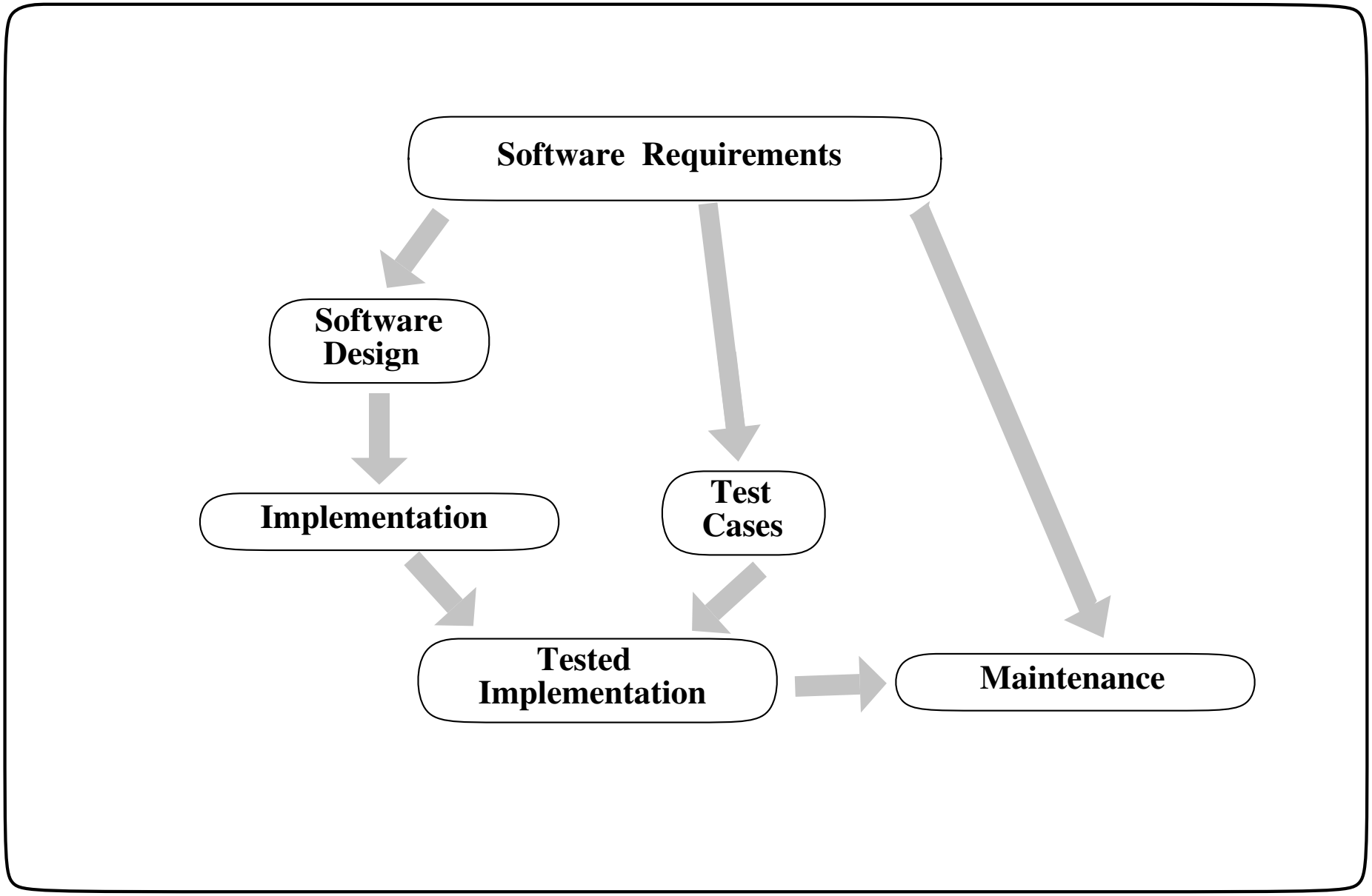
Automatic Analysis of Consistency Between Requirements And Designs

Marsha Chechik

Department of Computer Science
University of Maryland

Software Development Process



High Assurance Systems

"High assurance computer systems are systems where compelling evidence is required that the system delivers its services in manner that satisfies certain critical properties."

Examples:

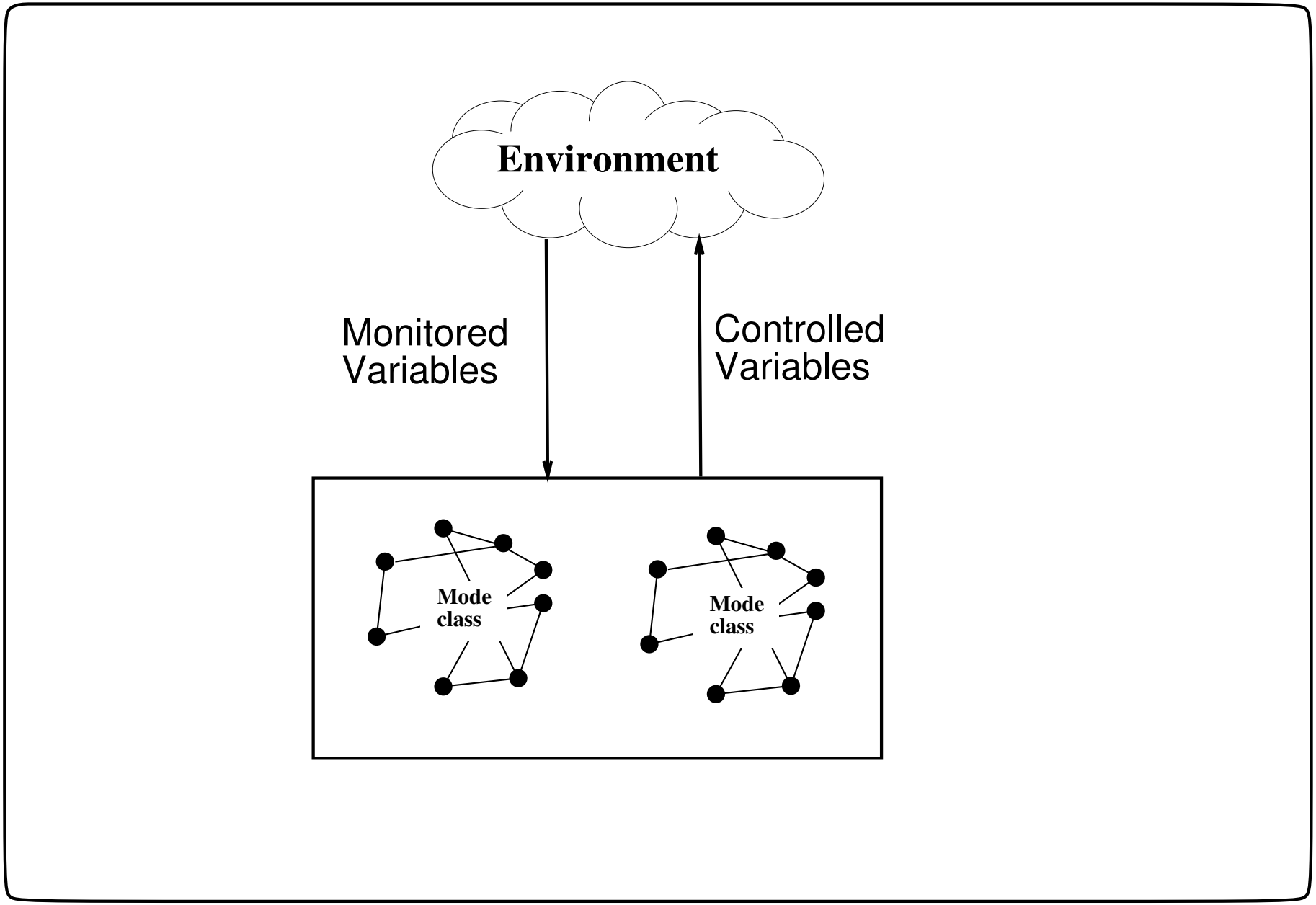
- Military command and control systems
- Nuclear power plants
- Telephone networks
- Medical systems (e.g., patient monitoring system)
- Air traffic control systems
- Flight control systems

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

SCR overview



SCR Requirements

Conditions and Events

- a *condition* is a state variable

  Running, ButtonPressed

- an *event* is a point in time when a condition's value changes:

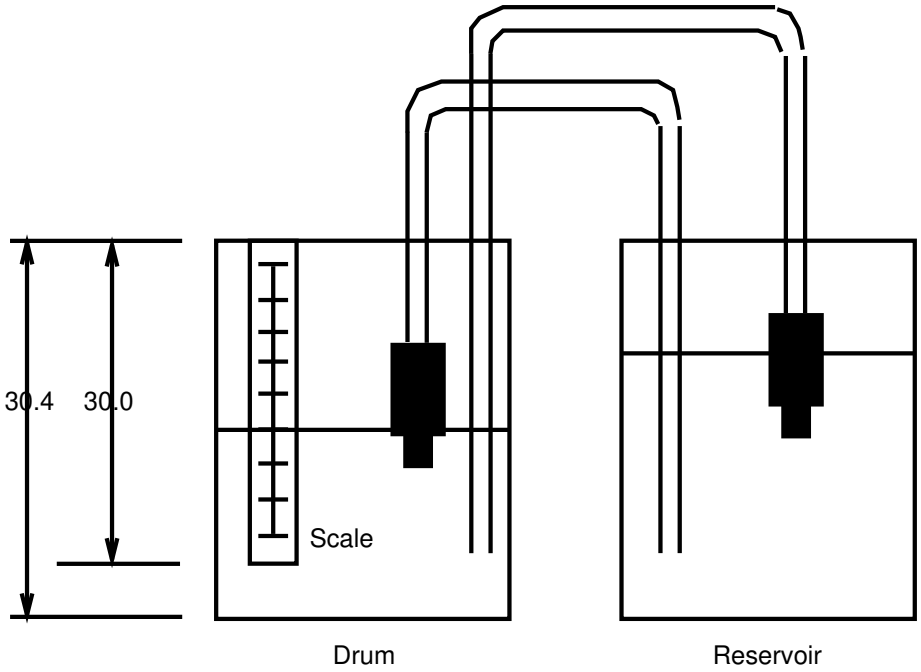
  @T(*ButtonPressed*)

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

Modes and Transitions

- a *mode* is a set of system states
- a *mode transition* between modes occurs as a result of events
- a *mode class* is a set of modes (union must cover state space)
- a system is in exactly one mode of each mode class at all times

Water–Level Monitoring System



Modes

STANDBY, OPERATING, SHUTDOWN, TEST

Selected monitored variables

WithinLimits	Low level < Water level < High level
InsideHysRange	(Low level + 0.5cm) < Water level < (High level – 0.5cm)
SlfTestPressed	Self–test button is being pressed
SlfTestPressed500	Self–test button has been pressed for 500ms or more

Water–Level Monitoring System

Requirements

2 modeclasses (NORMAL and FAILURE) with 7 total modes

18 monitored variables

7 controlled variables

Mode Transition Table for Modeclass NORMAL

Current Mode	*Inside HysRange*	*Within Limits*	*SlfTest Pressed*	*SlfTest Pressed500*	*InTest 14000*	*Reset Pressed3000*	*ShutDown Time200*	New Mode
STANDBY	true	–	–	–	–	@T	–	**OPERATING**
	–	–	–	@T	–	–	–	**TEST**
OPERATING	–	@F	false	–	–	–	–	**SHUTDOWN**
	–	–	true	@T	–	–	–	**TEST**
SHUTDOWN	@T	–	false	–	–	–	false	**OPERATING**
	–	–	false	–	–	–	@T	**STANDBY**
	–	–	–	@T	–	–	–	**TEST**
TEST	–	–	–	–	@T	–	–	**STANDBY**

Initial: **STANDBY if** *(~SlfTestPressed & ~InTest14000 & ~ResetPressed3000 & ~ShutDownTime200)*

Code Fragment from Case-Study

```
while(TRUE) {
   ..
   GET_MODE(OperatingClass, OPCM);
   WATER_LEVEL(&LVLCM);
   INLIM=((LVLCM < (float)DPAR_HWL) && (LVLCM > (float)DPAR_LWL));
   GET_TIME(SHUTER, &SHUTTM);
   if (ERR_VAR_F(CLK_ERR) == CLK_OVERFLOW) {
     RESET_TIMER(SHUTER);
     SHUTTM = 0;
   }
   if (MODE_EQ(OPCM, Shutdown) && INLIM && (SHUTTM < DPAR_SLT)
     && !PRESSED(SlfTst)){
     SET_MODE (OperatingClass, Operating);
  }
  ...
  GET_MODE(OperatingClass, OPCM);
  if ((!MODE_EQ(OPCM, Test))) && (PRESSED(SlfTst))) {
    SET_MODE(OperatingClass, Test);
    INIT_DRVS();
  }
  ...
}
```

Goal of Our Project

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

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

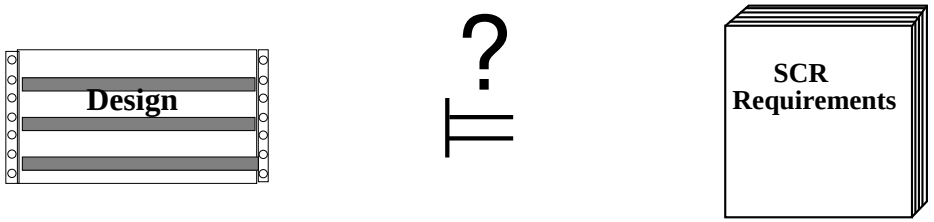
  Verify that requirements are consistent, i.e., that they satisfy global safety properties

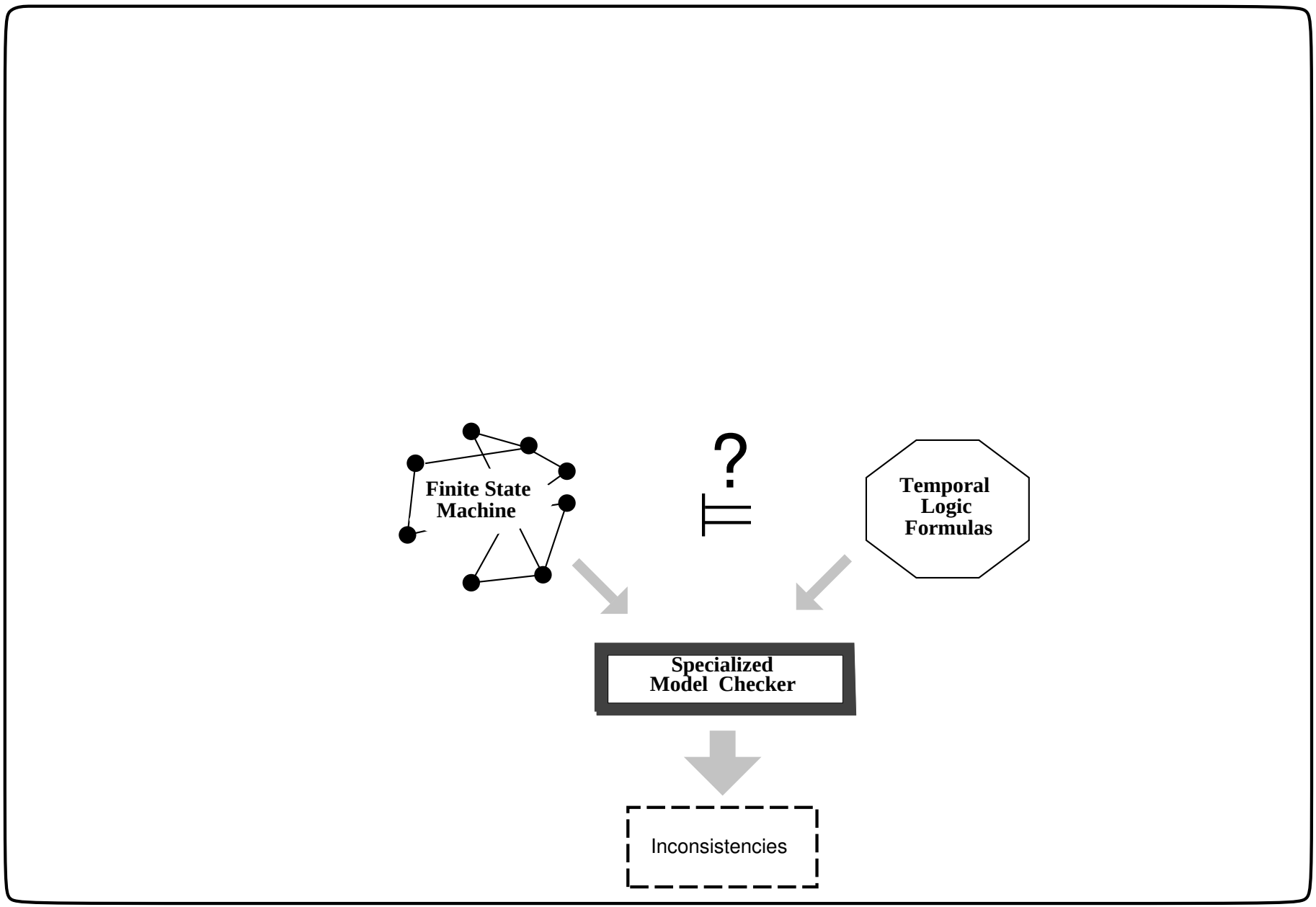
  Verify that designs are consistent with requirements

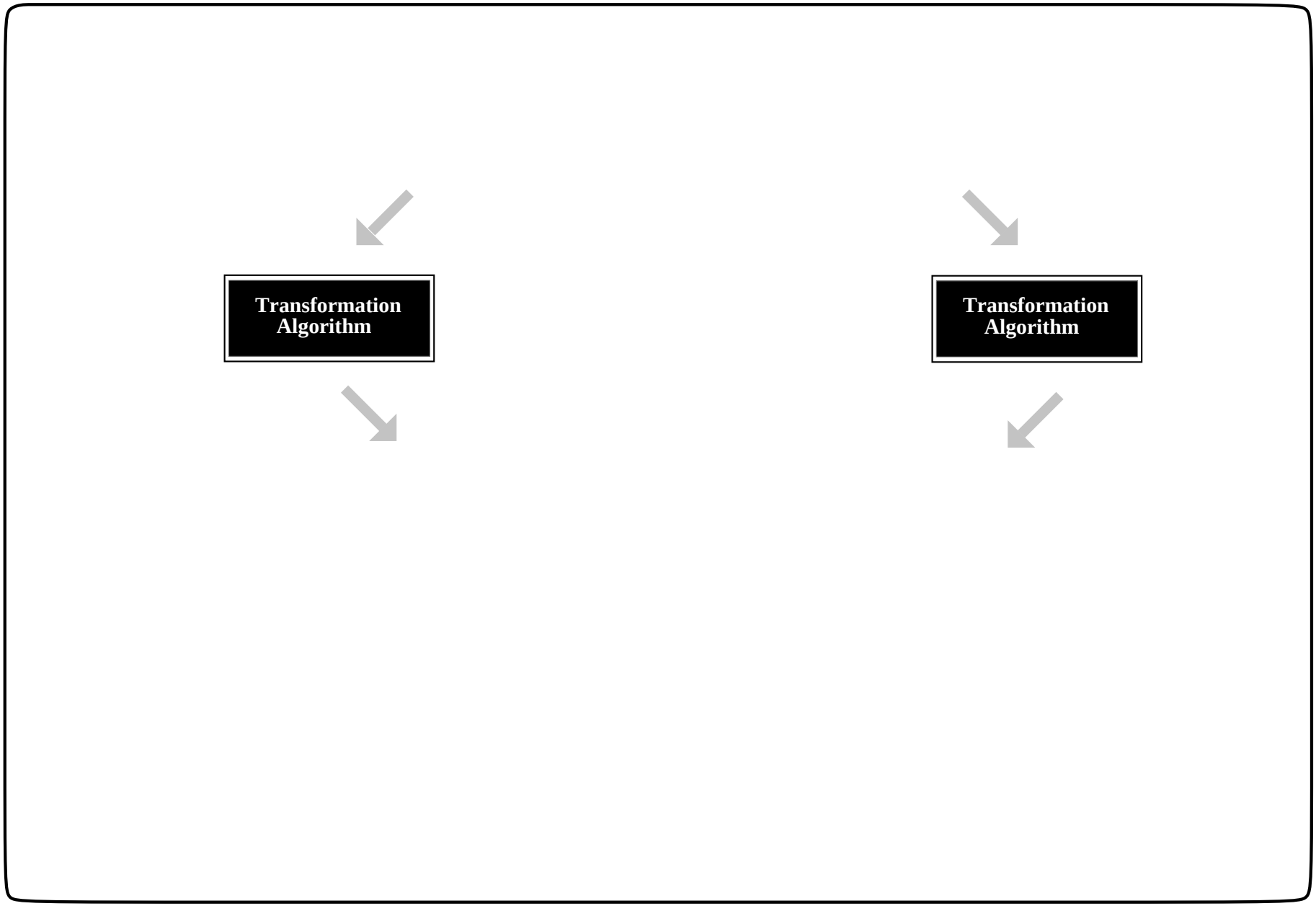
- **Reduce analysis effort through automation**

  Build or use existing automatic tools to perform this analysis

Overview




Finite State
Machine
?
Temporal
Logic
Formulas
Specialized
Model Checker
Inconsistencies


Transformation Algorithm
Transformation Algorithm

Simple Water–Level Monitoring System (SWLMS)

Mode Transition Table

Current Mode	*SwitchOn*	*PumpFail*	New Mode
Off	@T	f	Operating
	--	@T	Error
Operating	@F	f	Off
	--	@T	Error

Initial: Off if ~SwitchOn & ~PumpFail

Notion of Consistency

Mode Transition Table:

Current Mode	SwitchOn	PumpFail	New Mode
Off	@T	f	Operating
	–	@T	Error
Operating	@F	f	Off
	–	@T	Error

Initial: Off if (~SwitchOn & ~PumpFail)

Generated Properties:

- All specified transitions are implemented (ALT)

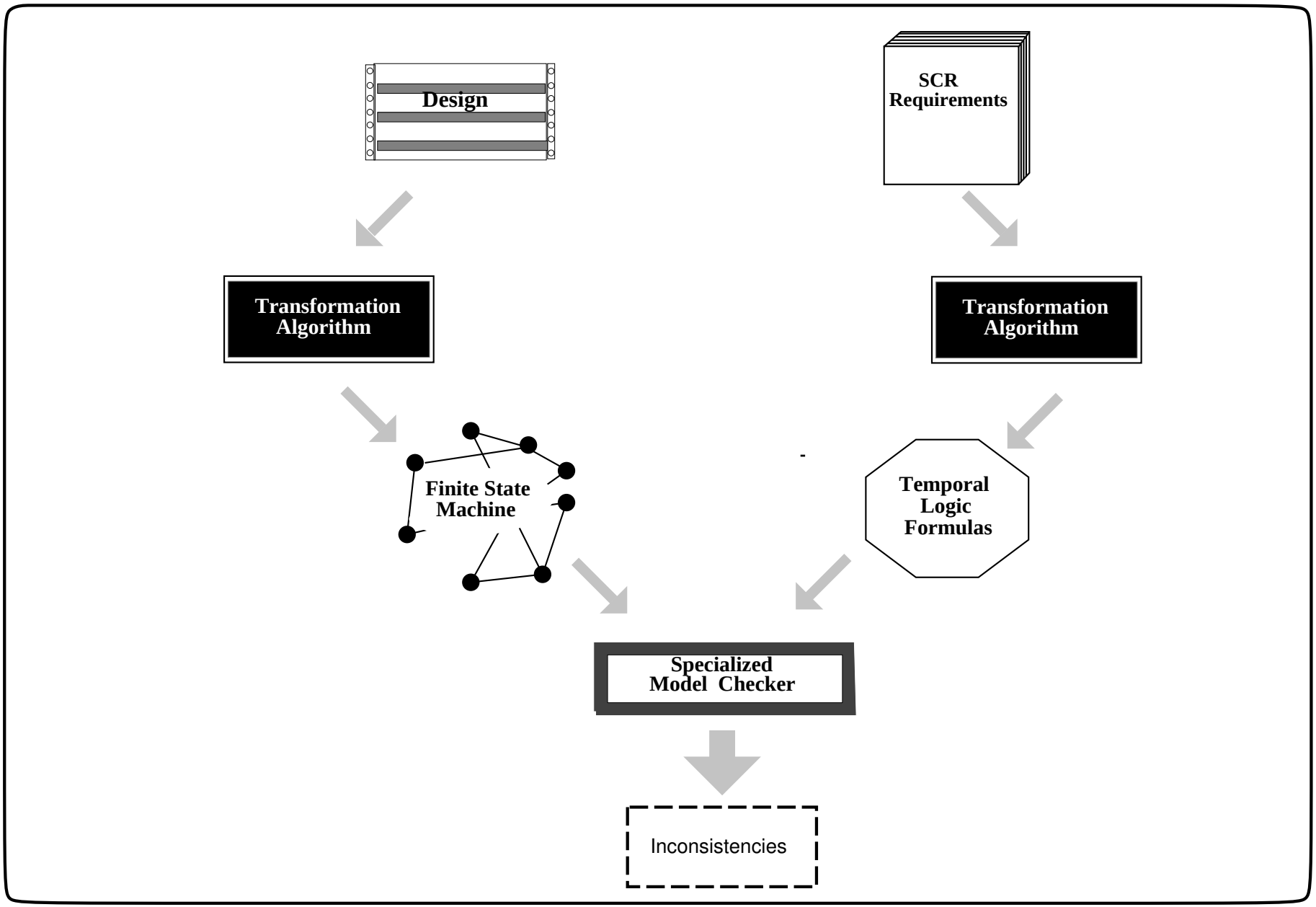
$$P_1 = EF(\text{Off} \;\&\; @T(\text{PumpFail}) \;\&\; EX(\text{Error}))$$

$$P_2 = EF(\text{Operating} \;\&\; @T(\text{PumpFail}) \;\&\; EX(\text{Error}))$$

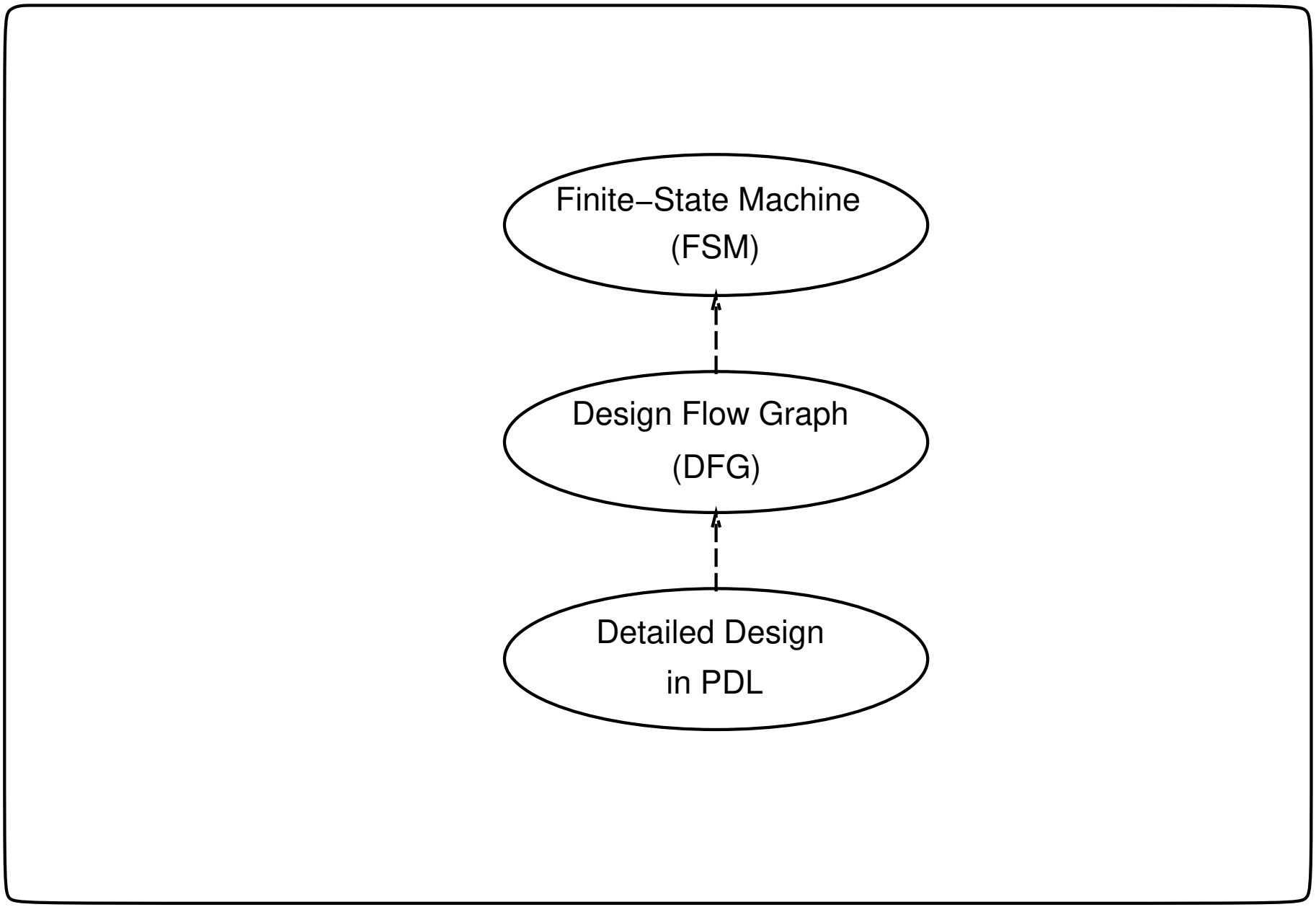
- No unspecified transitions are implemented (OLT)

$$P_3 = AG(EX(\text{Error}) \rightarrow (\text{Error} \;\vee\; (\text{Operating} \;\&\; @T(\text{PumpFail})) \;\vee\; (\text{Off} \;\&\; @T(\text{PumpFail}))))$$

Designs |=? Requirements



Layers of Abstraction



Program Design Language

Outer syntax: C control structures

Inner syntax: Comments describing values of requirements variables

Initial: assigns values to variables in start state

Update: assigns values to variables

Assert: checks and restricts values of variables

```
@@ update Top(PumpFail);  /* read device monitor */
if (pump_fail() && ...) {
      @@ assert PumpFail;
      @@ update Error;
}
else {
      @@ assert ~PumpFail;
      ...;
}
```

– capture local state changes rather than invariant properties

SWLMS Design

```
{...
  @@ Initial Off&~SwitchOn&~PumpOn
  &~TooHigh&~TooLow&~PumpFail;
  while (1) {
    /* assume no device failures */
    @@ Assert ~PumpFail;
    READ_SWITCH; /* read monitor */
    @@ Update Top(PumpFail);
    if (SWITCH_ON) && IN_OFF(Sys){
      @@ Assert SwitchOn & Off;
      @@ Update Operating;
    }
    else if (IN_OPERATING(Sys)) {
      @@ Assert Operating;
      whiel (1) {
        READ_SWITCH();
        @@ Update Top(SwitchOn);
        if (!SWITCH_ON()) {
          @@ Assert ~SwitchOn;
          @@ Update Off;
          break;
        }
        /* compute water level */
        @@ GET_WATER_LEVEL(&Water);
          @@ Update Top(TooHigh) &
             Top(TooLow);
          if (IS_HIGH(Water) ||
              IS_LOW(Water)) {
            @@ Assert TooHigh |
               TooLow;
            @@ Update PumpOn;
          }
          else {
            @@ Assert ~TooHigh &
               ~TooLow;
            @@ Update ~PumpOn;
          }
        }
      }
      READ_DEVICE()
      /* read device monitor */
      @@ Update Top(PumpFail);
      if (PUMP_FAIL()) {
        @@ Assert PumpFail;
        @@ Update Error & ~PumpOn;
      }
    }
    ...
  }
```

Building DFG

- Finite set of nodes corresponding to splits, joins and annotations
- Construct set–based approximation of attainable values of requirements variables
- Each state contains reaching values (RV) and condition (Cond) information
  - 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

Construction of CFG

- Compute gen and kill sets for each node containing an annotation

Example:

@@ Assert B
gena = {(B, {True})}
killa = {(B, {False})}

@@ Update M(M2) & D
gena = genu = {(M,{M2}), (D,{True})}
killa = killu = {(M,{M1,M3}), (D,{False})}

- Compute in and out sets using the following equations:

e2

e3

e1

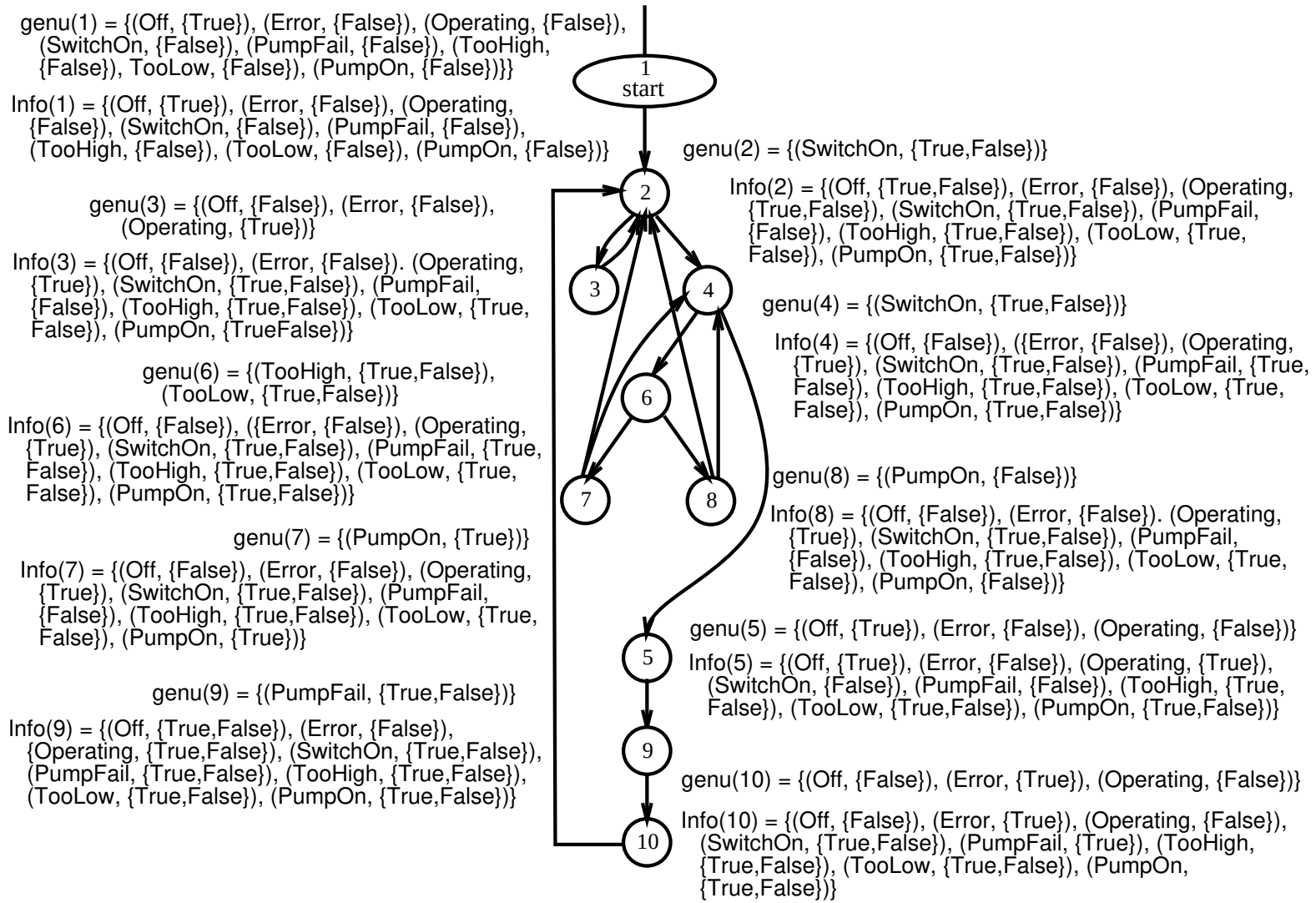
Building a Finite–State Abstraction

- Remove all nodes except those corresponding to Initals and Updates
- Remove any node n for which Cond(n) $\not\sqsubseteq$ RV(n)
- For all remaining nodes n:

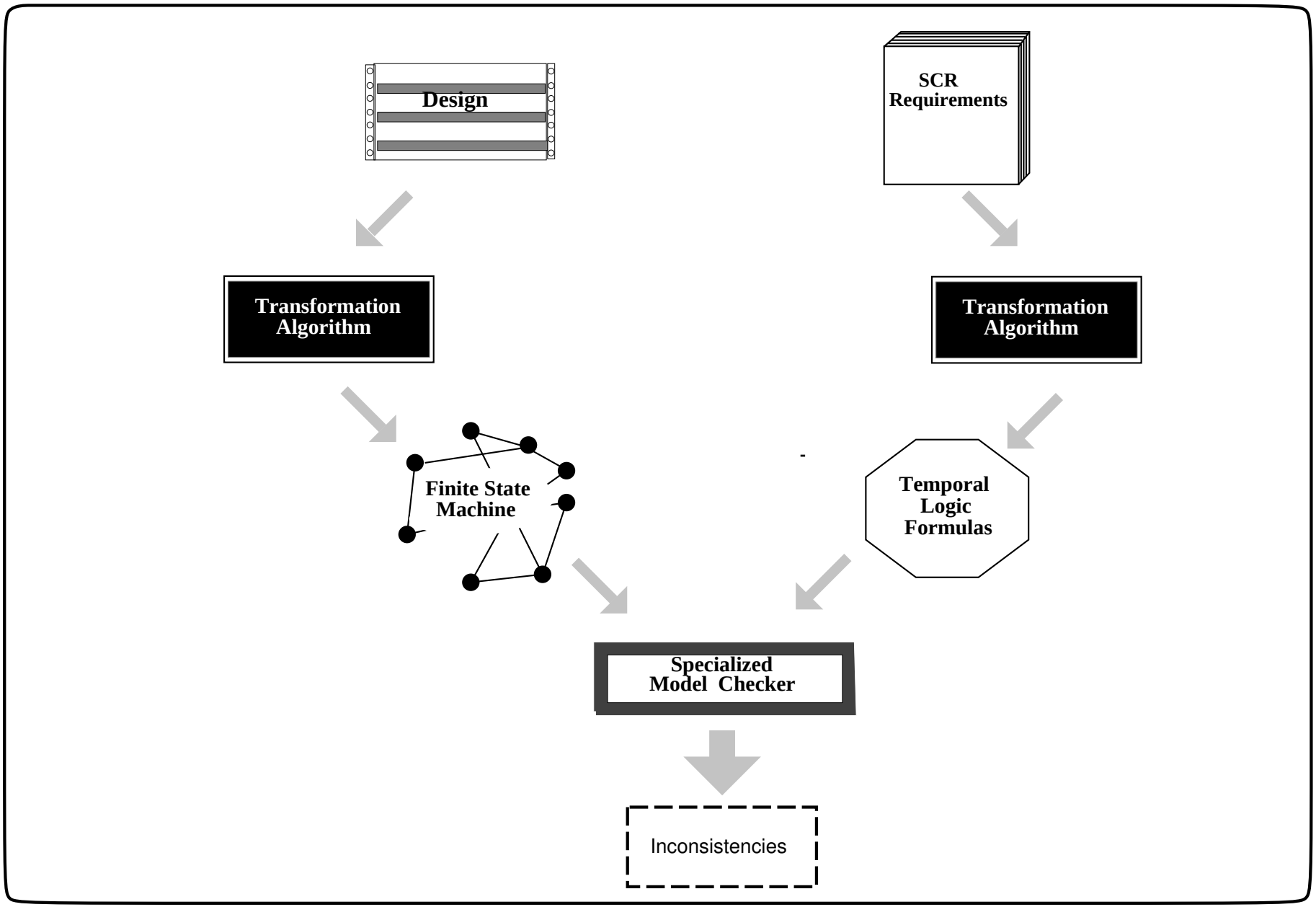
  Compute Info(n) = Cond(n) $\sqcap$ RV(n)

  Keep genu(n) and Info(n) and remove all other sets

Finite–State Abstraction of SWLMS



Designs |=? Requirements



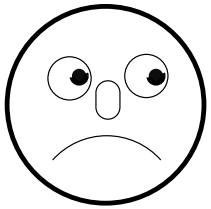
Property Verification

Static analysis complicates this task:

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

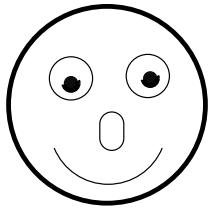
So, we divide properties:

- 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.

Property Verification

Values of atomic propositions

- P(S): P's value in state is a set
- Pf: Assertions about P in formulas are treated as assertions about sets of values

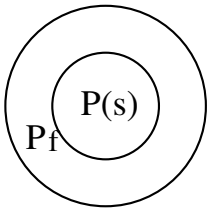
Examples:

SwitchOn is treated as (SwitchOn, {True})

~Error is treated as

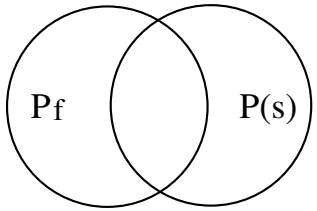
{(Error,{False}), (Operating,{True,False}, (Off,{True,False})}

Optimistic and pessimistic analysis



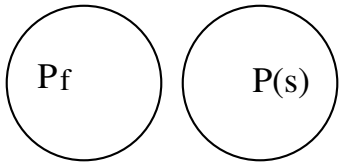
$s \models_o P$

$s \models_p P$



$s \models_o P$

$s \not\models_p P$



$s \not\models_o P$

$s \not\models_p P$

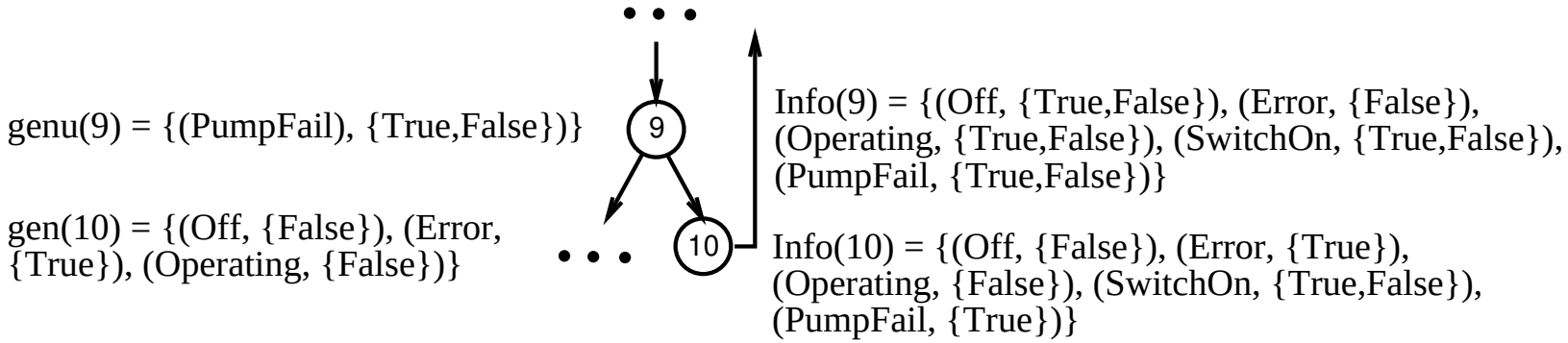
Computing Events

To calculate a node's event, for each predecessor

Triggering condition is its genu set.

WHEN condition is its Info set.

Example:



Calculated events and transitions:

from Off to Error and from Operating to Error:

@T(PumpFail) WHEN [SwitchOn = {True,False} & Off = {True,False} &

Operating = {True,False} & Error={False}]

Verification of ALT Properties

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

Example ALT properties:

P1 = EF(Off & @T(PumpFail) & EX(Error))

P2 = EF(Operating & @T(PumpFail) & EX(Error))

P1 is satisfied at node 9 via transition from Off to Error

P2 is satisfied at node 9 via transition from Operating to Error

Verification of OLT Properties

- OLT properties are verified pessimistically, in one traversal of FSM

Example OLT property:

P3 = AG(EX(Error) -> (Error \/ (Operating & @T(PumpFail))
\/ (Off & @T(PumpFail))))

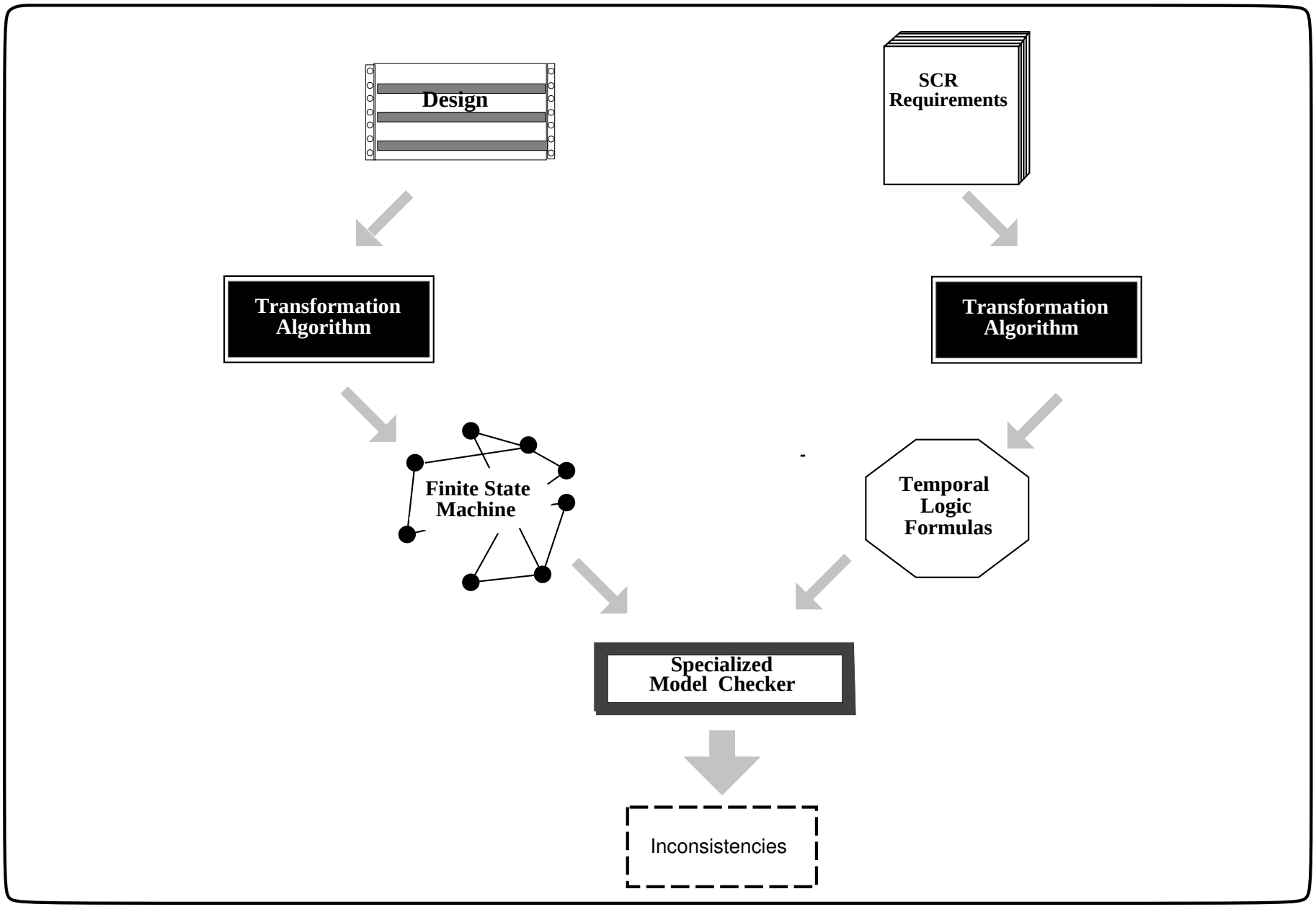
- Check this formula for those states where Error is False and in whose successors Error is True (node 9 in our example)

- Report errors as discovered

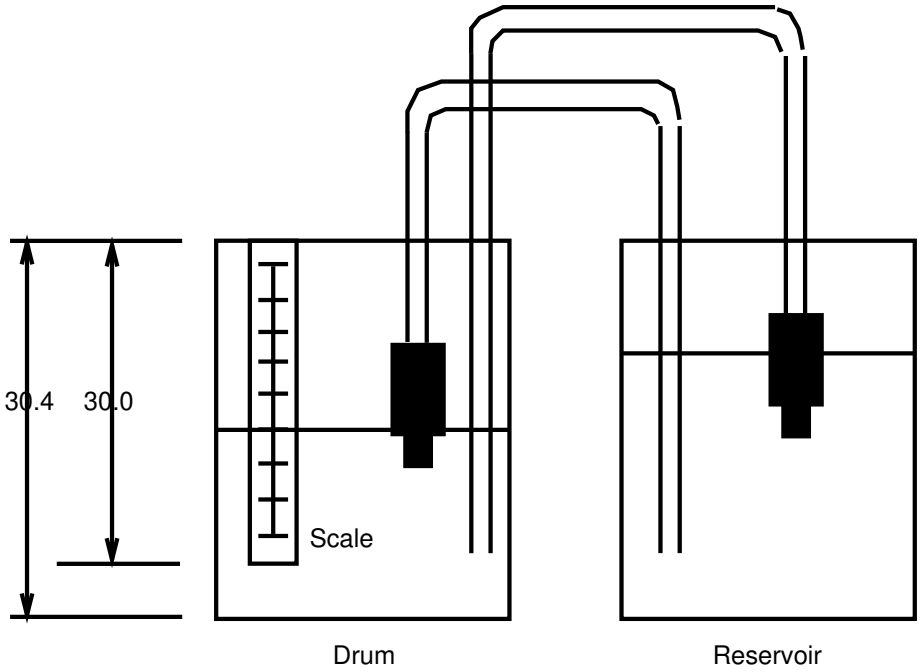
Complexity of the Analysis

• Construction of DFG – data–flow analysis of annotations. n – number of decision, join, and nodes with annotations	$O(n^2)$
• Construction of FSM	$O(n)$
• Property verification m – number of nodes in FSM. $m < n$	$O(m)$
All together	$O(n^2)$

Designs |=? Requirements



Water–Level Monitoring System



Modes

STANDBY, OPERATING, SHUTDOWN, TEST

Selected monitored variables

WithinLimits	Low level < Water level < High level
InsideHysRange	(Low level + 0.5cm) < Water level < (High level – 0.5cm)
SlfTestPressed	Self–test button is being pressed
SlfTestPressed500	Self–test button has been pressed for 500ms or more

Water–Level Monitoring System: Case Study

Requirements

2 mode classes (NORMAL and FAILURE) with 7 total modes

18 monitored and 7 controlled variables

Original Implementation

1000+ lines (54 functions) in FORTRAN and Assembler

by Van Schowen

Detailed Design (reverse–engineered)

250 lines (8 functions)

49 Update and 60 Assert annotations

Case Study: Water–Level Monitoring System – Results

Results of design verification:

- Wrong monitored variable was checked to enable mode transitions
- Watchdog timers were not set on control buttons
- No transitions to mode corresponding to complete system failure

Errors found:

- OLT properties violated 4 times (3 locations) for modeclasses and 53 times (7 locations) for controlled variables
- ALT properties violated 9 times for modeclasses and 28 times for controlled variables

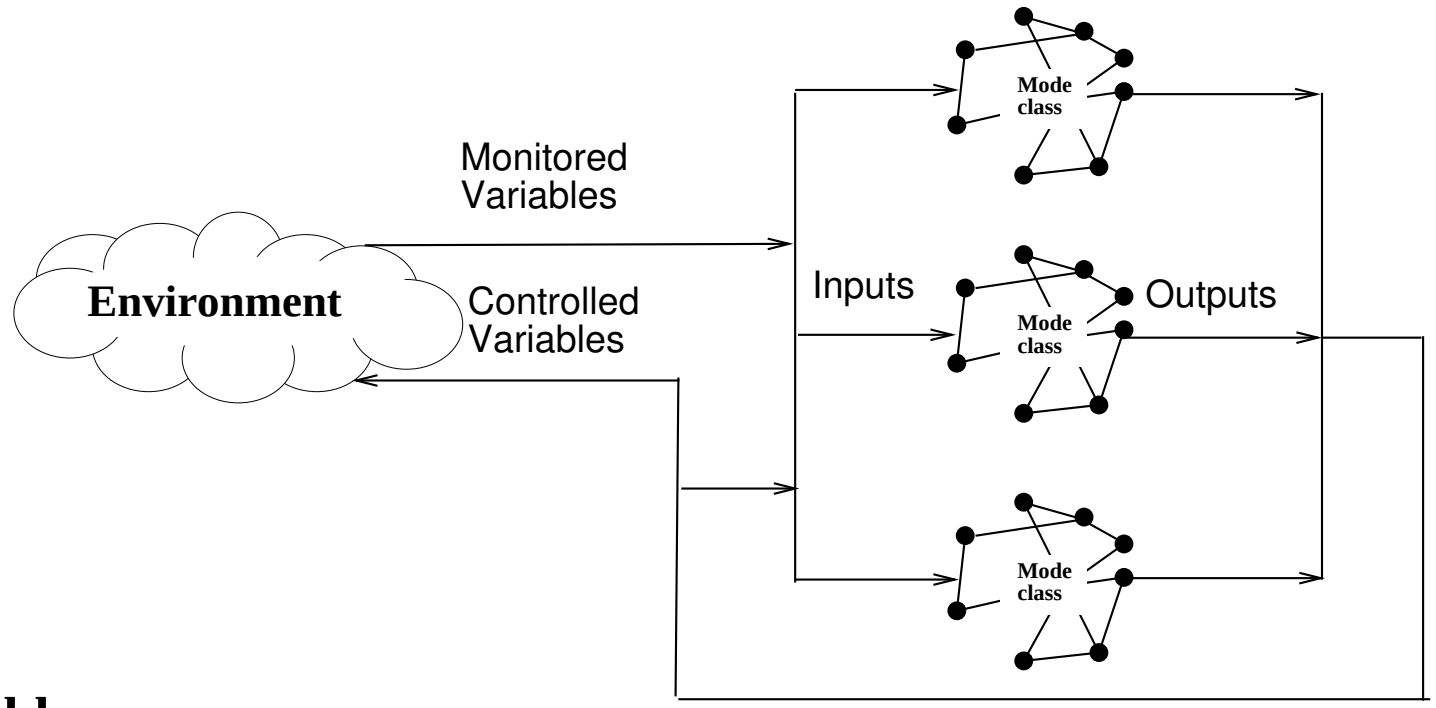
Verifying Concurrent Designs

– SCR requirements naturally model concurrently executing processes

Approach:

- Extend SCR notation to formalize the semantics of communicating mode classes
- Extend PDL to include communication information
- Extend data–flow techniques to enable computation of state information (including values of variables) without building a system's reachability graph
- Conduct a realistic case–study

Verifying Concurrent Designs (Cont.)



Problem:

Need to extend this notation further to be able to process identical processes. Need arrays of modes and modeclasses, as in StateCharts and RSML.

Extending PDL

- Read and Write annotations are used to identify points at which processes exchange information

- The exchange is done via shared buffers with blocking Reads and Writes

  @@ Buf: Read – a process reads values of its inputs from buffer Buf

  – Process sends a message to buffer's Read() routine, which waits until there is information in the buffer, removes it, and returns to the process

  @@ Buf: Write – a process stores values of its outputs in buffer Buf

  – Process sends a message to buffer's Write() routine, specifying values to be stored. Write() waits until the buffer is empty, stores the values and returns control to the process.

Evaluation of Semantics

Why this semantics?

- Reduces the chances for starvation and deadlock
- Ensures that all events are "fresh"
- Ensures that no events are missed

Why not?

- Might not be appropriate for all applications
- Other semantics would result in weaker analysis

So, what do we do?

- Case–studies
- Evaluate design and analysis tradeoffs among different semantics

Design of Manager and SubSys1

```
      /* 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:}
```

```
      /* subsystem 1 */

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:}
```

Conclusion

- Defined a notion of consistency between SCR-style requirements and a detailed design
- Defined a method for creating a FSM from a design
- Conducted a case-study which proved usefullness of the approach and its applicability to small but realistic systems

Research Directions

Analyzing consistency of concurrent designs

Defining a process of writing designs

- Define a set of heuristics which help to write correct designs
- Conduct case−studies using Analyzer as validation tool

Connecting annotations with source code

- Build implementation around PDL constructs
- Use representation mappings and dataflow analysis on implementation−level variables to determine if PDL constructs correspond to implementation

Generating test oracles from requirements

- Generate code to test annotations