CS 5/6110, Software Correctness Analysis, Spring 2023

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Outline

- Coherence
- Complexity of coherence-checking of a trace
  - It is NP-complete
    - Good to understand the “hardness” of a coherence assertion
      - Trace-verification per trace is hard to exhaust …
        - But even if we do, it can be complex
    - Real verification : see below
- Coherence protocol implementation
  - Implementations are horrendously complex (90 pages of TLA+ rules which expands to 3x that many pages of Murphi rules – at least)
    - 30 message types , handling orphaned messages , etc
  - ALL for speed
    - Else your phone will respond to your button-push of today tomorrow ☺
- ”meta-circular assume/guarantee” of parameterized cache coherence protocols that can be practiced within Murphi
  - General param verif problem is undecidable
    - Reduction from at TM that halts in k-steps when started on an empty tape to a ring of k automata
- How the CMP method looks like
- This brings us to the topic of writing parameterized protocols using Murphi’s scalar sets
- How to write one such model for our favorite locking protocol
  - And how verification scales with and without scalar sets
  - And how ROMP is able to make a big dent (ROMP is our random-walk Murphi)

What is memory coherence?

- A view of shared memory from the point of view of
  - One address
  - More than one thread
  - It is basically sequential consistency, but for a memory system with exactly one address
    - What is sequential consistency? [Lamport'78]
      - Very good descriptions here: https://en.wikipedia.org/wiki/Sequential_consistency
- Different from sequential consistency?
  - Yes!
    - An execution like this

An example demonstrating the diff starkly

P0	P1 (“P” = processes or shared-memory threads)
Initially a==b==0	
Store(a,1);	Store(b,2);
Load(b,0);//returns 0	Load(a,0);//returns 0

If you wear a pair of goggles that just sees black or just sees brown
There is an “SC explanation” -- try it
But if you see both colors together, then no SC explanation

Hence coherent (one address has always the latest value, since the real-time order of local operations is irrelevant at the global level)

Complexity of trace-verification for coherence

Suppose someone gives you a trace

Px:S(a,0); Py:L(a,1); Pz:S(a,33); Pz: (a,44); Px: S(a,3); ….

Where the P are processes (or threads) that share the SINGLE address a

And each Px: …. ; …. ; Px: … is listed in program order per Px
With the other Py , y ~= x listed according to some interleaving

Then what is the complexity of checking that such a trace and declaring whether it is coherent or not?

Complexity of trace-verification for coherence

NP-Complete!

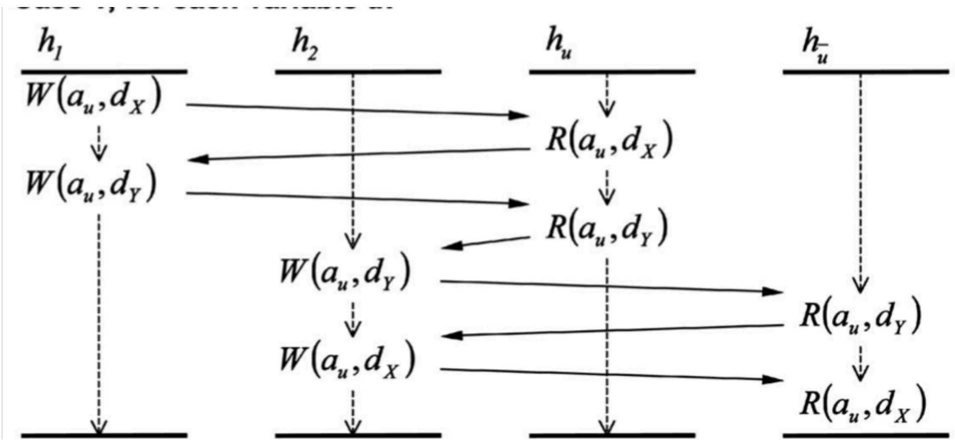
Where is a good (accessible) proof? – ANS : Cantin's paper

Definition 1: Verifying Memory Coherence.

INSTANCE: Data value set D, address a, and finite set H of process histories, each consisting of a finite sequence of read and write operations.

QUESTION: Is there a coherent schedule S for the operations of H with address a?

Where is a good (accessible) proof? – ANS : Cantin's paper



Given four processes (or hardware threads) and reads/writes
Per location ("a_u" in this case) explain read-value outcomes.
Here we explain as if this interleaving took place.
The inability find such an explanation means the system is incoherent
Cool piece of encoding by Cantin: Given a 3SAT formula, he generates a shared memory execution such that the formula is SAT iff the execution is coherent!

Cantin's reduction from SAT (general SAT which is = 3SAT wrt complexity)

Recall or know
That
2-sat
Is P-time
(= 2-coloring)

SAT:

$$U \equiv \{u_1, u_2, \ldots, u_m\}$$
$$C \equiv \{c_1, c_2, \ldots, c_n\}$$
$$Q \equiv c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

VMC:

$$H \equiv \{h_1, h_2, h_3, h_{u_1}, \ldots, h_{u_m}, h_{\bar{u}_1}, \ldots, h_{\bar{u}_m}\}$$
$$D \equiv \{d_{u_1}, \ldots, d_{u_m}, d_{\bar{u}_1}, \ldots, d_{\bar{u}_m}, d_{c_1}, \ldots, d_{c_n}, d_I, d_F\}$$

$D_l \equiv$ An ordered list of values d_c for each literal l, such that $l \in c$

h_1	h_2	h_{u_i}	$h_{\bar{u}_i}$	h_3
$W(d_{u_1})$	$W(d_{\bar{u}_1})$	$R(d_{u_i})$	$R(d_{\bar{u}_i})$	$R(d_{c_1})$
$W(d_{u_2})$	$W(d_{\bar{u}_2})$	$R(d_{\bar{u}_i})$	$R(d_{u_i})$	$R(d_{c_2})$
$\vdots$	$\vdots$	$W(D_{u_i}[1])$	$W(D_{\bar{u}_i}[1])$	$\vdots$
$W(d_{u_m})$	$W(d_{\bar{u}_m})$	$\vdots$	$\vdots$	$R(d_{c_n})$
		$W(D_{u_i}[\lvert D_{u_i}\rvert])$	$W(D_{\bar{u}_i}[\lvert D_{\bar{u}_i}\rvert])$	$W(d_{u_1})$
				$\vdots$
				$W(d_{u_m})$
				$W(d_{\bar{u}_1})$
				$\vdots$
				$W(d_{\bar{u}_m})$
				$W(d_F)$

Test literal

Satisfy clauses

Conjoin satisfied clauses

Cleanup

Truth Assignment

Clause Evaluation
(1 process per literal, $1 \le i \le m$)

Fig. 1. General SAT to VMC reduction.

Recap / Summary

- To define memory views as shared by multiple processors, we need to define a formal shared memory consistency model
- Coherence is one of the basic models
  - Each location has a latest data that every reader agrees on
  - Also known as “per-location sequential consistency”
- There are some interesting complexity results that help us understand coherence
  - Given a “finished execution trace”, checking coherence is NP-Complete
    - Very insightful proof by Jason Cantin et al
    - https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1435343

Implementing coherence

- Snoopy-bus protocols (still present at smaller scales)
- Directory-based protocols (more scalable)
- See https://www.morganclaypool.com/doi/pdf/10.2200/S00962ED2V01Y201910CAC049 for Thu's colloq speaker's book

Coherence Verification

- Given the complexity of coherence protocols, formal methods (model-checking mainly) is essential
- Let us take a look at an academic protocol called The German Protocol
- How does coherence verification scale?
  - Not well – today, large protocols take days to cover for one bit of data and 3 processors
  - Solutions
    - Derive cutoff bounds – Emerson and Kahlon
      - The bounds are large (7-8) and automatically computing bounds is not practical for large protocols
    - Do a parametric verification
      - Prove coherence for all "N" – N is the number of cores/threads
      - This involves modeling 2-3 nodes explicitly and involves a manual abstraction/refinement loop – called CEGAR
        - Counter-Example Guided Abstraction Refinement
      - Involves designer input of non-interference lemmas
    - Do formal synthesis
      - Prof. Vijay Nagarajan will be presenting this when he joins us!

Basics about Transition Systems

- Before we study the German protocol and how the parametric verification method I'm going to present works, let us discuss basic concepts about formal transition systems

Almost all specs for safety property checking look like this

- Based on Guarded Commands

  Rule1: g1 ==> a1

  Rule2: g2 ==> a2

  …

  RuleN: gN ==> aN

  Invariant P

- Supported by tools such as Murphi (Stanford, Dill's group)
- Presents the behavior declaratively
  - Good for specifying "message packet" driven behaviors
  - Sequentially dependent actions can be strung using guards
- "Rule Sets" can specify behaviors across axes of symmetry
  - Processors, memory locations, etc.
- Simple and Universally Understood Semantics

Let us understand how we may safely transform such rule-based specifications (this is what we have to do in the parametric verification approach to be presented). The method is called the CMP method named after its inventors at Intel.

The name of the game – "invariants" !!

- Much like loop invariants, except
  - These are invariants pertaining to formal state-transition systems
- We are really talking about inductive invariants
  - Invariants P such that
    - They are true in the initial state
    - When held at a state, every transition rule preserves it
- This is a hugely important topic (inductive invariant discovery)
  - Then you can do program-proofs by walking each path or transition rule!

Invariants, Inductive Invariants

- Consider the transition system
  - Init : X == 1 (X is an int var … i.e. +, 0, -)
  - Tr1 : true ➔ x += 2
  - Tr2 : true => x += 3
  - An invariant is x != 0
  - But is it inductive?
    - (X != 0) ==?==Tr1➔ (X!=0) ?
    - (X != 0) ==?==Tr2➔ (X!=0)?

Invariants, Inductive Invariants

- Consider the transition system
  - Init : X == 1 (X is an int var … i.e. +, 0, -)
  - Tr1 : true ➔ x += 2
  - Tr2 : true => x += 3
  - An invariant is x != 0
  - But is it inductive?
    - (X != 0) ==?==Tr1➔ (X!=0) ?
    - (X != 0) ==?==Tr2➔ (X!=0)?
  - No, for X == -2, it is not true!
    - So what is one inductive invariant for this transition system?

Invariants, Inductive Invariants

- Consider the transition system
  - Init : X == 1 (X is an int var … i.e. +, 0, -)
  - Tr1 : true ➔ x += 2
  - Tr2 : true => x += 3
  - An invariant is x != 0
  - But is it inductive?
    - (X != 0) ==?==Tr1➔ (X!=0) ?
    - (X != 0) ==?==Tr2➔ (X!=0)?
  - No, for X == -2, it is not true!
    - So what is one inductive invariant for this transition system?
      - How about X >= 0 ?
      - How about X >= -1 ?
      - How about X >= 1 ?
      - …
    - What is the strongest inductive invariant of the above three?
    - What is the strongest inductive invariant (period) ?
      - Answer : = reachable state space!

Tools to interactively discover invariants

- The I4 tool, and this hotos paper
  - https://web.eecs.umich.edu/~manosk/assets/papers/i4-hotos19.pdf
- https://web.eecs.umich.edu/~manosk/projects.html
- https://web.eecs.umich.edu/~manosk/assets/papers/i4-hotos19-slides.pdf

- Things like Paxos have been treated – which is nice
- https://researchr.org/publication/fmcad-2021

Let us understand how we may safely transform such rule-based specifications. The first few will be warmups. Then the real thing!

- Observation: Weakening a guard is sound
- Suppose we add a disjunct as below (Cond1) and still manage to show that P is an invariant, then without adding Cond1, the result must still hold

  Rule1: g1 \/ Cond1 ==> a1

  Rule2: g2 ==> a2

  Invariant P

- Reason: Rule1 fires more often with Cond1 added!
- May get false alarms (**P** may fail if Rule1 fires spuriously)
- For many "weak properties" **P**, we can "get away" by guard weakening
  - This is a standard abstraction, first proposed by Kurshan (E.g. removing a module that is driving this module, letting inputs "dangle")
- BUT in the CMP method, we won't do this – rather we will do guard strengthening!
- Except it is useful to know this disjunction property in thinking about certain steps of CMP

But… Guard Strengthening is, by itself, Unsound

- Strengthening a guard is not sound

  Rule1: g1 /\ Cond1 ==> a1

  Rule2: g2 ==> a2

  Invariant P

- Reason: Rule1 fires only when **g1 /\ Cond1**
- So, less behaviors examined in checking **P**
  - **Thus, verifying _with_ Cond1 means nothing for verification without Cond1**
- **But hang on, there is more in the CMP method ☺**

Guard Strengthening can be made sound, if the conjunct is implied by the guard

- This is sound
  - **Rule1: g1 /\ Cond1 ==> a1**
  - **Rule2: g2 ==> a2**
  - **Invariant P /\ (g1 ==> Cond1)**
- Reason: Rule1 fires only when **g1 /\ Cond1**
- BUT, **Cond1** is always implied by **g1; we are showing g1 ➔ Cond1 is an invariant also** , so no real loss of states over which Rule1 fires…
  - Call this "Guard Strengthening Supported by Lemma"
- Except, you are showing the invariant in the same modified system!
  - This is fine:
    - Initial state satisfies P, and also (g1 ➔ Cond1). Thus, whenever g1 is true in the initial state, Cond1 is an implied fact. So g1 /\ Cond1 is like g1 by itself.
    - Thus "a1" can be conducted in the initial state , to obtain the next set of states. (No change wrt g2 and a2, so they are like before.)
  - In general
    - At state t (by induction over time), we have P true and (g1 ➔ Cond1) true.
    - Thus , g1, when true, implies Cond1 at time t. Thus g1 /\ Cond1 is like g1 (same truth status) at time t
      - Thus we can obtain the state at t+1 safely via "a1"

Summary of Transformations so far (checkmark shows what's safe)

```
rule g1 ==> a1;
rule g2 ==> a2;
invariant P;
```

```
rule g1 /\ cond1 ==> a1;
rule g2 ==> a2;
invariant P;
```

X

```
rule g1 \/ cond1 ==> a1;
rule g2 ==> a2;
invariant P;
```

✓

```
rule g1 /\ cond1 ==> a1;
rule g2 ==> a2;
invariant P /\ (g1 => cond1);
```

✓

The CMP Approach

- Weaken to the Extreme
- Then Strengthen Back Just Enough (to pass all properties)

Weaken to the Extreme sounds crazy at first!

Rule1: g1 \/ True ==> a1

Rule2: g2 ==> a2

Invariant P

The transition system above

can be transformed to the one below without any issues

(except the proof of P being an invariant might not go through) – but that will be fixed momentarily

Rule1: True ==> a1

Rule2: g2 ==> a2

Invariant P

Strengthen Back Some

Rule1: True /\ C1 ==> a1
Rule2: g2 ==> a2
Invariant P /\ (g1 => C1)

"Not Enough!" may be
the outcome of strengthening.
That is, while we added C1 back,
it may not be strong-enough.

How to pick C1 will be discussed soon.

Strengthen Back More…

Rule1: True /\ C1 ==> a1
Rule2: g2 ==> a2
Invariant P /\ g1 => C1

"Not Enough!"

Rule1: True /\ C1 /\ C2 ==> a1
Rule2: g2 ==> a2
Invariant P /\ (g1 => C1) /\ (g1 => C2)

"OK, just right!"

A Variation of Guard Strengthening Supported by Lemma: Doing it in a meta-circular manner (i.e., the temporal induction I alluded to earlier…)

rule g1 ==> a1;

rule g2 ==> a2;

invariant P;

→

rule g1 /\ cond1 ==> a1;

rule g2 ==> a2;

invariant P /\ (g2 => cond2);

rule g1 ==> a1;

rule g2 /\ cond2 ==> a2;

invariant P /\ (g1 => cond1);

✓

This is the approach in our work

Now the secret: in the CMP method, the designer decouples all nodes beyond k (typically 2) explicitly modeled nodes. Then, brings back the N-k nodes, but in 'spirit' – i.e., in terms of the non-interference conditions they must obey (note that "N" is a free parameter).

The Ci are thus the non-interference lemmas. Each is discovered upon seeing a counterexample, and then added back into the system! If the coherence invariant is proved (usually this happens), then you ended up having used a model-checker to prove a parametric theorem – which is a big deal!

```
rule g1 ==> a1;
rule g2 ==> a2;
invariant P;
```

→

```
rule g1 /\ cond1 ==> a1;
rule g2 ==> a2;
invariant P /\ (g2 => cond2);
```

```
rule g1 ==> a1;
rule g2 /\ cond2 ==> a2;
invariant P /\ (g1 => cond1);
```

✓

This is the approach in our work

This method has been perfected at Intel and in production use!

- See https://www.cs.utexas.edu/~hunt/FMCAD/FMCAD09/slides/talupur.pdf
- https://dl.acm.org/doi/pdf/10.5555/1517424.1517434
- Designers write “protocol flow” diagrams as part of standard documentation
  - These are mined to obtain the non-interference lemmas
- See http://formalverification.cs.utah.edu/presentations/fmcad04_tutorial2/chou/ctchou-tutorial.pdf for details of how this is done
- NOTE the table-style specification recommended in the above tutorial at fmcad04 !!

We will now present highlights of the German Protocol to tell you how coherence protocols look like (but this is like a “hello world” of cache protocols)

- See german.m , german.pdf , and abs-german.pdf in the class directory
  - Lec19
  - Lec19/protocols
- We will note down some highlights in the coming slides

CMP highlights from paper of C-T Chou

```
abs-german.m          Mon Nov  1 20:00:47 2004          1

const  ---- Configuration parameters ----

  NODE_NUM : 2;
  DATA_NUM : 2;

type   ---- Type declarations ----

  NODE : scalarset(NODE_NUM);              -- NODE now consists of
                                           -- only concrete nodes
  DATA : scalarset(DATA_NUM);

  ABS_NODE : union {NODE, enum{Other}}; -- ABS_NODE consists of both
                                           -- concrete and abstract nodes
```

CMP highlights from paper of C-T Chou

We now do the model checking, which produces the following counterexample to CtrlProp: node n_1 sends a ReqE to home; home receives the ReqE and sends a GntE to node n_1; node n_1 receives the GntE and changes its cache state to E; node n_2 sends a ReqS to home; home receives the ReqS and is about to send an Inv to node n_1; but suddenly home receives a bogus InvAck from Other (via ABS_RecvInvAck), which causes home to reset ExGntd and send a GntS to node n_2; node n_2 receives the GntS and changes its cache state to S, which violates CtrlProp because node n_1 is still in E. The bogus InvAck from Other is clearly where things start to go wrong: if there is a node in E, home should not receive InvAck from any other node. We can capture this desired property as a ***noninterference lemma***:

```
invariant "Lemma_1"
  forall i : NODE do
    Chan3[i].Cmd = InvAck & CurCmd != Empty & ExGntd = true ->
    forall j : NODE do
      j != i -> Cache[j].State != E & Chan2[j].Cmd != GntE
  end end;
```

CMP highlights from paper of C-T Chou

which says that if home is ready to receive an `InvAck` from node `i` (note that the antecedent is simply the precondition of `RecvInvAck` plus the condition `ExGntd = true`, which is the only case when the `InvAck` is to have any effect in `ABS_RecvInvAck`), then every other node `j` must not have cache state `E` or a `GntE` in transit to it. (We are looking ahead a bit here: if the part about `GntE` is omitted from `Lemma_1`, the next counterexample will compel us to add it.) If `Lemma_1` is indeed true in GERMAN, then we will be justified to ***refine*** the offending abstract ruleset `ABS_RecvInvAck` as follows:

CMP highlights from paper of C-T Chou

```
rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true
==>
  ExGntd := false; undefine MemData;
end;
```

The top-left rule is changed to the bottom-right rule in the CMP method, by introducing non-interference lemmas to strengthen the guards. These lemmas are proven in the same model being refined.

```
rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true &
  forall j : NODE do
    Cache[j].State != E & Chan2[j].Cmd != GntE
  end
==> ... end;
```

German protocol

```
const -- configuration parameters --
NODE_NUM : 6;
DATA_NUM : 2;

type -- type decl --

NODE : scalarset(NODE_NUM);
DATA : scalarset(DATA_NUM);

CACHE_STATE : enum {I, S, E};
CACHE : record State : CACHE_STATE; Data : DATA; end;

MSG_CMD : enum {Empty, ReqS, ReqE, Inv, InvAck, GntS, GntE};
MSG : record Cmd: MSG_CMD; Data : DATA; end;
```

German protocol

```
var -- state variables --

Cache : array [NODE] of CACHE;       -- Caches

Chan1 : array [NODE] of MSG;         -- Channels for Req*
Chan2 : array [NODE] of MSG;         -- Channels for Gnt* and Inv
Chan3 : array [NODE] of MSG;         -- Channels for InvAck

InvSet : array [NODE] of boolean;    -- Nodes to be invalidated
ShrSet : array [NODE] of boolean;    -- Nodes having S or E copies
ExGntd : boolean;                    -- E copy has been granted
CurCmd : MSG_CMD;                    -- Current request command
CurPtr : NODE;                       -- Current request node
MemData : DATA;                      -- Memory data
AuxData : DATA;                      -- Latest value of cache line
```

German protocol

```
-- Initial States --

ruleset d : DATA do startstate "init"
--
-- All nodes: init all cmd channels to be empty, Cache States I,
-- the set of nodes to be invalidated is empty
-- and nodes having S or E copies empty
--
  for i : NODE do
    Chan1[i].Cmd := Empty;
    Chan2[i].Cmd := Empty;
    Chan3[i].Cmd := Empty;
    Cache[i].State := I;
    InvSet[i] := false;
    ShrSet[i] := false;
  end;
  ExGntd := false;
  CurCmd := Empty;
  MemData := d;
  AuxData := d;
end end;
```

German protocol

```
-- State Transitions --
--------------------------------------------
ruleset i : NODE do
-- Any node with cmd req channel empty and cache I can request ReqS
  rule "SendReqS"
    Chan1[i].Cmd = Empty &
    Cache[i].State = I
    ==>
    Chan1[i].Cmd := ReqS;  -- raises "ReqS" semaphore
  end
end;
```

German protocol

```
------------------------
ruleset i : NODE do
-- Any node with cmd req channel empty and cache I/S can request ReqE
  rule "SendReqE"
    Chan1[i].Cmd = Empty &
   (Cache[i].State = I |
    Cache[i].State = S)
    ==>
    Chan1[i].Cmd := ReqE;  -- raises "ReqE" semaphore
  end
end;
```

German protocol

```
-------------------------------------------
ruleset i : NODE do
-- For any node that is waiting with ReqS requested, with CurCmd Empty
-- we set CurCmd to ReqS on behalf of node i (setting CurPtr to point to it).
-- Then void Chan1 empty.
-- Now Set the nodes to be invalidated to the nodes having S or E copies.
  rule "RecvReqS" -- prep action of dir ctrlr
    CurCmd = Empty &
    Chan1[i].Cmd = ReqS
    ==>
    CurCmd := ReqS;
    CurPtr := i; -- who sent me ReqS
    Chan1[i].Cmd := Empty;         -- drain its cmd
    for j : NODE do InvSet[j] := ShrSet[j] end; -- inv = nodes with S/E
  end
end;
```

German protocol

```
-----------------------
ruleset i : NODE do
-- For any node that is waiting with ReqE requested, with CurCmd Empty
-- we set CurCmd to ReqE on behalf of node i (setting CurPtr to point to it).
-- Then void Chan1 empty.
-- Now Set the nodes to be invalidated to the nodes having S or E copies.
  rule "RecvReqE"
    CurCmd = Empty &
    Chan1[i].Cmd = ReqE
    ==>
    CurCmd := ReqE;
    CurPtr := i; -- who sent me ReqE
    Chan1[i].Cmd := Empty;       -- drain its cmd
    for j : NODE do InvSet[j] := ShrSet[j] end; -- inv = nodes with S/E
  end
end;
```

German protocol

```
--------------------------------------------
ruleset i : NODE do
-- For every node with Chan2 Cmd empty and InvSet true (node to be invalidated)
-- and if CurCmd is ReqE or (ReqS with ExGnt true), then
-- void Chan2 Cmd to Inv, and remove node i from InvSet (invalidation already set out)
  rule "SendInv"
    Chan2[i].Cmd = Empty &
    InvSet[i] = true &  -- Gnt* and Inv channel
    ( CurCmd = ReqE |                   -- DC: curcmd = E
      CurCmd = ReqS & ExGntd = true )   -- DC: curcmd = S & ExGntd
  ==>
    Chan2[i].Cmd := Inv; -- fill Chan2 with Inv
    InvSet[i] := false;
  end
end;
```

German protocol

```
----------------------
--
-- When a node gets invalidated, it acks, and when it was E
-- then the node (i) coughs up its cache data into Chan3
-- Then cache state is I and undefine Cache Data
--
ruleset i : NODE do
  rule "SendInvAck"
    Chan2[i].Cmd = Inv &
    Chan3[i].Cmd = Empty
    ==>
    Chan2[i].Cmd := Empty;
    Chan3[i].Cmd := InvAck;
    if (Cache[i].State = E) then Chan3[i].Data := Cache[i].Data end;
    Cache[i].State := I; undefine Cache[i].Data;
  end
end;
```

German protocol

```
------------------------
ruleset i : NODE do
  rule "RecvInvAck"
    Chan3[i].Cmd = InvAck &
    CurCmd != Empty
    ==>
    Chan3[i].Cmd := Empty;
    ShrSet[i] := false;
    if (ExGntd = true) then ExGntd := false;
    MemData := Chan3[i].Data;
    undefine Chan3[i].Data end;
  end
end;
```

German protocol

```
-------------------------------------------
ruleset i : NODE do
  rule "SendGntS"
    CurCmd = ReqS &
    CurPtr = i &
    Chan2[i].Cmd = Empty &
    ExGntd = false
    ==>
    Chan2[i].Cmd := GntS;
    Chan2[i].Data := MemData;
    ShrSet[i] := true;
    CurCmd := Empty;
    undefine CurPtr;
  end
end;
```

German protocol

```
----------------------
ruleset i : NODE do
  rule "SendGntE"
    CurCmd = ReqE &
    CurPtr = i &
    Chan2[i].Cmd = Empty &
    ExGntd = false &
    forall j : NODE do ShrSet[j] = false end -- nodes having S or E status
    ==>
    Chan2[i].Cmd := GntE;
    Chan2[i].Data := MemData;
    ShrSet[i] := true;
    ExGntd := true;
    CurCmd := Empty;
    undefine CurPtr;
  end
end;
```

German protocol

```
-----------------------
ruleset i : NODE do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS
    ==>
    Cache[i].State := S;
    Cache[i].Data := Chan2[i].Data;
    Chan2[i].Cmd := Empty;
    undefine Chan2[i].Data;
  end
end;
```

German protocol

```
-----------------------
ruleset i : NODE do
  rule "RecvGntE"
    Chan2[i].Cmd = GntE
    ==>
    Cache[i].State := E;
    Cache[i].Data := Chan2[i].Data;
    Chan2[i].Cmd := Empty;
    undefine Chan2[i].Data;
  end
end;
```

German protocol

```
------------------------
ruleset i : NODE;           -- for every node i
        d : DATA            -- for every data d
  do
    rule "Store"
      Cache[i].State = E    -- if node is in E
      ==>
      Cache[i].Data := d;   -- store d into Cache[i].Data
      AuxData := d;         -- Also update latest cache line value
    end                     -- The node in E can get any "D" value
end;

------------------------------------------------
```

Invariants of the German protocol

```
----------------------------------------------


---- Invariant properties ----

invariant "CtrlProp"
forall i : NODE do
  forall j : NODE do
   i!=j ->
    (Cache[i].State = E -> Cache[j].State = I) &
    (Cache[i].State = S -> Cache[j].State = I |
                           Cache[j].State = S)
  end
end;

invariant "DataProp"
( ExGntd = false -> MemData = AuxData ) &
 forall i : NODE
 do Cache[i].State != I ->
    Cache[i].Data = AuxData
 end;


----------------------------------------------
```

Locking protocols, Scalar Sets, etc

- https://www.cs.utexas.edu/~shmat/courses/cs395t_fall04/cs395t_murphi.html
- http://www.cfdvs.iitb.ac.in/download/Docs/verification/tools/murphi/html/murphiinfo.html

How to initialize the dist locking protocol w/o breaking symmetry required by scalar sets!

```
-------------------------------------------------------

Ruleset n:procT Do
Startstate
For p:procT Do
    initq(request_bufs[p]);
    prob_owners[p] := n;  -- designate some n in procT  to be the owner
    initq(waiters[p]);
    ar_states[p] := ENTER;
    hstates[p] := HANDLE;
    mutexes[p] := false;
End;
Endstartstate;
Endruleset;
```

Code … ralize)

```
---------------------------------------------------------------------------
-- Murphi code for the locking protocol
-- Author : Ganesh Gopalakrishnan, written circa year 2000
-- Derived from Dilip Khandekar and John Carter's work
-- Reference to the work:
--
---------------------------------------------------------------------------
Const
  Nprocs         : 7; -- >= 2 reqd to satisfy request_bufT type declaration.

-------------------------------------------------------

Type
  procT : 0..Nprocs-1; -- Scalarset (Nprocs);

  request_bufT :record
                        Ar: Array[0..Nprocs-2] of procT;
                        Count: -1..Nprocs-2 -- legal range is 0..Nprocs-2
                                            -- -1 acts as empty indicator
                end;

                /* With Nprocs=1, we get 0..-1 which makes sense
                   mathematically (empty) but perhaps not in Murphi.
                   So, avoid Nprocs <= 1. Similar caveats apply for
                   all array declarations of the form 0..N-2. */

  stateT : enum { ENTER, TRYING, BLOCKED, LOCKED, EXITING };
  hstateT: enum { HANDLE, TRYGRANT };

-------------------------------------------------------
```

Code from dist t ... eed to generalize)

```
Var
   request_bufs   : Array [procT] of request_bufT;
   prob_owners    : Array [procT] of procT;
   waiters        : Array [procT] of request_bufT;
   mutexes        : Array [procT] of Boolean;

   ar_states      : Array [procT] of stateT;
   hstates        : Array [procT] of hstateT;


-------------------------------------------------------------------


procedure initq(var queue: request_bufT);
               -- queue of Array range 0..Nprocs-2
begin
   for i:0..Nprocs-2 Do
         Undefine queue.Ar[i]
   end;
   queue.Count := -1 -- empty queue
end;

function frontq(queue: request_bufT): procT;
               -- queue of Array range 0..Nprocs-2
begin
   if (queue.Count < 0)
   then Error "Front of empty queue is undefined"
   else return queue.Ar[0]
   endif
end;
```

Code from di … to generalize)

```
function nonemptyq(queue: request_bufT) : boolean;
begin
  return (queue.Count >= 0)
end;

function emptyq(queue: request_bufT) : boolean;
begin
  return (queue.Count = -1)
end;

procedure dequeue(var queue: request_bufT);
                  -- queue of Array range 0..Nprocs-2
begin
 if queue.Count = -1
 then Error "Queue is empty"
 else queue.Count := queue.Count - 1;
      if queue.Count = -1
      then Undefine queue.Ar[0]
      else for i := 0 to queue.Count do
             queue.Ar[i] := queue.Ar[i+1]
           end;
           Undefine queue.Ar[queue.Count+1]
      endif
 endif
end;
```

Code from dist termination… how initialization is done + how invariants are done (need to generalize)

```
procedure enqueue(var queue: request_bufT; pid: procT);
                  -- queue of Array range 0..Nprocs-2
begin
 if      queue.Count = Nprocs-2
 then    Error "Queue is full";
 else    queue.Count := queue.Count + 1;
         queue.Ar[queue.Count] := pid
 endif;
end;

-------------------------------------------------------

procedure place_request(prob_owner, p : procT);
begin
        enqueue(request_bufs[prob_owner], p)
end;
```

```
procedure copytail(var source_queue, destination_queue : request_bufT);
/* Called only when source_queue is non_empty, i.e. source_queue.Count >= 0.
   Copies the tail of the queue "source_queue" into
   "destination_queue" (which, in actual use, happens to be
    at the new prob_owner),  and also undefines "source_queue" and
   the unused locations of "destination_queue" (at the new probable owner).
   If source_queue.Count = 0, there is no tail to be copied, and we are done.
*/
begin
        if source_queue.Count > 0 -- non-empty and has >= 1 element
        then
           for i := 1 to source_queue.Count do
                destination_queue.Ar[i-1] := source_queue.Ar[i];
                Undefine source_queue.Ar[i];
           end;
           destination_queue.Count := source_queue.Count - 1;
           Undefine destination_queue.Ar[source_queue.Count];
           Undefine source_queue.Ar[0];
           source_queue.Count := -1
        else
           Undefine source_queue.Ar[0];
           source_queue.Count := -1
        endif
end;
```

Code from generalize)

```
-------------------------------------------------------

Ruleset p:procT Do
    Alias       request_buf: request_bufs[p]  Do
    Alias       prob_owner : prob_owners[p]   Do
    Alias       waiter     : waiters[p]       Do
    Alias       state      : ar_states[p]     Do
    Alias       hstate     : hstates[p]       Do
    Alias       mutex      : mutexes[p]       Do

        Rule "Try acquiring the lock"
                ((state = ENTER) & !mutex)
                        ==> mutex := true;
                            state := TRYING;
        Endrule;
        -------------------------------------------------------

        Rule "If the lock is around, grab it"
                ((state = TRYING) & (prob_owner = p) & mutex)
                        ==> mutex := false;
                            state := LOCKED;

        Endrule;
        -------------------------------------------------------

        Rule "If the lock isn't around, send request out"
                ((state = TRYING) & (prob_owner != p) & mutex)
                        ==> mutex := false;
                            place_request(prob_owner,p);
                            state := BLOCKED;
        Endrule;
        -------------------------------------------------------
```

```
--------------------------------------------------------------------------------
-- Contains injected bug:
--   This model contains a working and verified representation of a
--    N peterson algorithm, but with an injected "bug."
--   That adds a buffer of length `BL` which records the order
--    in which processes get the lock of a resource until it is full.
--   When full it will declare the order in the buffer as a sequence of
--    lock ownerships that if any subsequence of length `MSL` or more
--    occurs in any following `N` length sequences of locking orderings,
--    is declared to be a "bug."
--  To do this we also keep a buffer of length `N` that is filled with a
--   new set before it is checked for said bug.
-- Concepts:
--  These two buffers and the accompanying variables in the statespace,
--   greatly increase the statespace to a point that no current Murphi
--   model checker could ever efficiently hold them all in a set.
--  Therefore, it is known that unless a bug is encountered
--   a traditional Murphi model checker must either end
--   due to state hash saturation, or a built in timeout condition is hit
--   in cases of nivea state sets that have undefined behavior
--   when fully saturated.
--  The "bug" state is designed to be variably difficult to find, but
--   also defined that there is no possible progression that will never
--   meet the criteria... MAYBE
--------------------------------------------------------------------------------
```

Snippets fro

```
Const
  N: 4; -- 7; -- the number of processes
  LHL: N*4; -- the size of the history buffer
  BL: (N*2); /*N;*/ -- the length of the total bug sequence
  MSL: 3; -- the minimum length of the bug sub sequence

Type
  --  The scalarset is used for symmetry, which is implemented in Murphi 1.5
  --  and not upgraded to Murphi 2.0 yet
  pid: scalarset (N);
  pid_r: 0..(N-1);  -- math workable range version of pid
  b_ind_t: 0..(BL-1); -- index type for bug array
  lh_t: 0..(LHL-1); -- index type for bug array
  -- pid: 1..N;
  priority: 0..N;
  label_t: Enum{L0, -- : non critical section; j := 1; while j<n do
                L1,  -- : Beginwhile Q[i] := j
                L2,  -- : turn[j] := i   (asking the other process to take t
stem)
                L3,  -- : wait until (forall k != i, Q[k] < j) or turn[j] !=
t for your turn) <-- you get lock here
                L4   -- : critical section; Q[i] := 0  (your turn)
                };
Var
  P: Array [ pid ] Of label_t;       -- stores the current state of each pro
  Q: Array [ pid ] Of priority;      -- stores what priority each process ha
  turn: Array [ priority ] Of pid;   -- maps each priority to a process
  localj: Array [ pid ] Of priority; -- maps each process to it's current pr

  -- locking history buffer
  lock_hist: Array [ lh_t ] Of pid;
  bug: Array [ b_ind_t ] Of pid; -- where we store first locking order to us
  lh_size: lh_t;  -- start and end of circ buffer
```

Snippets from Pet

```
  lh_size: lh_t;  -- start and end of circ buffer
  b_size: b_ind_t;
  lh_is_full, bug_is_full: boolean;  -- if we circling yet (aka

-- Procedure print_bug(tmp:boolean)
-- Begin
--   put "Bug := [ ";
--   put bug;
--   put "\n\n"
-- EndProcedure;


Ruleset i: pid  Do

  Rule "L0 execute inc j and while"
    P[i] = L0  ==>
  Begin
    localj[i] := 1;
    P[i] := L1;
  End;

  Rule "L1 execute assign Qi j"
    P[i] = L1  ==>
  Begin
    Q[i] := localj[i];
    P[i] := L2;
  End;

  Rule "L2 execute assign TURNj i"
    P[i] = L2  ==>
  Begin
    turn[localj[i]]  := i;
    P[i] := L3;
  End;
```

Snippets from Peterson’s protocol with programmable bug-depth

```
  Rule "L3 execute wait until"
    P[i] = L3   ==>   -- any process at L3 can do
  Begin
    If ( Forall k: pid Do
              ( k!=i ) -> ( Q[k]<localj[i] )   -- if
           End --forall
         | ( turn[localj[i]] != i ) ) -- ?!? if ou
our process ?!?
    Then                                    -- ?!?  |->
to keep the always action below from executing p
      localj[i] := localj[i] + 1;   -- always inc
a)
      If ( localj[i]<N )
      Then
        P[i] := L1; -- go update the Q again
      Else          -- when localj[i] = N -> it
        P[i] := L4; -- this assigns the lock to
      End; --If
    End; --If
  End;
```

Snippets from Peterson’s protocol with programmable bug-depth

```
Rule "L4 execute critical and assign Qi 1"
  P[i] = L4  ==>
Begin
  Q[i] := 1;
  P[i] := L0;
  -- we have fully reached the lock (update
  If (bug_is_full)
  Then
    lock_hist[lh_size] := i;
    lh_size := (lh_size + 1) % LHL;
    lh_is_full:= (lh_size = 0)
  Else
    Assert (!bug_is_full) "Bug is full, but
    bug[b_size] := i;
    b_size := (b_size + 1) % BL;
    If (b_size = 0)
    Then
      bug_is_full := True;
      -- print_bug(True);
    EndIf;
  EndIf;
EndRule;
```

Snippets from Peterson's

```
End; --Ruleset

Startstate "init"
Begin
  For i:pid Do
    P[i] := L0;
    Q[i] := 0;
  End; --For

  For i: priority Do
    Undefine turn[i];
  End; --For

  Clear localj;

  -- hist bug stuff
  lh_is_full := False;
  bug_is_full := False;
  lh_size := 0;
  b_size := 0;
  Undefine lock_hist;
  Undefine bug;
  If (MSL<2)
    Then Error "CONFIG ERROR: `MSL` needs
  EndIf;
  If (MSL>LHL)
    Then Error "CONFIG ERROR: `MSL` needs
  EndIf;
End;
```

Snippets from Peterson's protocol with programmable bug-depth

```
-------------------------------------------------------

Ruleset n:procT Do
Startstate
For p:procT Do
    initq(request_bufs[p]);
    prob_owners[p] := n;  -- designate some n in procT  to be the owner
    initq(waiters[p]);
    ar_states[p] := ENTER;
    hstates[p] := HANDLE;
    mutexes[p] := false;
End;
Endstartstate;
Endruleset;
```

```
----------------------------------------------
Invariant "Bug3"
!(
(prob_owners[0]=1)&(prob_owners[1]=2)&
(prob_owners[2]=5)&
(prob_owners[3]=4)&(prob_owners[4]=5)&
(prob_owners[5]=6)&
(prob_owners[6]=6)

)


----------------------------------------------
-- Invariant "Parametric-Bug TBD..."
-- TBD
```