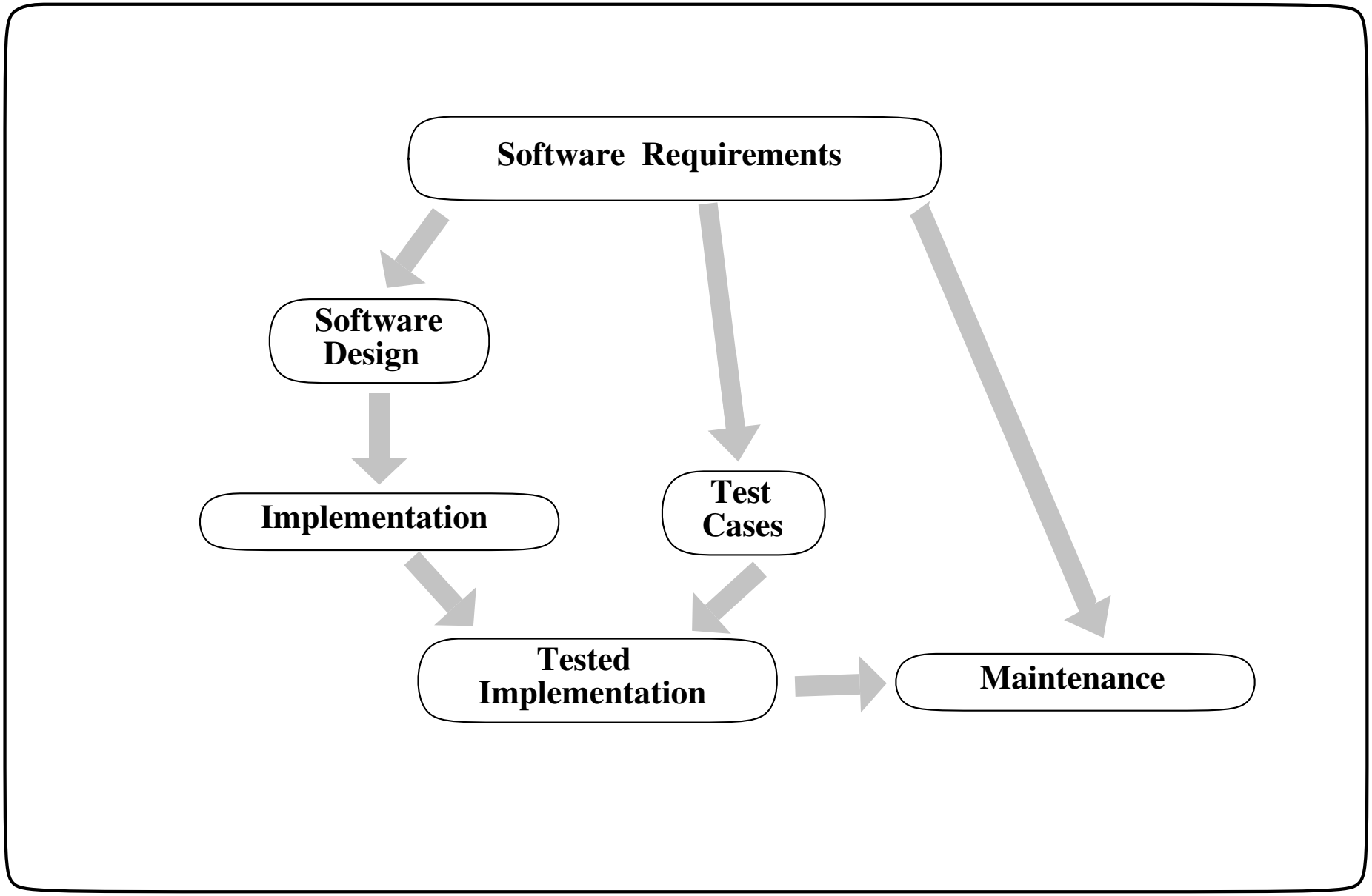
# Verification of Consistency Between Concurrent Program Designs And Their Requirements

Marsha Chechik John Gannon

Department of Computer Science
University of Maryland

June 20, 1996

# Software Development Process



## Formal Requirements Notations

– Efforts have been made to assure that requirements are easy to review, scalable, complete and consistent

- Software Cost Reduction (SCR) notation [NRL–developed]
  – the A–7E aircraft
- Requirements State Machine Language (RSML) [Levenson et al]
  – the FAA's Traffic Collision and Avoidance system
- Statecharts [Harel]
  – the Lavi fighter's avionics system

## Goal of Our Project

"To aid in acceptance and use of precise documentation of requirements."

- **Amortize its creation cost across several analysis activities**

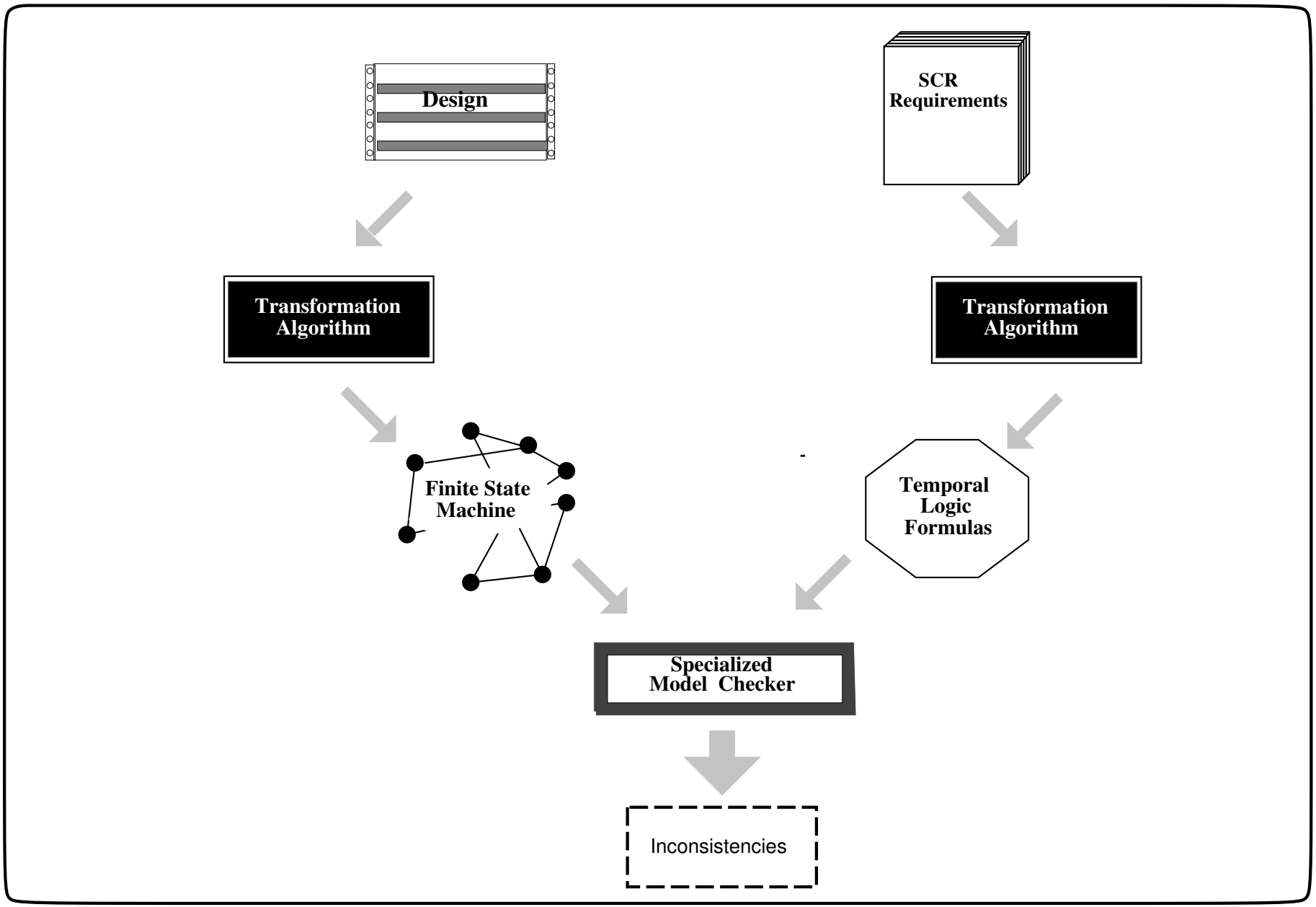
  Verify that requirements are consistent, i.e., that they satisfy global safety properties

  Verify that designs are consistent with requirements

- **Reduce analysis effort through automation**

  Build or use existing automatic tools to perform this analysis

# Designs |=? Requirements



## Talk Outline

1. SCR requirements
   a. Overview
   b. Concurrent extensions
   c. Notion of consistency with SCR requirements
2. Program Design Language
3. Building the model
   a. Dataflow analysis
   b. Removing unreachable states
   c. Building Finite-State Machine
4. Verification of Properties
   a. Optimism and Pessimism
   b. Computing Events
   c. Verifying automatically-generated properties
   d. Verifying user-specified properties
5. Conclusion

## SCR Requirements

- *state* – mapping of variables to values
- *modeclass* – set of states, called *modes*, which partition the monitored environment's state space

  one mode is designated as the *initial* mode
- a *condition* is a predicate on monitored or modeclass variables

  *Running, ButtonPressed, In(System=FailureMode)*
- an *event* is a point in time when a condition's value changes:

  @T( *ButtonPressed* )

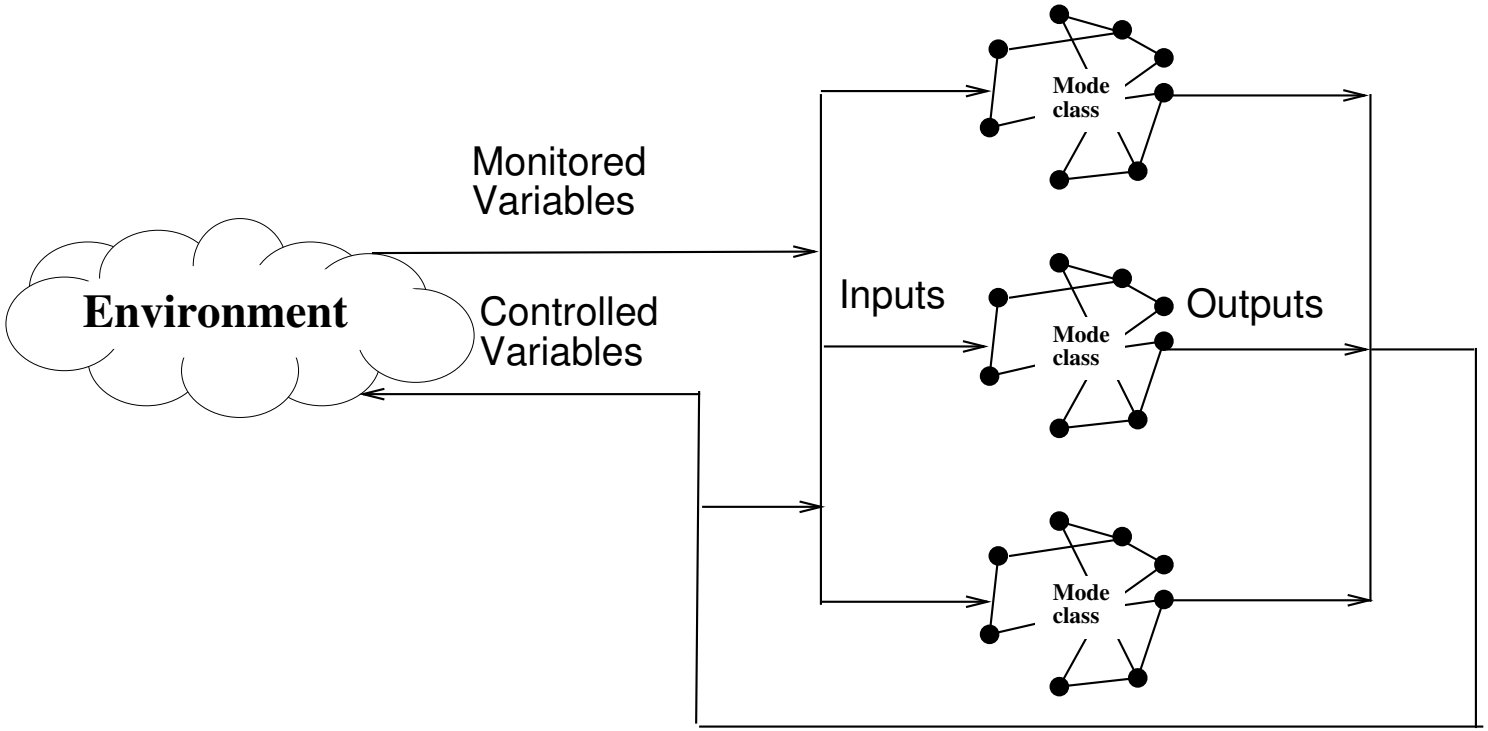
  @F( *Running* ) WHEN [ *ButtonPressed* ]

**Assumption:**

- monitored and controlled variables are boolean
- modeclass variables are enumerated types

## Modeling Concurrency

- each modeclass constitutes a separate *process*




- *communication variables* – output of one process and input to another
  They have the same name in both processes

- no two events can occur at the same time

## Example – Mutual Exclusion System

- Two identical systems (SubSys1 and SubSys2) with critical sections
- Separate process (Manager) gives out permissions to the other processes
- Communication is done by message−passing
- Subsystems request and return grants

Communication variables:

- S1Wait, S2Wait – which system is waiting for grant
- S1Done, S2Done – which system is returning grant
- Grant1, Grant2 – which subsystem gets permission to proceed

# Transition Tables for Mutual–Exclusion System

## Mode Transition Table for SubSys1

| Current Mode | *Time* | *In(Manager=Grant1)* | *Done* | New Mode |
|---|---|---|---|---|
| S1Done | @T | –– | –– | S1Wait |
| S1Wait | –– | @T | –– | S1RecGrant |
| S1RecGrant | –– | –– | @T | S1Done |

Initial: S1Done if ~Time & ~Done

## Mode Transition Table for Manager

| Current Mode | *In(SubSys1=S1Wait)* | *In(SubSys2=S2Wait)* | *In(SubSys1=S1Done)* | *In(SubSys2=S2Done)* | New Mode |
|---|---|---|---|---|---|
| Stable | @T | f | –– | –– | Grant1 |
| | f | @T | –– | –– | Grant2 |
| Grant1 | –– | –– | @T | –– | Stable |
| Grant2 | –– | –– | –– | @T | Stable |

Initial: Stable if In(SubSys1=S1Done) & In(SubSys2=S2Done)

## System Goals

- specified by user, in addition to transition tables
- can be local to one process or global to the whole system

  – reach(f) – the system can reach a state where f holds

  **Examples:**

```
reach(Manager=Grant1)
reach(Manager=Grant2)
```

  – inv(f) – f holds in every node of the system

  **Example (Mutual Exclusion):**

```
inv(~(SubSys1=S1RecGrant) & (SubSys2=S2RecGrant))
```

# Notion of Consistency

## Mode Transition Table for Manager

| Current Mode | *In(SubSys1=S1Wait)* | *In(SubSys2=S2Wait)* | *In(SubSys1=S1Done)* | *In(SubSys2=S2Done)* | New Mode |
|---|---|---|---|---|---|
| Stable | @T | f | –– | –– | Grant1 |
| | f | @T | –– | –– | Grant2 |
| Grant1 | –– | –– | @T | –– | Stable |
| Grant2 | –– | –– | –– | @T | Stable |

Initial: Stable if In(SubSys1=S1Done) & In(SubSys2=S2Done)

## Generated Properties:

– All specified transitions are implemented (ALT)

$$P1=EF(Manager=Grant1 \wedge @T(In(SubSys1=S1Done)) \wedge EX(Manager=Stable))$$

$$P2=EF(Manager=Grant2 \wedge @T(In(SubSys2=S2Done)) \wedge EX(Manager=Stable))$$

– No unspecified transitions are implemented (OLT)

$$P3=AG(Manager=Stable \vee (EX(Manager=Stable) -> (Manager=Stable) \vee (Manager=Grant1 \wedge @T(In(SubSys1=S1Done))) \vee (Manager=Grant2 \wedge @T(In(SubSys2=S2Done))))))$$

## Program Design Language (PDL)

– consist of control–flow constructs and annotations reflecting communication information and changes to values of requirements variables

– annotations start with @@

- **Initial annotation**

  indicates the starting state for each modeclass

- **Update annotation**

  assigns values to variables identifying points at which program changes its state

- **Assert annotation**

  captures designer's hypotheses that variables have particular values in the current system state

- **Read annotation**

  indicates a state in which a process receives a message

- **Write annotation**

  indicates a state in which a process sends a message

## Semantics of Communication Constructs

**Motivation:**

- Processes should be able to react to changes of states of other processes
- These events should be "fresh"

**Solution – shared buffers:**

A:Read reads from buffer A
A:Write writes to buffer A

**Semantics of Read:**

| Process: | Buffer: |
| --- | --- |
| envoke A.Read() routine | wait until there is info in buffer |
| wait until it returns info | remove it |
| | return values of relevant values |

**Semantics of Write:**

| Process: | Buffer: |
| --- | --- |
| envoke A.Write() routine, giving it values of vars to be stored | wait until buffer is empty |
| | store values of variables |
| wait until it returns | return |

# Detailed Design of MES

## Manager:

```
1:{ @@ INITIAL Manager(Stable);
2:  while(1) {
3:    if (manager is free) {
4:      @@ Assert Manager(Stable);
5:      @@ A: READ;
6:      /* someone requests grant */
7:      if (this is system two) {
8:        @@ ASSERT In(SubSys2=S2Wait);
9:        @@ UPDATE Manager(Grant2);
10:     }
11:     else {
12:       @@ ASSERT In(SubSys1=S1Wait);
13:       @@ UPDATE Manager(Grant1);
14:     }
15:     /* tell processes */
16:     @@ B: WRITE;
17:   }
18:   else{
19:     /* one subsystem is done */
20:     @@ Assert ~Manager(Stable);
21:     @@ C: Read;
22:     @@ UPDATE Manager(Stable);
23:   }
24: }
25:}
```

## SubSys1:

```
1: { @@ INITIAL SubSys1(S1Done) &
        ~Time & ~Done;
2:   while (1) {
3:     /* wait for some time */
4:     @@ UPDATE Time;
5:     @@ UPDATE SubSys1(S1Wait)
          & ~Done;
6:     @@ A: WRITE;
7:     @@ B: READ;
8:     /* assume grant is given */
9:     @@ ASSERT In(Manager=Grant1);
10:    @@ UPDATE SubSys1(S1RecGrant)
          & ~Done;
11:    /* work for a while */
12:    @@ UPDATE Done;
13:    @@ UPDATE SubSys1(S1Done);
14:    @@ C: WRITE;
15:  }
16:}
```

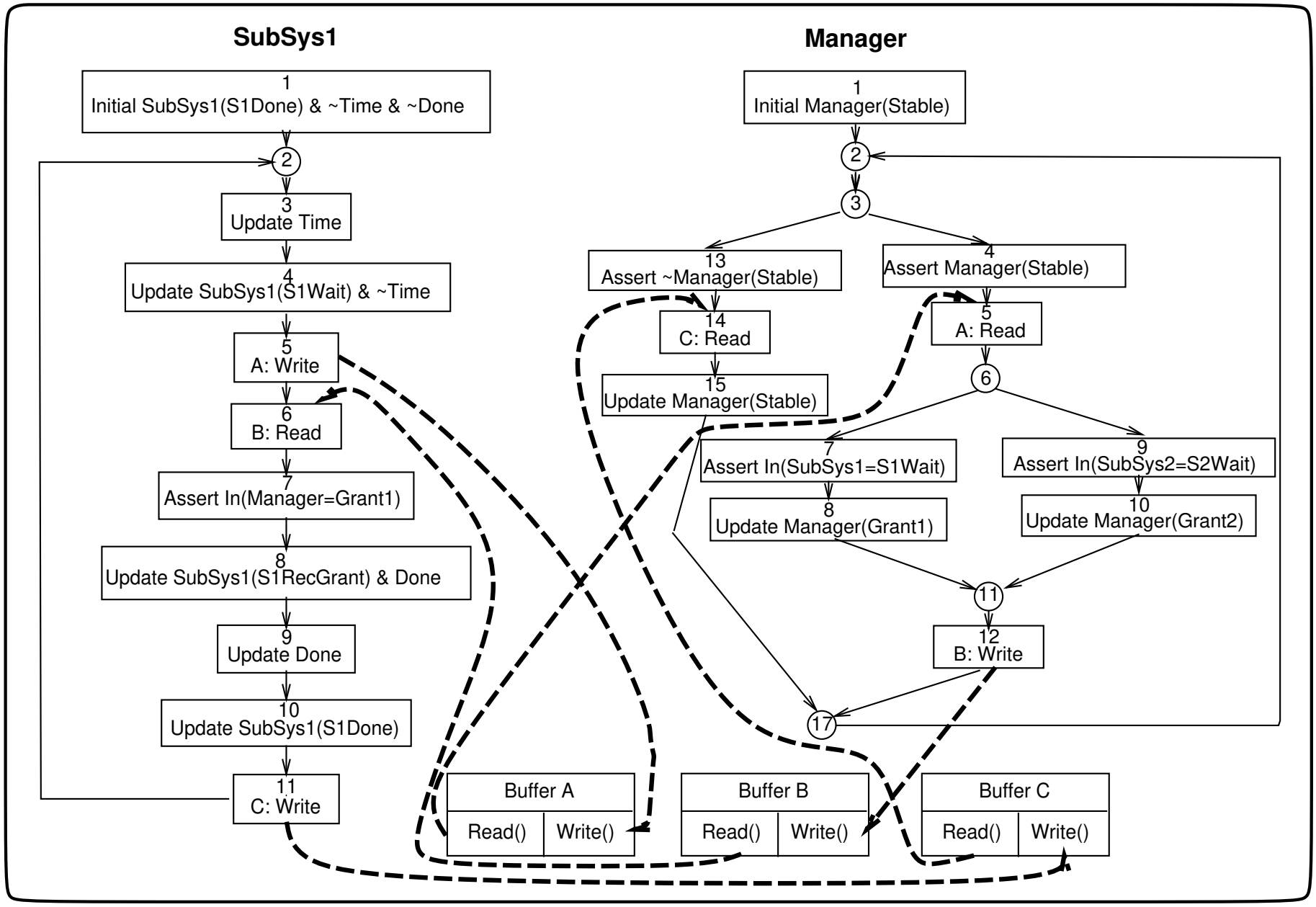
## Building Concurrent Design Flow Graph (CDFG)

- CDFG is a finite set of nodes corresponding to splits, joins, annotations, and Read() and Write() routines of buffers
- nodes are connected by sequential and synchronization edges
- propagate variable information throughout the graph via dataflow analysis techniques, computing set–based approximation of attainable values of requirements variables
- for each state, compute reaching values (RV) and condition (cond) info

  RV of a variable at a node is a set of values that it may attain if control reaches this node

  Cond of a variable at a node is a set of values that it "must" attain if control reaches this node

# CDFG of Mutual–Exclusion System



## Goals of Propagating Information

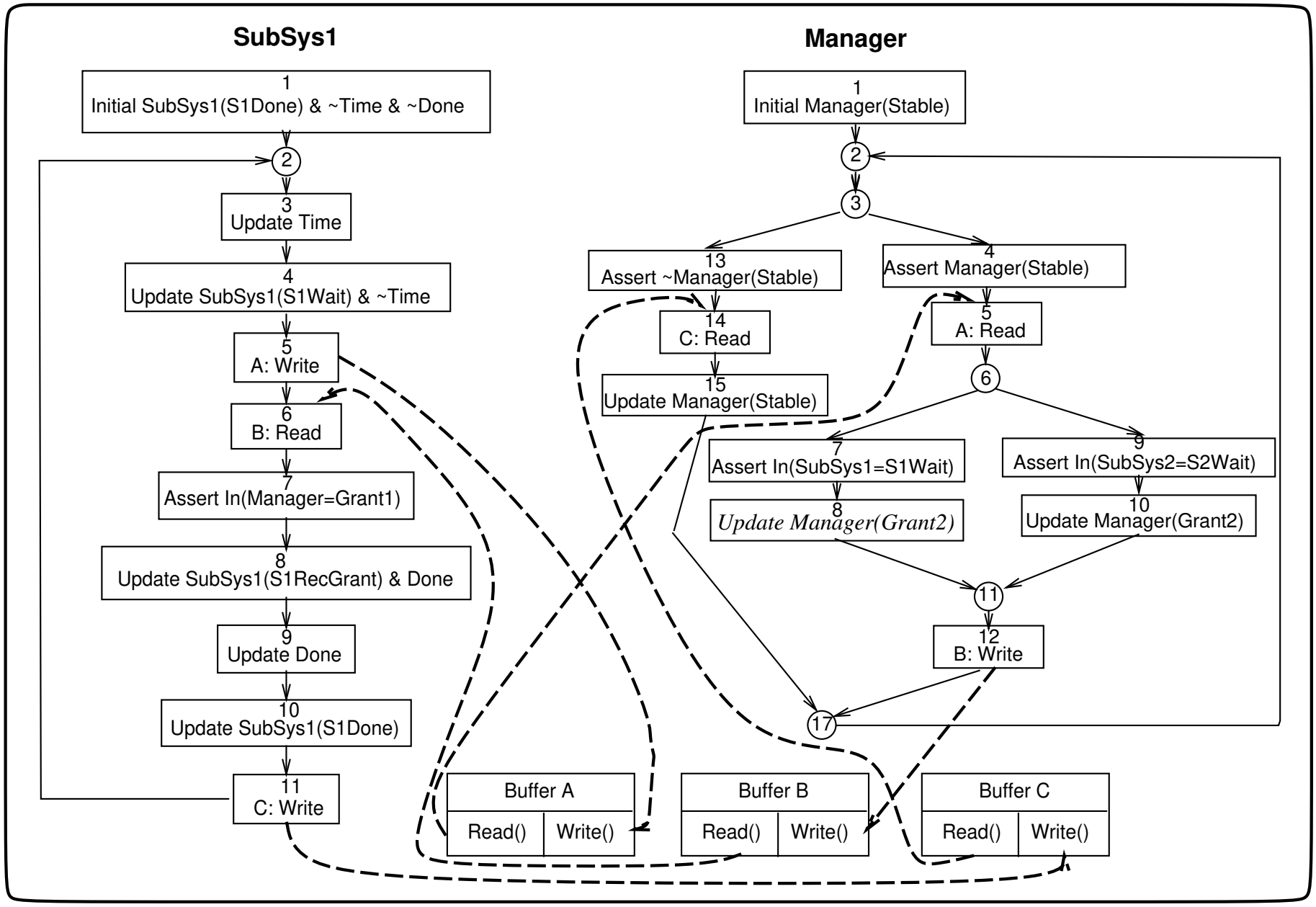
- A state n is unreachable if

$$\text{Cond}(n) \not\sqsupseteq \text{RV}(n)$$

- Find unreachable states and eliminate values they contribute to state space from consideration

- Initially, assume that all states are reachable

# CDFG of Mutual–Exclusion System

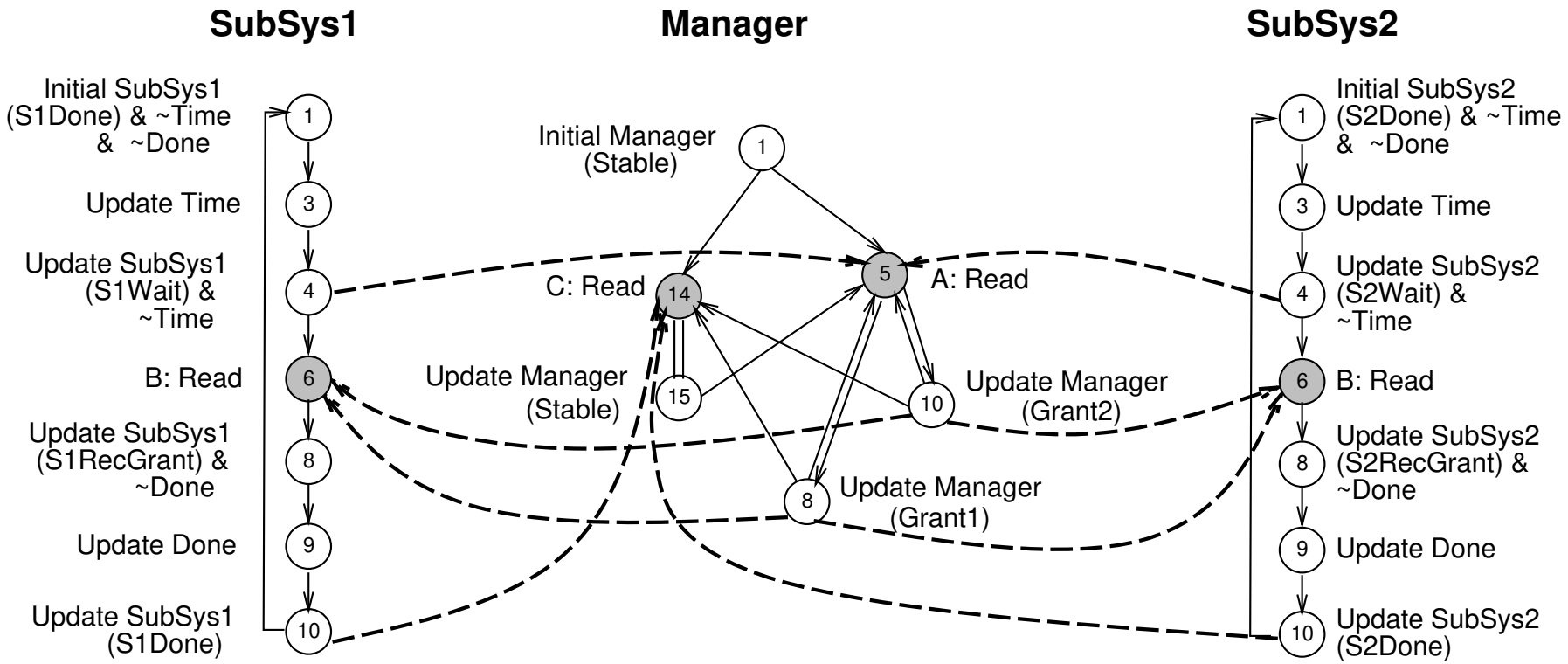


## Finite–State Machine

– Remove unreachable nodes, buffers and nodes not corresponding to Initial, Update and Read annotations

– For each remaining node, compute

$$\text{Info}(n) = \text{RV}(n) \sqcap \text{Cond}(n)$$



## Property Verification

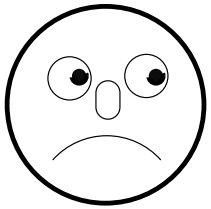
Static analysis complicates this task:

- Static analysis results in sets of values for each variable
- We cannot determine which paths are feasible

So, we divide properties:

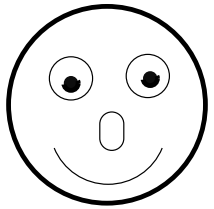
- pessimistic properties are quantified over all system states.

When analysis reports no violation, then it holds in design.



- optimistic properties are quantified over some system states.

When analysis reports a violation, it does not hold in design.

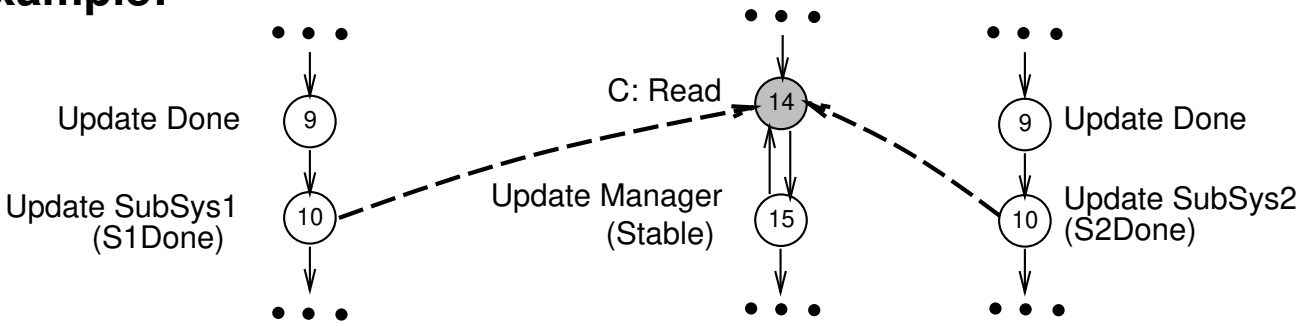


# Computing Events

For each predecessor (follow synchronization edges if necessary)

- Triggering condition is computed from information generated and killed at this node
- When condition is this node's Info set
  (in case of synchronization, it is unioned with Info at Read node)

**Example:**




Calculated events and transitions:

from Grant1 to Stable and from Grant2 to Stable:

@T(SubSys1=S1Done) WHEN [In(SubSys1=S1Wait)={False} /\ Manager={Grant1,Grant2}
/\ In(Manager=Grant1)={True,False} /\ In(Manager=Grant2)={True,False}]

@T(SubSys2=S2Done) WHEN [In(SubSys2=S2Wait)={False} /\ Manager={Grant1,Grant2}
/\ In(Manager=Grant1)={True,False} /\ In(Manager=Grant2)={True,False}]

## Verification of Automatically-Generated Properties

- ALT properties are verified optimistically
- Satisfied properties are marked during FSM traversal
- Afterwards, unmarked properties are reported

**Example ALT properties:**

$$P1 = EF(Manager=Grant1 \wedge @T(In(SubSys1=S1Done)) \wedge EX(Manager=Stable))$$

$$P2 = EF(Manager=Grant2 \wedge @T(In(SubSys2=S2Done)) \wedge EX(Manager=Stable))$$

- OLT properties are verified pessimistically, in one traversal of FSM
- Errors are reported as discovered

**Example OLT property:**

$$P3 = AG(Manager=Stable \vee (EX(Manager=Stable) \rightarrow (Manager=Stable \vee (Manager=Grant1 \wedge @T(In(SubSys1=S1Done))) \vee (Manager=Grant2 \wedge @T(In(SubSys2=S2Done))))))$$

## Output of Analysis

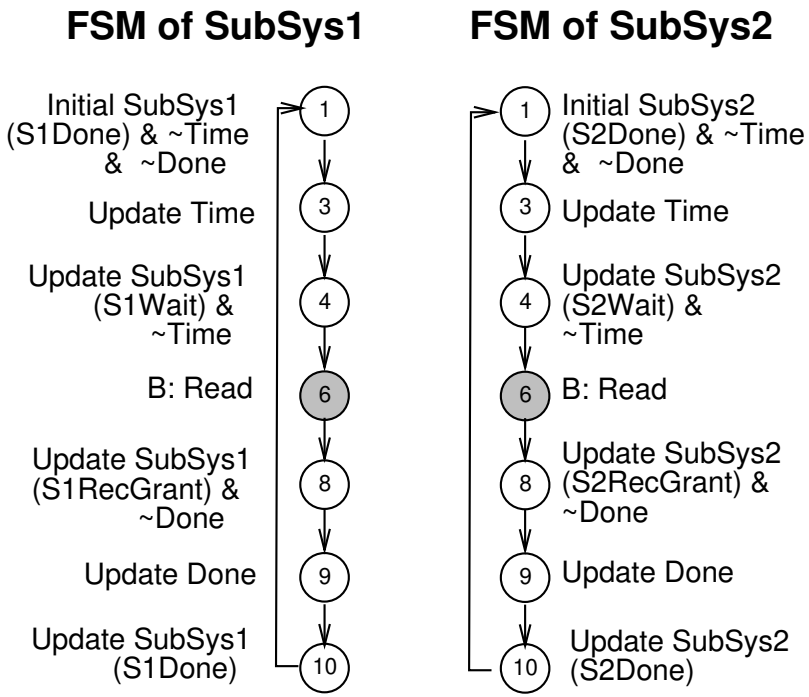
```
Found an event on line 13, process SubSys1
The event is @T(In(SubSys1=S1Done)) WHEN [In(SubSys1=S1Wait)=
  {False} & Manager={Grant1,Grant2} & In(Manager=Grant1)={True,
  False} & In(Manager=Grant2)={True,False}]

Found rule:
  Grant1    @T(In(SubSys1=S1Done))    Stable

Error on line 22, process Manager, in modeclass MANAGER:
  no transition from mode(s) {Grant2} to mode(s) {Stable}

Found an event on line 13, process SubSys2
The event is @T(In(SubSys2=S2Done)) WHEN [In(SubSys2=S2Wait)=
  {False} & Manager={Grant1,Grant2} & In(Manager=Grant1)={True,
  False} & In(Manager=Grant2)={True,False}]

Found rule:
  Grant2    @T(In(SubSys2=S2Done))    Stable

Error on line 22, process Manager, in modeclass MANAGER:
  no transition from mode(s) {Grant1} to mode(s) {Stable}
```

# Verification of User–Defined Global Properties

- Check reach properties optimistically, inv properties pessimistically
- Check each property on combinations of states of each process's FSM

**Example:**

```
inv(~(SubSys1=S1RecGrant & SubSys2=S2RecGrant))
```




Violations are reported in states (8, 8), (8, 9), (9, 8), (9, 9)

## Conclusion

- Extended SCR notation to reason about concurrency
- Defined a notion of consistency between SCR-style requirements and a detailed designs
- Created techniques to propagate values of variables throughout the system without having to build reachability graph
- Created model-checking approach allowing verification of most properties without building reachability graph
- Technique works well for process with complicated control-flow and not extensive communications

## Limitations

- Buffers are not the most effective means of modeling synchronization
- Need to be able to verify liveness and fairness properties