Automatic Analysis of Consistency Between Requirements and Designs

Marsha Chechik

University of Toronto

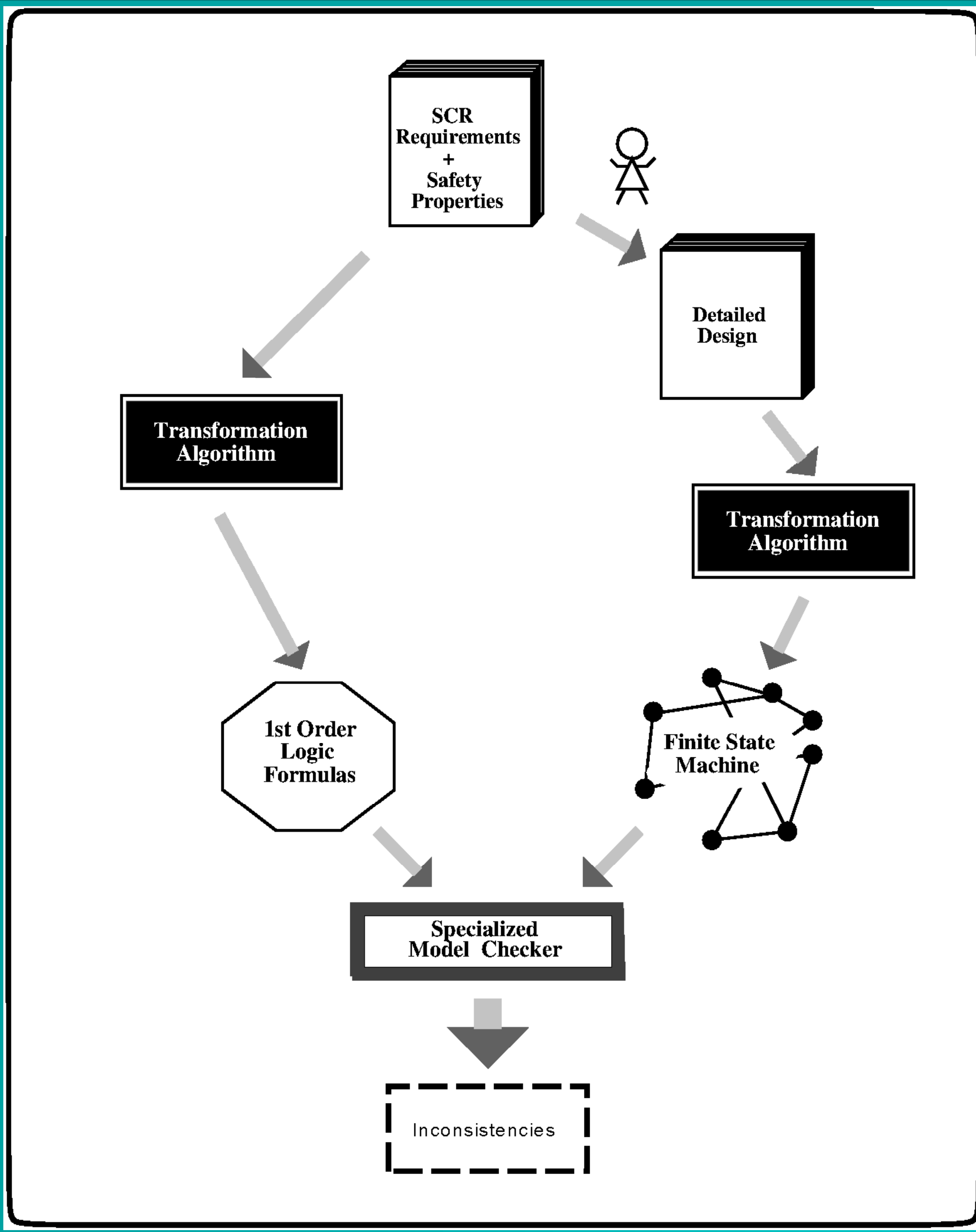
Department of Computer Science

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 designs are consistent with requirements
  - Verify that requirements are consistent
- Reduce analysis effort through automation
  - Build or use existing automatic tools to perform this analysis

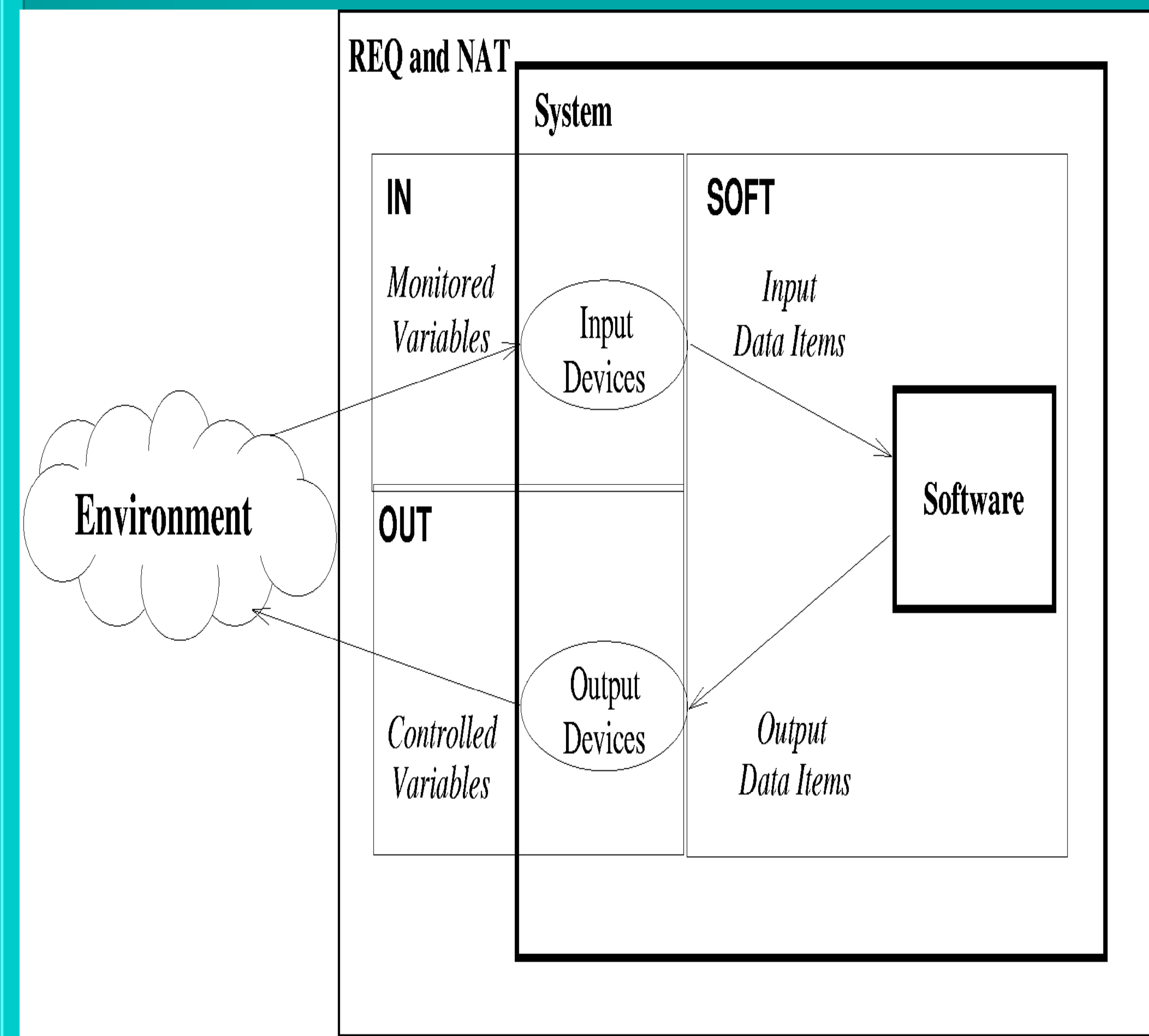
Overview of the Approach



Talk Outline

- Requirements
  - SCR Notation and Semantics
  - Meaning of Consistency with Requirements
- Building the Model
  - Annotations
  - Data-flow Analysis of Annotations and Control-Flow Info
  - Constructing Finite-State Machine
- Verifying Properties
  - Computing Transitions and Events
  - Verification Algorithms and Their Meaning
- Case-study
- Some ideas for future work
- Conclusion

SCR and 4-Variable Model



SCR Requirements

- A *state* is a mapping of variables to values
- A *modeclass* is a set of system 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 variables

  Running, *ButtonPressed*
- An *event* is a point in time when a condition's value changes:

  @T(*buttonpressed*)

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

SCR Requirements (Cont'd)

Assumption - Specification is finite, i.e.

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

Simple Water-Level Monitoring System (SWLMS)

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

Initial: Off if ~SwitchOn & ~PumpFail

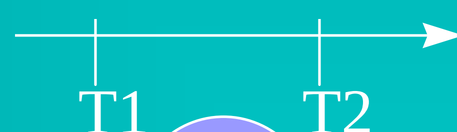
Environmental Assumptions

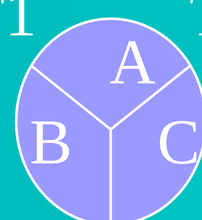
- Declarative information about system variables
- Helps to increase clarity
- Improves readability and reduces annotation effort

Sample relationships:

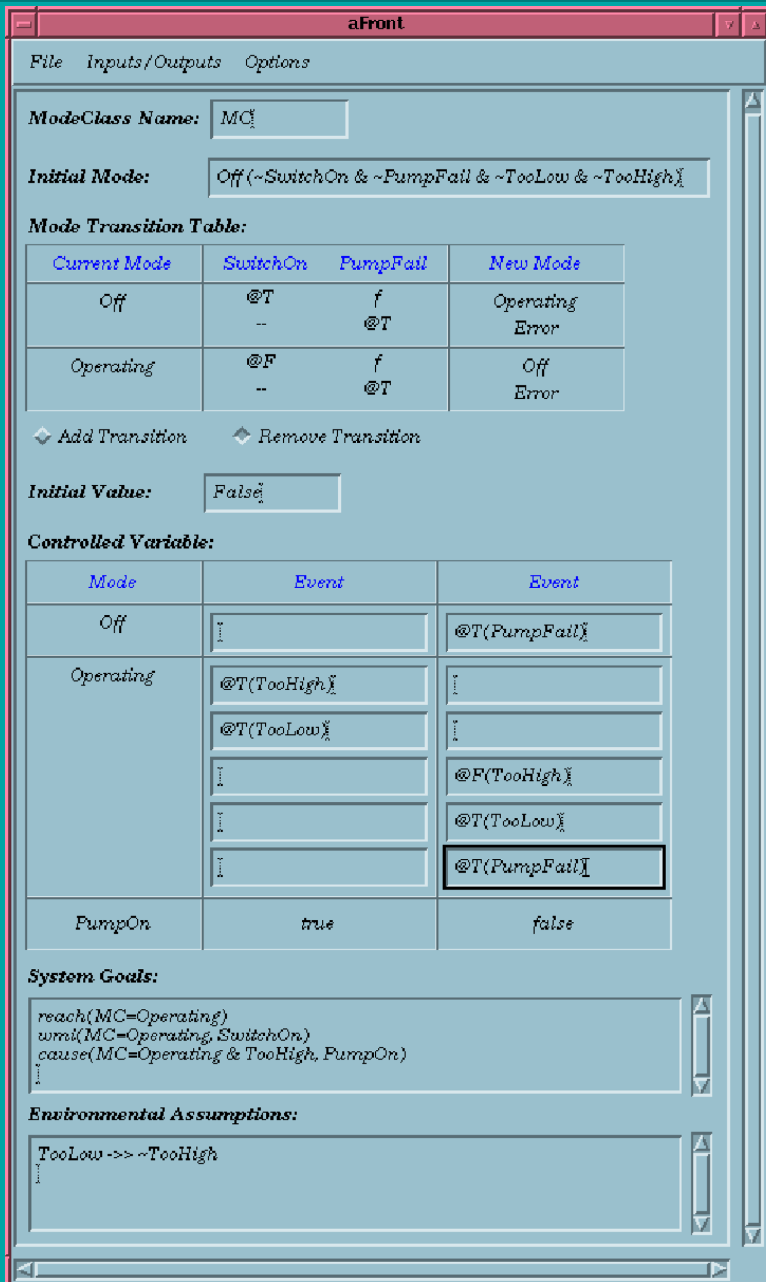
- Equivalence *(A=B)*
- Implication *(A->B)*

  TooLow->~TooHigh
- Timeline *(T1 < T2)*
- Enumeration *(A | B | C)*





User-Interface for CORD



Formal Model of SCR

Formal Model of SCR [Heitmeyer et al]

$$(m = m_i) \land (a = false) \land$$
$$(b = true) \land (a' = true) \land (m' = m_j)$$

Our model:

$$(m' = m_i) \land (a = false) \land$$
$$(b' = true) \land (a' = true) \land (m'' = m_j)$$

Notion of Consistency

Definition

Let a set of SCR requirements $\mathcal{R} = <\mathcal{B},\mathcal{E},\mathcal{G}>$ be given. A program artifact $\mathcal{A}$ constrained by environmental assumptions $\mathcal{E}$ (called $\mathcal{A}_{\mathcal{E}}$) is consistent with its requirements $\mathcal{R}$ if

- $\mathcal{A}_{\mathcal{E}}$ implements all state transitions specified in $\mathcal{B}$ (ALT properties), and
- $\mathcal{A}_{\mathcal{E}}$ does not implement any state transitions which are not specified in $\mathcal{B}$ (OLT properties).

Consistency Properties

- OLT property

$$P_1 = \forall n, (n \models (MC = Error) \rightarrow (\forall p \in pred(n),$$
$$(p,n) \models (MC = Error) \vee (p,n) \models (@T(PumpFail) \wedge$$
$$(MC = Operating)) \vee (p,n) \models (@T(PumpFail) \wedge$$
$$(MC = Off)))$$

- ALT properties

$$P_2 = \exists n, \exists p \in pred(n), n \models (MC = Error) \wedge$$
$$(p,n) \models ((MC = Off) \wedge @T(PumpFail))$$

$$P_3 = \exists n, \exists p \in pred(n), n \models (MC = Error) \wedge$$
$$(p,n) \models ((MC = Operating) \wedge @T(PumpFail))$$

Program Design Language

- Motivation:
  – Designs should look like real programs (I.e., contain control-flow)
  – Designs should capture the essence of what happens in the code
- PDL:
  – Consists of control-flow constructs and annotations reflecting changes of values of requirements variables
  – annotations start with @@
  – annotations Capture local state changes rather than invariant properties

Program Design Language - Example

- Initial: assigns values to variables in start state
- Update - assigns values to variables indicating a state change
- Assert - checks and restricts values of variables

```
@@ Update PumpFail=Top
/* Read device monitor */
If (pump_fail() && …) {
    @@ Assert PumpFail=true;
    @@ Update MC=Error;
}
Else {
    @@ Assert PumpFail=false;
    …;
}
```

Design of SWLMS

```
{…
@@ Initial MC=Off & switchon=false & pumpon=false & toohigh=false
      & toolow=false & pumpfail=false;
While(1) {
  Read_device();
  @@Update pumpfail=Top;
  If (PUMP_FAIL()) {
     @@Assert pumpfail=true;
     Break; }
  @@ Assert pumpfail=false;  /* assume no device failures */
  READ_SWITCH();              /* read switch monitor */
  @@Update switchon=Top;
  If (SWITCH_ON() && IN_OFF (System)) {
    @@Assert switchon=true & MC=Off;
    @@Update MC=Operating; }
  Else if (IN_OPERATING(System)){
    @@Assert MC=Operating;
    While(1) {
      Read_device();
      @@Update pumpfail=Top;
      If (PUMP_FAIL()){
        @@ Assert pumpfail=true;
        Break;  }
      @@ Assert pumpfail=false;
      Read_switch();
      @@ Update switchon=Top;
      If (!SWITCH_ON()) {
        @@ Assert switchon=false;
        @@ Update MC=Off;
        Break;}
      @@ Assert switchon=true;
       GET_WATER_LEVEL(&Water);  /* compute water level */
       @@ Update toohigh=Top & toolow=Top;
       If (IS_HIGH(Water) || (IS_LOW(Water)) {
          @@ Assert toohigh=true | toolow = true;
          @@ Update pumpon=true;}
      Else {
          @@ Assert toohigh=false & toolow=false;
          @@Update pumpon=false;}
      }}}
  @@ Assert pumpfail=true;
  @@ Update MC=Error & pumpon=false;  }
```

Another Way of Writing SCR Designs

Central notion is an event - a *change* of value of a condition.

May occur when a condition is assigned some value or when the value is tested.

Explicit event handling:

```
While (1) {
   For each sensor {
      Copy previous value of corresponding variable
      Read the sensor
      Determine if the event occurred by comparing old and new values
      Determine state changes
      Change values of controlled variables
   }
}
```

Building a Finite-State Abstraction

- Build a control-flow graph consisting of splits, joins and annotations (DFG)
- Compute information generated by each annotation
- Detail this information using environmental assumptions and check for violations of the assumptions
- Propagate this information throughout the DFG in a manner similar to constant propagation

Building DFG (Cont'd)

Information is computed using one of the two techniques:

- *Set-based approximation* - at a given node each variable is associated with a set of values it may attain if control reaches this node
- *Exact analysis* - a given node is associated with a set of variable-value pairs, where each set represents information for some path leading to this node.

Building DFG (Cont'd)

Set-based approximation

Example:

{(PumpOn,{true}), (PumpFail, {true, false}), (MC, {Error, Off})}

Running time:

$$O(|V|^2 \times \sum_{r \in R} |T(r_j)|$$

Exact analysis

Example:

{{(PumpOn, true), (MC, Error),(PumpFail, true)},
{(PumpOn, true), (MC, Off), (PumpFail, false)},
{(PumpOn, true), (MC, Off), (PumpFail, true)}}

Running time:

$$O(|V|^2 \times \prod_{r \in R} |T(r_j)|+1)$$

Building DFG (Cont'd)

- Compute `gen` and `known` sets for each annotation

@@ Assert SwitchOn=False & TooLow=true

`known` = {(SwitchOn,{false}), (TooLow,{true})}

@@ Update MC=Operating

`gen` = {(MC, {Operating})}

- Use environmental assumptions to make `gen` and `known` sets more precise

Environmental assumption:

TooLow -> ~TooHigh

So,

`known` = {(SwitchOn,{false}), (TooLow,{true}), (TooHigh,{false})}

Propagating Information

$$in(n) = \coprod_{\forall k, s.t. (k,n) \in E} out(k)$$

$$out(n) = F(in(n))$$

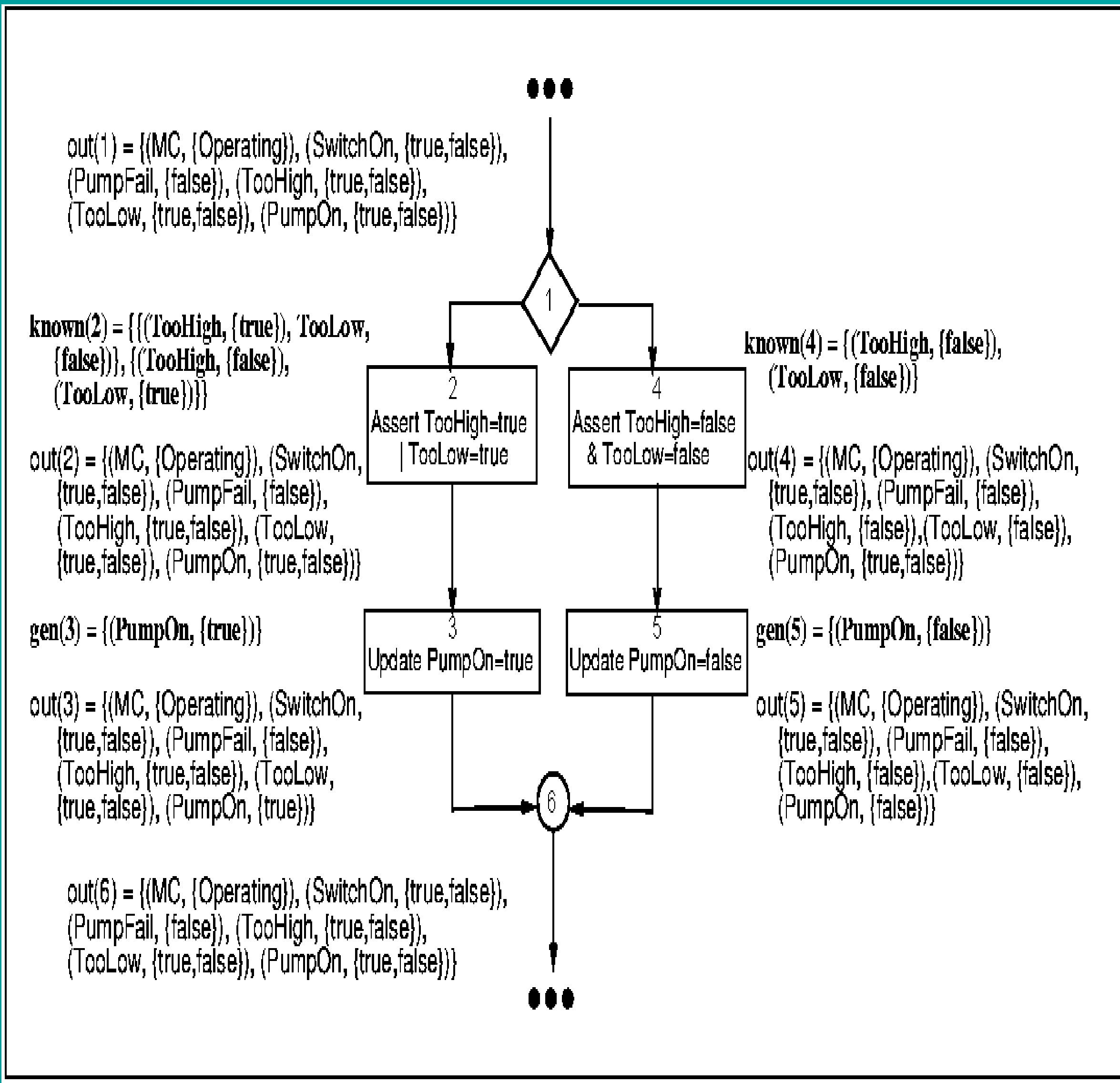
F is a transfer function, defined as

Annotation at node n	Transfer function
Initial	$\texttt{out}(n) = \texttt{gen}(n)$
Update	$\texttt{out}(n) = repl(\texttt{in}(n), \texttt{gen}(n))$
Assert	$\texttt{out}(n) = \texttt{in}(n) \coprod \texttt{known}(n)$
none	$\texttt{out}(n) = \texttt{in}(n)$

Example: DFG of SWLMS

```
if (IS_HIGH(Water) || IS_LOW(Water)) {
  @@ Assert TooHigh=true | TooLow=true;
  @@ Update PumpOn=true;
}
else {
  @@ Assert TooHigh=false &
    TooLow=false;
  @@ Update PumpOn=false;
}
```

DFG of SWLMS



out(1) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
1
known(2) = {{(TooHigh, {true}), TooLow, {false})}, {(TooHigh, {false}), (TooLow, {true})}}
2
Assert TooHigh=true | TooLow=true
4
Assert TooHigh=false & TooLow=false
known(4) = {(TooHigh, {false}), (TooLow, {false})}
out(2) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
out(4) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {false}),(TooLow, {false}), (PumpOn, {true,false})}
gen(3) = {(PumpOn, {true})}
3
Update PumpOn=true
5
Update PumpOn=false
gen(5) = {(PumpOn, {false})}
out(3) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true})}
out(5) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {false}),(TooLow, {false}), (PumpOn, {false})}
6
out(6) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

Constructing FSM

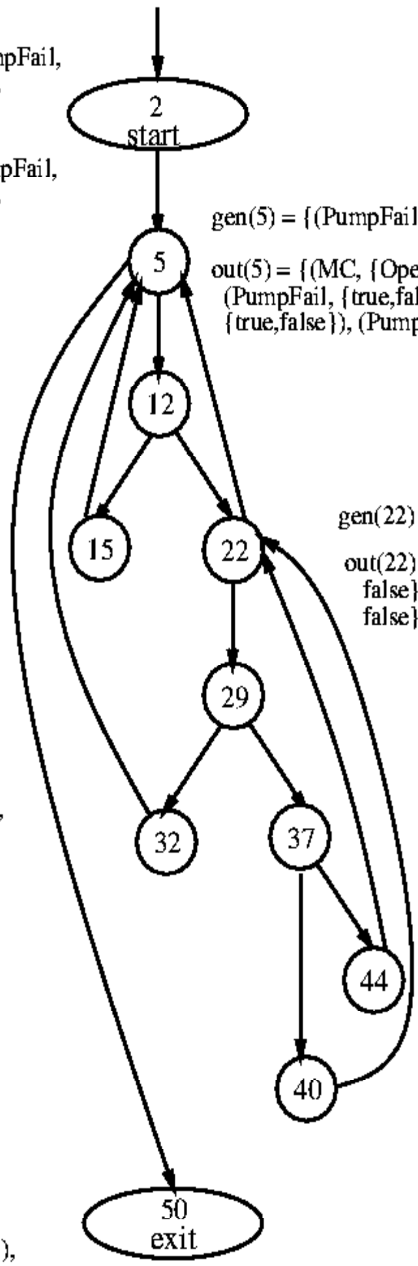
- Remove all DFG nodes except those corresponding to Initial and Update annotations
- For each node, check an implicit reachability property:

$$\forall n \in S, (\forall r_j \in R, v(r_j, \mathtt{out}(n)) \neq \{\})$$

FSM of SWLMS

gen(2) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {false}), (TooLow, {false}), (PumpOn, {false})}}

out(2) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {false}), (TooLow, {false}), (PumpOn, {false})}



gen(5) = {(PumpFail, {true,false})}

out(5) = {(MC, {Operating,Off}), (SwitchOn, {true,false}), (PumpFail, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

gen(12) = {(SwitchOn, {true,false})}

out(12) = {(MC, {Operating,Off}), (SwitchOn, {true,false}),(PumpFail, {false}), (TooHigh, {true,false}),(TooLow,{true,false}), (PumpOn, {true,false})}

gen(15) = {(MC, {Operating})}

out(15) = {(MC, {Operating}), (SwitchOn, {true}),(PumpFail, {false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}

gen(22) = {(PumpFail, {true,false})}

out(22) = {(MC, {Operating}), (SwitchOn, {true, false}), (PumpFail, {true,false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}

gen(29) = {(SwitchOn, {true,false})}

out(29) = {(MC, {Operating}), (SwitchOn, {true, false}), (PumpFail, {false}), (TooHigh, {true. false}), (TooLow, {true,false}), (PumpOn, {true, false})}

gen(37) = {(TooLow, {true,false}),(TooHigh, {true,false})}

out(37) = {(MC, {Operating}), (SwitchOn, {true}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

gen(32) = {(MC, {Off})}

out(32) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow,{true,false}), (PumpOn, {true,false})}

gen (44) = {(PumpOn, {false})}

out(44) = {(MC, {Operating}), (SwitchOn, {true}), (PumpFail, {false}), (TooHigh, {false}), (TooLow, {false}), (PumpOn, {false})}

gen (40) = {(PumpOn, {true})}

out(40) = {(MC, {Operating}), (SwitchOn, {true}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true})}

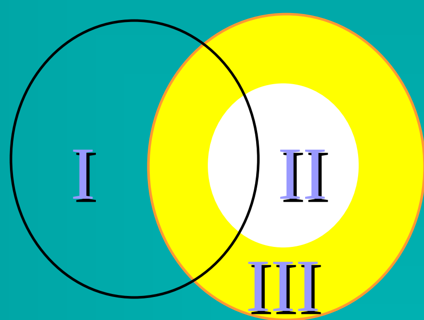
gen(50) = {(MC, {Error}), (PumpOn, {false})}

out(50) = {(MC, {Error}), (SwitchOn, {true,false}), (PumpFail,{true}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {false})}

Optimism and Pessimism

- A property is checked pessimistically if all of its violations in the design are detected, but the analysis may incorrectly identify violations at points in the design at which the property actually holds
- A property is checked optimistically if all detected violations are present in the design, but the analysis is sometimes unable to find all violations of the property

Verifying Properties - Overview



I & ~III	ALT violation
I & III & ~II	ALT violation, not found (optimism)
I & III & II	no ALT violation
~I & ~III	no OLT violation
~I & III & ~II	extra OLT violation (pessimism)
~I & III & II	OLT violation

Computing Transitions

A *transition* is a tuple

`(from, to, triggering, when)`

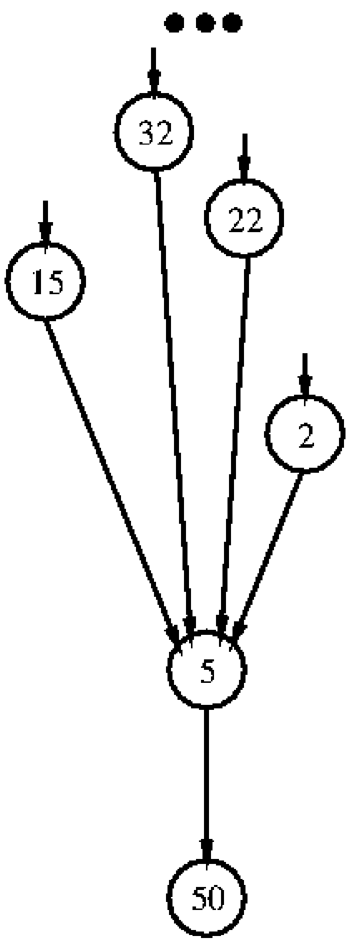
Triggering conditions:

- Computed from `gen` sets of predecessor nodes
- Checked against their predecessor nodes to see if an event occurred
- Checked against designer's assumptions

When conditions:

- Computed from `out` sets of predecessor nodes
- Checked against designer's assumptions

Computing Transitions - Example



out(32) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}

out(22) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {true,false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}

out(15) = {(MC, {Operating}), (SwitchOn, {true}), (PumpFail, {false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}

out(2) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {false}), (TooLow, {false}), (PumpOn, {false})}

gen(5) = {(PumpFail, {true,false})}

out(5) = {(MC, {Operating, Off}), (SwitchOn, {true, false}), (PumpFail, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

in(50) = {(MC, {Operating,Off}), (SwitchOn, {true,false}), (PumpFail, {true}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

gen(50) = {(MC, {Error})}

out(50) = {(MC, {Error}), (SwitchOn, {true,false}), (PumpFail, {true}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

SWLMS Transitions

I_1: Transition from MC=Off to MC=Error:

@T(PumpFail) WHEN [(SwitchOn,{true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})]

I_2: Transition from MC=Off to MC=Error:

NONE WHEN [(SwitchOn,{true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})]

I_3: Transition from MC=Operating to MC=Error:

@T(PumpFail) WHEN [(SwitchOn,{true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})]

I_4: Transition from MC=Operating to MC=Error:

NONE WHEN [(SwitchOn,{true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})]

Verifying OLT Properties

- OLT properties are verified pessimistically, in a single FSM traversal
- Errors are reported as discovered

Example OLT property:

$$P_1 = \forall n, (n \models (MC = Error) \rightarrow (\forall p \in pred(n),$$
$$(p,n) \models (MC = Error) \vee$$
$$((p,n) \models @T(PumpFail) \wedge (MC = Operating)) \vee$$
$$((p,n) \models @T(PumpFail) \wedge (MC = Off)))$$

No transitions satisfy this property , so an error is reported:

Error on line 50 of function main in modeclass MC:
no triggering condition for transition
from mode(s) {Operating,Off} to mode(s) {Error}

Verifying ALT Properties

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

Example ALT properties:

$$P_2 = \exists n, \exists p \in pred(n), n \models (MC = Error) \wedge$$
$$(p,n) \models ((MC = Off) \wedge @T(PumpFail))$$

$$P_3 = \exists n, \exists p \in pred(n), n \models (MC = Error) \wedge$$
$$(p,n) \models ((MC = Operating) \wedge @T(PumpFail))$$

Node 50 is reachable. P_2 is satisfied by I_1. P_3 is satisfied by I_3.

Output of Analysis of SWLMS

CORD (Version 3.02)

Error on line 15 of function main in modeclass MC:
no triggering condition for transition from mode(s) {Off} to mode(s) {Operating}

Error on line 32 of function main in modeclass MC:
illegal transition from mode(s) {Operating} to {Off}

Event is (@T(TooHigh) & @T(TooLow)) or (@T(TooHigh) & @F(TooLow))…

WHEN [(PumpOn, {true,false}), (SwitchOn, {false}), (PumpFail, {false})]

….

*** MISSING TRANSITIONS ****

FROM	EVENT	TO
Operating	@F(SwitchOn) WHEN [PumpFail=false]	Off

Output of Analysis of SWLMS (Cont'd)

VARIABLE	EVENT	VALUE
PumpOn	@T(PumpFail) WHEN [MC=Off]	false
PumpOn	@T(PumpFail) WHEN [MC=Operating]	false

…..

***** VIOLATION SUMMARY *****

OLT properties for modeclasses:	1 time
OLT properties for controlled variables:	1 time
OLT properties (no events found):	5 times
ALT properties for modeclasses:	1 time
ALT properties for controlled variables:	2 times

Case-Study - WLMS

WLMS monitors and displays the water level in a container, raising visual and audio alarms and shutting off its pump when the level is out of range or when monitoring system fails. Two push buttons, SelfTest and Reset, permit operator to test the system and return it to normal operation.

Requirements:

- 2 mode classes (NORMAL and FAILURE) with 7 total modes
- 18 monitored and 7 controlled variables

Case-Study (Cont'd)

Original implementation:

- 1000+ lines (54 functions) in FORTRAN and Assembler

Translation:

- Into C with Xlib interface

Detailed Design (Reverse-engineered):

- 250 lines (8 functions)
- 49 Update and 60 Assert annotations

Some WLMS Code

```
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_OVEFLOW) {
      RESET_TIMER(SHUTER);
      SHUTTM = 0;
   }
   else (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);
      ….
   }
```

Results of WLMS Verification

Results of design verification

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

Errors Found

- OLT properties violated 16 times (4 locations) for modeclasses and 53 times (7 locations) for controlled variables
- ALT properties violated 9 times for modeclasses and 28 times for controlled variables

Successes

- Defined a notion of consistency between SCR-style requirements and designs
- Looked at ways of implementing SCR requirements and developed a simple PDL
- Developed techniques for building finite-state machines from detailed designs
- Gave precise definition of how results of verification of properties should be interpreted (optimism and pessimism)
- Performed a case-study and found a number of defects, missed during testing

Limitations

- The program design language is primitive and not very natural. Variables can only be boolean and enumerated types. No data structures are allowed
- The technique does not support a notion of hierarchical verification
- Our analysis is inexact, so the answers require manual checking
- No attempt has been made in dealing with timing

Research Directions

- Connecting annotations with source code
- Generating code from design construct
- Developing approaches to verify design slices
- Developing hybrid techniques: combination of static analysis and automatic test generation
- Developing a richer type system
- Conducting more case-studies