PSYNC: A partially synchronous language for fault-tolerant distributed algorithms

Cezara Drăgoi, Thomas A. Henzinger, Damien Zufferey

(POPL ’16)

Overview

PSYNC

- is a domain-specific language (DSL)
- for programming fault-tolerant distributed algorithms
- in a high-level, round-based model with lockstep semantics

- which compiles into ***equivalent***, efficient asynchronous programs
- and is suitable for semi-automated verification

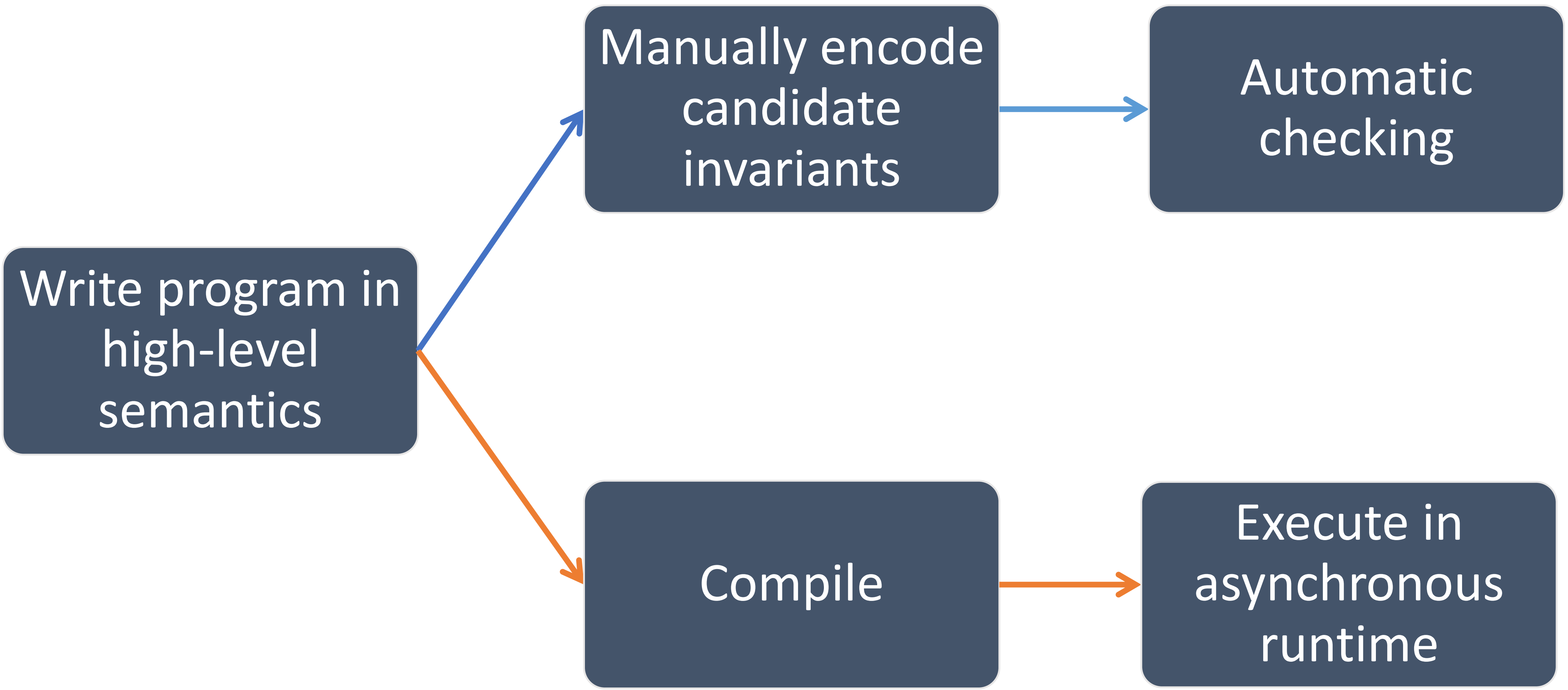
PSYNC is **descriptive** enough
to specify lots of dist. algos

PSYNC is **sufficiently restricted** so programs
can be semi-automatically proven

Unify **modelling**, **programming** and **verification** of fault-tolerant distributed algorithms

PSYNC can be **compiled**
into asynchronous programs

Workflow


Manually encode candidate invariants
Automatic checking
Write program in high-level semantics
Compile
Execute in asynchronous runtime

Why does it matter?

- Fault-tolerant distributed systems are complex:
  - packets get re-ordered, duplicated or dropped
  - machines fail & potentially restart
  - concurrency is hard
- For critical applications, we want assured *correctness*
  - we need **formal verification**
  - but want development and verification to be easy
- PSYNC provides a solution

Internals

How does it work?

- **Communication-closed rounds** with **lockstep semantics**:
  - a PSYNC program is defined as a sequence of rounds
  - all processes execute the same round
  - messages sent within a round are either delivered in that round or dropped forever

- Each round consists of two operations, executed in this order:
  1. Send – send messages
  2. Update – update local state based on messages received in this round

PSYNC is based on the Heard-Of ($\mathcal{HO}$) model

- a distributed system is a set of processes & an **adversarial environment**
- every round, the environment decides which messages processes receive

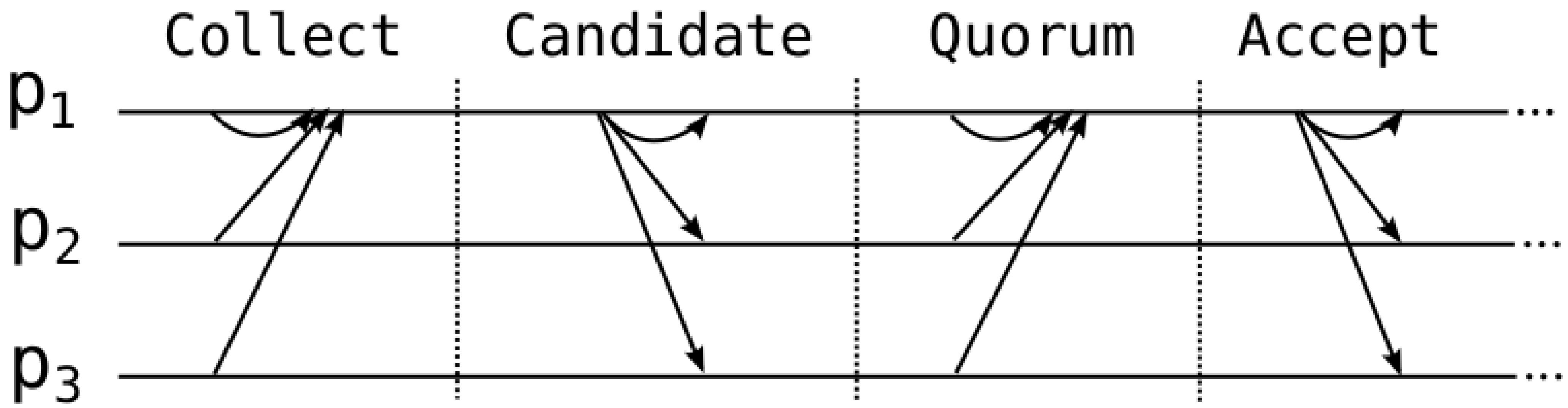
- each process p has a *Heard-Of set*, $\mathcal{HO}(p)$ = *the set of processes p hears from*
- in a given round, process p receives a message from process q if q sends a message to p and $q \in \mathcal{HO}(p)$

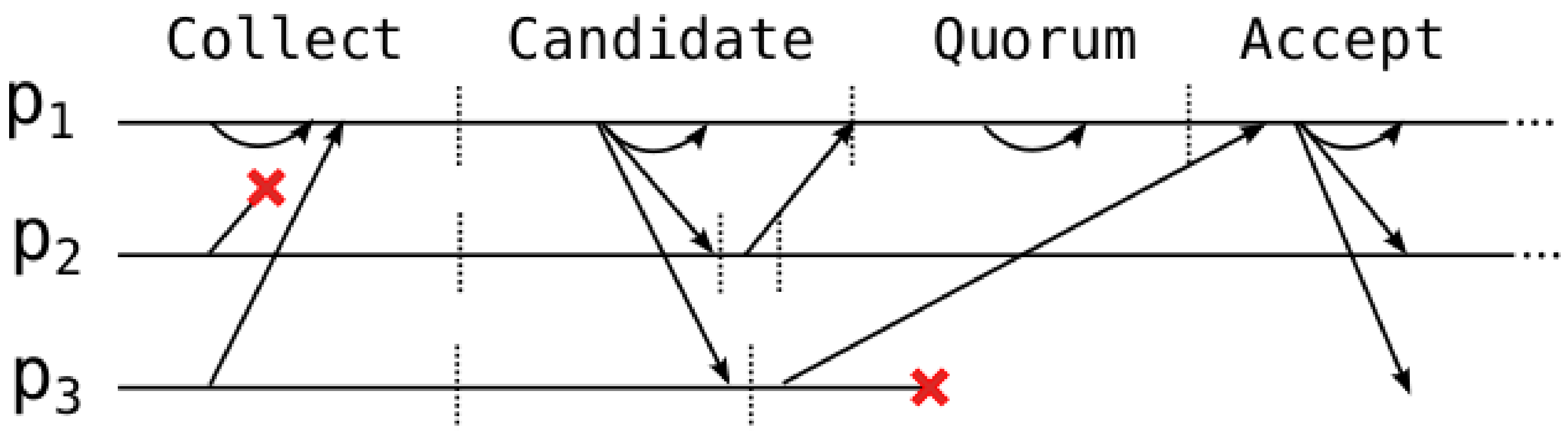
- $\mathcal{HO}$ uniformly models *asynchronous* behaviour and *faults* while providing the illusion of a lockstep semantics

- A dropped message sent from q to p can be modelled as:
  - $q \notin HO(p)$

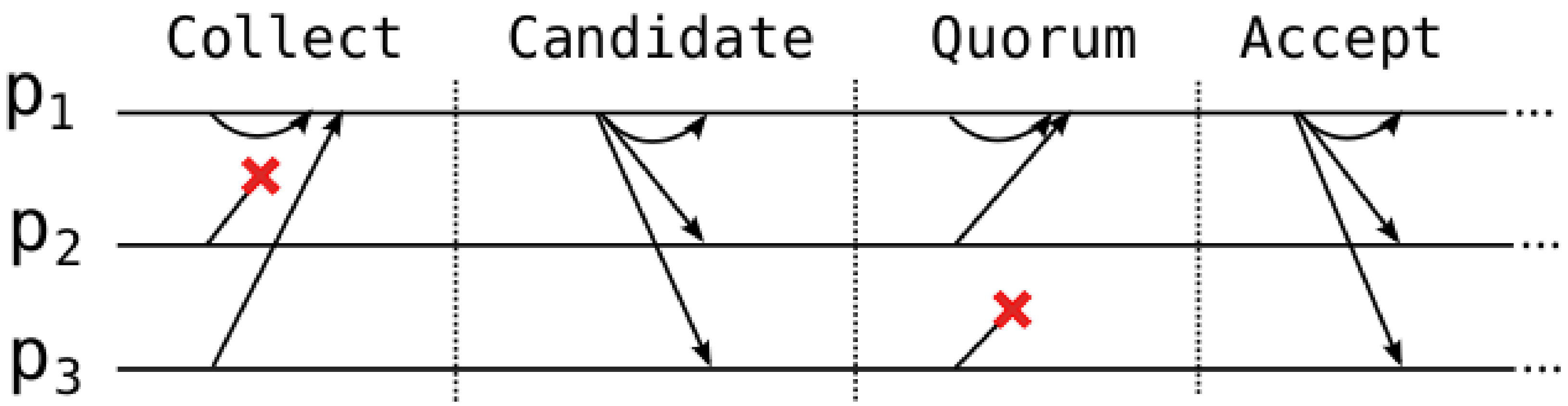
- A crashed process q can be modelled as:
  - $\forall p,\ q \notin HO(p)$

- Generally, assumptions regarding the $\mathcal{HO}$ sets can be given as linear temporal logic (LTL) formulas


Collect
Candidate
Quorum
Accept
p_1
p_2
p_3



(a) An asynchronous, faulty execution of the *LastVoting*



(b) Corresponding indistinguishable lockstep execution

LastVoting: Paxos adapted to $\mathcal{HO}$ model

```
interface
    init(v: Int); out(v: Int)

variable
    x: Int; ts: Int; vote: Int
    ready: Boolean; commit: Boolean
    decided: Boolean; decision: Int

//auxiliary function: rotating coordinator
def coord(phi: Int): ProcessID =
    new ProcessID((phi/phase.length) % n)

//initialization
def init(v: Int) =
    x := v
    ts := -1
    ready := false
    commit := false
    decided := false
```

```
val phase = Array[Round]( //the rounds
  Round /* Collect */ {
    def send(): Map[ProcessID, (Int,Int)] =
        return MapOf(coord(r) → (x, ts))
    def update(mbox: Map[ProcessID, (Int,Int)]) =
        if (id = coord(r) ∧ mbox.size > n/2)
            vote := mbox.valWithMaxTS
            commit := true },
  Round /* Candidate */ {
    def send(): Map[ProcessID, Int] =
        if (id = coord(r) ∧ commit) return
          broadcast(vote)
        else return ∅
    def update(mbox: Map[ProcessID, Int]) =
        if (mbox contains coord(r))
            x := mbox(coord(r))
            ts := r/4 },
```

```
  Round /* Quorum */ {
    def send(): Map[ProcessID, Int] =
        if ( ts = r/4 ) return MapOf(coord(r) → x)
        else return ∅
    def update(mbox: Map[ProcessID, Int]) =
        if (id = coord(r) ∧ mbox.size > n/2)
            ready := true },
  Round /* Accept */ {
    def send(): Map[ProcessID, Int] =
        if (id = coord(r) ∧ ready) return broadcast(vote)
        else return ∅
    def update(mbox: Map[ProcessID, Int]) =
        if (mbox contains coord(r) ∧ ¬decided)
            decision := mbox(coord(r))
            out(decision)
            decided := true
        ready := false
        commit := false })
```

Properties for *LastVoting*

- Safety (agreement)

$$\Box(\forall p, p'.\, p.decided \land p'.decided \implies p.decision = p'.decision)$$

with the (simplified) invariant:

$$\begin{array}{lll} & \forall p. & p.\mathtt{decided} = false \\ \lor & \exists v, t, A. & A = \{p \mid p.\mathtt{ts} \geq t\} \land |A| > n/2 \\ & & \land \forall p.\, p \in A \Rightarrow p.\mathtt{x} = v. \end{array}$$

- Liveness

$$\Diamond(\forall p.\, p.decided)$$

```
    val noDecision: Formula = P.forall( i => !i.decided && !i.ready)

    val majority: Formula =
      V.exists( v => V.exists( t => {
          val A = P.filter( i => i.ts >= t )
          A.size > n/2 &&
          r > 0 &&
          t <= r/4 &&
          P.forall( i => (A.contains(i) ==> (i.x == v) ) &&
                         (i.decided ==> (i.decision == v) ) &&
                         (i.commit ==> (i.vote == v) ) &&
                         (i.ready ==> (i.vote == v) ) &&
                         ((i.ts == r/4) ==> coord.commit ))
      }) )

    val keepInit: Formula = P.forall( i => P.exists( j1 => i.x == init(j1.x) ))

    val safetyInv = And(keepInit, Or(noDecision, majority))
```

Semantics, runtime & verification

Init

$$\frac{\forall p \in P.\ * \xrightarrow{\texttt{init}(v_p)} s(p)}{* \xrightarrow{\emptyset, \{\texttt{init}_p(v_p) \mid p \in P\}} \langle Snd, s, 0, \emptyset, \mathsf{HO} \rangle}$$

Send

$$\frac{\begin{array}{c}\forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r]\texttt{.send}(m_p)} s(p) \\ msg = \{(p, t, q) \mid p \in P \wedge (t, q) \in m_p\}\end{array}}{\langle Snd, s, r, \emptyset, \mathsf{HO} \rangle \xrightarrow{\{\texttt{send}_p(m_p) \mid p \in P\}, \emptyset} \langle Updt, s, r, msg, \mathsf{HO}' \rangle}$$

Update

$$\frac{\begin{array}{c}\forall p \in P.\ mbox_p = \{(q, t) \mid (q, t, p) \in msg \wedge q \in \mathsf{HO}(p)\} \\ \forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r]\texttt{.update}(mbox_p), o_p} s'(p) \\ r' = r + 1 \qquad O = \{o_p \mid p \in p\}\end{array}}{\langle Updt, s, r, msg, \mathsf{HO} \rangle \xrightarrow{\{\texttt{update}_p(mbox_p) \mid p \in p\}, O} \langle Snd, s', r', \emptyset, \mathsf{HO} \rangle}$$

Runtime

- The lockstep semantics (LS) can be translated into asynchronous semantics (AS)

- Rounds are implemented using a **timeout**
  - *Paper describes how an appropriate timeout should be determined*

- If the specification is closed under indistinguishability, properties that hold in LS also hold in AS
  - *Consensus, k-set agreement and lattice agreement are closed under indist.*

Verification

- Can verify both *safety* and *liveness*
- Semi-automated, i.e. programmer manually writes invariants that are auto-checked by an SMT solver (**deductive verification**)
- Could potentially support **model-checking**, as lockstep semantics is sufficiently restricted

Conclusion

Take-aways

- *PSYNC strikes a balance between high-level constructs, performance, and automated verification*
- *The $\mathcal{HO}$ model along with communication-closed rounds and lockstep execution, greatly simplifies implementing & verifying dist. algos*
- *For an important class of specifications including consensus, if a PSYNC program satisfies the specification, then its runtime system satisfies it as well*