# Using formalism to design secure systems

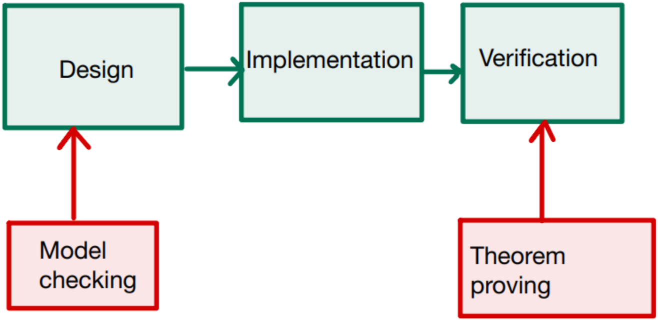
Stefan Nožinić (stefan@lugons.org)

July 6, 2023

# Agenda

- What are formal methods
- How to formalize systems
- Several examples from literature
- One real world example
- Conclusion

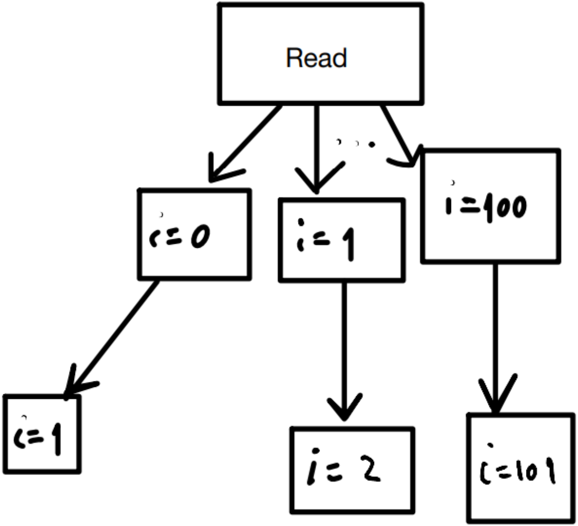
# Process


Design
Implementation
Verification
Model checking
Theorem proving

```
int main() {
    int i = readInteger();
    i++;
    return 0;
}
```

## How to model this simple program formally as state machine?

# State diagram



```
---------------- MODULE simple_increment_sm ----------------
EXTENDS TLC, Integers

VARIABLES i, pc

init ≜ i = 0 ∧ pc = "start"

next ≜ IF pc = "start" THEN
(i' ∈ 0 .. 1000) ∧ (pc' = "middle")
 ELSE IF pc = "middle" THEN
(i' = i + 1) ∧ (pc' = "done")
 ELSE FALSE
=============================================================
```

# Two-phase commit

```
—— MODULE twoPhase ——
EXTENDS TLC

CONSTANT RM
VARIABLE rmState
VARIABLES tmState, tmPrepared, msgs

Messages == [type : {"prepared"}, rm : RM] \cup
    [type : {"commit", "abort"}]



TPTypeOK ==
    /\ rmState \in [RM -> {"working", "prepared",
        "committed", "aborted"}]
    /\ tmState \in {"init", "done"}
    /\ tmPrepared \subseteq RM
    /\ msgs \subseteq Messages
```

```
TPInit ==
    /\ rmState = [r \in RM |-> "working"]
    /\ tmState = "init"
    /\ tmPrepared = {}
    /\ msgs = {}
```

```
TPNext ==
    \/ TMAbort
    \/ TMCommit
    \/ (\E r \in RM : RMPrepare(r))
    \/ (\E r \in RM : RMChoosetoAbort(r))
    \/ (\E r \in RM : TPRecvPrepared(r))
    \/ (\E r \in RM : RMChooseToCommit(r))
    \/ \E r \in RM : RMAbort(r)

====
```

```
TPRecvPrepared(r) ==
    /\ tmState = "init"
    /\ [type |-> "prepared", rm |-> r] \in msgs
    /\ tmPrepared' = tmPrepared \cup {r}
    /\ UNCHANGED << tmState, msgs, rmState >>
```

```
TMCommit ==
    /\ tmState = "init"
    /\ \A r \in RM : r \in tmPrepared
    /\ msgs' = msgs \cup {[type |-> "commit"]}
    /\ tmState' = "done"
    /\ UNCHANGED <<rmState, tmPrepared >>

TMAbort ==
    /\ tmState = "init"
    /\ tmState' = "done"
    /\ \E r \in RM : rmState[r] = "aborted"
    /\ msgs' = msgs \cup {[type |-> "abort"]}
    /\ tmPrepared' = {}
    /\ UNCHANGED  <<rmState>>
```

```
RMPrepare(r) ==
    /\ rmState[r] = "working"
    /\ msgs' = msgs \cup
         {[rm |-> r, type |-> "prepared"]}
    /\ rmState' = [rmState
         EXCEPT ![r] = "prepared"]
    /\ UNCHANGED <<tmPrepared, tmState>>

RMAbort(r) ==
    /\ rmState[r] = "working"
    /\ rmState' = [rmState
         EXCEPT ![r] = "aborted"]
    /\ UNCHANGED
         <<tmPrepared, tmState, msgs>>
```

```
RMChoosetoAbort(r) ==
    /\ rmState[r] = "working"
    /\ [type |-> "abort"] \in msgs
    /\ rmState' = [rmState
        EXCEPT ![r] = "aborted"]
    /\ UNCHANGED
        <<tmState, tmPrepared, msgs>>
```

```
RMChooseToCommit(r) ==
    /\ rmState[r] = "working"
    /\ [type |-> "commit"] \in msgs
    /\ rmState' = [rmState
         EXCEPT ![r] = "committed"]
    /\ UNCHANGED
         <<tmState, tmPrepared, msgs>>
```

====

# PlusCal

- A little more progreammer-friendly
- We specify processes and TLC will check all behaviours

# Simple clock

- We start from anywhere between 1 and 12 (including 1 and 12)
- in every iteration, we increase by one
- when we reach 12, we reset back to 1
- We also state that "x will be eventually one"

# Real world example - health monitor

- We have several nodes (lets say nodes are 1, 2 and 3)
- Every node can reboot and recover later on
- Every node has one instance of service called "replicator"
- When node is down, its replicator instance gets transferred to another node which is up
- When we detect that replicato instance is stuck, we kill it and restart it
- We state that eventually if replicator is stuck, this will lead to either it being killed or recovered by itself

```
▼ 14: Orchestrator in heartbeat >>
  ▶ alive (1)          {3}
    killed (0)         {}
  ▶ pc (4) M           (0 :> "RebootNode" @@ 1 :> "NodeDown" @@ 2 :> "NodeDown" ...
  ▶ replOwner (3)      <<2, 3, 3>>
  ▶ replStuck (3)      <<TRUE, TRUE, FALSE>>
▼ 15: RestartReplicator in heartbeat >>
  ▶ alive (1)          {3}
    killed (0)         {}
  ▶ pc (4) M           (0 :> "RebootNode" @@ 1 :> "NodeDown" @@ 2 :> "NodeDown" ...
  ▶ replOwner (3)      <<2, 3, 3>>
  ▶ replStuck (3)      <<TRUE, TRUE, FALSE>>
▼ 16: RebootNode in heartbeat >>
  ▶ alive (1)          {3}
    killed (0)         {}
  ▶ pc (4) M           (0 :> "Orchestrator" @@ 1 :> "NodeDown" @@ 2 :> "NodeDown" ...
  ▶ replOwner (3)      <<2, 3, 3>>
  ▶ replStuck (3)      <<TRUE, TRUE, FALSE>>
▼ 17: Orchestrator in heartbeat >>
  ▶ alive (1)          {3}
    killed (0)         {}
  ▶ pc (4) M           (0 :> "RebootNode" @@ 1 :> "NodeDown" @@ 2 :> "NodeDown" ...
  ▶ replOwner (3)      <<2, 3, 3>>
  ▶ replStuck (3)      <<TRUE, TRUE, FALSE>>
▼ 18: P in heartbeat >>
  ▶ alive (1)          {3}
    killed (0)         {}
  ▶ pc (4) M           (0 :> "RebootNode" @@ 1 :> "NodeDown" @@ 2 :> "NodeDown" ...
  ▶ replOwner (3)      <<2, 3, 3>>
  ▶ replStuck (3)      <<TRUE, TRUE, FALSE>>
▶ 15: Back to state >>
```

## Status

Check again Full output

Checking heartbeat.tla / heartbeat.cfg

Success : Fingerprint collision probability: 4.4E-11

Start: 13:23:48 (Jul 4), end: 13:23:55 (Jul 4)

## States

| Time | Diameter | Found | Distinct | Queue |
|---|---|---|---|---|
| 00:00:00 | 0 | 1 | 1 | 1 |
| 00:00:03 | 13 | 36 691 | 9 543 | 2 062 |
| 00:00:05 | 21 | 69 801 | 14 637 | 0 |
| 00:00:06 | 21 | 69 801 | 14 637 | 0 |

## Coverage

| Module | Action | Total | Distinct |
|---|---|---|---|
| heartbeat | Init | 1 | 1 |
| heartbeat | P | 11 895 | 5 256 |
| heartbeat | CheckIfStuck | 11 004 | 4 365 |
| heartbeat | RestartReplicator | 15 465 | 570 |
| heartbeat | NodeDown | 0 | 0 |
| heartbeat | Orchestrator | 9 894 | 2 964 |
| heartbeat | RebootNode | 7 245 | 208 |
| heartbeat | MakeReplicatorStuck | 14 637 | 1 273 |
| heartbeat | Terminating | 0 | 0 |

# Conclusion

- Formal specification can help us reason about systems and communicate better in teams
- There are tools to help us formally specify systems and to check its validity
- More granular we go, more validation we get

# Gossip session