A (Yet) Another Look at Checking Program Correctness

Marsha Chechik

University of Toronto

Department of Computer Science

September 26, 1997

http://www.cs.toronto.edu/~chechik

Methods of Program Verification

- Verify that program corresponds to its specification
  - “Classical” methods: Hoare, Dijkstra
- Check some properties:
  - Check that property is implied by program (theorem proving) [PVS, HOL, etc]
    - » Very effective, although requires considerable skill
    - » Used in Ontario Hydro now for Darlington reengineering

Methods of Program Verification

- Model all possible behaviors and check temporal properties on this model (model-checking) [clarke86]
  - "can the system deadlock or survive a particular fault or give response within certain time?"
  - ideal for hardware and distributed algorithms like protocols

  Problem: state-space grows exponentially to the # of vars

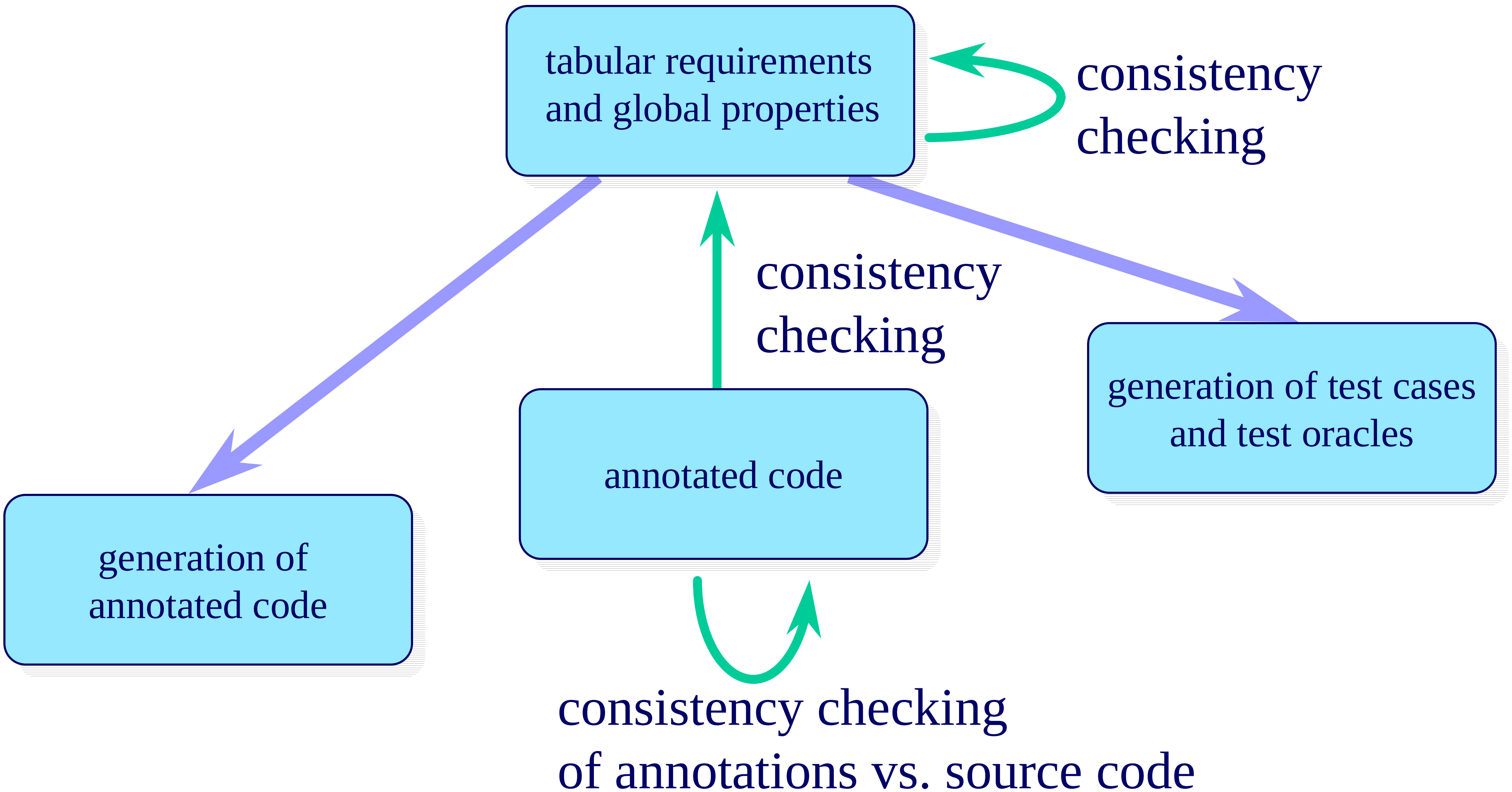
  - develop better model-checkers (explore symmetries, symbolic)
  - verify abstractions of programs [Wing95, Jackson96]
    - » problem of reporting meaningful errors

Our Approach to Formal Methods

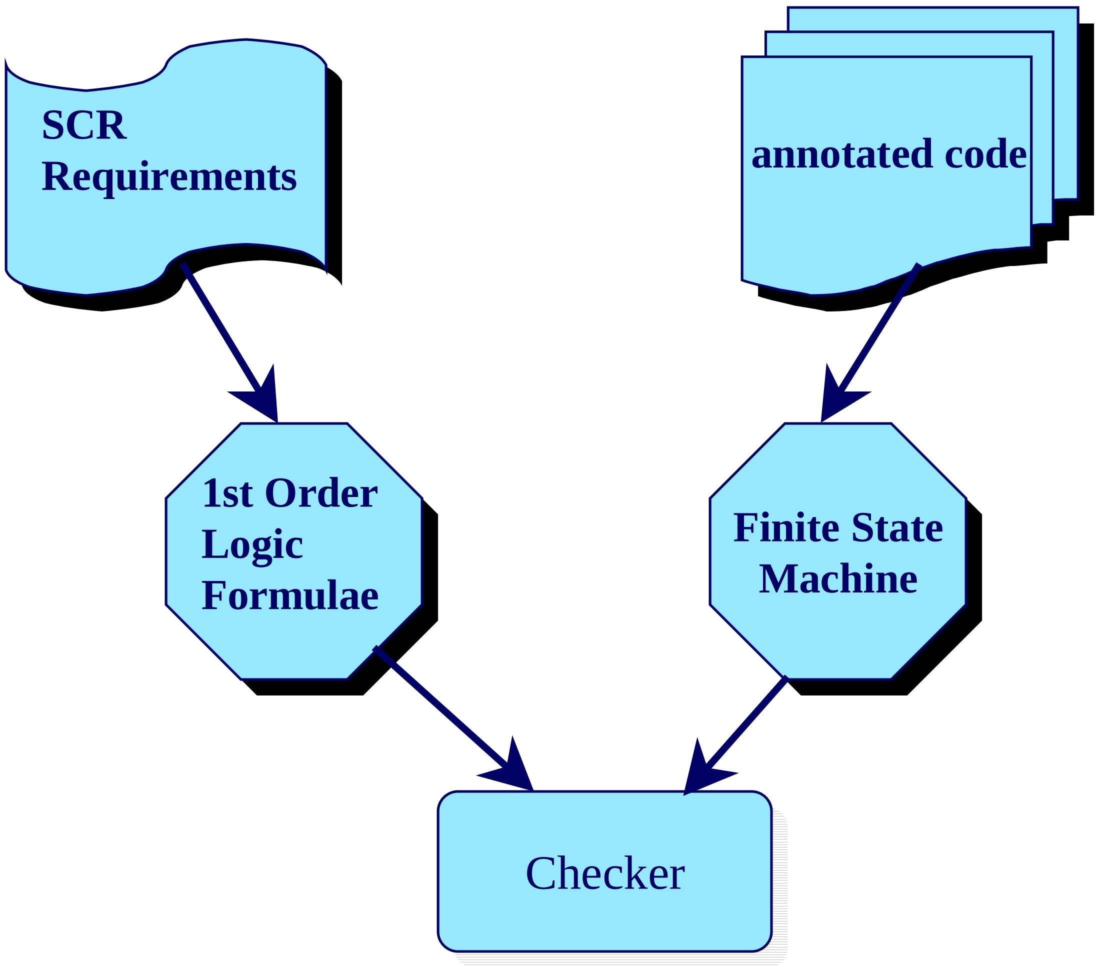
Formal methods "light"

- Use easy to read and review requirements notations (StateCharts, RSML, SCR)
- Develop partial specifications and analyze them mechanically
- Don't check for all properties - just important ones
- Sacrifice exactness for scalability
- Apply formal techniques more for debugging than assurance
- Amortize use of formal specs in other parts of the development

Formal Methods Research at UofT



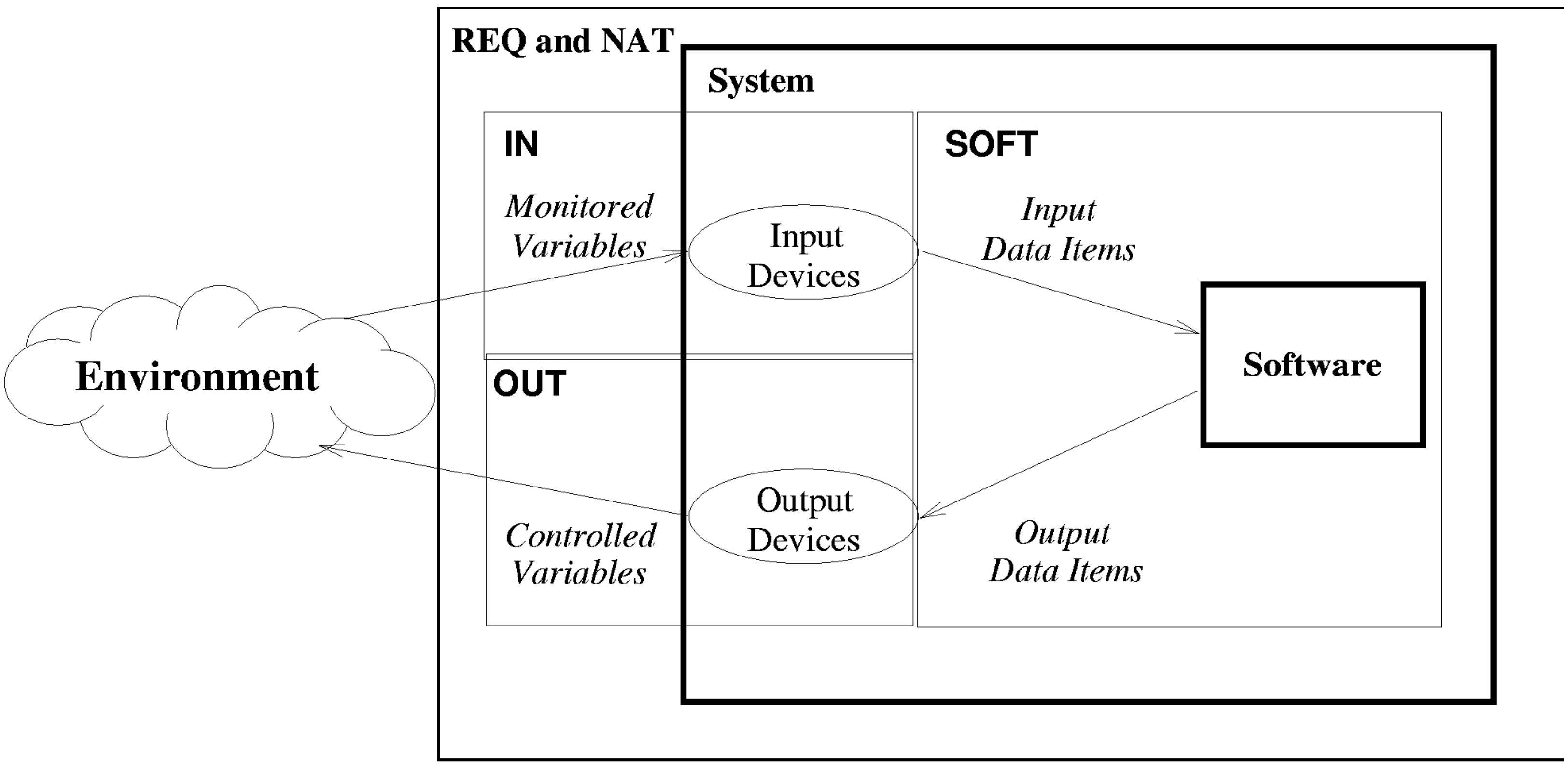
Overview of the Approach



Overview of the talk

- SCR Requirements
- Notion of Consistency
- Annotating programs
- Building a finite-state machine
- Checking Properties
- Case study

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

Total size of state space:

$$2^{|M| \cup |C|} \times \prod_{1 \le j \le |MD|} T(md_j)$$

Simple Water-Level Monitoring System (SWLMS)

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

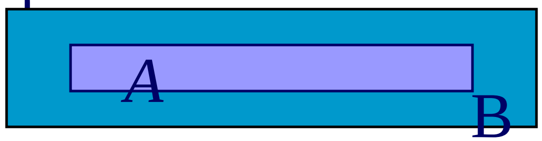
Initial: Off if ~SwitchOn & ~PumpFail

Environmental Assumptions

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

Sample environmental assumptions:

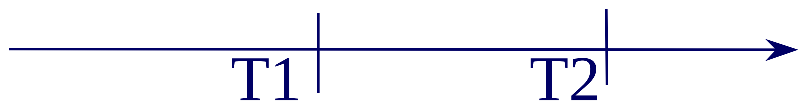
- Equivalence *(A=B)*



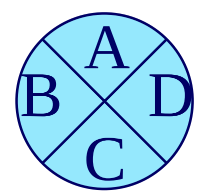
- Implication *(A->B)*

TooCold -> *~TooHot*

- Timeline $(T1 < T2)$



- Enumeration *(A | B | C | D)*



Notion of Consistency

Definition

Let a set of SCR requirements $\mathcal{R} = \langle \mathcal{B}, \mathcal{E}, \mathcal{G} \rangle$ 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

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

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

Annotation Language

- Goal: document code with events on the requirements level. These events can be used to check consistency with requirements
- Language constructs - KISS principle:
  - Initial: assigns values to variables in starting state
  - Update: assigns values to variables indicating a state change
  - Assert: checks and restricts values of variables indicating a condition

Example - 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);
    …
```

Annotated WLMS Code

```
while (true) {
  …
  if (MODE_EQ (OPCM, Shutdown) && INLIM && (SHUTTM < DPAR_SLT) &&
     !PRESSED(SlfTst)) {
     @@ Assert Normal=Shutdown & WithinLimits = true &
     ShutdownLockTime200=false & SlfTstPressed=false;
      SET_MODE (OperatingClass, Operating);
     @@ Update Normal=Operating;
   }
   …
   GET_MODE (OperatingClass, OPCM);
   if ((!MODE_EQ(OPCM, Test))) && (PRESSED(SlfTst))) {
      @@ Assert ~ Normal=Test &  SlfTstPressed = true;
      SET_MODE(OperatingClass, Test);
      @@ Update Normal=Test;
   }
   …
```

Checking annotations vs source code

- User gives associations between annotation and source-code variables

```
Correspondences:
   {OperatingClass} = {Normal}
   {Pressed(SlfTst)} = {SlfTstPressed}
   {Shuttm} = {ShutdownLockTime}
   {INLIM} = {WithinLimits}
```

- Static checker checks that assignments and tests of source-code variables correspond to Updates and Asserts on annotation 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)

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

Propagating Information

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

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

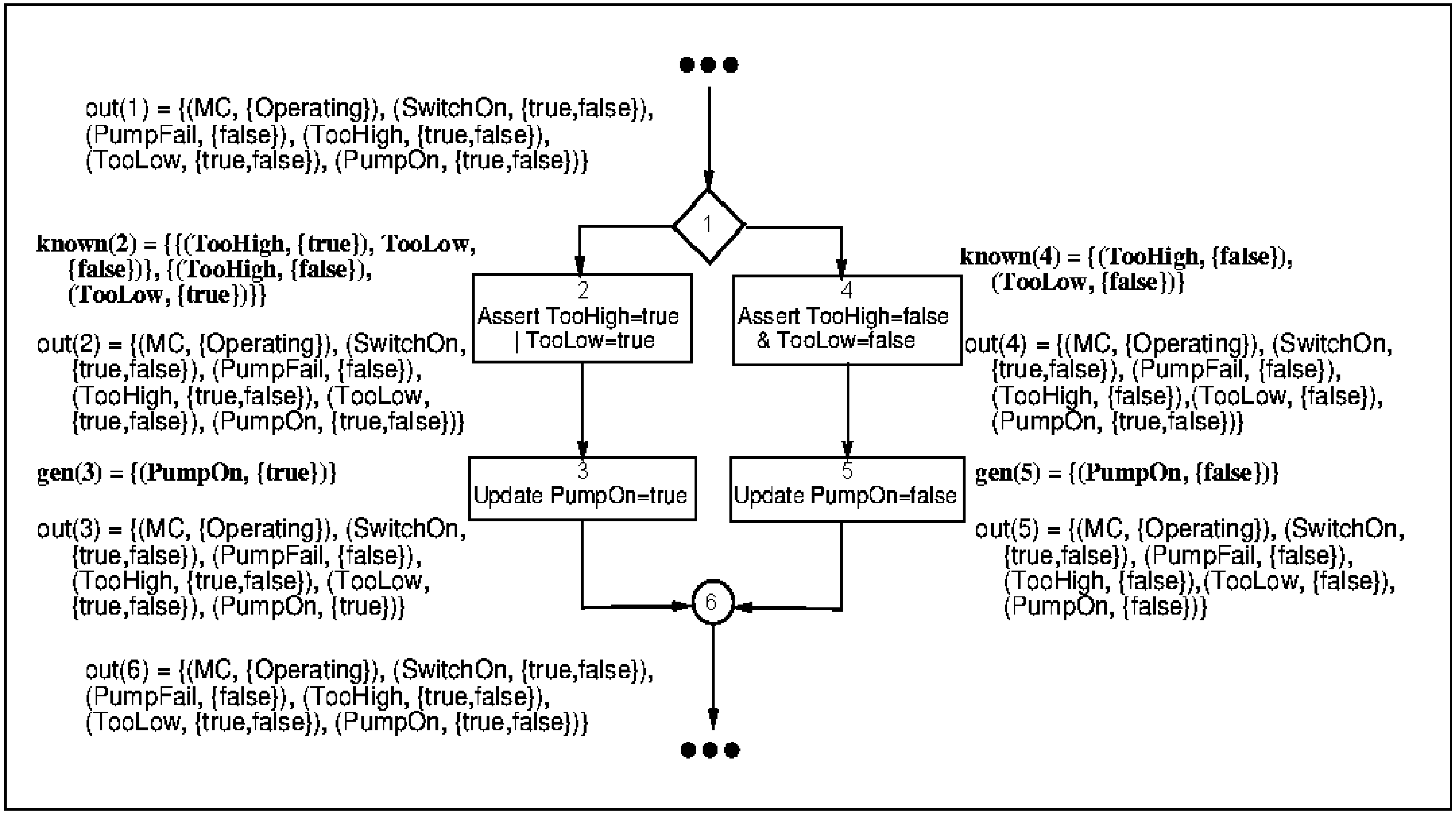
F is a transfer function, defined as

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

Example: Fragment of SWLMS

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

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

Running time:

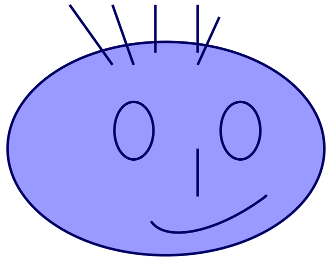
$$O(V \times (|MC| + |M| + |C|))$$

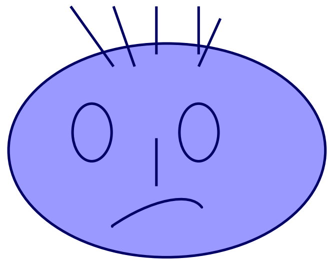
Resulting # of nodes is at most the number of Update and Initial annotations

Optimism and Pessimism

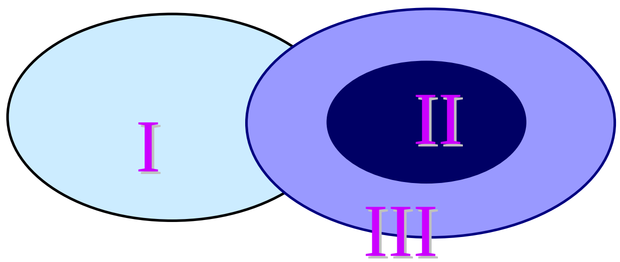
- A property is checked pessimistically if all of its violations in the design are detected, but the analysis may incorrectly identify violations at points in the design at which the property actually holds

- 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)`

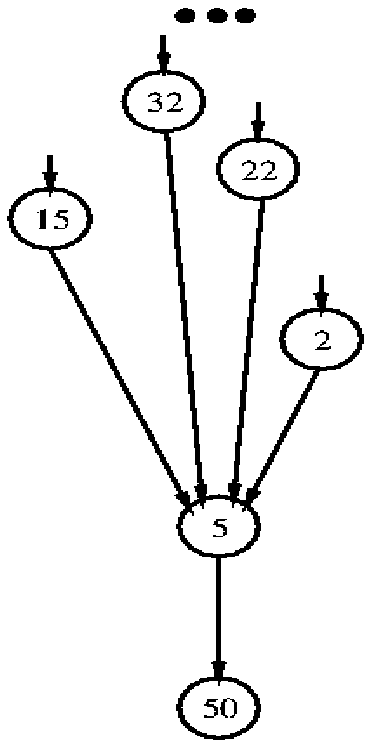
Triggering conditions:

- 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       Off
                 [PumpFail=false]
```

Output of Analysis of SWLMS (Cont'd)

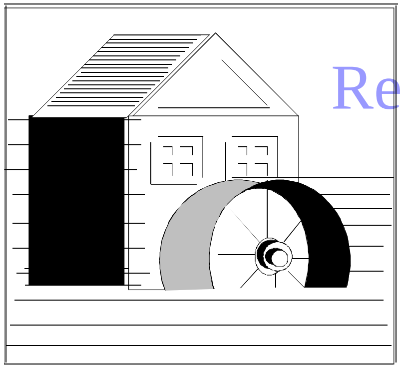
```
VARIABLE                    EVENT                        VALUE
-------------------------------------------------------------
PumpOn       @T(PumpFail) WHEN [MC=Off]               false
PumpOn   @T(PumpFail) WHEN [MC=Operating]             false
                                             …..
          ***** 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
```

Running Time for Verifying Properties

$$O(|V|^2 \times \text{num\_properties})$$

Water-Level Monitoring System

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

SCR Table for WLMS

Current Mode	*Inside HysRange*	*Within Limits*	*SlfTst Pressed*	*SltTst Pressed 200*	*Test Interval*	*Reset Interval*	*Shutdown LockTime*	**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 (~SlfTestInterval & ~TestInterval & ~ResetInterval & ~ShutdownLockTime)

Global Properties of WLMS

- "If the system is in mode Operating, the self-test button has not been pressed for 500 ms and either the water level is within limits or the self-test button is being pressed"

  inv(Operating -> (~SlfTestPressed500 & (WithinLimits | SlfTestPressed)))

- "If the system is in Shutdown, the water is not inside hysterysis range and the system has been in mode Shutdown for less than 200 ms"

  inv(Shutdown -> (~InsideHysRange & ~ShutdownLockTime200))

- "If the system is in Standby, the self-test button has been pressed for 500 ms"

  inv(Standby, ~SlfTestPressed500)

- "When the self-test button is pressed for 500 ms, the system should go into self-test mode"

  cause(SlfTestInterval, Test)

Case-Study (Cont'd)

- Original implementation:
  - 1000+ lines (54 functions) in C with Xlib
  - 49 Update and 60 Assert annotations
- Analyzed annotated implementation (time ~ complication)

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 properties

Conclusion

- Formal methods, when properly applied, can significantly improve quality of the product and decrease cost
- In non safety-critical systems, FMs are best used in specifications
- At UofT, we support formal documentation by
  - using familiar notations for specification
  - amortizing cost of formal spec in various analysis activities
  - using/developing automatic tools, requiring minimum user-guidance

Research Interests

- Development of hybrid (formal + informal) methods for specification of software
- Improvement of annotation language and tools
- Industrial-size case-study (so far, only 2-3 K lines)
- Specification-based testing
- Use of existing techniques like model-checking and theorem-proving to aid in analysis of timing properties