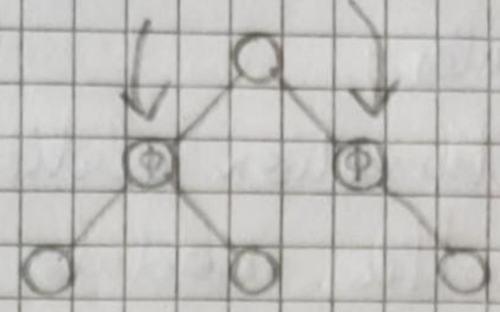
$Sat(\neg\phi) = S / Sat(\phi)$ ← we remove from S all states that dont satisfy ϕ;

$Sat(EX\phi) = \{\sigma \mid (Reach_1(\sigma) \cap Sat(\phi)) \neq \{\}\}$ ← next step exists one path;

$Sat(AX\phi) = \{\sigma \mid (Reach_1(\sigma) \quad Sat(\phi))\}$ ← in one step all successors satisfy ϕ;

$Sat(EX(\phi))$

$Sat(AX(\phi))$

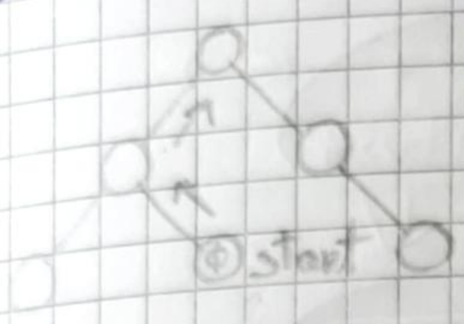


$Sat(EF(\phi)) = \{\sigma \mid (Reach(\{\sigma\}) \cap Sat(\phi)) \neq \{\}\}$ ← states that in any number of steps reach ϕ

↳ ANOTHER ALGORITHM:

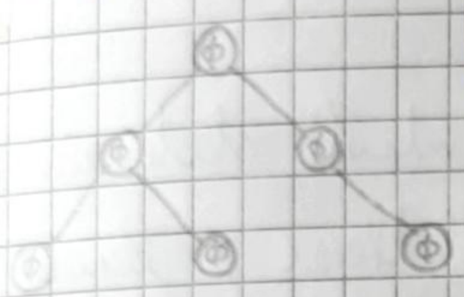
$Sat(EF\phi) =$ WE START FROM ϕ & GO BACK

$R := Sat(\phi)$ while $\sigma \to \sigma'$ with $\sigma \notin R$ & $\sigma' \in R$ do $R := R \cup \{\sigma\}$



$Sat(AG(\phi)) = \{\sigma \mid (Reach(\sigma) \subseteq Sat(\phi))\}$ ← for all paths I always have ϕ

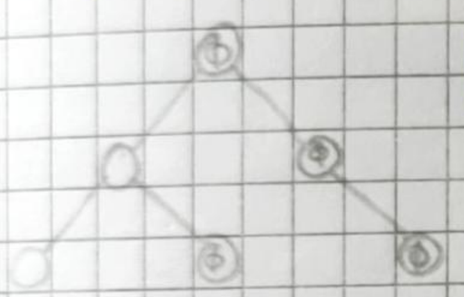
↳ OTHERWISE: $Sat(AG(\phi)) = \neg(EF(\neg\phi))$



$Sat(EG(\phi)) = \cap_n F^n(Sat(\phi))$ ← exists a path in every state

↳ WHERE: $F(S') = \{\sigma \in S' \mid (Reach_1(\{\sigma\}) \cap S' \neq \{\}\}$

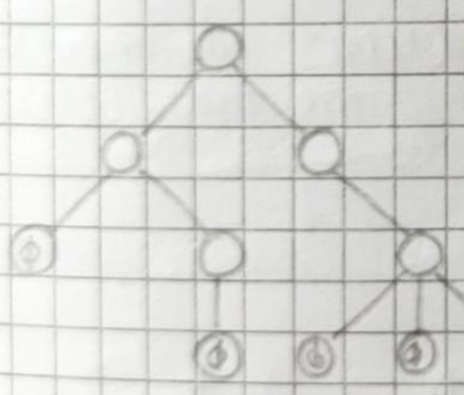
$Sat(\phi) = F^0(Sat(\phi))$ & then we remove states until each remaining state has a successor within the resulting set



$Sat(AF(\phi)) = S/(\cap_n F^n(S/Sat(\phi)))$ ← for all paths we eventually reach ϕ

↳ WHERE: $F(S') = \{\sigma \in S' \mid (Reach_1(\{\sigma\}) \cap S') \neq \{\}\}$

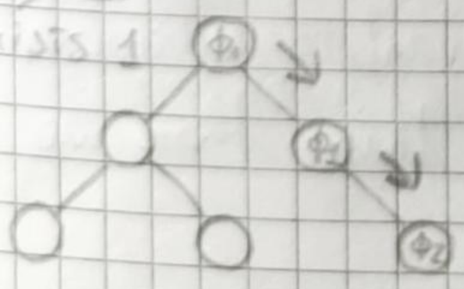
↳ OTHERWISE: $Sat(AF(\phi)) = \neg(EG(\neg\phi))$



ONTIC

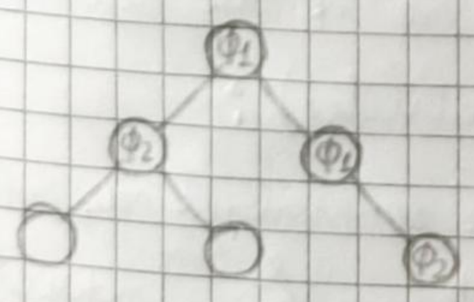
$Sat(E(\phi_1 \cup \phi_2)$ ← it satisfies only if ϕ_1 holds till we reach ϕ_2

EXISTS 1



ALL PATHS

$Sat(A(\phi_1 \cup \phi_2))$



MODEL CHECKER BASED ON A TRANSITION SYSTEM & CTL: NuSMV

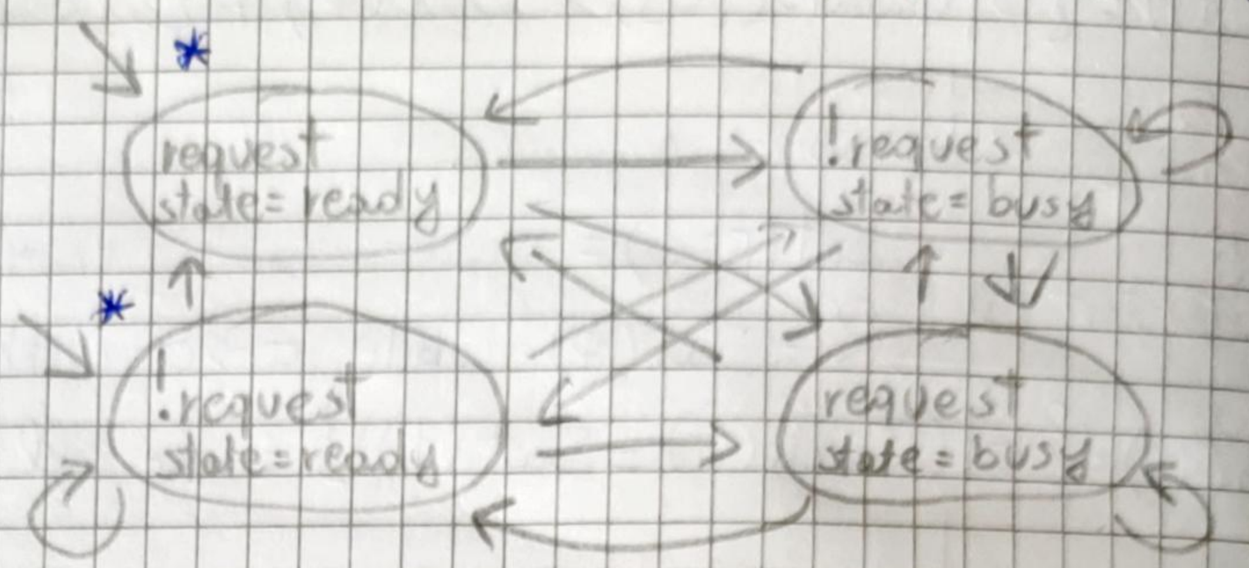
↳ System = program of 1 or more modules;

↳ Module = variables + 1 initial state + transition relation defined over states;

↳ Modules instantiated as vars. in other modules;

↳ Modules can be Synchronous (all do 1 st. together) or Asynchronous;

↳ EXAMPLE: FSM




```
MODULE main
VAR
request : boolean;   <- INPUT VALUE
state {ready, busy};
ASSIGN
init(state) := ready;
next(state) := case
  state = ready & request = TRUE : busy;   <- STATE BECOMES BUSY
  TRUE : {ready, busy};   <- IF NONE OF THE ABOVE WE DO THIS, IT MOVES RANDOMLY
esac;
```

* ATT! Initial state can be any of these 2 because I set init(STATE) = ready

⇓ WE CAN CHECK FORMULAS:

```
SPEC AG (state = busy | state = ready);
SPEC EF (state = busy);
SPEC EG (state = busy);
- SPEC AG (state = ready & request = true) -> AX state = busy;
```

FOR ALL THE STATES THAT HAVE THESE CHARS WE HAVE A NEXT STATE THAT SATISFIES

CYBERSECURITY ENGINEERING: (SLIDE PAGE 12)

① CONCEPT PHASE: Basis of all the activities is the "item definition" = item + operational environ.

↳ we collect: -1) existing information; -2) item boundary; -3) item functions; -4) prelim. architecture; -5) information about operational enviroment of the item relevant to cybersecurity

After "item definition" we need to identify cybersecurity goals, in this phase we do TARA methods (each phase is input to the next):

1) Asset identification: we can have many for same item definition;

2) Damage scenario: for each possible scenario I have an asset & I see if the cybersecurity properties are affected (confidentiality, Integrity & Availability):

3) Threat scenario identification: identify target asset, cause of compromise, compromised sec. property

4) Impact rating: To each damage scenario we assign a "Impact category" & "Impact rating" (SFOP);

5) Attack path analysis: for every threat scenario we build a tree that has the threat as root & all the steps in the attack path:

6) Attack feasibility rating: We assign this to each attack path depending on the damage/effort needed;

7) Risk value determination: for each threat scenario, the risk determined from the impact of the damage scenarios and the attack feasibility of the associated attack paths;

8) Risk treatment decision: for each threat scenario, considering the risk value, one or more of the following treatment option is determined:
- avoid risk; - reduce risk; - share risk; - retain risk;

At this point we identify the cybersecurity goals & the cybersecurity claims if we retain/share the risks!

After we design the cybersecurity concept: cybersecurity requirements for the item and for the operating environment, & the measures that must be implemented to modify the risk.

FM3

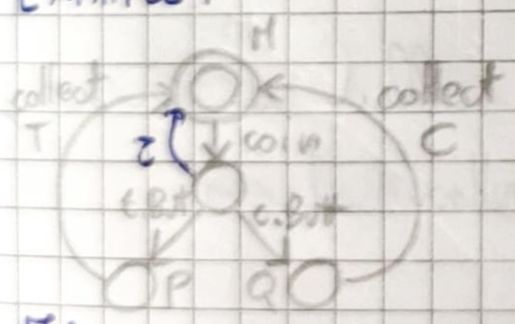
- Process algebras: tool to model concurrent systems. We will assume models using CCS: ⇝ CALCULUS OF COMMUNICATING SYSTEMS

⇝ system = set of communicating processes;

⇝ each process executes actions & sync with other processes;

⇝ τ = special unobservable action that is internal

EXAMPLE:



NON DETERMINISTIC CHOICE

ACTION PERFORMED

M = coin: (P+Q)

P = teaButton: collectTea: M

Q = coffeeButton: collectCoffee: M

If machine can fail we add an internal action τ as above!

R = TAU: M;

The semantics of process algebras are Labelled Transition Systems (LTS)

↳ An LTS is a 4-tuple $A = (X, x^0, Act \cup \{\tau\}, \rightarrow)$

↳ X = finite set of states;

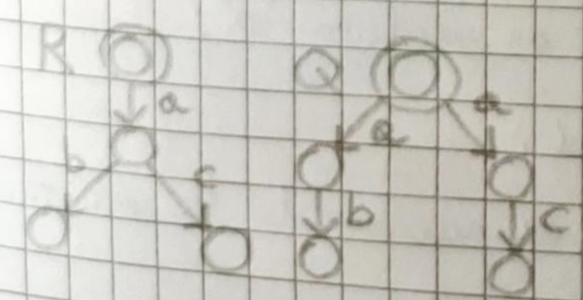
x^0 = initial state;

Act = finite set of observable actions;

$\rightarrow$ = transition relation;

} $x \xrightarrow{a} x'$, $a \in Act \cup \{\tau\}$, is a transition from state x to x' executing action a;

EXAMPLE:



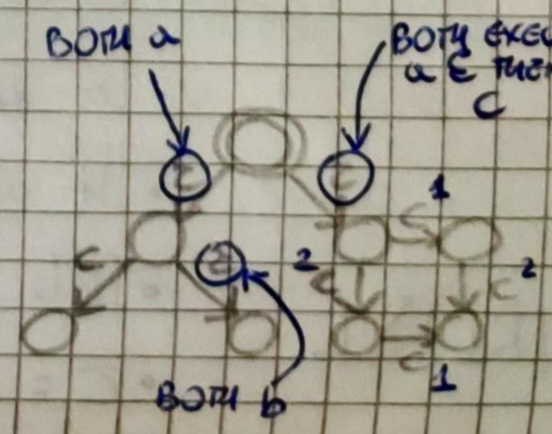
⇒ $P = (R \| Q) \backslash a \backslash b$

PARALLEL COMPOSITION

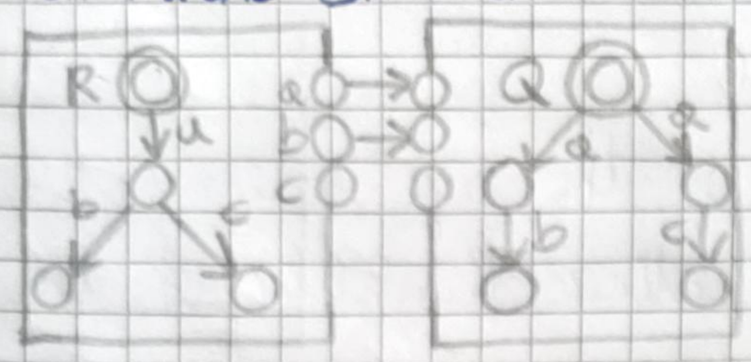
ACTION RESTRICTION

SO BOTH a OR b OR NO ACTION

ACTION CAN BE PERFORMED ONLY WITHIN A COMMUNICATION



GRAPHICAL SPECIFICATION OF PROCESS P:



ACTL: Action-based Computation Tree Logic ← based on actions rather than states; ⇝ interpreted over LTSs.

↳ uses 3 types of formulae:

1) action formulae: permits to express restraints on the actions that can be observed;
2) state formulae: gives a characterization about the possible ways an execution can proceed after a state has been reached;
3) path formulae: states properties of an execution;

def: (Traces equivalance):
Those systems that perform the same sequences of actions are equivalent.
For example Q & R have same traces: {ac, ab};

def: (Strong simulation): ← DOESN'T WORK IF WE HAVE τ!
$A = (Q, q^0, Act, \rightarrow)$ & S: binary relation over Q ⇝ then S is called a strong simulation over $(Q, Act, \rightarrow)$ if, whenever pSq, if $p \xrightarrow{a} p'$ then there exists $q' \in Q$ such that $q \xrightarrow{a} q'$ & $p'Sq'$
For example Q & R have strong simulation;

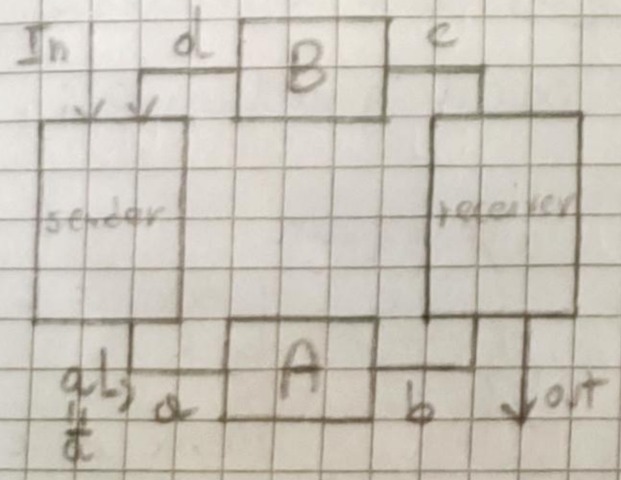
def: (Strong bisimulation):
$A = (Q, q^0, Act, \rightarrow)$ & S: binary relation over Q ⇝ then S is called a strong bisimulation over $(Q, Act, \rightarrow)$ if both S & S^{-1} are strong simulations.
For example Q & R are not strong bis;

def: (Weak bisimulation):
Equal to strong but we consider even τs! ⇝ we can have τs & they can be different but in the end observable actions are the same!

def: (Strongly bisimilar): 25

def: (observational equivalence): 28

ALTERNATING-BIT PROTOCOL:

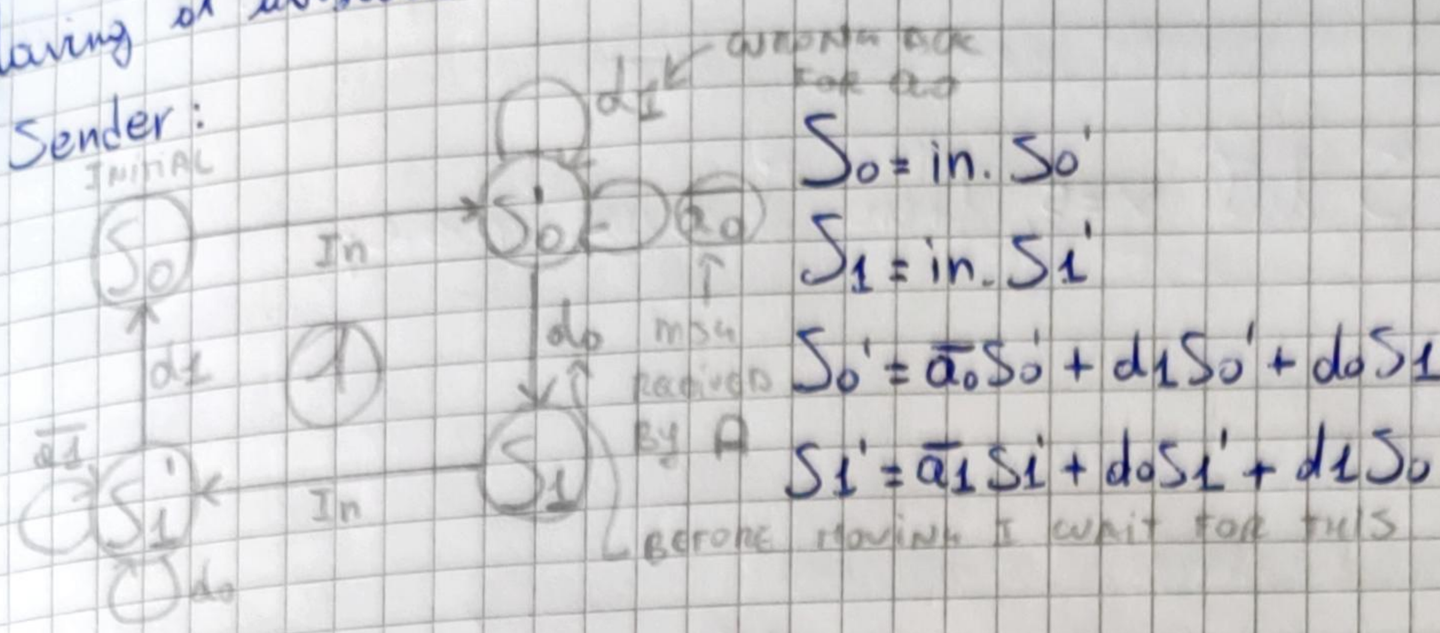


$Sys = (S_0 | A | B | R_1) \setminus L$

- For every input we have an output (synchronized);
- Faults happen when I lose a message;
- For example "Sender" & "A" have to sync on action "a";
- If we send a_0 next time we send a_1;
- After bit is received, receiver send c_0 or c_1 according to the one received;
- τ = omission of messages or acknowledgments;

Having at abstract level: $P = in.out.P$, are P and Sys weak bisimulation equivalent?

- Sender:



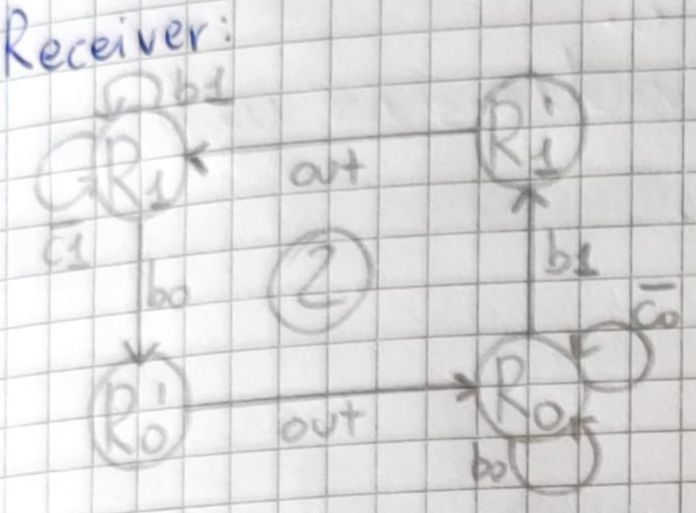
$S_0 = in.S_0'$

$S_1 = in.S_1'$

$S_0' = \bar{a}_0 S_0' + d_1 S_0' + d_0 S_1$

$S_1' = \bar{a}_1 S_1' + d_0 S_1' + d_1 S_0$

- Receiver:



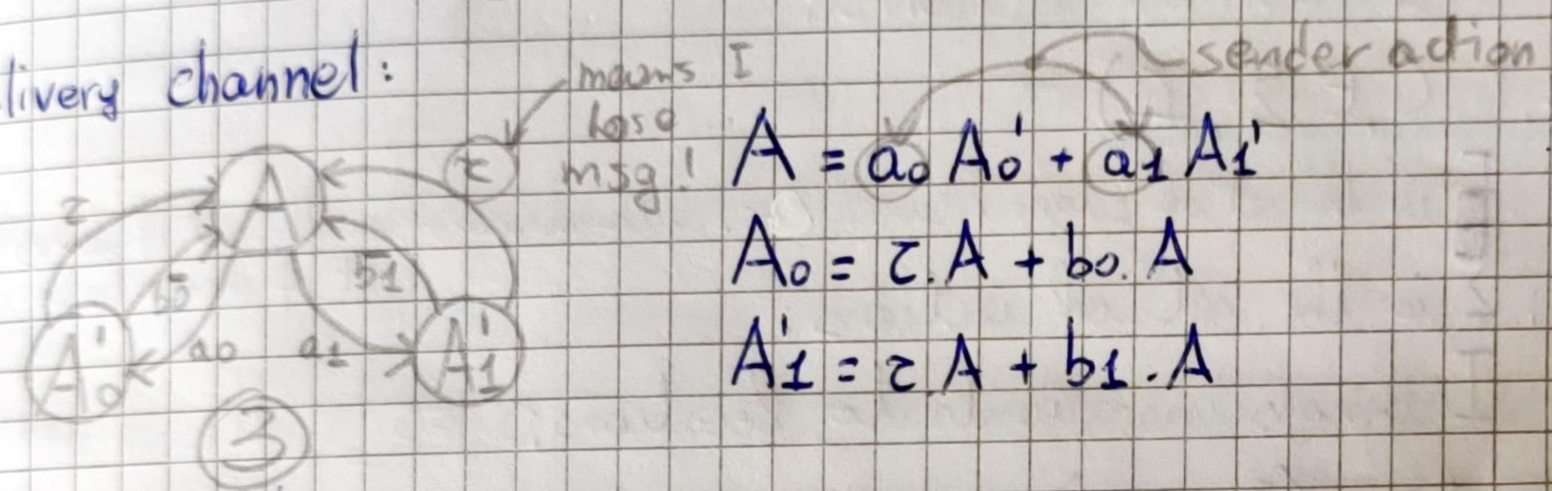
$R_0' = out.R_0$

$R_1' = out.R_1$

$R_0 = b_0 R_0 + \bar{c}_0 R_0 + b_1 R_1'$

$R_1 = b_1 R_1 + \bar{c}_1 R_1 + b_0 R_0'$

- Delivery channel:

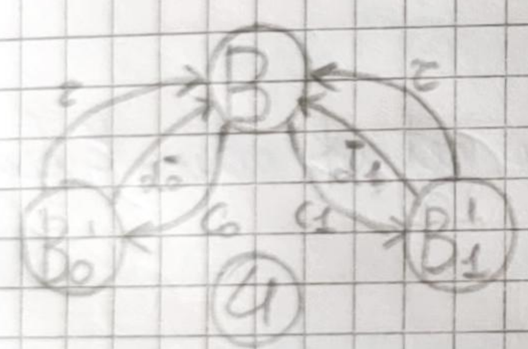


$A = a_0 A_0' + a_1 A_1'$

$A_0 = \tau.A + b_0.A$

$A_1' = \tau.A + b_1.A$

- Ack channel:

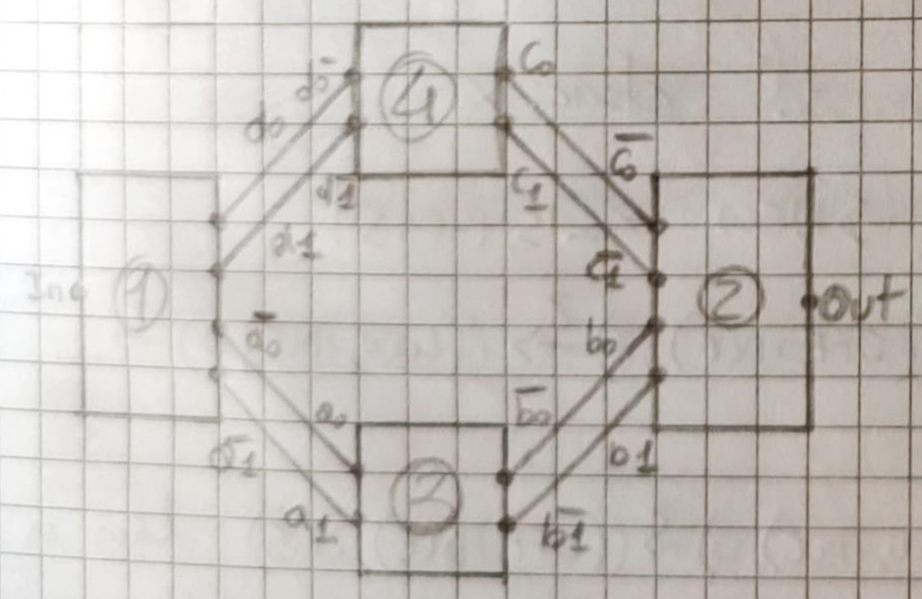


$B = c_0 B_0' + c_1 B_1'$

$B_0' = \tau B_0 + \bar{d}_0 B$

$B_1' = \tau B_0 + \bar{d}_1 B$

- The system Sys:



All actions are synchronized so they are all hidden!

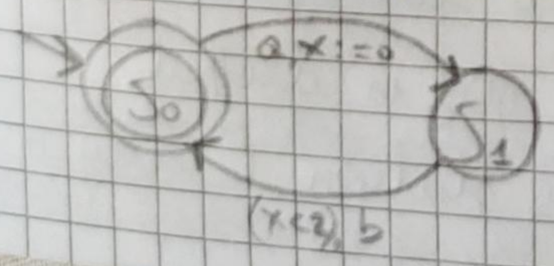
LTS of Sys is generated automatically by the network of the process

↳ P is observational equivalent to Sys: $P \approx Sys$

↳ yes it is bisimulation equivalent

Timed Automata:

- Automaton and real-valued variables for measuring time between transitions, named clocks;
- Clocks progress synchronously with time & can be reset by transitions;
- Transition = guard + action + reset of clocks;

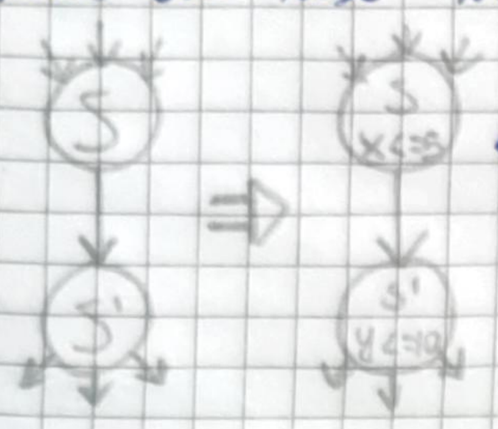


← 1 clock: x

↳ x:=0 is a reset of clock

↳ x<2 is a guard

Every action can be done or not so we may remain in a state forever! To avoid this we use "invariants":



← I can stay in S till $x=5$, then I must move;

NETWORK OF TIMED AUTOMATA:

→ Complementary actions: $a?$ & $a!$ are complementary or matching.

In a network of timed automata, the complementary actions are synch actions and become the network internal action, denoted by τ.

↳ COMPLEMENTARY ACTIONS;

Timed automaton: $A = (L, l_0, C, E, \Sigma, I)$

L = finite set of locations; E is the set of edges between loc. with an action, on the set of clocks & a set of clocks to be reset;

$l_0 \in L$ is the initial location; Σ is the set of actions;

C is a finite set of clocks; I maps invariants to locations;

Semantics of a timed automaton: $\langle S, S_0, \rightarrow \rangle$ = LABELED TRANSITION SYSTEM;

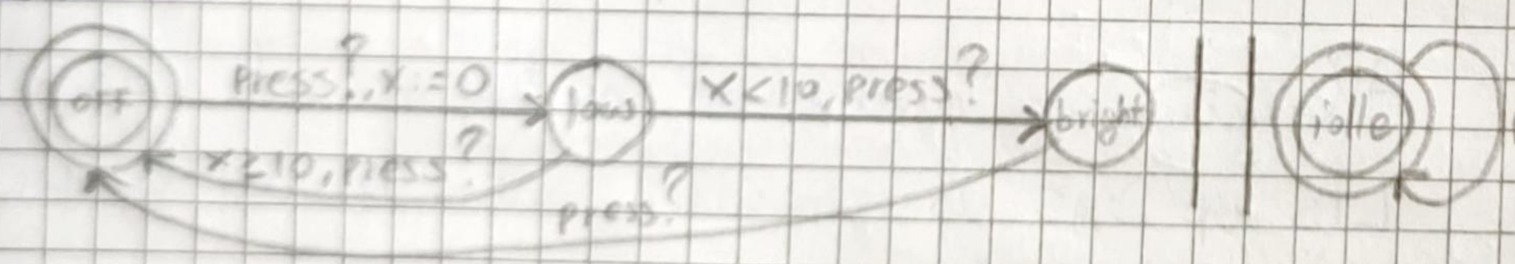
S = set of states; S_0 = initial state; $\rightarrow$ = transition relation;

Network of timed automata: n timed automata over the same set of actions & clocks:

$A_i = (L_i, l_i^0, C, E_i, \Sigma, I_i)$;

Semantics of a network of timed automata: $(S, S_0, \rightarrow)$

EXAMPLE:



clock: x

↳ NEW: $x \leq 15$, press!

$((OFF, IDLE), 0) \xrightarrow{\tau} ((LOW, IDLE), 0) \xrightarrow{\tau} ((BRIGHT, IDLE), 5.1) \xrightarrow{\tau} ((OFF, IDLE), 7) \xrightarrow{\tau} ((LOW, IDLE), 0) \xrightarrow{\tau} \ldots$

What happens now if I insert a guard to idle?

WE CANNOT DO SOMETHING LIKE THIS: $((OFF, IDLE), 0) \xrightarrow{\tau} ((LOW, IDLE), 0) \xrightarrow{\tau} ((OFF, IDLE), 22.5)$ this is due to the fact that after reaching $x > 15$ we cannot use the button because the guard will never be satisfied.

WSN PROTOCOL:

→ Flooding: 1-TO-N routing protocol where a dedicated node needs to communicate general information to all the nodes of the network. A node forwards the msg to the neighbors which did not already received it, otherwise the msg is dropped.

↳ PROPERTY P: every node receives all the msgs sent by the base station, & every msg that was received is forwarded only once → FROM 96 TO 144 SE NO TEMPO

FMa:

To model systems using logic we use PVS:

- ⇝ specification of the system;
- ⇝ animation of the specification;
- ⇝ formal proofs;

Nodes are numbered from Ø;

We use 2 basic elements: network graph & network state;

↳ function execute: given a network graph, a network state & a protocol function, applies the protocol to the network graph & state & returns a new network state.

TUTTA ROBA DI PROGRAMMAZIONE GUARDA SLIDE!

DOMANDE:

① What is the hamming code? How many errors can we fix?

The hamming code is one of the possible information redundancies techniques. We represent information with more bits than strictly necessary, for example if we have a n-bit information we add c bits to get a m bit information; In the end we have only 2^n acceptable code words over the 2^m possible.

We can define separable code (attend original information with new inf.) or non-separable code (requires more complicated decoding).

We use hamming distance: number of bit positions on which 2 code words differ ⇝ this is equal to the number of independent single bit errors that the code can detect ⇝ efficency = $2^n/2^m$

↳ with hamming distance = K we can detect K-1 single bit errors & correct up to d errors where $d=(K-1)/2$

EXAMPLE: Parity bits spread through all the data word:

BIT POSITION		1	2	3	4	5	6	7	8	9	10	11
ENCODED DATA BITS		P1	P2	d1	P4	d2	d3	d4	P8	d5	d6	d7
PARITY BIT COVERAGE	P1	✓		✓		✓		✓		✓		✓
	P2		✓	✓			✓	✓			✓	✓
	P4				✓	✓	✓	✓				
	P8								✓	✓	✓	✓

Parity bits: all bits positions power of 2;

Data bits: all the others

IMPORTANT: "Parity bit P_s covers all bits whose position has the s least significant bit equal to 1"

② What is the Model Checker? Come si usa? Come si formalizza? Come si scrivono le proprietà del modello? Come è la forma delle proprietà della Ctl?

A Model Checker is a program that can determine wether or not a CTL formula holds on a transition system. It derives from the model checking that is a fully automated method for verifying properties of programs/systems. We use

temporal logic, which is a logic for representing and reasoning about propositions qualified in terms of time ⇝ we have 2 notions:

• Branching time: CTL (COMPUTATION TREE LOGIC) ; • Linear time: LTL

A transition system is a tuple: $(S, I, \rightarrow, AP, L)$

⇝ S = non-empty set of states;

⇝ I = non-empty set of initial states;

⇝ $\rightarrow$ = transition relation from 1 state to another;

⇝ AP = set of atomic propositions;

⇝ L = labelling function for states;

↳ ATOMIC PROPOSITION: ϕ

REACHABILITY IN 1 STEP: $EX\,\phi$ & $AX\,\phi$

REACHABILITY: $EF\,\phi$ & $AF\,\phi$

UNAVOIDABILITY: $EG\,\phi$ & $AG\,\phi$

$\sigma \models \phi$ ← a formula ϕ holding in a state σ of model

Passing to MODEL CHECKERS, we have to make some assumptions:

1) There are transitions leaving all states; 2) S is finite

We introduce: $Sat(\phi)$ = set of states satisfying $\phi = \{\sigma \mid \sigma \models \phi\}$, this procedure performs a recursive descent over the formula given as argument; ← FOR MORE FM2!

③ Consensus problem?

Used to achieve agreement on a set of distributed processors that send values.

We use Byzantine generals.

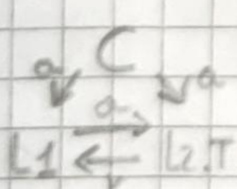
We assume: • n generals; • Each general comunicates $v(i)$ to others;

• $v(i)$ option of general i; • Final decision obtained by majority;

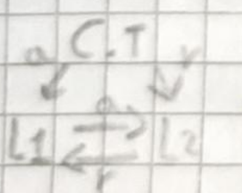
We have "Interactive consistency", using 1 commanding lieutant & $n-1$ generals:

IC1: All loyal generals obey the same command;

IC2: If loyal general then he has to agree with the order of commanding lieutant;



← In this case no solution, if we do not add the rule that in case of disagreement between Ls, the loyals agree with commander. ⇝ satisfy IC2!



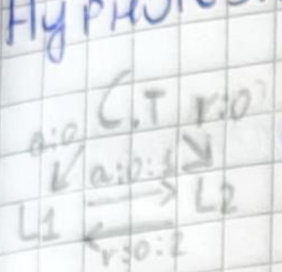
← We can't reach agreement, because IC1 not satisfied! ⇝ we need 4!

We use "Oral message algorithm"

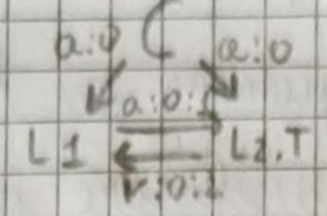
HYPHOTESIS: System is synchronous; 2 processes have direct communication through a network, which cannot fail & has no delay; Sender of msg can be identified;

IMPLEMENTATION: It is a recursive algorithm ⇝ $n-1$ invocations of $OM(m-1)$, each invokes $n-2$ invocations of $OM(m-2)$, ... ⇝ COMPLEXITY: $O((3m+1)^{m+1})$

To resolve any number of traitors we need: $n \geq 3m+1$, with m = n. of traitors;
Since we want to reduce the complexity we impose a limitation:
~> Traitor cannot lie ~> New algorithm;
We use "Signature msg algorithm"
HYPHOTESIS: Anyone can verify the autenticity



← They know C is traitor;



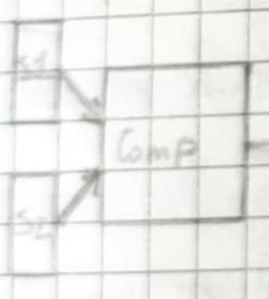
← If L2 tries to forge C signature L1 knows he is traitor!

Consensus problem cannot be solved if system is asynchronous. Since we never know if process has crashed or is very slow!

④ Error detection & Error recovery?

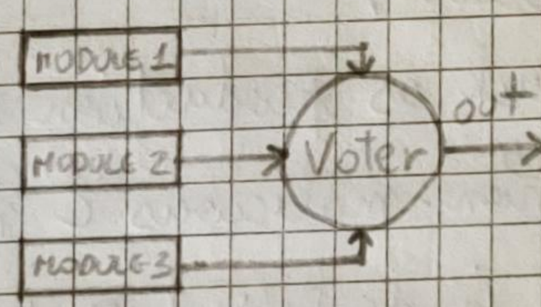
There are 2 techniques of the Fault Tolerance, which is one of the 4 dependability means (prevention, tolerance, removal & forecasting). Generally fault tolerance deals with faults at run-time & delivers correct service in presence of activated faults & errors. This is done through "① error detection & system recovery" or "② error compensation".
① Error detection and recovery -> error handling -> rollback or rollforward;
② recovery -> error handling -> compensation; (known also as fault masking)

Example of error detection:



error if 2 inputs to the comparator are different

Example of error compensation:



error is neutralized without the error detection.

Error detection:

~> types of checks: 1) Replication: comparator;
2) Reasonable: use known semantic of data;
3) Run-time: in hardware;
4) Specification: if result is correct;
5) Reversal: inverse computation;
6) Structural;
7) Timing;
8) Codes;

~> prevent propagation: 1) minimum privilege;
2) sys. design fault tolerance principle;
3) modularization;
4) clear hierarchy & connectivity of components;
5) partitioning;
6) atomic action;

~> measured by: • Coverage; • Latency; • Damage confinement;

Error recovery: 1) forward recovery; 2) backward recovery; 3) Exception handling;
1) Requires to asses the damage caused by the detected error or by errors propagated before detection; (ex: sys. can recover by skipping its response to a missing sensor input);
2) Requires to store a previous state of a system (CHECKPOINT), if we have permanent error then is useless ~> not useful for hw faults & design faults;

3) catch() clauses ~> • interface exception; • internal local exceptions; • failure exceptions.

⑤ Dependability attributes? Chain of threats? Fault classification?

Dependability attributes:

1) Availability: readiness for correct service;
2) Reliability: continuity of correct service;
3) Safety: absence of catastrophic consequences on the user(s) and the environment;
4) Confidentiality: absence of unauthorized disclosure of information;
5) Integrity: absence of improper system alterations;
6) Maintenability: ability to undergo modifications & repairs;

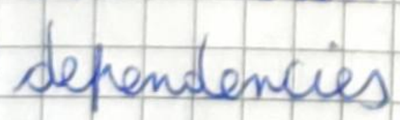
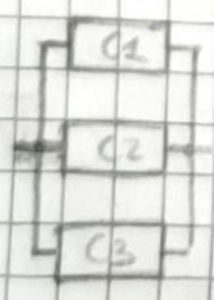
Chain of threats: Fault -> Error -> Failure. It is the error that causes the system to fail ~> fault can be active, if it causes an error, or dormant. Moreover they can be either external or internal. Fault activation: application of an input to a component that activate the fault. Error propagation within a component: it is caused by the computation process. Propagation from component A to B: happens when B receives service from A, service failure of A appears as an external fault to B & propagates the error.

Fault Classification: 8 groups but we divide in 3 major partially overlapping groups 1) Development; 2) Physical; 3) Interaction;

~> Natural faults: are hw faults, such as production defects, these can be internal or external
~> Human-made faults: divided in ① non-malicious & ② malicious;
① nondeliberate or deliberate (done in development or interaction) or accidental or incompetence
② introduced to alter functioning of the system ~> logic (malwares), or intrusion (operational external faults ~> physical means);
~> Interaction faults: during use phase, so all operational faults and external;
~> Permanent faults: continuous & stable;
~> Transient faults: appear & disappear within a very short period of time;

⑥ Components in series or //, how do we ensure reliability? TMR example? Reliability formula TMR?

We use combinatorial models with the following properties:
1) independent components; 2) each component has a failure rate λ; 3) model is constructed on the structure of the system; ATT! Inadequate for sys. with complex dependencies; ~> $R(t) = e^{-\lambda t}$

Series =>	Parallel => at least 1!	Parallel => M out of N
		
$R_{sys} = \prod_{i=1}^{n} R_i(t)$	$Q_{sys}(t) = \prod_{i=1}^{n} Q_i(t)$	$R_{M\text{-}of\text{-}N}(t) = \sum_{i=0}^{N-M} \binom{N}{i} R^{N-i}(t)\,(1-R(t))^i$
$\lambda_{sys} = \lambda_1 + \lambda_2 + \dots$	$R_{sys}(t) = 1 - Q_{sys}(t) = 1 - \prod_{i=1}^{n}[1 - R_i(t)]$	

TMR vs symplex:

$R_m = e^{-\lambda t}$

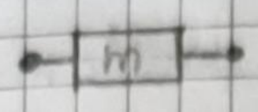
$R_V = 1$

We need 2-outof-3 ~>

$\sum_0^1 \binom{3}{i} (e^{-\lambda t})^{3-i} (1-e^{-\lambda t})^i =$

$3e^{-2\lambda t} - 2e^{-3\lambda t} \leftarrow R_{TMR}(t)$

$MTTF_{TMR} = \frac{3}{2\lambda} - \frac{2}{3\lambda} = \frac{5}{6}\lambda$



$R = e^{-\lambda t}$

$MTTF = 1/\lambda$

Notice: $R_{TMR} > R_{sym}$ if $R_m > 0{,}5$!
The MTTFs tell us that TMR is worse than simplex system, but in reality the reliability of TMR is higher for the first 6000 hrs!

⑦ Explain "random process". Markov chains omogeneous and not? Symplex sys? Diff. between continuos & discrete Markov chain?

Random process = family of random variables indexed by numbers expressing points in time

↳ EXAMPLE: Random process: $\{X_t\}$, with time $t = \{0,1,2,3,\dots\} \rightsquigarrow \{X_0, X_1, X_2, \dots\}$ random vars, if they do not depend one another they are independent.

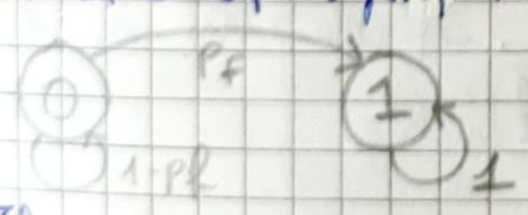
In dependability, these represent the values of the state of the system randomly changing over time.

ATT! A Markov chain is a process that satisfies the "Markov process", which means that the state of the process at time $t+1$ depends only on the state at time t, and it is independent on any state before t. PROP: $P\{X_{t+1}=j \mid X_t=i\} = P\{X_1=j \mid X_0=i\}$ ~ "steady state transition probability", which means that the prob of a transition between i & j doesn't depend by time. ~> DTMC IF $t=0,1,2,3,\dots$

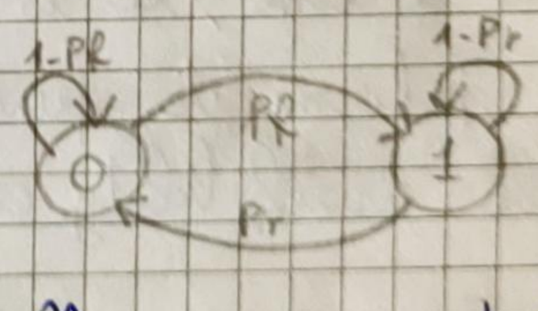
If markov chain satisfies property of "steady-state probability" is homogeneous, otherwise is non-homogeneous.

If markov chain is finite state, then it can be represented by a matrix

Example of symplex system:



$$P = \begin{array}{c} \\ \\ \end{array}\begin{array}{cc} 0 & 1 \\ \end{array} \leftarrow \text{NEXT STATE}$$
$$P = \begin{bmatrix} 1-P_f & P_f \\ 0 & 1 \end{bmatrix} \begin{array}{c} 0 \\ 1 \end{array} \text{ CURRENT STATE}$$

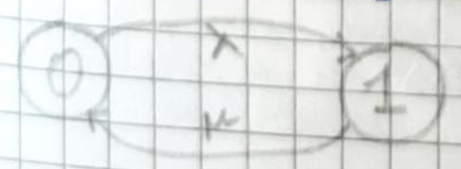


$$P = \begin{array}{cc} 0 & 1 \end{array}$$
$$P = \begin{bmatrix} 1-P_r & P_r \\ P_r & 1-P_r \end{bmatrix} \begin{array}{c} 0 \\ 1 \end{array}$$

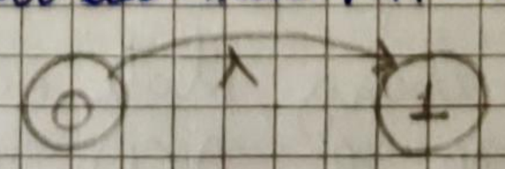
The example above is a discrete time MC. Now we want to analyze CTMCs.

Properties: 1) state transitions may occour at any time; 3) we consider steady state probability;
2) transitions are assigned transition rates; ↳ $P\{X_{t+\tau}=j \mid X_t=i\} = P\{X_\tau=j \mid X_0=i\}$
3) T_i = time spent in a state = continuos r.v with exponential distribution;

This model fits assumption that failure rates are costant, leading to exponential distribution of interarrivals of failures ~> for each state i: $T_i = e^{-a_i}$



$$Q = \begin{bmatrix} -\lambda & \lambda \\ \mu & -\mu \end{bmatrix}$$



$$Q = \begin{bmatrix} -\lambda & \lambda \\ 0 & 0 \end{bmatrix}$$

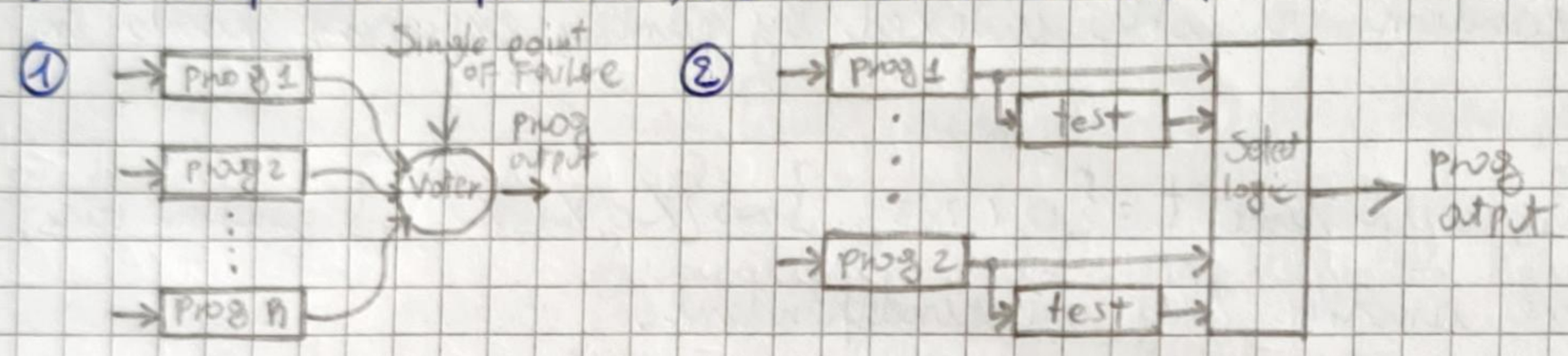
⑧ Advantages of having independent softwares? What faults can we have?

Software redundancy is used to tolerate design faults and operational faults. These design faults are mistakes on the interpretation or implementation of the specification.

We use multi-version approaches ⇝ replicate software, used in safety-critical systems.

↳ N-version programming ①: independently develope and write N codes, then vote on the N produced codes. We compare the outputs of the softwares and through a voter we know the result! Problems in implementing the voter due to sw diversity.

We try N-self-checking programming ②: we have acceptance tests that tell us if the software passed the examination;



⑨ TMR & Symplex? Availability? Failure rate?

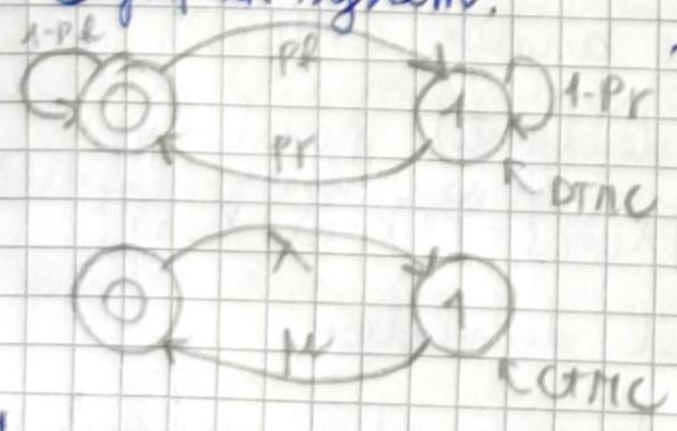
TMR is also called triple modular redundancy & is used for fault masking. We can have 1 of 3 module failure and have the system to still work. Failure rate in a system is given as λ if constant or otherwise as $\lambda(t) = f(t)/R(t)$, which is the failure probability density function ($dQ(t)/dt$) over the reliability function. If we want to compare a symplex system with a TMR we hypothetize that TMR is made we 3 symplex modules and a voter with reliability = 1.

Symplex: $R(t) = e^{-\lambda t}$; MTTF = $1/\lambda$; FAILURE RATE = λ;

TMR: $R(t) = \sum_0^1 \binom{3}{i} (e^{-\lambda t})^{3-i} (1-e^{-\lambda t})^i$; MTTF = $(5/6\lambda)$; FAILURE RATE = $\frac{6}{5}\lambda$;

Availability is a function of time, it is the probability that the system is operating correctly & available to perform its functions at the istant t. If we use a state based model to describe availability we need the repair probability.

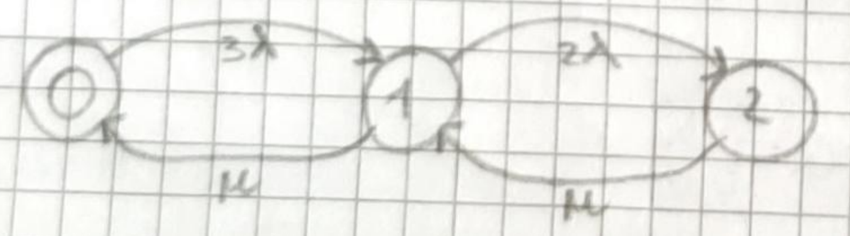
↳ Symplex system:



⇝ To evaluate availability we need to see the prob. that the system is available at time t ⇝ in this case it happens when we are in state 0 ⇝ $\pi_0(t) = A(t)$, but we need to use a CTMC ⇝ $\pi_0(t) = \frac{\mu}{\mu+\lambda} + \frac{\lambda}{\mu+\lambda} e^{-(\mu+\lambda)t}$ ← STEADY STATE

→ IN DTMC STEADY-STATE PROB. is: $\lim_{t\to\infty} \pi(t)$ ← limiting behavior

↳ TMR: CTMC



⇝ STEADY STATE BEHAVIOUR: $\pi(t) = \pi(t-1)P$

The sys. is available until it reaches the state 2, so using state occupancy vectors the availability will be: $A(t) = 1 - \pi_2(t)$

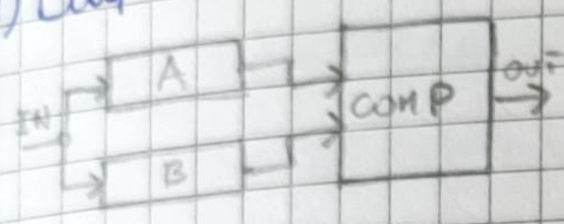
⑩ Active & Passive redundancies? Advantages & disadvantages? Hybrid TMR?

Passive fault tolerance is done through fault masking, but we cannot fix the problem because we do not have fault detection. To have it we need to use active fault tolerance which gives us fault detection, fault location & fault recovery. ⇝ Both HW redundancy techniques.

In passive we have TMR & NMR (coverage of $m=(N-1)/2$ nodes) ⇝ good for transient faults.

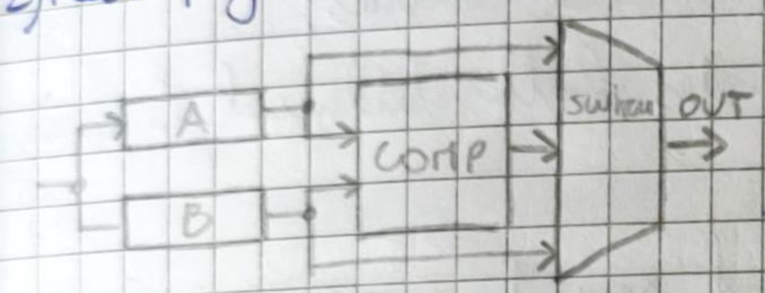
In active we have:

1) Duplication with comparison scheme:



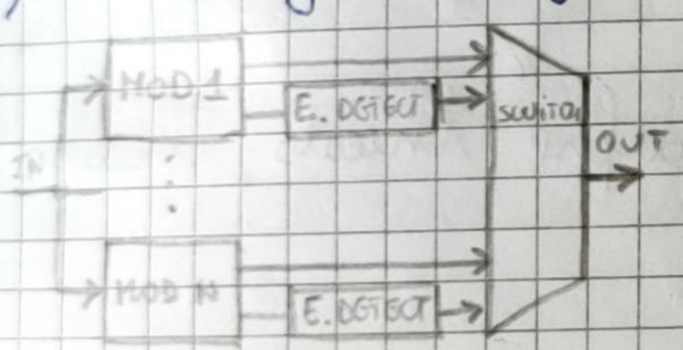
If A & B out are different then we receive an error signal. We must assume that A & B can't fail together & that the comparator cannot fail;

2) Reconfigurable duplication:



If A & B are different through the switch chooses the correct one, the comparator performs checks to see which is the faulty module & the disconnect itself + faulty mod;

3) Stand-by sparing:

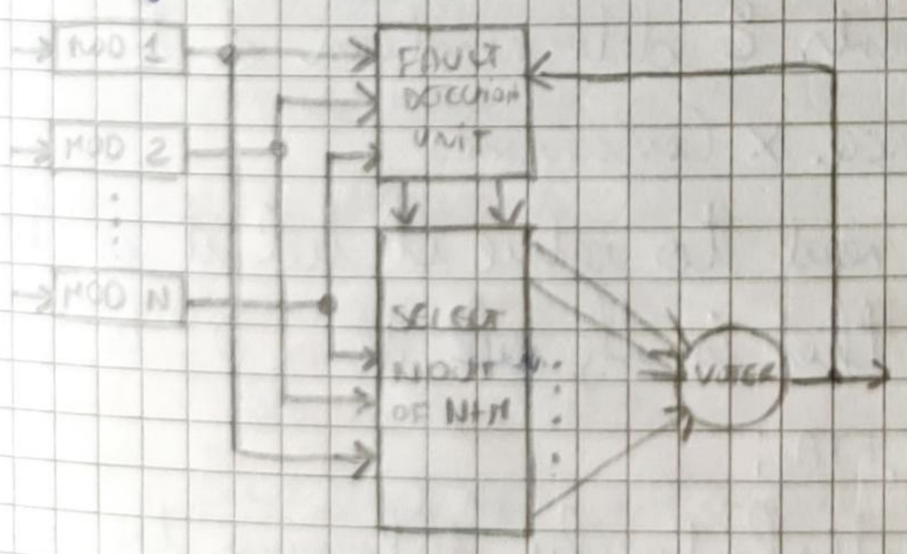


Only part of the modules are operational, when 1 fails it is disconnected ⇝ HOT SPARES + WARM SPARES + COLD SPARES;

↳ PAIR & SPARE APPROACH, WE HAVE 1) INSTEAD OF MOD + E. DETECTION

If we both want active and passive we need a hybrid approach:

Reconfigurable NMR:



This lets us more faults, detect them & repair them. We need always to avoid internal errors in the voter, switch & fault detection unit.

⑪ Threat Modelling? What did we use other than Mokus & Advise? CVSS what is it? What is risk analysis?

We saw 4 other risk assement tools: DREAD, CVSS, OWASP & SAHARA

DREAD: 1) Damage potential: Ranks the extent of damage if a vulnerability is exploited;
2) Reproducibility: either high or low; 3) Exploitability: effort required to exploit vulnerability;
4) Affected users: number of users affected compared to total; 5) Discoverability measures: often we assign max value to it; ⇝ $RISK = \frac{D+R+E+A+D}{5}$ (↳ MAX 10 TO EACH) or $RISK = D+R+E+A+D$ (↳ MAX 3 TO EACH)

CVSS: It is a published standard (common vulnerability scoring system). This captures the principal characteristics of a vulnerability and produce a numerical score reflecting its severity.

↳ based on 3 metric groups: Base, Temporal & Environmental;

Base:
- Exploitability metrics (Access vector, Access complexity, Authentication);
- Impact metrics (C.I.A Impacts);
- Scopus;

Temporal:
- Exploitability;
- Remediation level;
- Report confidence;

(CHANGE OVER TIME)

Environmental:
- Security requirements (C.I.A Requirements);
- Modified base metrics;

↳ RISK = Base metrics (SUM) + refined by temporal & environmental metrics;

OWASP: Open Web Application Security Project. It estimates both technical & business impact factors ⇝ RISK = LIKELIHOOD * IMPACT;

SAHARA: It allows the evaluation of the impact of security issues on safety at the system level. (Security aware Hazard Analysis & Risk Assesment)

Threat quantified according to:
- Required resources; - Know-How that are required to define threats; threats criticality;

Hazard & Risk Analysis:

↳ Analysis of hazards: risk class matrix = combination of 6 categ. of occurence + 4 categ. of consequences; ⇝ frequent to incredible / catastrophic to negligible;

↳ Risk Analysis: Identification of hazardous events & determination of the necessary risk reduction for these events ⇝ RISK = HAZARD FREQ. × CONSEQUENCES

ATT! If we have a high risk for a EUC, we need to reduce it till it is tolerable, after that we have the residual risk that is the diff. between 0 & the residual.

To evaluate risks we use: HAZOP, FMEA, FMEDA, ETA, FTA;

Safety Integrity: prob. of safety related sys. satisfactorily performing the required safety functions under all the stated conditions within a stated period of time;

↳ SIL ⇝ from 1 to 4.

⑫ ECC?

It means error correction codes ⇝ we use two dimensional parity. In the second row we detect an error because parity value is different between the row & the column.

nbit word					row parity
1	0	1	...	0	1
0	0	1	...	1	1 ←
1	1	1	...	0	0
1	0	0	...	0	0

⑬ Attack trees?

Are part of "Attack path analyses", which is used to identify the attack path that realizes threat scenarios. Eventually they are part of TARA that is made up by 7 phases:

-> ITEM + OPERATIONAL ENVIRONMENT

1) Asset identification => Input: item def.; Output: identification of damage scenarios;
2) Threat scenario identification => Input: previous In & Out; Output: id. of threat scenarios;
3) Impact rating => Input: previous In & Out; Output: damage scenarios in impact categories SFOP + rating;
4) Attack path analyses => Input: previous In & Out; Output: build attack trees;
5) Attack feasibility rating => Input: previous In & Out; Output: add feasibility rating to each path;
6) Risk value determination => Input: previous In & Out; Output: build risk matrix;
7) Risk treatment decision => Input: previous In & Out; Output: decide what to do with the risk;

All of this is done to determine cybersecurity goals & claims that then create the cybersecurity concept = cybersecurity requirements for the item and for the operating environment, & the measures that must be implemented to modify the risk.

↑ ALL IN CYBERSEC. CONCEPT PHASE: ITEM DEF. + CYBERSEC. GOALS + CYBERSEC CONCEPT;

⑭ Fault trees? Fault tree THR? Fault event?

Fault tree analysis is a failure analysis in which an undesired state of a system is analyzed using boolean logic to combine a series of lower-level events. ~> Allows us to study how the sys. can fail.

The top event is the status of the system & it is described in terms of the states of the basic events. ~> Fault propagates from leaves to root.

Cut set = set of elementary events that lead to the occurance of the top event.

The reliability of the system is given by $R_{sys} = 1 - \sum Q_{S_i}(t)$, where $Q_{S_i}(t)$ is the prob. that all the components in the minimal cut set S are faulty.

Till now we assumed that we had components, which fail independently, if we have a duplicated component we need to use instead conditional fault trees.

↳ $Q_S(t) = Q_{S|C\,Fail}(t)\, Q_C(t) + Q_{S|C\,notFails}(t)(1 - Q_C(t))$

Generally fault trees are used in conjuction with FMEA (failure mode effect analysis), which is a failure analysis for identifying the risk of failure of a system. We must identify the functionality of the system + all potential failure modes. Then we build a FMEA table to then calculate the risk priority number or RPN = S * O * D = severity * occurance * detectability;

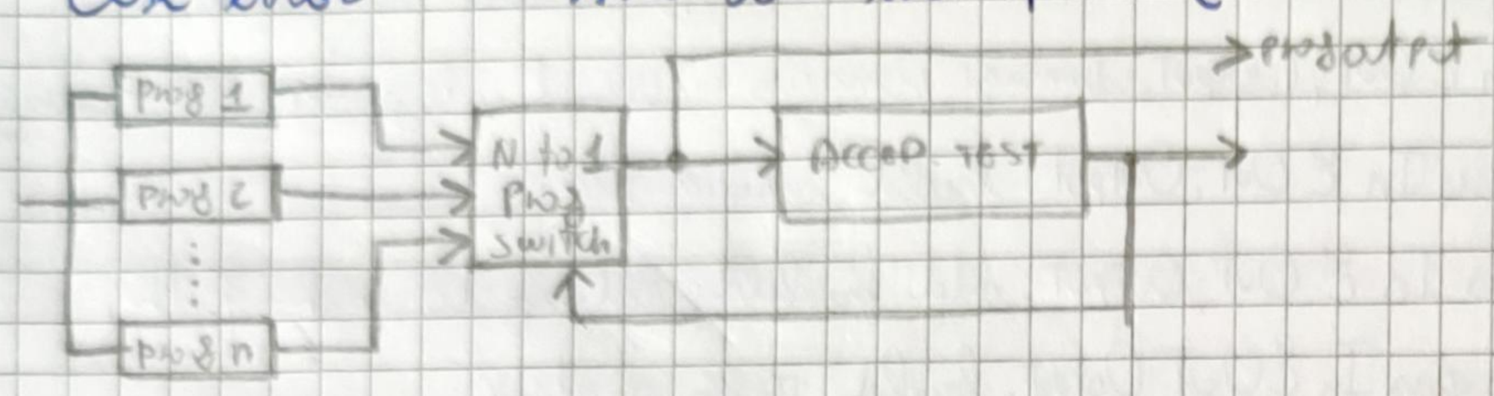
⑮ Functional Safety?

It provides the assurance that the safety-related systems will offer the necessary risk reduction required to achieve safety in presence of malfunctions.

⑯ Recovery Block?

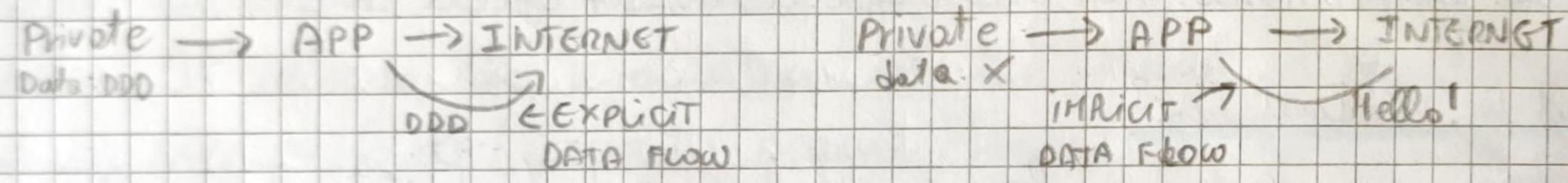
It is based on one acceptance test and a single alternate is run at a time

↳ Ensure T ⇝ We have a checkpoint before trying to ensure T, each time that
By P the test returns an error by a program we return to the
Else by Q previous ch & retry, this is done till the acceptance test is passed or all fail. (Error detection + Backward error recovery).
Else error



⑰ Data leakage def? Examples?

It happens when secure or confidential data is given to unauthorized users.
Explicit data leakage: when data is leaked and is not "encrypted";
Implicit data leakage: when data is leaked and means something that is not recognized

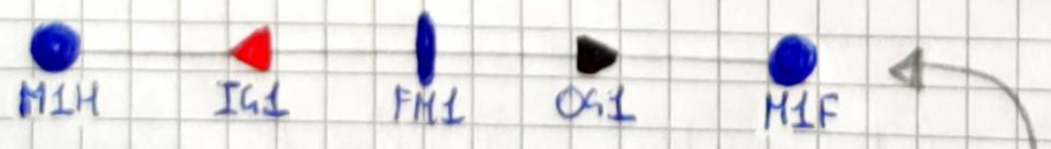


APPUNTI LAB:

TMR MOBIUS SAN: 3 modules + voter ↓ ATOMIC

1) create a place ● & name it Module1Healthy token = 1 to say it is healthy, if it goes to ∅ is faulty;
2) we introduce a timed activity ▮ & name it FaultModule1, the rate is "lambda" & the case quantity = 1
3) we define lambda as a double in the global variables in edit pannel.
4) we introduce the input gate IG1 ◀
5) we introduce the output gate OG1 ▶
6) we connect the timed activity to IG1 & OG1. AT! IN to activity, activity to AT!
7) In IG1 in input predicate we write: "Module1Healthy -> Mark() > 0" & input function we write: "Module1Healthy -> Mark() -- ;"
8) create new place ● & name it Module1Faulty
9) In IG0 in output function we write: "Module1Faulty -> Mark() ++;"

Diagram:



Since we have TMR we replicate this 3 times!

Now we need a voter!!

1) We introduce a new place ● called VoterH & we set its token to 1
2) We introduce a input gate IGV ◀ & in input predicate: "VoterH->Mark()>0" &
in input function: "VoterH->Mark()--;"
3) We introduce a timed activity ▮ called VoterFault & to rate "lambaV" & use quantity=1
4) We introduce a new output gate OGV ▶ & in output function: "VoterF->Mark()++;"
5) We introduce a new place ● called VoterF;
6) Connect like before!
Now we define a new reward & connect it to atomic model;
1) Add variable: • Reliability TMR; • Reliability single module; • Reliability voter;
2) Select "Reliability single module variable" & in Reward function we write:
if (TMR->Module1Healthy->Mark()>0) return 1;
else return 0;
3) Select time ~> Instant of time ~> select range ~> configure;
4) Select simulation ~> Reward estimation;
5) Do all the above for other 2 vars;
6) ATT! For var "Reliability TMR" we need to insert in Rate reward:
if((TMR->M1H->Mark() + TMR->M2H->Mark() + TMR->M3H->Mark() > 1) && (TMR->VoterH->Mark()==1)) return 1;
else return 0;
Now we define the study for global vars. ~> we set lamba to 0.001 & lambaV to 10^{-32}!
↳ We can add experiments to try different values.
Now we need to define the transformer, which is the component that transforms the high level representation of the TMR to a markov chain
Now we define solver as a Simulator
Other way to implement it:
1) module ● with 3 tokens
2) activity ▮ with rate = lamba * module->mark() & use = 1
3) IG1 ◀
4) OG1 ▶

TMR WITH REPMR:

ATT! READ LABO2!
↑ failure detection repair done!!!

Bank tutorial

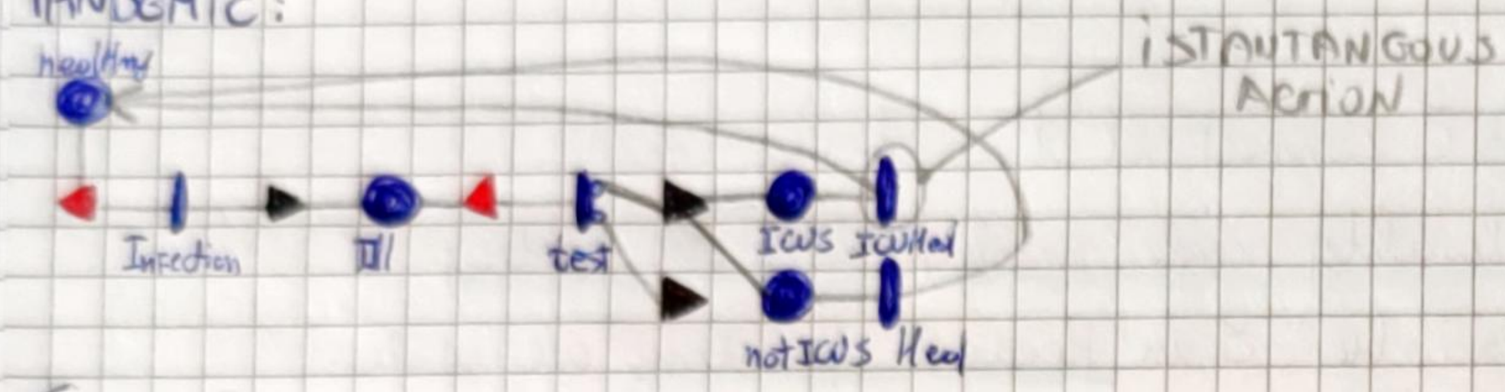
↳ Things adversary needs: Security access, Bank PC, Security manager PC?

↳ Adversary profile ~> combination of Cost & Detection

↳ General they use a backdoor ~> because low detection

↳ Do diff. experiments with various costs & Detection with diff. paths in NOISE model

PANDEMIC:



CYBERPHYSICAL SYSTEMS ~> Distributed control, we need to do error detection + recovery.

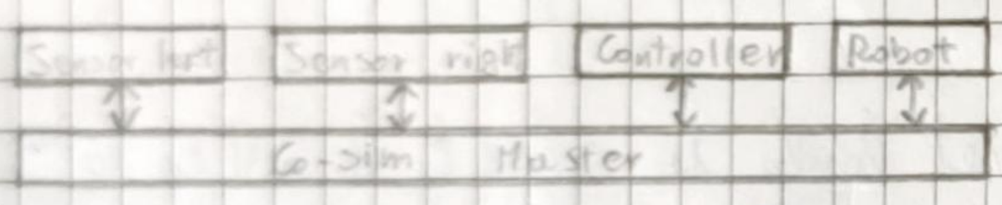
↳ high lvl representation: control component (Alg) with continuos time plant (sensors + actuators)

↳ LFR: Line follower robot ~> up to 4 ligth sensors, 2 wheels, 1 Arduino, 6 batteries.

↳ COSIMULATION = coupling different simulations together

↳ to base case we use a standar co-simulation ~> Functional Mock-up interface, with this we export a model as a fmu & case simulation

↳ LFR schema for co-simulation master:

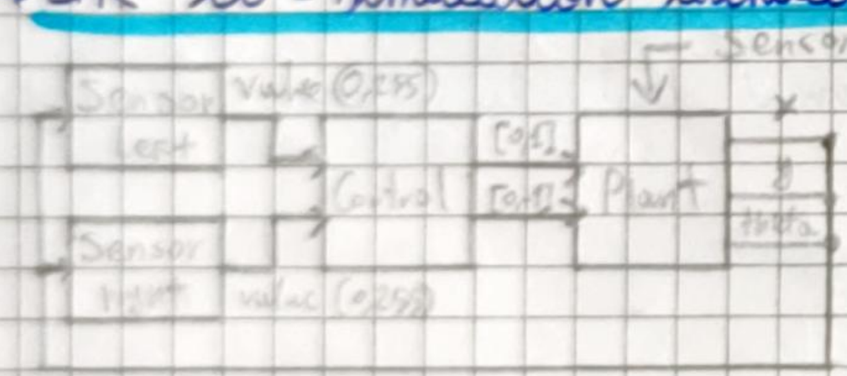


} 4 fmus

IF WE ADD ATTACK fmu WE NEED TO ATTACK BOTH SENSORS!!!

↳ Sensor -> attack -> controller

↳ LFR co-simulation architecture:



The control tells which to activate & Plant uses diff. equations to control x, y & theta

INTO-CPS: multi-models folder tells us how the FMUs are connected between each other. ~> after clicking one of them I can go to configuration, where we have FMUs and we can add or remove them. (this are checked automatically to see if they are supported). For each FMU we can create 1 or more istances, then we can create connections between the istances.

↳ For example: body to sensor we do:

IDENTIFIER OF ISTANCE PREVIOUSLY ADDED

Output: body -> Output var.: robot_x -> Input: sensor1 -> Input var.: robot_x

We can modify parameters of istances too.

In Overview tab we can see all connections

After clicking on a multi-model we can see that co-sim appears on the left under the model ~> we can edit this configuration selecting the algorithm, we use Fixed step: all FMUs perform steps together. We can create simulation with graphs.

PVS: PVS10 USED TO USE FUNCTIONS OF PVS

PVS:
↳ Formal system: system that we use to prove the truth of sentences by deductions, by using sentences, one that we want to prove & one that we know that is true.
A formal system consists in a set of axioms & a set of inference rules.
↓
Formal system F, set of axioms A (GENERALLY TRUE) & set of inference rules R:
We want to prove that a formula S (FOR EXAMPLE SAFETY) follows a set H of hypotheses
↓
Theorem prover is a computer program that implements a formal system. It takes in a formal def. (H & S) & tries to build a proof by application of inference rules.
PVS is an interactive theorem prover
↳ Sequent calculus logic used by PVS:
It is all based on sequents ⇝ which are a certain number of sentences that yeld another certain number of sentences:

$$\underbrace{A_1, A_2, \dots, A_n}_{\text{ANTECEDENTS}} \overset{\text{TURNSTILE}}{\vdash} \underbrace{B_1, \dots, B_m}_{\text{CONSEQUENTS}}$$

↳ Proofs are constructed backwards from the goal sequent, which has form $H \vdash S$;
Inference rules are applied backwards. Since a rule may have 2 premises, proving a goal produces a tree of sequents, called the proof tree.
The proof is completed when (and if!) all branches terminate with a true sequent.
↳ Proved sequent: if one of the following is verified than the sequent is proved:
1) any antecedent is false;
2) any consequent is true;
3) any formula occours both as an antecedent & as a consequent;

x	y	x ⇒ y (IMPLIES)	
F	F	T	1) & 2)
F	T	T	
T	F	F	
T	T	T	2)

⇝ (grind) ← semi automatic part of prover! ⇝ does the proof for us on the branches. If we run this immediately we cannot prove anything, we need before another command.
⇝ To open PVS: open in terminal hidden folder .PVS 6.0 ecc... & then .\pvs, then we need to change the context, to do that we need to press Alt+X, then we write change-context ⇝ Enter ⇝ move to Examples

Theory Landing Weather:

Lemmas should be proved.

↳ To prove them we move cursor on them, then Alt+x & call x-prove. We will have a sequent & the proof tree.

↳ sequent is empty at the start & equal to the lemma that we want to verify

↳ LEMMA raing => not clear

① We have an implication as a consequent, so we use command called (FLATTEN) ⇝ applies change & we have a sequent

{-1} raing

{-2} clear ← THIS IS HERE & NOT AFTER THE |--- BECAUSE IT WAS "NOT CLEAR"

|-------

② Now we can use axioms ⇝ to use them we use (lemma)

If we use (lemma cond_ax1) we import the axiom as an antecedent

{-1} raing => cloudy

[-2] raing

[-3] clear

|------

③ Now we need to split raing => cloudy, we can't use (FLATTEN) because it is an antecedent, but we can use (split) ⇝ creates 2 subgoals, so we have 2 branches

①

{-1} cloudy

[-2] raing

[-3] clear

|------

&

②

[-1] raing

[-2] clear

|------

{1} raing ← MATCHES 3RD CONDITION!!!

④ We use (postpone) to move to ② branch and it tells us that it is verified, so moves us back to ①

We import 3rd axiom:

{-1} clear IFF NOT cloudy

↑ IF & ONLY IF

[-2] cloudy

[-3] raing

[-4] clear

|------

⑤ We can use (flatten to simplify it) ← BECAUSE IT IS AN IMPLICATION & WE HAVE AN AND

{-1} clear implies NOT cloudy
{-2} NOT cloudy Implies clear
[-3] cloudy
[-4] rainy
[-5] clear
|------

⑥ Now we use (split) but it lemmates both!!!

→ IT PUTS AS CONSEQUENT BOTH CLEAR & CLOUDY.

↳ LEMMA snowy => NOT clear

① (Flatten)
② (lemma cond_ax2)
③ (split)
④ (postpone)
⑤ (lemma cond_ax3)
⑥ (Flatten)
⑦ (split)

↳ LEMMA clear => Favorable

① (Flatten)
{-1} clear
|------
{1} Favorable
② (expand Favorable)
[-1] clear
|------
{1} NOT rainy & NOT snowy
③ (split)
④ axiom_1 & axiom_3 for both brenches
(grind) cannot import axioms!!!

RULE:
- Flatten: when and (if +) is in antecedent sentence
- split: when and (if +) is in consequence sentence
  ↳ other case we use inverse!
    ↳ OR + =>

ATTACKS TO CO-SIM:
→ Formal methods provides results with general validity. → diff. experiments at once + prevent users to make mistakes. → but we have some drawbacks anyway, such as proving formulae
→ We can merge co-sim & formal methods to remove drawbacks (effort by cosim & validity by FM).
→ maximum_step_attack: if attack robot it will never be able to move.

GENERAL ARGUMENTS:

INFORMATION REDUNDANCY

SELF-CHECKING CIRCUITRY: automatically detect a fault during the normal course of operations

→ IDEA: FAULT FREE + INPUT = CODEWORD
FAULT + INPUT = CODEWORD OR NON CODEWORD

↳ def: self-testing circuit: if, for every fault from the set, the circuit produces a non code output for at least 1 code input;

def: fault-secure circuit: if, for every fault from the set, the circuit never produces an incorrect code output for a code input;

↳ TOTALLY SELF-CHECKING IF BOTH VERIFIED. ⇝ COMPARATORE TSC.

TIME REDUNDANCY: reduce hw at expense of additional time

↳ repetition of computations → GENERATE DATA BY PERFORMING 2ND COMPUTATION

↳ use a minimum of extra hw to also detect permanent faults;

PETRI NETS: Places + Transitions + Arcs ⇝ places contain tokens;
When they have no transitions left they go in deadlock
The execution is non deterministic, so we may have some conflicts

- Timed transitions: activity that needs some time to be executed, we assign a delay to each transition, these activate after a timer elapses; delay d
  $(\tau_{k1}, t_{k1}), \dots, (\tau_{kn}, t_{kn})$ = (time, transition)

STOCHASTIC PETRI NETS: Delay d of a timed transition is a random var
ATT! If d is a negative exp. distr. the reachability graph is a markov chain

SAN (STOCHASTIC ACTIVITY NETWORK): Places + Input gates + Output gates + Activities;
→ DESCRIBES CONDITIONS FOR ENABLING

- transitions fired based on prob.;
- Activity can be timed or istantaneous; → IF WE CONNECT 2 PLACES WITHOUT A GATE THROUGH AN ACTIVITY.

FORMAL METHODS: mathematically-based techniques that can be used in the design & development of computer based systems ⇝ consists of 1) a language; 2) a set of tools; 3) a methodology for its application;
We can use them for the analysis of: ① programs; ② systems;

① A program P is a sequence of instructions <C>. The state of the program can be <C, m> command + initial memory + <m> single memory in the final state
ATT! Since the program is deterministic we have only 1 final state

↳ CONTROL FLOW GRAPH: directed graph (V, E) where V is a set of nodes and E is a set of edges connecting nodes. Nodes = instructions & E = contain edge between 2 addresses

If we use a low level language like bytecode we need a new operand: Stack = λ!

② PROGRAM GRAPHS: a control flow graph with edges labelled by actions. The initial & final node are identified as special. ~> Used to compute some actions based on input. Semantics consists in a memory m & a semantic function specifying the meaning of the actions ~> give a initial memory & a final memory.

A configuration is a pair (q, m), sem. m assigned to q.

Deterministic system = for the same input I always have the same output.

Evolving system = program does not stop in a stuck configuration.

Program Verification: verify correctness of a program attaching to each node a predicate & this should be satisfied whenever the node is reached during execution. (cannot be fully automated).

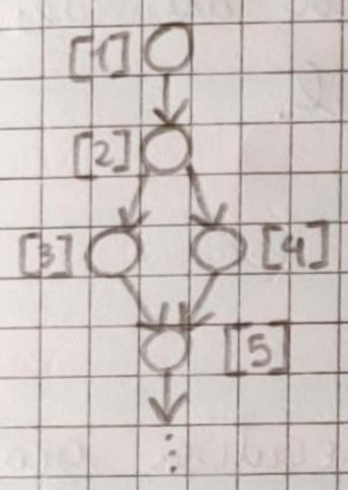
Program Analysis: fully automated approach to prove properties, it is a static technique since we do not execute the program with real values. ~> RULE: we use abstractions of the memories rather than the real memories.

ATT! The analysis of the program is an assignment A of abstract memories with each node in the program graph: A -> PowerSet(Mem)

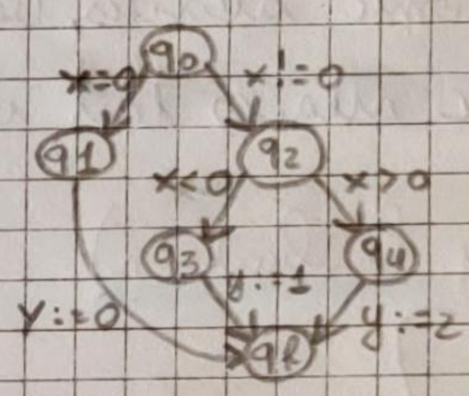
~> DIFF. CONTROL FLOW GRAPHS:

①

1: y := 5
2: if y > 1
then
3: x := 1
else
4: x := 0
5: z := x
6: ...



②



SECURITY POLICY: • a set of security-motivated constraints that must be satisfied by an organization or a computer system.

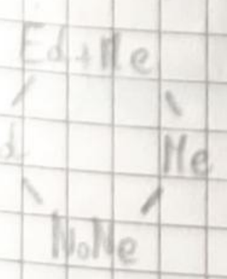
↳ Multilevel security policy: a sec. policy that allocates the classification of data & users based on a set of hierarchical security levels.

def: $\mathcal{L} = (S, \sqsubseteq)$ = (set of sec. levels, ordering relation) = is a lattice of sec. level

↑ sec policy

example:

$S = \{N, Ed, Me, Ed+Me\}$, $\sqsubseteq$ defined as: $N \sqsubseteq Ed$, $N \sqsubseteq Me$, $Ed \sqsubseteq Ed+Me$, $Me \sqsubseteq Ed+Me$, so the least upper bound ($\sqcup$): $Ed+Me = Ed \sqcup Me$



Secure information flow property: low output do not reveal inf. on the high lvl input & viceversa.

Non-interference property: the sec. domain private is non-interfering with domain public if no input by private can influence subsequent outputs that can be seen by public.

ENCHANTED OPERATIONAL SEMANTICS:

I-SEC IS GUARANTEED BY

- $(S, \mathcal{E}) \rightsquigarrow S = (l, h)$ & $l \sqsubseteq h$;
- Program P is a triple $\langle C, H, L \rangle$ with $H \cup L = Var(C)$
- Transition system $C(P, M)$ = P program + M initial enchanted memory

I-Security: when program terminates the value of the low vars do not change.

Safe-Memory: if & only if each low var. of P holds a low value in M, after having assigned I values to them.

ImpI: guarantees that there are no possible leakages.

STRIDE: is a threat model ⇝ has 6 threat categories

↳ SPOOFING
TAMPERING
REPUDIATION
INF. DISCLOSURE
DENIAL OF SERVICE
ELEVATION OF PRIVILEDGE

ATTACK TREES: describe sets of events that can lead to the goal in a combinatorial way. The root of each tree is the goal of an attacker that can damage the system operation. Leaf nodes = atomic attacks, these are assigned attributes.
Minimal cut-set = set of atomic attacks that achieve a goal.

ADVISE: (Adversary View Security Evaluation)

↳ 3 parts: 1) A system;
2) An adversary view;
3) Security metrics;

} are build on Attack execution graph. (AEG)

An AEG contains:

A: attack step: [ATTACK]
R: access domains in the system: [PROCESS]
K: knowledge required: (K)
S: skills of adversary △SK
G: adversary attack goals (GOAL)

The attractiveness of the all available attack steps is evaluated from the view point of the adversary with the criteria:

- COST; - DETECTABILITY; - EXPECTED PAYOFF IN THE NEXT STATE;

SIF: (Secure information flow) ⇝ a program P has secure information flow if in each final state of A(P), each $x : \sigma$ holds a value $\tau \sqsubseteq \sigma$

↓

SIF ANALYSIS: we compute least upper bound between memories instead of using a memory associated with a node of the control flow graph. The memory assigned to a state is the memory with the highest security levels assigned to variables during the abstract execution of the program.

JAVA CARDS: each application is executed in isolation, but this is not enough because we do not control the propagation of the information through Packets.

AUTOSAR: we have trust lvl of a software component & security requirement of a communication link. It provides information of which functional entity reads from or writes onto ports of components.

SIFT: (Software - Implemented Fault Tolerance), it is a multiprocessor computer designed for advanced commercial aircraft applications.
HW: processors are physically isolated from each other to avoid fault propagation. We have main & I/O. Each processor and its associated memory form a processing module.
↳ we have fault detection & reconfiguration.
Reliability is achieved by having each iteration of a task independently executed by a number of modules. ⇝ SIFT executes a set of tasks, each of which consists of a sequence of iterations, the input data to each iteration of a task are the output data produced by the previous iteration of some collection of tasks. ⇝ Output is chosen by 2-out-of-3 vote.
Synchronization is done through a vote only at the beginning of each iteration.
We ensure:
1) damage isolation; 2) Fault isolation; 3) Fault masking;
Scheduling: periodic method ⇝ faster tasks, higher priority;
Processor synchronization: Consensus problem;
Software: Application + Executive;

ADVISE:

In the ATTACK STEP:
1) ATTACK COST: how much adversary pays to do it
2) ATTACK TIME: time needed for the attack, we have diff distributions.
3) PRECONDITIONS: when the adv. will perform the attack.
4) OUTCOMES: ⇝ outcome prob.; detection prob.
5) EFFECTS: what happens when the attack is finalized.
In ADVERSARY:
1) Planning horizon ⇝ min equal to n. of attack steps.
2) Weights for discount & Preference of Cost, Detection & Payoff.
3) Starting skills, knowledge & accesses;
4) Goals that he wants to achieve + payoff for reaching it.

Now we need to identify rewards.

↳ 3 rewards: for the 2 knowledges & 1 goal (example)

↳ Set time !!! ⇝ for all of them.

↳ for roterewards we just return the move() of itself !!!