CAV, Chicago Illinois, 4:30pm 18 July 2000

Verification Diagrams ~~Revisited~~ Rediscovered: Disjunctive Invariants for An Easy Verification

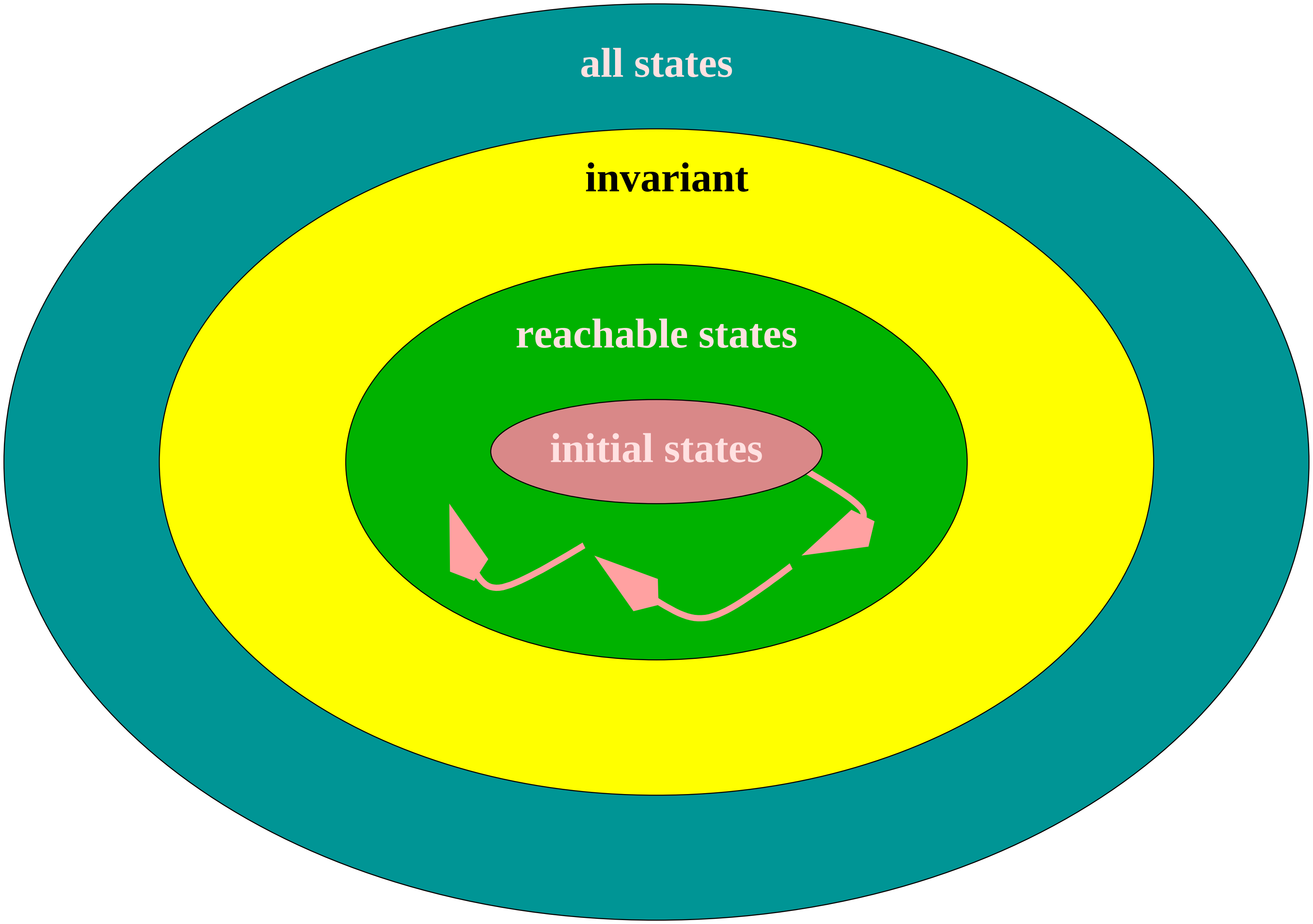
John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Background: Verification of Invariants

- Invariants characterize properties (aka. safety properties) that are true of all the reachable states in a system
  - Equivalently, are supersets of the reachable states
- Model checkers can verify an invariant by enumerating the reachable states and checking the property at each one
  - Only works when the number of reachable states is finite (and relatively small)
  - Or when there is a verifiably property-preserving abstraction of the system
  - (Traditional model checking employs aggressive abstractions and is used for debugging, not verification)
- Otherwise, we need to employ deductive methods
  - Methods based on theorem proving
  - Powerful but hard to use

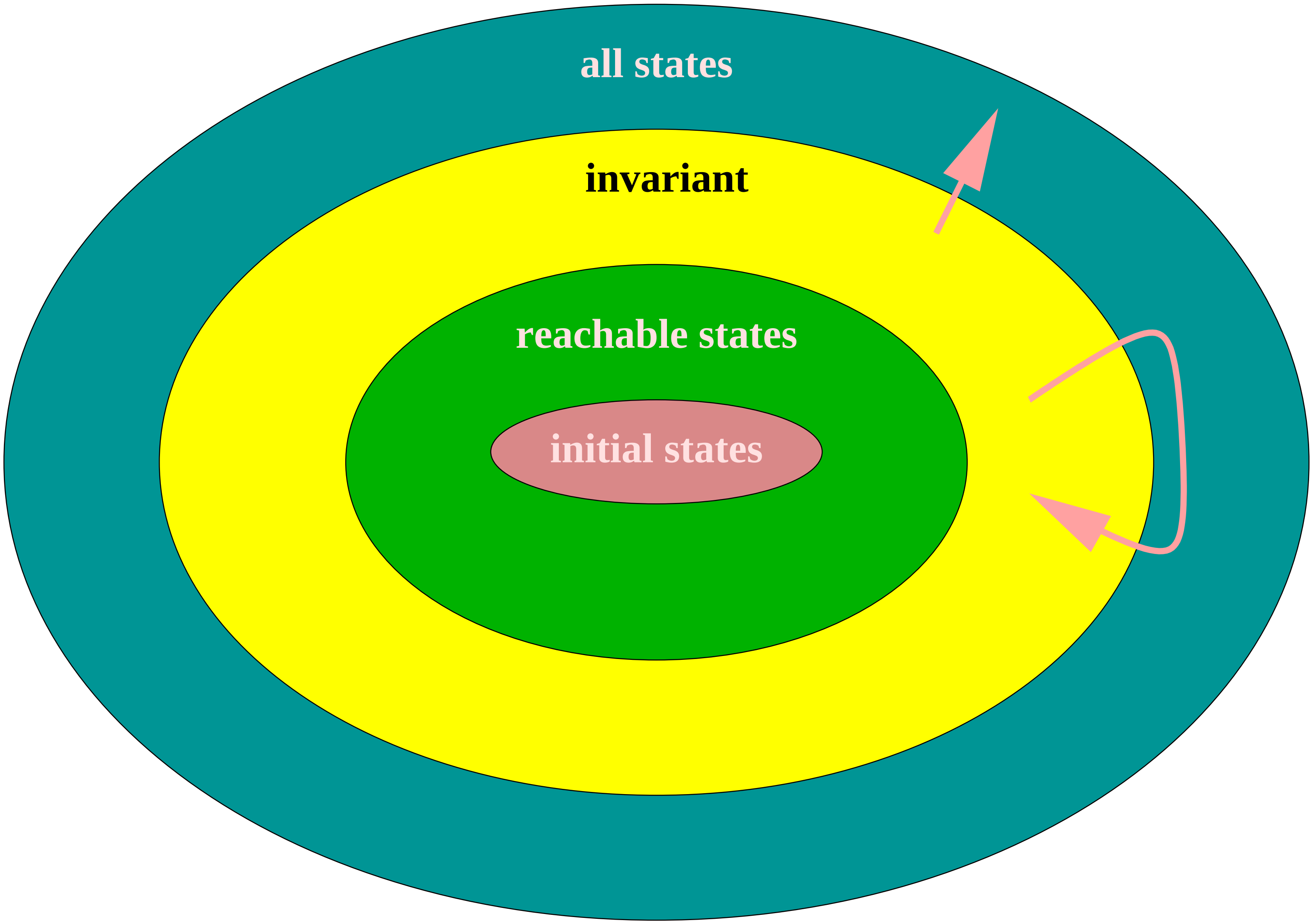
Invariants In Pictures



Background: Deductive Verification of Invariants

- With theorem proving, we cannot enumerate the reachable states directly
  - Too many of them
- So instead usually try to prove that the putative invariant G is inductive (on transition relation T, with initial states I)
  - Includes all the initial states: $I(s) \supset G(s)$
  - Is closed on the transitions: $G(s) \wedge T(s,t) \supset G(t)$
- The reachable states are the smallest set that is inductive, so inductive properties are invariants
- Trouble is, naturally stated invariants are seldom inductive
  - The second condition is violated
- Need to make them smaller (stronger) to exclude the states that take you outside the invariant

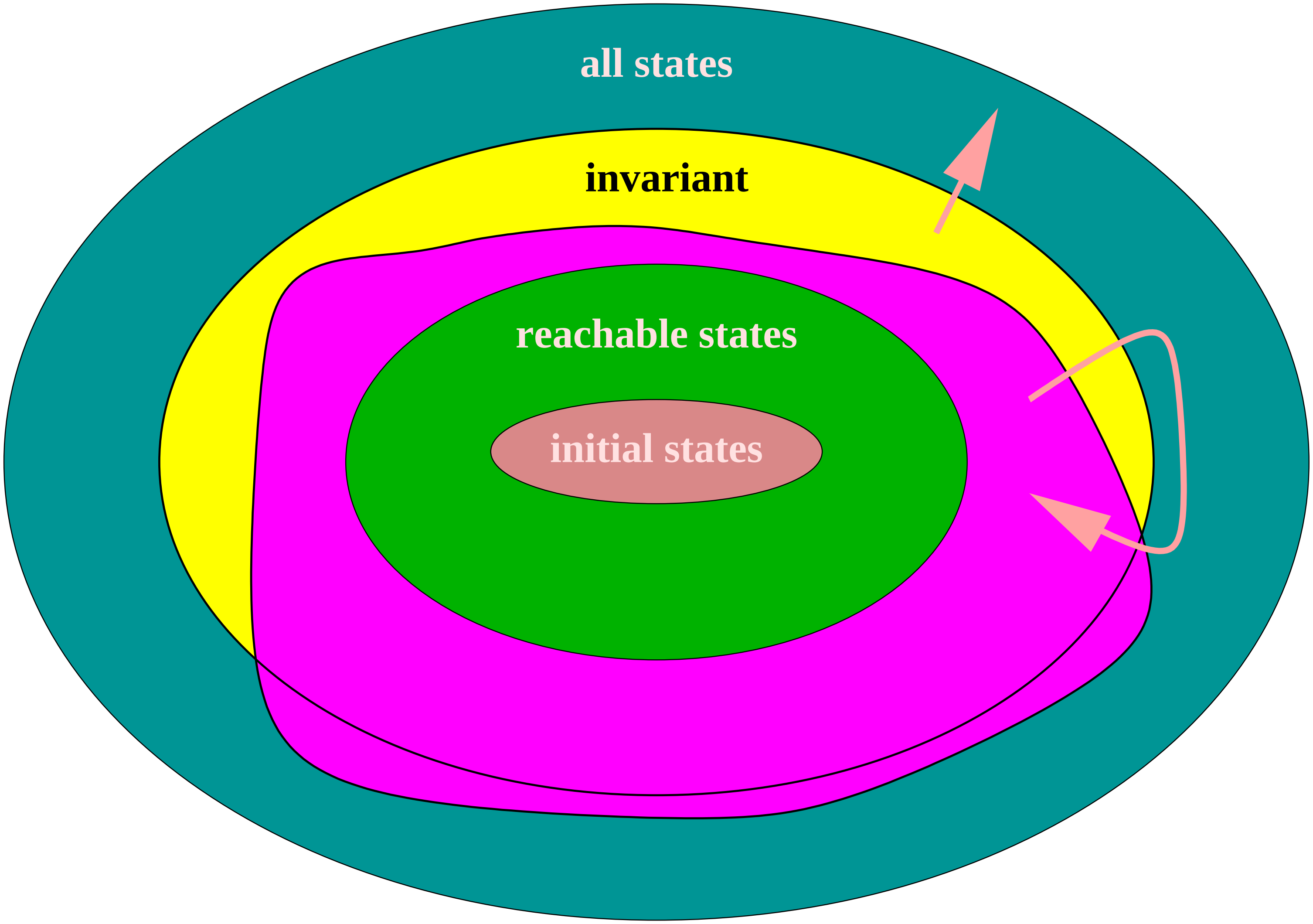
Noninductive Invariants In Pictures



Background: Strengthening Invariants To Make Them Inductive

- Postulate a new invariant that excludes the states (so far discovered) that take you outside the desired invariant
- Show that the conjunction of the new and the desired invariant is inductive
- Iterate until success or exasperation
- Process can be made systematic, but it always tedious
- Bounded retransmission protocol required 57 such iterations
  - Took a couple of months to complete
    (Havelund and Shankar)
- Notice that each conjunct must itself be an invariant
  (the very property we are trying to establish)

Inductive Invariants In Pictures



Background: Strengthening Invariants To Make Them Inductive

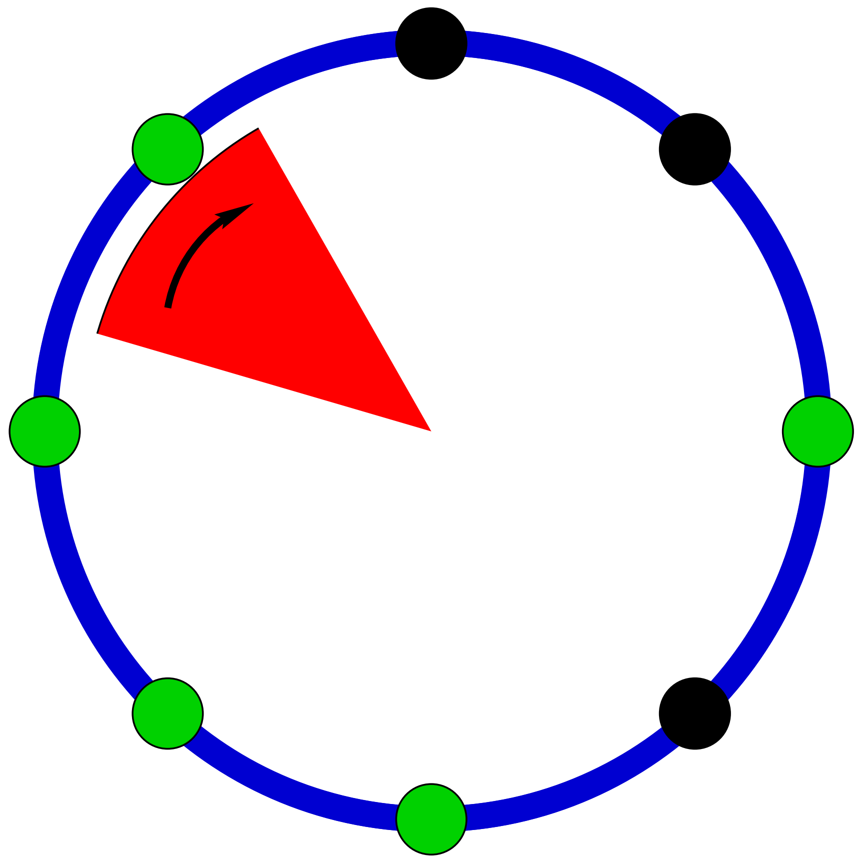
- Postulate a new invariant that excludes the states (so far discovered) that take you outside the desired invariant
- Show that the conjunction of the new and the desired invariant is inductive
- Iterate until success or exasperation
- Process can be made systematic, but it always tedious
- Bounded retransmission protocol required 57 such iterations
  - Took a couple of months to complete (Havelund and Shankar)
- Notice that each conjunct must itself be an invariant (the very property we are trying to establish)

Example: The Time-Triggered Architecture (TTA)

- An architecture for safety-critical real-time embedded control
- Specifically, automobiles
  - Drive-by-wire, not "body electronics"
- It's based on a protocol called the Time-Triggered Protocol (TTP)
- It's been adopted by Daimler-Benz, and Motorola have committed to adding TTP controllers to the their embedded processor chips
- It targets the same 10^{-9} requirement as flight-critical avionics
- And has quite a lot in common with Arinc 659 (the Honeywell SAFEbus) used in the Boeing 777 AIMS
- And may be used by Airbus in the A3XX

Example: The Time-Triggered Protocol (TTP)

Creates a synchronous, TDMA ring on a broadcast bus



Example: The Time-Triggered Protocol (TTP)

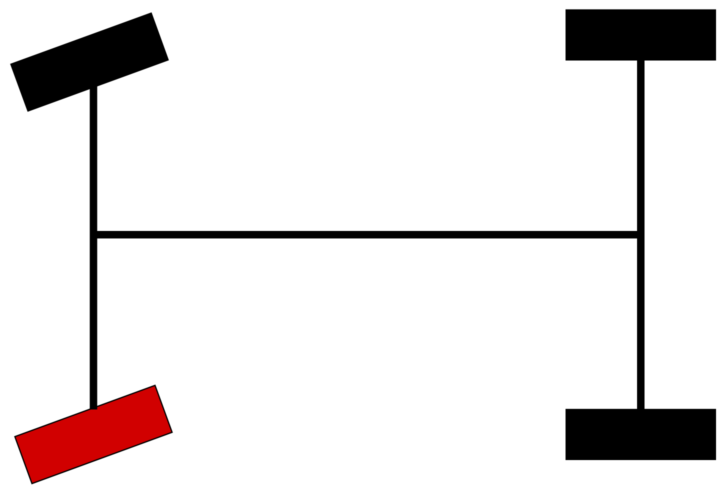
TTP provides several services in a tightly integrated manner

- Clock synchronization (verified in PVS by Pfeifer, Schwier, and Friedrich W. von Henke, DCCA '99)
- Time-triggered implementation of synchronous algorithms (verified in PVS by Rushby, DCCA '97, TSE '99)
- Time Division Multiple Access (TDMA)
    - Deterministic mutual exclusion
    - Data communication
- Group membership (the example considered here)

Assumes (and enforces) fail silence

The Need for Group Membership

- Consider a brake-by-wire application
- Separate computers at each wheel adjust braking force according to inputs from brake pedal, accelerometers, steering angle, wheel-spin sensors etc.
- Suppose one of these computers fails



- The others need to redistribute the braking force
- So must have consistent opinion about who has failed

Requirements For Group Membership

Each processor maintains a membership set

Validity: the membership sets of nonfaulty processors contain all the nonfaulty processors

- And, ideally, nothing else—but this is not possible because it takes some time to diagnose a faulty processor
- So allow at most one faulty processor in the membership

Agreement: all nonfaulty processors have the same membership sets

Self-Diagnosis: faulty processors eventually remove themselves from their own membership sets (and fail silently)

Subject to assumptions about possible fault modes, fault arrival rate, and maximum number of faults

Group membership: Algorithms

- Must be achieved with very low overhead
- WDAG algorithm (Katz, Lincoln, Rushby WDAG'97) does it with one bit added to each broadcast
- I'll describe a similar, but simpler algorithm, that does not achieve full self-diagnosis
- Each processor maintains an ack bit and appends it to its message when it broadcasts
    - Interpretation is

      "I heard the previous expected broadcast"

        - ⋆ For WDAG it is "I retained the previous expected broadcaster in my membership"
- The actual TTP algorithm is different: uses two bits, encoded in the CRC checksum, much more complicated

Fault Assumptions

Fault modes: very restrictive assumptions

Send Fault

- A faulty processor might not broadcast (in some rounds)
  - Broadcasts are received either by all or none of the nonfaulty processors

Receive Fault

- A faulty processor might not receive some broadcasts

Fault arrivals: faults must be "rare"

- Roughly, no more than one per round
- More precisely, may arrive only when system is "stable"
  - And after a fault it returns to a stable state very quickly

Maximum faults: no restriction

States and Participants

- There are n processors, initially all are nonfaulty
- Time is marked by slots numbered $0, 1, 2, \ldots$
- In slot t processor $t \bmod n$ is the broadcaster
- Processor p has membership set $\mathtt{mem}(p)$ and ack bit $\mathtt{ack}(p)$
- The broadcaster b sends its ack bit $\mathtt{ack}(b)$ in its message
  - Provided $b \in \mathtt{mem}(b)$
- So $p \neq b$ has access to mem(p), ack(p), and ack(b) (but can only change the first two)
- p only expects to receive a message if $b \in \mathtt{mem}(p)$
- And can detect the absence of an expected message (i.e., we are dealing with a synchronous system)

Group membership: Algorithm

Broadcaster: Let $b = i \bmod n$.

$$
\begin{array}{llll}
(a) & b \in \texttt{mem}(b) & \longrightarrow & \texttt{mem}(b)' = \texttt{mem}(b), \quad \texttt{ack}(b)' = \mathit{true} \\
& \text{otherwise} & \longrightarrow & \text{no change.}
\end{array}
$$

Receiver: Consider an arbitrary processor $p \neq b$. If $b \in \texttt{mem}(p)$ and $p \in \texttt{mem}(p)$, the appropriate guarded command from the following list is executed:

$$
\begin{array}{lllll}
(b) & \texttt{ack}(p) \wedge \text{no msg} & \longrightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{b\}, & \texttt{ack}(p)' = \mathit{false} \\
(c) & \texttt{ack}(p) \wedge \texttt{ack}(b) & \longrightarrow & \texttt{mem}(p)' = \texttt{mem}(p), & \texttt{ack}(p)' = \mathit{true} \\
(d) & \texttt{ack}(p) \wedge \neg\texttt{ack}(b) & \longrightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{b\}, & \texttt{ack}(p)' = \mathit{true} \\
(e) & \neg\texttt{ack}(p) \wedge \text{no msg} & \longrightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{p\} & \\
(f) & \neg\texttt{ack}(p) \wedge \neg\texttt{ack}(b) & \longrightarrow & \texttt{mem}(p)' = \texttt{mem}(p), & \texttt{ack}(p)' = \mathit{true} \\
(g) & \neg\texttt{ack}(p) \wedge \texttt{ack}(b) & \longrightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{p\} & \\
& \text{otherwise} & \longrightarrow & \text{no change.} &
\end{array}
$$

The Environment and The Initiality Predicate

- The environment can perform only a single action: it can cause a new fault to arrive—provided no other fault has arrived "recently" (defined later)
- Let the_mem denote the current set of nonfaulty processors

  Fault Arrival: $\exists x \in \mathtt{the\_mem} : \mathtt{the\_mem}' = \mathtt{the\_mem} - \{x\}$
- Initially, all processors are nonfaulty, their membership sets contain all processors, and their ack bits are *true*

  Initially: $\mathtt{the\_mem} = \mathtt{fullset} \wedge \mathtt{mem}(p) = \mathtt{the\_mem} \wedge \mathtt{ack}(p)$

Desired Safety Properties

The safety properties are specified as follows.

Validity:

$$p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \vee \exists x : \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\}$$

Agreement: $p \in \texttt{the\_mem} \wedge q \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{mem}(q)$

The Published WDAG Proof

- Was a conventional inductive invariance proof
- It is incorrect
  - And the algorithm has a bug
    - ⋆ Found independently by Shankar (inspection), and
    - ⋆ Sadie Creese and Bill Roscoe (model checking)
- But is fairly easy to correct
- However, it defeated attempts by Pat Lincoln, Shmuel Katz, and me to formally verify it in PVS
  - Because of its horrible complexity

The Published WDAG Invariant

The invariant has the following conjuncts.

1. All nonfaulty processors have the same membership sets.
2. All nonfaulty processors are in their own membership sets.
3. All nonfaulty processors have the same value for `ack`.
4. For each processor p, $\texttt{ack}(p)$ is true iff in the most recent previous step in which p expected a broadcast from a processor b, either p was b, or $arrived(b, p) \wedge (\texttt{ack}(b) \vee \neg\texttt{ack}(p))$ in that step.
5. If a processor p became faulty less than n steps ago and q is a nonfaulty processor, either p is the present broadcaster or the present broadcaster is in p's local membership set iff it is in q's.
6. If a receive fault occurred to processor p less than n steps ago, then either p is not the broadcaster or $\texttt{ack}(p)$ is $false$ while all nonfaulty q have $\texttt{ack}(q) = true$, or p is not in its local membership set.
7. If in the previous step b is broadcaster, p is a nonfaulty processor, and $arrived(b, p)$ does not hold, then b is faulty in the current step.
8. If the broadcaster b is expected by a nonfaulty processor, then b is either nonfaulty, or became faulty less than n steps ago.

Disjunctive Invariants

- When G fails to be inductive, replace it by some smaller G^1
- And then iterate with smaller $G^2, \ldots, G^i$ until you discover one that is inductive.
- The usual way to form G^i is by conjoining additional predicates:

$$G^i_\wedge = G \wedge G_1 \wedge \cdots \wedge G_i$$

- Instead, conjoin it with a disjunction of predicates

$$G^i_\vee = G \wedge (G_1 \vee \cdots \vee G_i)$$

- This is equivalent to $G^i_\vee = (G \wedge G_1) \vee \cdots \vee (G \wedge G_i)$
- Generally easier to use $G^i_\vee = G'_1 \vee \cdots \vee G'_i$ and then prove $G'_i \supset G$ for each G'_i
- The G'_is are called configurations

Verification Conditions (Proof Obligations)

- We need $(G'_1(s) \vee \cdots \vee G'_i(s)) \wedge T(s,t) \supset G'_1(t) \vee \cdots \vee G'_i(t)$
- Case-split across the configurations in the antecedent

$$G'_i(s) \wedge T(s,t) \supset G'_1(t) \vee \cdots \vee G'_i(t)$$

- Then case-split on the right by introducing transition conditions $C_{i,j}(s)$ such that, for each configuration i

$$\bigvee_j C_{i,j}(s)$$

- And the verification conditions then become

$$G'_i(s) \wedge T(s,t) \wedge C_{i,j}(s) \supset G'_j(t)$$

  for each transition condition j of each configuration i

- Some $C_{i,j}$ may be identically *false* (VC then vacuously *true*); not necessary that the $C_{i,j}$ for different j be disjoint

Verification Conditions (ctd.)

Also, initiality predicate must imply some disjunction of configurations

- $I(s) \supset G'_1(s) \vee \cdots \vee G'_i(s)$

(typically there is just a single starting configuration)

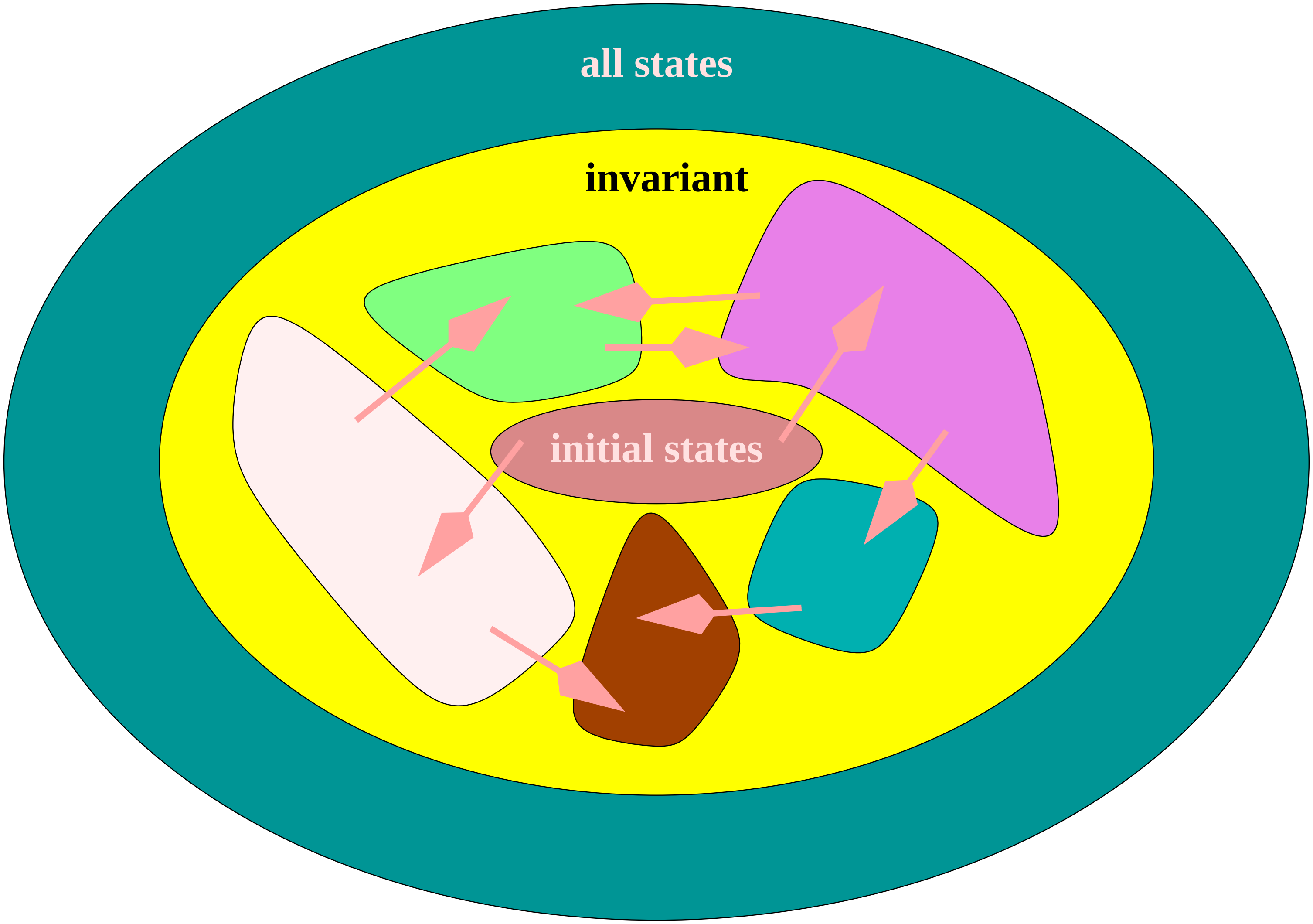
Reminder of the other VCs

- $G'_i \supset G$
- $\bigvee_j C_{i,j}(s)$
- $G'_i(s) \wedge T(s,t) \wedge C_{i,j}(s) \supset G'_j(t)$

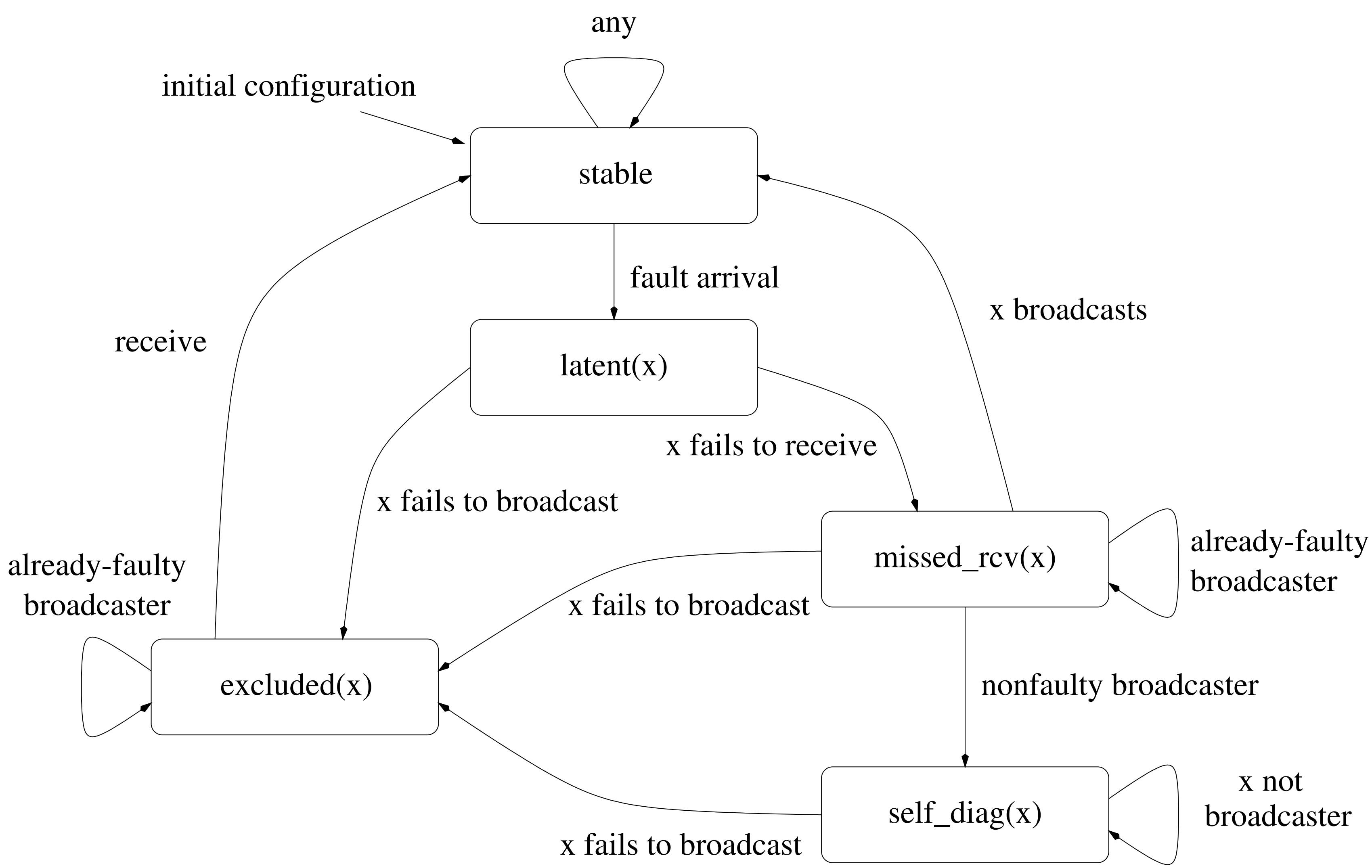
Disjunctive Invariants: Model Checking Intuition

- With theorem proving, we cannot enumerate the reachable states individually
  - Too many of them
- So try grouping them into sets called configurations and do a forward reachability calculation on the configurations
  - Mechanized by symbolic evaluation
- We are done if the configurations are closed under the transition relation, and each reachable configuration is a subset of the desired invariant
  - Equivalently, if each configuration is characterized by a property that implies the desired invariant
- (A symbolic model checker is implicitly doing the same thing)

Disjunctive Invariants In Pictures



There is a Natural Diagrammatic Representation



Systematic Construction

- Invent a starting configuration, show it is implied by the initiality predicate and implies the desired safety property
- For each unvisited configuration, do the following

  Invent some transition conditions and show their disjunction is *true*; then for each condition

  - Symbolically execute the algorithm, starting from the configuration concerned, under the chosen transition condition
  - Result is either a new configuration (must prove it implies the safety property); add to list of unvisited configurations
  - Or implies one previously encountered; may need to generalize previously encountered configuration and revisit its proofs
- Until all visited

Observations

- There are no new proof principles involved here: it is just a different way of organizing a traditional inductive invariance proof (the verification conditions discharge the standard proof obligations)
- Insight is needed to select the transition conditions, and to generalize the results of symbolic execution to yield suitable configurations
- Failure is manifested by inability to close the diagram: may indicate either an incorrect algorithm or poorly chosen configurations and transition conditions

The Configurations for Group Membership Algorithm

The starting configuration is the following

Stable: $p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{true}$

Fault arrival takes us to the following configuration

Latent(x): $x \notin \texttt{the\_mem}$

$\wedge\, p \in \texttt{the\_mem} \cup \{x\} \supset \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

If x is the next broadcaster (it will fail to broadcast) we go to

Excluded$_1$(x):

$x \notin \texttt{the\_mem} \wedge \texttt{mem}(x) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{false}$

Otherwise, x will fail to receive a good broadcast

Missed_rcv(x):

$x \notin \texttt{the\_mem} \wedge \texttt{mem}(x) = \texttt{the\_mem} \cup \{x\} - \{b\} \wedge \texttt{ack}(x) = \mathit{false}$

$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

The Configurations (ctd.)

Then if x fails to broadcast, we go to

Excluded$_2$(x):

$$x \notin \texttt{the\_mem} \wedge \texttt{mem}(x) = \texttt{the\_mem} \cup \{x\} - \{b\} \wedge \texttt{ack}(p) = \mathit{true}$$
$$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{false}$$

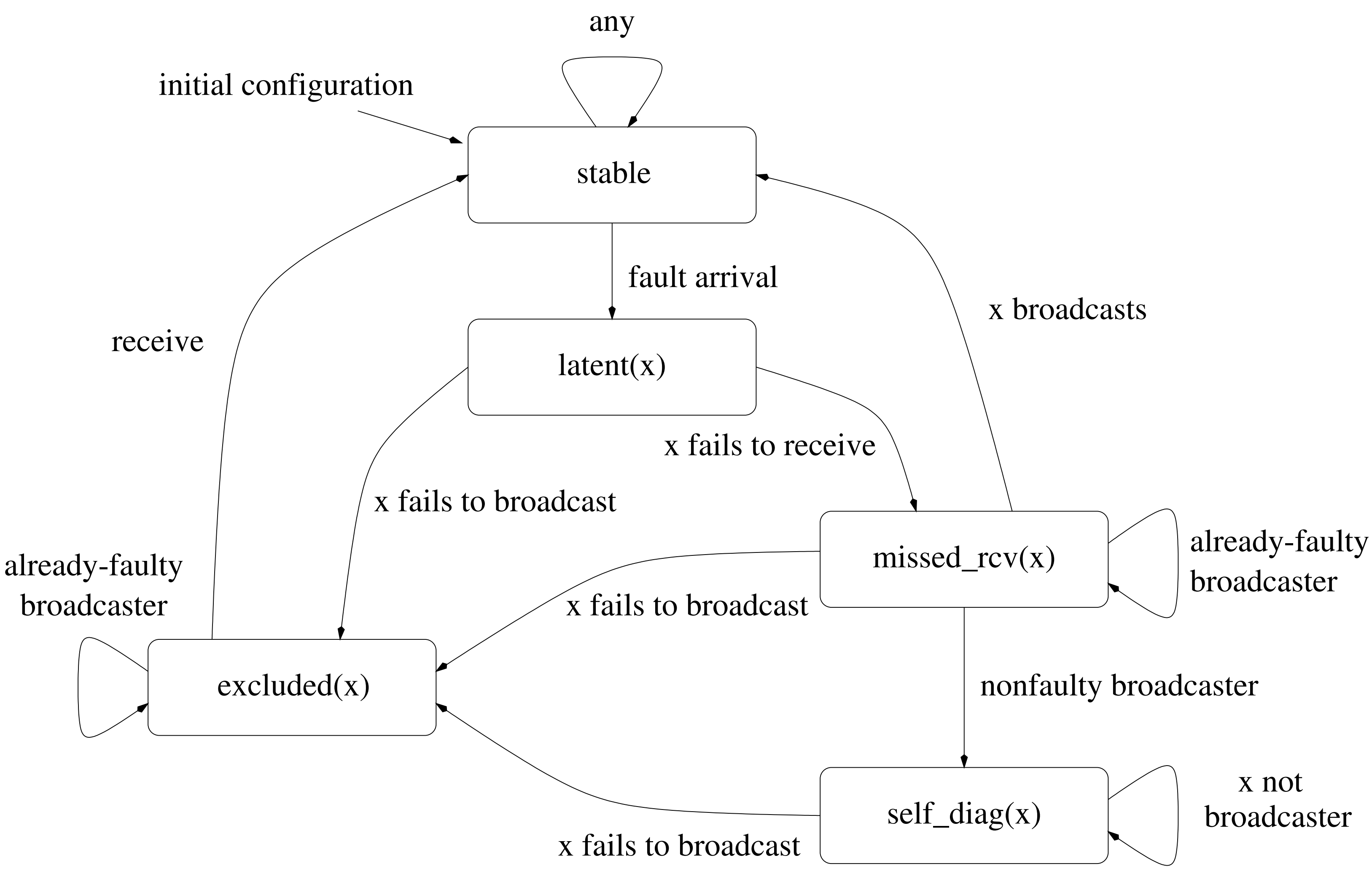
We recognize that *excluded*$_1(x)$ and *excluded*$_2(x)$ should each be generalized to yield the following common configuration.

Excluded(x): $p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{false}$

If the next broadcaster is nonfaulty, Missed_rcv(x) takes us to

Self_diag(x): $x \notin \texttt{the\_mem} \wedge x \notin \texttt{mem}(x)$
$$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$$

Configuration Diagram for Group Membership Protocol



Theorems

I have specified this example in PVS and proved the following

```
validity: THEOREM NOT faulty(t,p) IMPLIES
   membership(t,p) = the_mem(t)
    OR EXISTS x: faulty(t,x) AND membership(t,p) = add(x,the_mem(t))

agreement: THEOREM NOT faulty(t,p) AND NOT faulty(t,q)
     IMPLIES  membership(t,p) = membership(t,q)
```

Actual proof is complicated by need to establish an acceptable fault arrival rate: no more than one per round

- Assume faults arrive only when system is stable
- Prove system returns to a stable state following a fault

  ```
  liveness: THEOREM stable(t) IMPLIES EXISTS j: j>0 AND stable(t+j)
  ```

  And does so "very quickly": by inspection, $j < n$

Typical Lemmas

Bounded liveness complicates specification: need to remember when certain events happened

```
selfdiag_to_selfdiag: LEMMA
  self_diag(t, y, x)
    AND x /= broadcaster(t)
    AND the_mem(t+1) = the_mem(t)
        IMPLIES self_diag(t+1, y, x)

selfdiag_to_excluded: LEMMA
  self_diag(t, y, x)
    AND x = broadcaster(t)
    AND the_mem(t+1) = the_mem(t)
        IMPLIES excluded(t+1, x)

self_diag_exit: LEMMA
  self_diag(t, y, x) IMPLIES
     (FORALL j: x /= broadcaster(t+j))
     OR (EXISTS j: excluded(t+j, x))
```

Proofs

By defining some convenient strategies (macros), most of the proofs become very systematic

Example (selfdiag_to_selfdiag) from the full WDAG proof

```
(SKOSIMP)
 (FORWARD-CHAIN "selfdiag_is_nonstable")
 (FORWARD-CHAIN "fault_model1")
 (EXPAND "self_diag")
 (INSTALL-REWRITES :DEFS NIL :THEORIES ("sets") :REWRITES
  ("the_mem" "faults_latch1" "set_prop2" "set_prop1a"))
 (GROUND)
 (("1" (ASSERT))
  ("2"
   (SKOSIMP)
   (GROUND)
   (("1" (MEM "p!1")) ("2" (MEM "p!1")) ("3" (ACK "p!1"))
    ("4" (PNACK "p!1")))))
```

The WDAG Algorithm

I have also proved the corresponding properties for the full (corrected) WDAG Algorithm

```
validity: THEOREM IF faulty(t,p)
 THEN
    NOT membership(t,p)(p)
      OR subset?(membership(t,p), add(p,the_mem(t)))
 ELSE
    membership(t,p) = the_mem(t)
      OR EXISTS x: faulty(t,x) AND membership(t,p) = add(x,the_mem(t))
 ENDIF

 stable(t): bool =
     FORALL p: IF faulty(t, p) THEN NOT membership(t,p)(p)
               ELSE membership(t, p) = the_mem(t)
                    AND ack(t, p) AND NOT pnack(t, p)
               ENDIF
```

The configuration diagram is about twice as complex

Summary of Experience

- Holger Pfeifer (Ulm) has used the approach to verify the actual TTP algorithm (it is much less uniform than WDAG)
  - Configuration diagram twice as complex as WDAG
  - Will be presented at Forte/PSTV (Pisa, Italy) in October
- Disjunctive invariants made mechanized verification of these algorithms very intuitive and straightforward
- First construct a configuration diagram informally
  - Aids understanding and helps with debugging and design
- Then use it to guide the mechanized verification

Benefits

- Configurations need not be invariants
- Construction of the diagram is incremental; not required to reconsider entire proof as each configuration is added
    - Whereas conjunctive invariants require reconsideration and restatement of the entire verification each time a conjunct is added; discovery of each conjunct relies on interpretation of failed proofs
- The diagram conveys insight into the operation of the algorithm

Comparison With Proof Diagrams

The method is similar to Proof Diagrams introduced by Manna and Pnueli (1994) and subsequently generalized by Bjørner, Browne, de Alfaro, Sipma, and Uribe. Some differences:

Intuitions: disjunctive invariants *vs.* abstract state machine

Finiteness: proof diagrams seem to be finite, verification diagrams need not be (configurations are parameterized)

Arcs: in proof diagrams these correspond to transitions; in configuration diagrams to predicates

Mechanization: provided for proof diagrams by STeP (user proposes diagram, system generates the VCs); no special mechanization provided in PVS: just symbolic execution (automated by rewriting and simplification)

What's New?

- Disjunctive invariants arise naturally in asynchronous systems (case split across the possible transitions)
- Main novelty is the application to synchronous systems
  - No explicit control
  - Configuration diagram extracts the "control" (actually phase) structure
- An addition to the growing repertoire of systematic methods for deductive verification of special classes of systems
  - Aggregation for cache coherence; Park, Das, and Dill
  - Completion functions for processor pipelines; Hosabettu, Gopalakrishnan, and Srivas

Conclusions and Future Work

- Need to try the method on more applications
  - Only group membership examples, so far
  - These are all synchronous algorithms
- Clarify relationship to proof diagrams
- Proof diagrams also need more applications
  - No hard examples so far
  - Examples are all asynchronous algorithms
- Identify when these methods are effective
- Find ways to characterize and systematize these and other "systematic" methods
- Clarify relationship to predicate abstractions