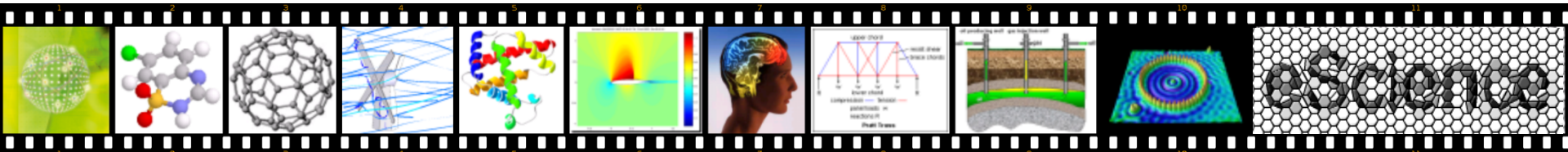
Verification of a Dynamic Channel Model using the SPIN Model Checker

Rune M. Friborg and Brian Vinter
eScience center, University of Copenhagen

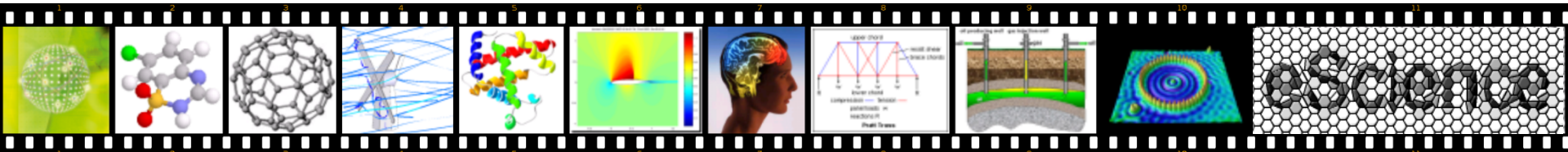
Introduction to PyCSP

- 2007 - PyCSP is presented. The synchronization model for channel communications is based on JCSP.
- 2009 - A PyCSP with a new synchronization model is presented. It is using the two-phase locking protocol to allow any2any channels supporting both input and output guards.



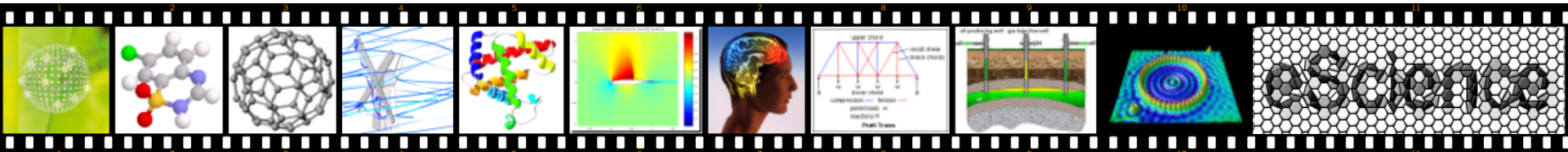
Python for eScience applications

- Python is widely accepted and used in the scientific community.
- Scientists have access to parallel hardware, but need tools to make their prototypes use this hardware
- Parallel hardware comes in many forms
  - Multi-core
  - Cluster
  - Grid
  - Net (XML) services



Communicating Sequential Processes

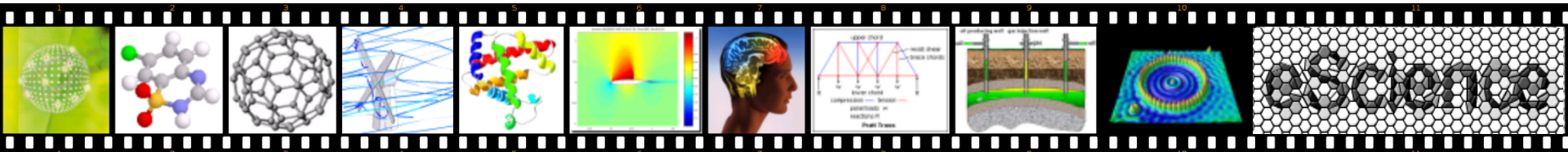
- Synchronized constructs for running a set of processes
  - In parallel
  - In sequence
- Synchronized communication through message passing
  - One-way channels
- Complete Process Isolation
  - No shared data-structures
  - No side-effects from processes
  - Compositional structure
  - Reuse of processes



Current PyCSP features

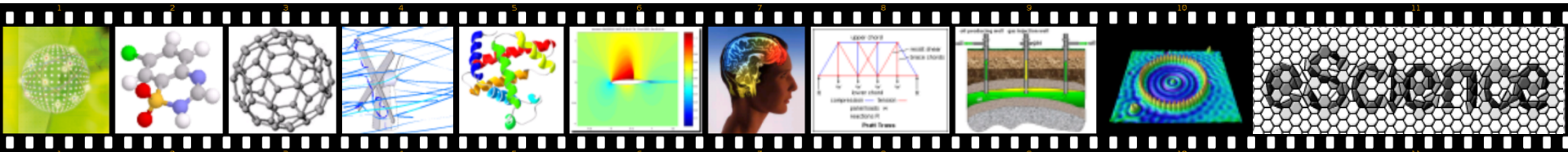
- Channel() is an any2any channel
- Alt performs the read or write operation when a guard has been selected
- Timeout and skip guards
- Controlled termination through poisoning and retiring
- Input and output guards can be combined in an alt operation.

No support for distributed communication?



Run anywhere

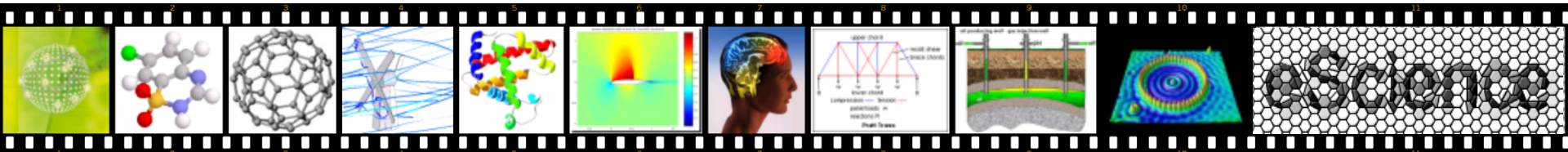
```
import pycsp
@pycsp.process
def worker(cin, cout):
        while(True):
                data = cin()
                partial_result = compute(data)
                cout(partial_result)

workChan, resultChan = Channel("work"), Channel("result")
pycsp.Parallel(
    master(data, workChan.writer(), resultChan.reader()),
    WORKERS * worker(workChan.reader(), resultChan.writer()
)
```



Control vs. flexibility

- More control – less flexibility
  - @grid_process(vgrid=DIKU ..)
  - @ssh_process(host=ip..)
  - @mpi_process(mpirun .. , count)
  - @process
  - @coroutine
- Less control – more flexibility
  - @process
  - Parallel(processes..., hint=local | striped | blocked | auto)
  - Spawn(processes..., hint=local | striped | blocked | auto)

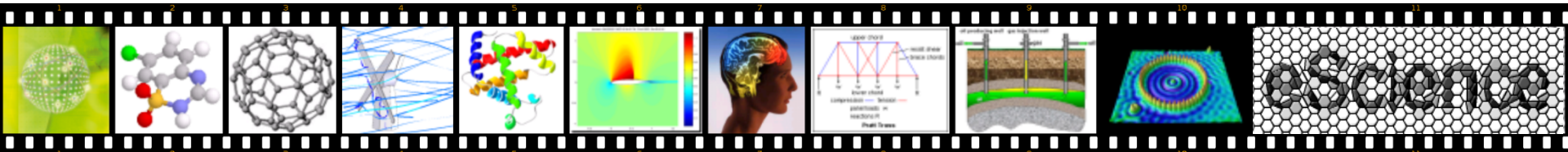
- both approaches need one channel type.



The Dynamic Channel

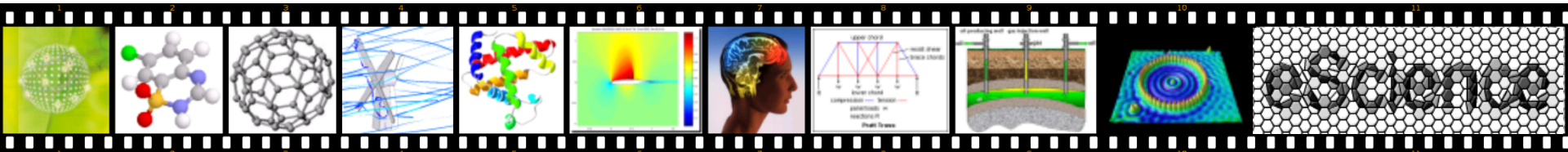
- A hybrid synchronisation model
- Combine channel synchronisation mechanisms with different specialities for better performance
  - Co-routine – co-routine communication
  - thread – thread communication
  - single node – single node communication
  - any node – any node communication
- Change synchronisation mechanism on-the-fly

- Finding a common approach to synchronisation



Classic approach : Two-phase commit protocol (barrier)

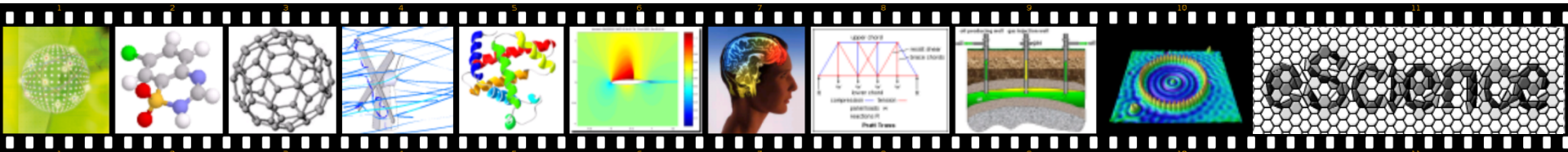
- Acquire lock for resource C (enter barrier)
- update C
- Release lock for resource C (leave barrier)



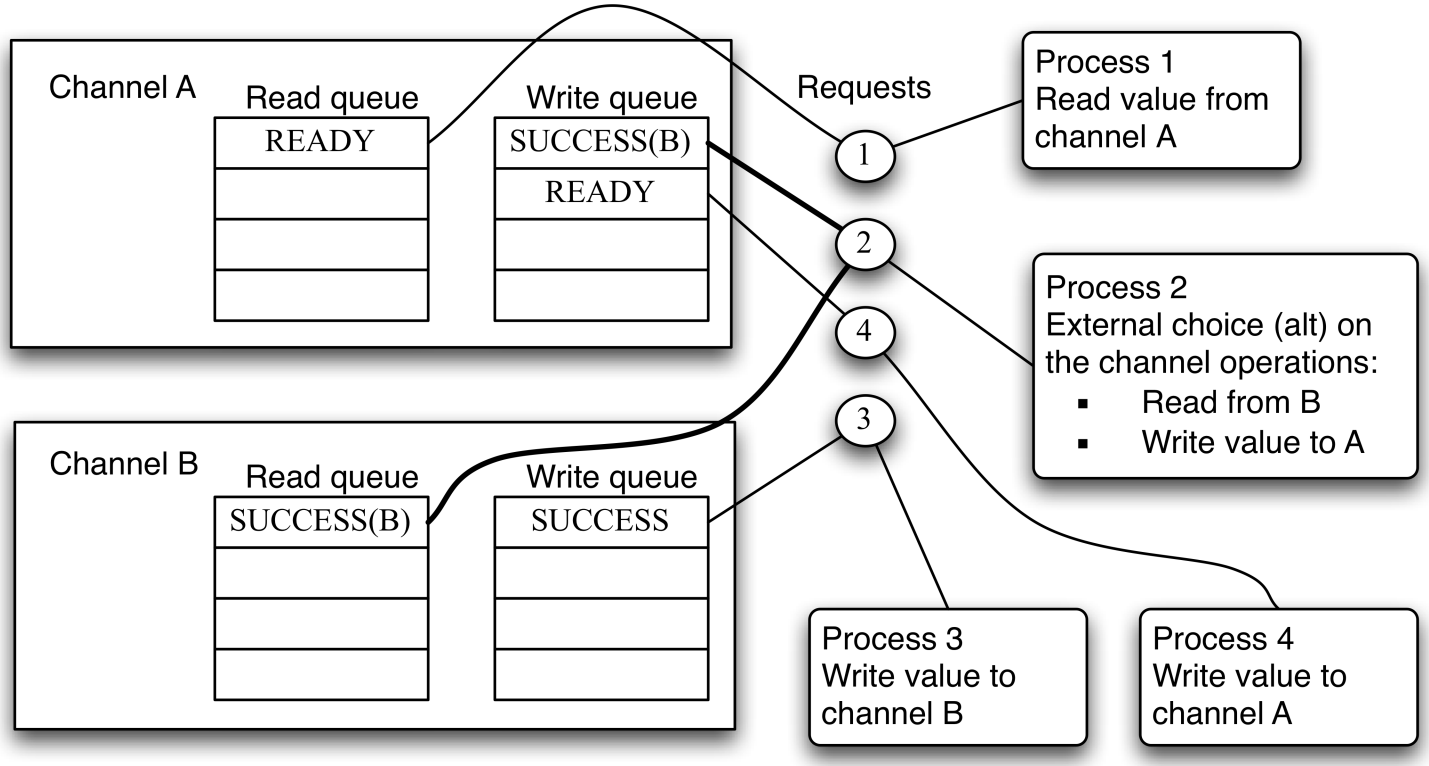
PyCSP approach : Two-phase locking protocol

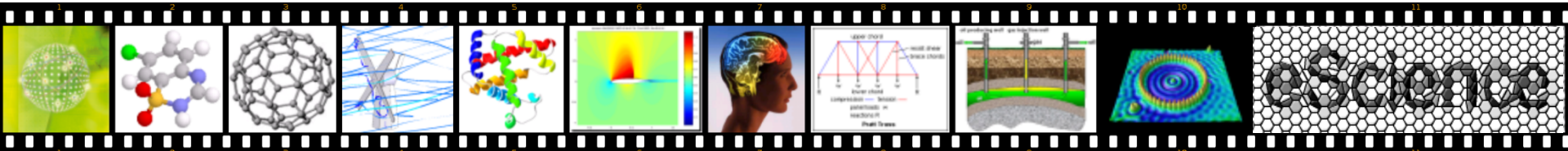
- Provide each process with a lock
- Acquire all process locks depending on resource C
- Update C
- Release all process locks depending on resource C

- Deadlocks are avoided using Roscoe's deadlock rule 7:
  - Locks are acquired in a global order indexed by the set (node id, memory address).



Two-phase locking protocol in use

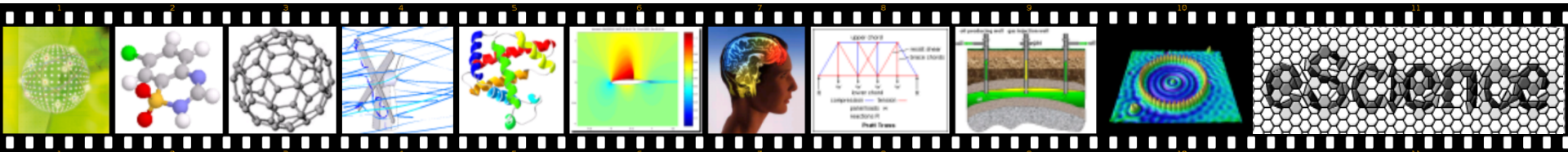




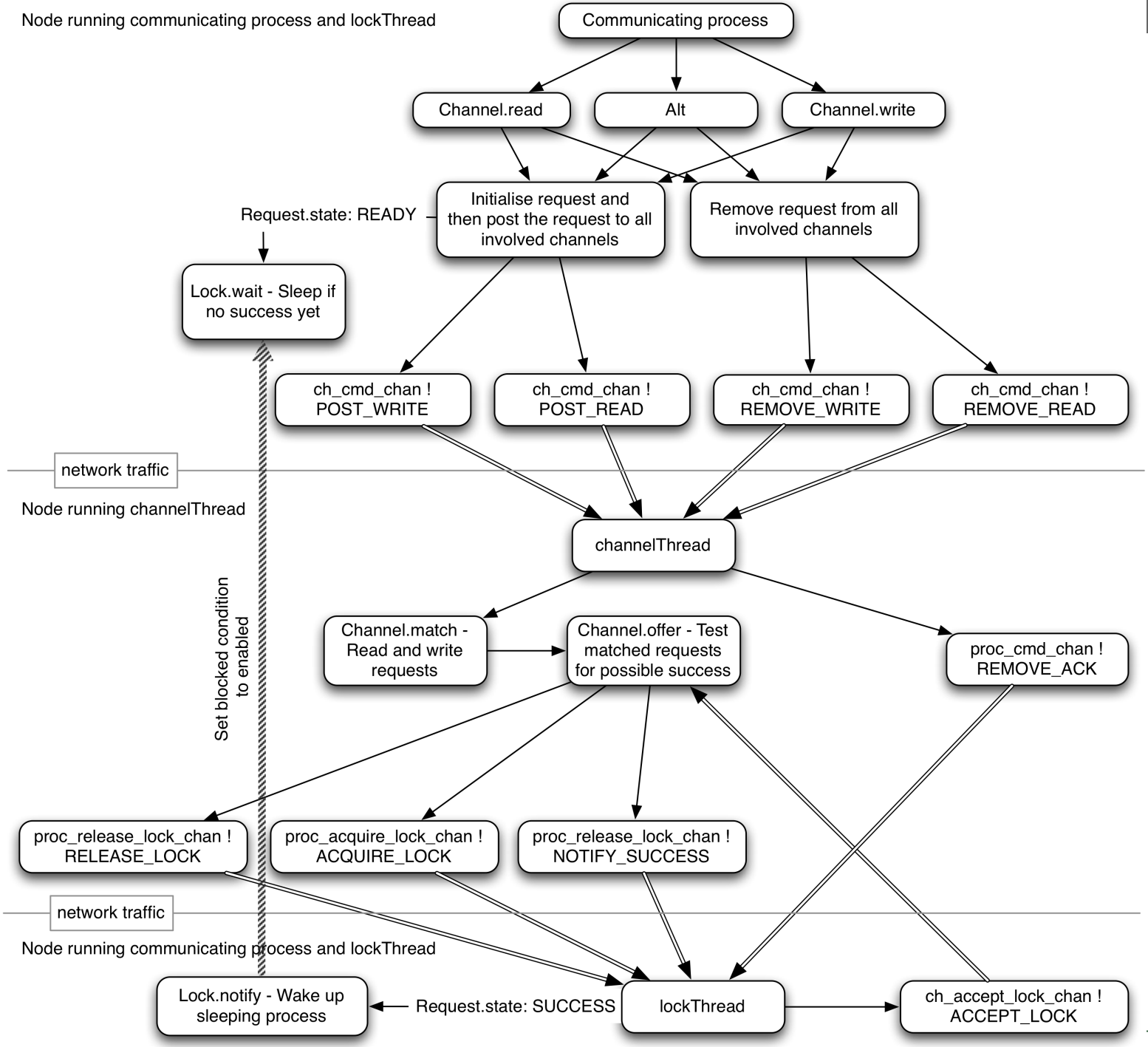
Promela and the SPIN Model Checker

- Promela is a process modeling language
- SPIN performs an exhaustive state space exploration of models written in Promela
- SPIN has been used to check for the presence of deadlocks, livelocks, starvation, race conditions and correct channel communication behaviour

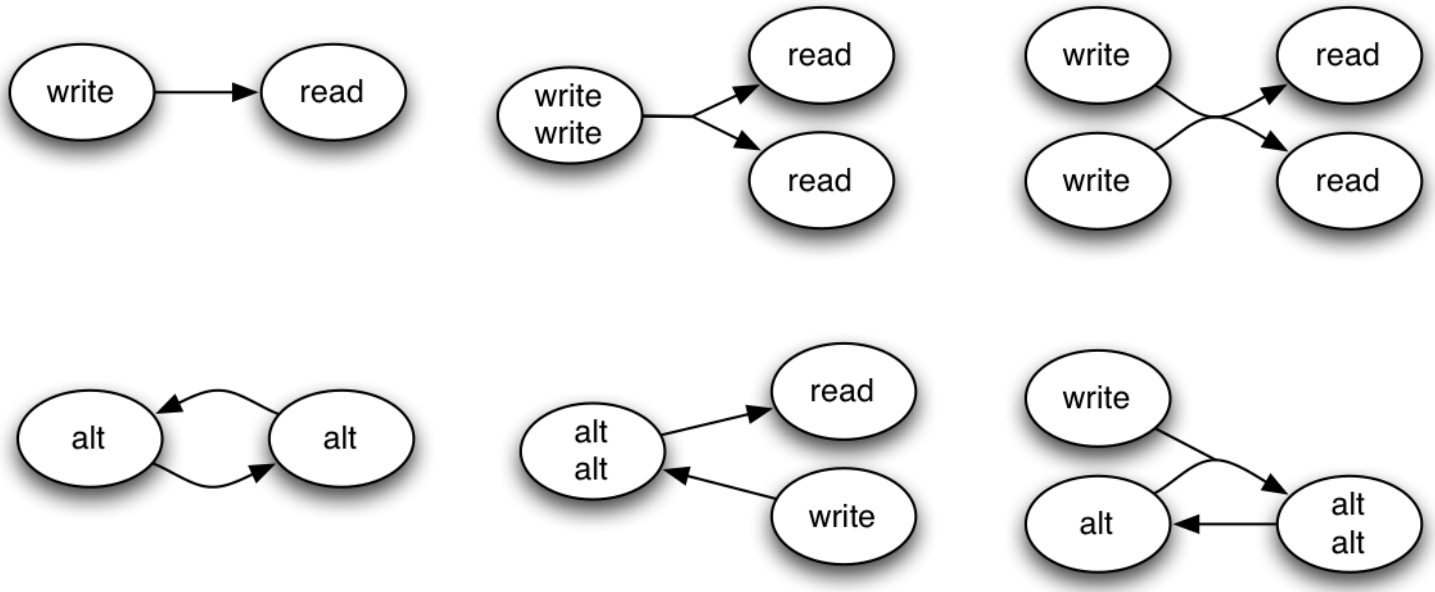
- We present three models
  - Local synchronisation model (current PyCSP implementation)
  - Distributed synchronisation model
  - Transition model



Distributed Model



Automatic Exhaustive Verification of the Distributed Model

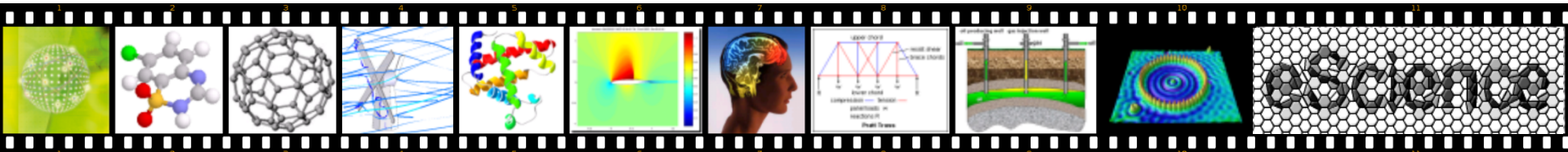


The SPIN model checker did not find any errors for the above process networks

The algorithm never backs down from a commit request

Some advantages of one channel type

- The programmer does not need to know anything about the location of the hardware any process might run on.
- All channel ends can be mobile.
- It is simple to combine various hardware in one single application.

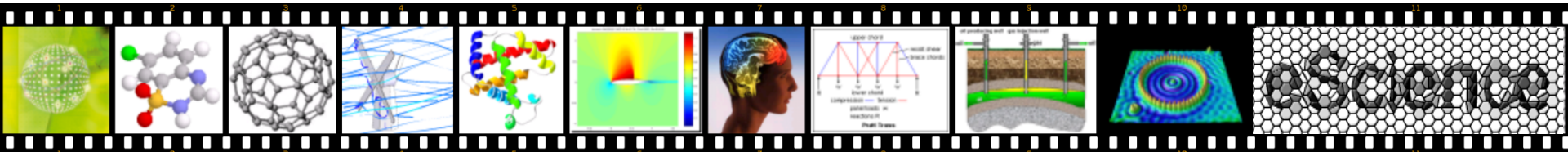


Transition Model

- The Transition Model enables switching between synchronisation mechanisms (levels) for channels

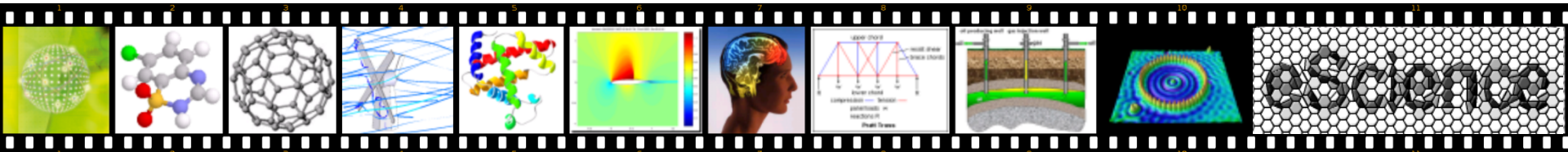
- The read, write and alt construct is split into three stages:
  - enter
  - wait
  - leave

- Enter - retrieves the channel level, stores it inside the channel request and posts the request to the channel(s)
- Wait – waits until notified. If the channel request has not yet committed, it retrieves the channel level and activates leave (old channel level) and enter (new channel level)
- Leave – Use the stored channel level inside the channel request to leave the channel(s)

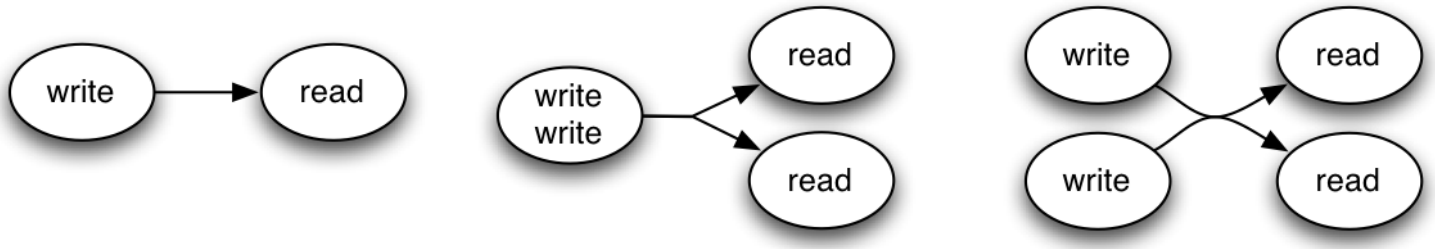


Transition Model – Activating a transition

- A transition is activated when there is a feature request, that are not supported at the current synchronisation level.
- Transition from level A to B
  - The channel level is updated to B
  - All posted channel requests on the channel is notified with a transition signal
  - done



Automatic Exhaustive Verification of the Transition Model



Alt was removed from the transition model, to reduce complexity

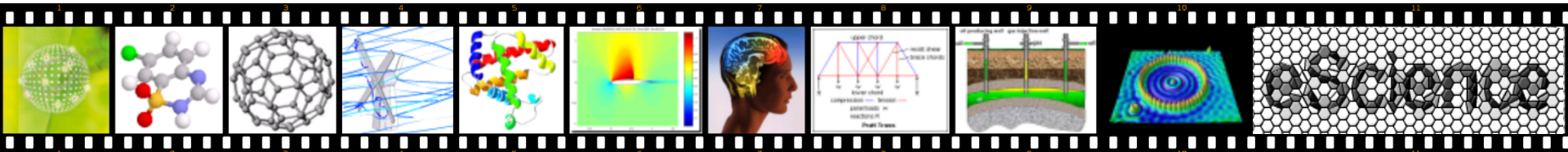
The SPIN model checker did not find any errors for the above process networks

The algorithm never backs down from a commit request

Conclusion

- The synchronization mechanism in the current PyCSP is model checked successfully
- We suggest that others may use these models as the basics for implementing a CSP library

The full model of the dynamic channel has **not** been verified, since the large state space may make it unsuited for exhaustive verification using a model checker



Future

The models will be the basis for a new PyCSP channel, that can start out as a simple pipe and evolve into a distributed channel spanning multiple nodes.

Questions?