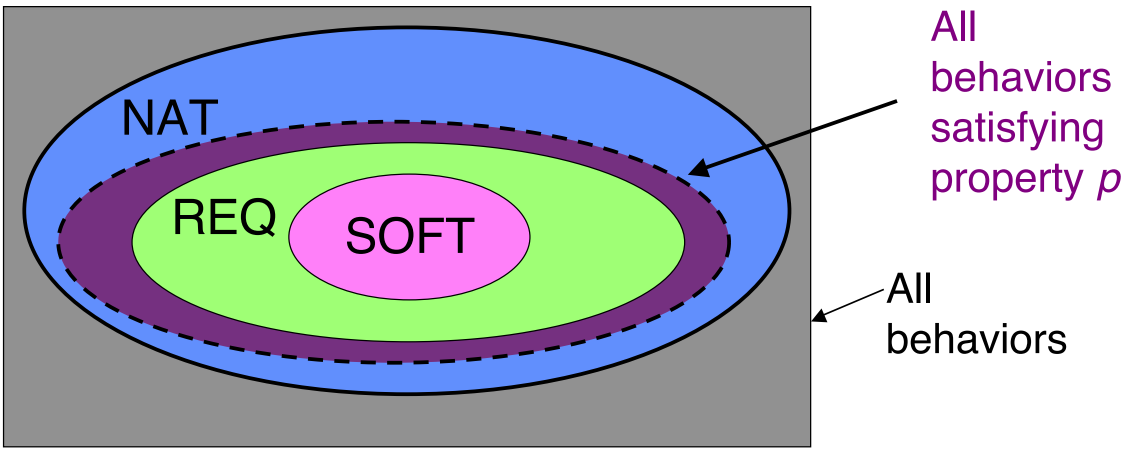
Requirements Models for System Safety and Security

Connie Heitmeyer
Naval Research Laboratory
Washington, DC 20375
heitmeyer@itd.nrl.navy.mil

International Summer School
Marktoberdorf
August 2010

BEHAVIORS (POSSIBLE VS. ACCEPTABLE) AND PROPERTIES



NAT: All possible behaviors satisfying natural laws, constraints on the system env
REQ: All acceptable system behaviors
SOFT: All acceptable software behaviors

OVERVIEW

- Introduction to the Requirements Problem
- Four Variable Model and SCR
  - Formal Requirements Model
  - Tools for Analyzing Requirements Models
  - Applying the Tools to Practical Systems
- Verifying Source Code for Security Properties: A Practical Application
- An incremental, model-based method for developing critical software
  - Example applying the method to fault-tolerance

SCR REQUIREMENTS MODEL

WHAT QUESTIONS DOES THE SCR MODEL ADDRESS?

- What units of discourse are useful in specifying the required software system behavior?
  - Monitored & controlled variables, terms, and modes
  - Conditions and events
- How are system outputs (i.e., controlled vars) represented as mathematical functions?
  - Role of terms and modes
  - Semantics of SCR tabular format
- How can the required behavior of a system be represented as a state machine?

DEFINITIONS: VARIABLES, TYPES, AND SYSTEM STATE

We assume the existence of a number of sets including

- *RF* is the set of *state variable names*
  - *RF* is partitioned into sets of mode class names, monitored variable names, term names, and controlled variable names
- *TS* is a union of *types*, where each *type* is a nonempty set of values
- For all *r* in *RF*, $TY(r) \subseteq TS$ is the *range type* of *r*
  - *TY(r)* is the set of possible values of *r*

A *system state* *s* is a function that maps each state variable name *r* in *RF* to a value in *TY(r)*

DEFINITION: SYSTEM

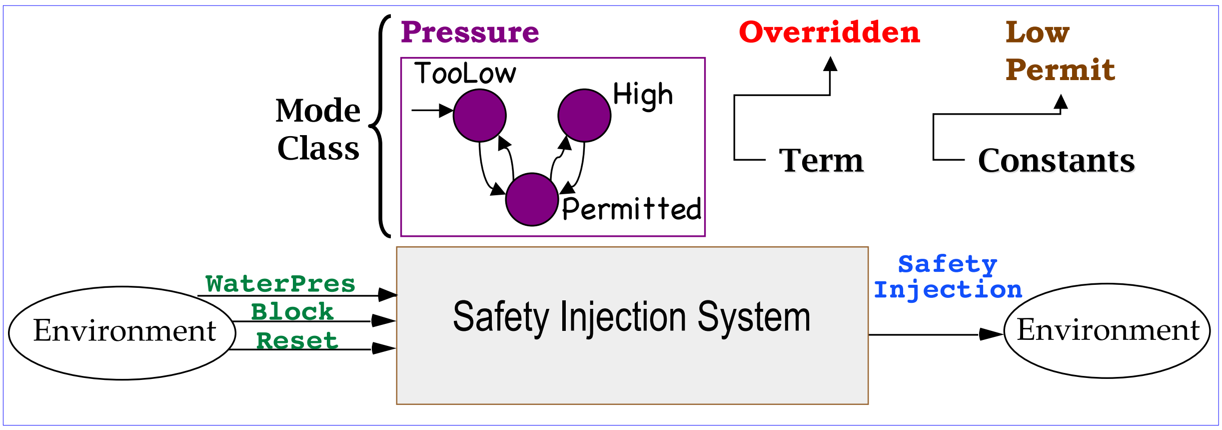
A software system is a state machine whose transitions from one state to the next are triggered by monitored events. Formally, a *software system* Σ is a 4-tuple $\Sigma = (E^m, S, S_0, T)$, where

- E^m is the set of possible monitored events,
- S is the set of possible system states,
- $S_0 \subseteq S$ is the set of initial states, and
- T is the system transform, a partial function from $E^m \times S$ into S. T is a partial function because not all monitored events are eligible to occur in a given state.

Note: Our state machine model is NOT a Mealy machine
→ the system outputs (i.e., controlled vars) are included in the state

EXAMPLE MODEL: CONTROL SYSTEM FOR SAFETY INJECTION (3)

- Mode Class Pressure - abstraction of WaterPres
- Term Overridden - denotes whether operator has overridden injection
- Controlled variable SafetyInjection - defined in terms of terms, modes, and monitored variables



EXAMPLE: SYSTEM STATE

The example control system contains the following sets:

Set of monitored variables: {Block, Reset, WaterPres}
Set of controlled variables: {SafetyInjection}
Set of terms: {Overridden}
Set of mode classes: {Pressure}

Type definitions associated with these sets are

TY(WaterPres) = {1, 2, ..., 2000}
TY(SafetyInjection = {On, Off}
TY(Block) = TY(Reset) = {On, Off}
TY(Overridden) = {true, false}
TY(Pressure) = {TooLow, Permitted, High}

variable name	WaterPres	Block	Reset	Pressure	Overridden	SafetyInjection
variable value	850	Off	On	TooLow	false	Off

Example of a System State

EXAMPLES: CONDITIONS AND EVENTS

Concept	Syntax	Semantics	Evaluation
simple condition	Reset=Off; WaterPres<900	Reset=Off; WaterPres<900	true in s and s′ true in s, false in s′
condition	Reset=Off AND WaterPres<900 Reset=Off OR WaterPres<900	Reset=Off AND WaterPres<900 Reset=Off OR WaterPres<900	true in s, false in s′ true in s and s′
→ primitive event	@T(WaterPres=900) @T(Reset=Off)	WaterPres≠900 AND WaterPres′=900 Reset≠Off & Reset′=Off	true in (s, s′) false in (s, s′)
→ conditioned event	@T(WaterPres=900) **WHEN** Reset=Off	WaterPres≠900 AND WaterPres′=900 AND Reset=Off	true in (s, s′)

states:

s_0	...	s	s'
Reset=Off WaterPres=25	• • •	Reset=Off WaterPres=899	Reset=Off WaterPres=900

DENOTING FUNCTIONS USING TABLES

Advantages of a tabular notation

- Less error-prone than, e.g., logic notation
  - Structure provided by tables eliminates whole classes of errors
- More scalable than many other notations
  - For example, graphic notations, such as finite state diagrams, do not scale well to practical applications
    - » The labels on the transitions are often too long
    - » Not practical when the number of states is large

EXAMPLE OF A CONDITION TABLE

Mode Pressure	**Condition**	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
SafetyInjection =	Off	On

Based on the new state dependencies set $D_n = \{\texttt{Pressure}, \texttt{Overridden}\}$ and the above condition table, the function F_6 defining the value of the controlled variable $r_6 = \texttt{SafetyInjection}$ is defined by

$\texttt{SafetyInjection} =$

$$F_6(\texttt{Pressure}, \texttt{Overridden}) = \begin{cases} \texttt{Off} \text{ if } \texttt{Pressure} = \texttt{High} \lor \texttt{Pressure} = \texttt{Permitted} \lor \\ \qquad (\texttt{Pressure} = \texttt{TooLow} \land \texttt{Overridden} = \mathit{true}) \\ \texttt{On} \text{ if } \texttt{Pressure} = \texttt{TooLow} \land \texttt{Overridden} = \mathit{false} \end{cases}$$

The table defines SafetyInjection as a function of a single state.

CONDITION TABLE: FORMAL DEFINITION

Mode	Condition			
m_1	$c_{1,1}$	$c_{1,2}$	$\cdots$	$c_{1,p}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
m_n	$c_{n,1}$	$c_{n,2}$	$\cdots$	$c_{n,p}$
r_i	v_1	v_2	$\cdots$	v_p

no ambiguity

no missing cases

Each **condition table** describes the value of a controlled variable or term r_i as a relation ρ_i on modes, conditions, and values:

$$\rho_i = \{(m_j, c_{j,k}, v_k) \in M_{\mu(i)} \times C_i \times TY(r_i)\}.$$

The relation ρ_i must satisfy the following properties:

1. The m_j are unique; the v_k are unique.
2. $\cup_{j=1}^{n} m_j = TY(\mu(i))$ (All modes in the associated mode class are included).
3. For all j: $\vee_{k=1}^{p} c_{j,k} = true$ (**Coverage:** The disjunction of the conditions in each row of the table is *true*).
4. For all $j, k, l,\ k \neq l$: $c_{j,k} \wedge c_{j,l} = false$ (**Disjointness:** The pairwise conjunction of the conditions in each row of the table is always *false*).

These four properties guarantee that the function is total

EXAMPLE OF AN EVENT TABLE

Mode Pressure	**Event**	
High	Never	@F(Pressure = High)
TooLow, Permitted	@T(Block = On) WHEN Reset = Off	@T(Pressure = High) OR @T(Reset = On)
Overridden' =	True	False

Based on the above event table and the new state and old state dependencies sets, {`Block`, `Reset`, `Pressure`, `Overridden`} and {`Block`, `Reset` `Pressure`}, the function defining the value of `Overridden`, denoted F_5, is described by

$$\mathtt{Overridden}' = F_5(\mathtt{Pressure}, \mathtt{Block}, \mathtt{Reset}, \mathtt{Overridden}, \mathtt{Pressure}', \mathtt{Block}', \mathtt{Reset}'\} =$$

$$\begin{cases} true & \text{if } \boxed{(\mathtt{Block}' = \mathtt{On} \wedge \mathtt{Block} = \mathtt{Off} \wedge \mathtt{Pressure} = \mathtt{TooLow} \wedge \mathtt{Reset} = \mathtt{Off})} \vee (\mathtt{Block}' = \mathtt{On} \wedge \mathtt{Block} = \mathtt{Off} \wedge \\ & \quad \mathtt{Pressure} = \mathtt{Permitted} \wedge \mathtt{Reset} = \mathtt{Off}) \\ false & \text{if } (\mathtt{Reset}' = \mathtt{On} \wedge \mathtt{Reset} = \mathtt{Off} \wedge \mathtt{Pressure} = \mathtt{TooLow}) \vee \\ & \quad (\mathtt{Reset}' = \mathtt{On} \wedge \mathtt{Reset} = \mathtt{Off} \wedge \mathtt{Pressure} = \mathtt{Permitted}) \vee \\ & \quad (\mathtt{Pressure}' = \mathtt{High} \wedge \mathtt{Pressure} \neq \mathtt{High}) \vee \\ & \quad ((\mathtt{Pressure}' = \mathtt{Permitted} \vee \mathtt{Pressure}' = \mathtt{TooLow}) \wedge \\ & \quad \neg(\mathtt{Pressure} = \mathtt{Permitted} \vee \mathtt{Pressure} = \mathtt{TooLow})) \\ \boxed{\mathtt{Overridden}} & \boxed{\text{otherwise}} \end{cases}$$

no change

Defines Overridden as a function of two states

EVENT TABLE: FORMAL DEFINITION

no missing cases

no ambiguity

Mode	Event			
m_1	$e_{1,1}$	$e_{1,2}$	$\cdots$	$e_{1,p}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
m_n	$e_{n,1}$	$e_{n,2}$	$\cdots$	$e_{n,p}$
r_i	v_1	v_2	$\cdots$	v_p

Each **event table** describes the value of a controlled variable or term r_i as a relation ρ_i on modes, events, and values:

$$\rho_i = \{(m_j, e_{j,k}, v_k) \in M_{\mu(i)} \times E_i \times TY(r_i)\}.$$

The relation ρ_i must satisfy the following properties:

1. The m_j are unique; the v_k are unique.
2. For all $j, k, l,\ k \neq l$: $e_{j,k} \wedge e_{j,l} = \mathit{false}$ (**Disjointness:** The pairwise conjunction of the events in each row of the table is always *false*).

The **One Input Assumption** (only one monitored event occurs at a time) and the two properties above guarantee that F_i is a function. The "no-change" part of F_i's definition guarantees totality.

EXAMPLE OF A MODE TRANSITION TABLE

Old Mode	Event	New Mode
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

Based on the above mode transition table and the old and new dependencies sets {WaterPres, Pressure} and {WaterPres}, the function defining the value of Pressure, denoted F_4, is described by

No transitions possible from TooLow to High and vice versa

$$\begin{cases} \texttt{TooLow} & \text{if} \quad \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres}' < \texttt{Low} \wedge \\ & \qquad \texttt{WaterPres} \not< \texttt{Low} \\ \texttt{High} & \text{if} \quad \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres}' \geq \texttt{Permit} \wedge \\ & \qquad \texttt{WaterPres} \not\geq \texttt{Permit} \\ \texttt{Permitted} & \text{if} \quad (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{WaterPres}' \geq \texttt{Low} \wedge \\ & \qquad \texttt{WaterPres} \not\geq \texttt{Low}) \vee \\ & \qquad (\texttt{Pressure} = \texttt{High} \wedge \texttt{WaterPres}' < \texttt{Permit} \wedge \\ & \qquad \texttt{WaterPres} \not< \texttt{Permit}) \\ \texttt{Pressure} & \text{otherwise.} \end{cases}$$

NAT: Pressure = TooLow $\Rightarrow$ Pressure' $\in$ {TooLow, Permitted} $\wedge$ …

MODE TRANSITION TABLE: DEFINITION

A mode transition table with this format which satisfies the four properties is a special case of an event table.

Current Mode	Event	New Mode
m_1	$e_{1,1}$ $\dots$ e_{1,k_1}	$m_{1,1}$ $\dots$ m_{1,k_1}
m_2	$e_{2,1}$ $\dots$ e_{2,k_2}	$m_{2,1}$ $\dots$ m_{2,k_2}
$\dots$	$\dots$	$\dots$
m_n	$e_{n,1}$ $\dots$ e_{n,k_n}	$m_{n,1}$ $\dots$ m_{n,k_n}

A mode transition table describes a mode class r_i as a relation ρ_i on modes, conditioned events, and modes. It is defined by

$$\rho_i = \{(m_j, e_{j,k}, m_{j,k}) \in M_{\mu(i)} \times E_i \times M_{\mu(i)}\}.$$

where E_i is the set comprised of "never" and conditioned events defined on the variables in RF, and each $e_{j,k}$ is an event (or "never") in a row containing mode m_j and a column containing value v_k.

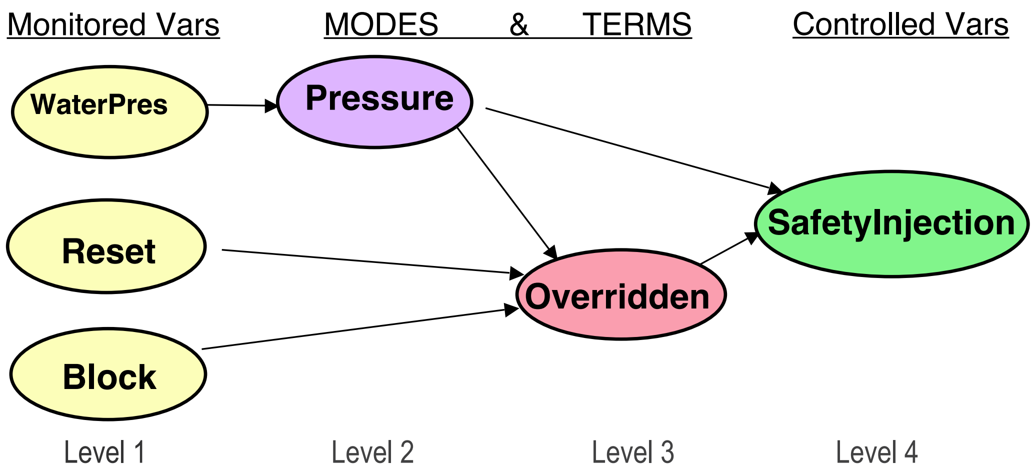
The relation ρ_i has the following properties:

1. The m_j are unique.
2. For all $k \neq l$, $m_{j,k} \neq m_{j,l}$.
3. For all j and for all k, $m_j \neq m_{j,k}$ (**No Self-Loops**).
4. For all $j, k, l,\ k \neq l$: $e_{j,k} \wedge e_{j,l} = false$ (**Disjointness:** The pairwise conjunction of the conditioned events in each row of the table is always *false*).

PARTIAL ORDERING OF THE VARIABLES

- Based on their dependencies, the state variables may be **partially ordered**.
  - Each monitored variable is independent of any other variable, including other monitored variables
  - Each mode class can only depend on the monitored variables, the mode classes and terms preceding it in the partially order, and similarly each term …
  - Each controlled variable can depend on the monitored variables, mode classes, terms, and any controlled variables that precede it in the partial order
- Thus the variables in RF can be ordered as a sequence R, a **topological sort** of RF, based on their dependencies

DEPENDENCIES AMONG THE VARIABLES



TRANSFORM FUNCTION

The system transform T is defined using a series of value functions V_i and a series of partial states z_i. The *partial states* z_i are defined by

$$z_i = \begin{cases} \emptyset & \text{for } i = 1 \\ z_{i-1} \cup \{(r_{i-1}, V_{i-1}(e,s))\} & \text{for } i = 2, 3, \ldots, P+1. \end{cases}$$

The complete new state z_{P+1} is computed by computing each z_i in turn.

If r_i is a monitored variable, the value function V_i is defined by

$$V_i(e,s) = \begin{cases} v & \text{if } r_i = r \\ s(r_i) & \text{otherwise.} \end{cases}$$

If r_i is defined by a condition table function F_i, the value function V_i is defined by

$$V_i(e,s) = F_{i,z_i} \, ,$$

where F_{i,z_i} denotes the evaluation of the single-state function F_i in partial state z_i.

If r_i is defined by an event table function F_i, the value function V_i is defined by

$$V_i(e,s) = F_{i,s,z_i} \, ,$$

where F_{i,s,z_i} denotes the evaluation of the two-state function F_i in state s and partial state z_i.

The system transform T is defined by the $(P+1)$st partial state. That is, $T(e,s) = z_{P+1}$.

GUARANTEED PROPERTIES OF THE TRANSFORM T

Lack of circularity and the conditions that the tables must satisfy guarantee important properties of the transform T:

1. T is **complete**: For each monitored event that may occur, at least one new system state is completely defined
2. T is **deterministic**: For each monitored event that may occur, at most one new system state is defined

REQUIREMENTS TOOLSET

AUTOMATICALLY GENERATING INVARIANTS

PROBLEM IN THEOREM PROVING

AUXILIARY INVARIANTS → How to obtain these??

State Machine Σ → THEOREM PROVER → Property p is an invariant

- Proving p invariant often fails without the aid of auxiliary invariants
- Major difficulty: Finding strong enough auxiliary invariants so that the proof succeeds

ONE SOLUTION

Automatically construct state invariants from specs

SCR GOAL: MAKE 'FORMAL METHODS' PRACTICAL

- Scalable tabular notation
- Integrated set of software tools
  - light-weight tools (easy to use)
  - heavy-duty tools (e.g., theorem prover)

INCREASING EFFORT, INCREASED EXPERTISE

SPECIFY THE SYSTEM PRECISELY

Use a TABULAR notation with an explicit formal semantics to specify the required behavior

APPLY "CONSISTENCY CHECKING"

Automatically check spec for syntax/type errors, missing cases, nondeterminism, circular defs, etc.

SIMULATE THE SYSTEM BEHAVIOR

Symbolically execute the system based on the (executable) req. specs

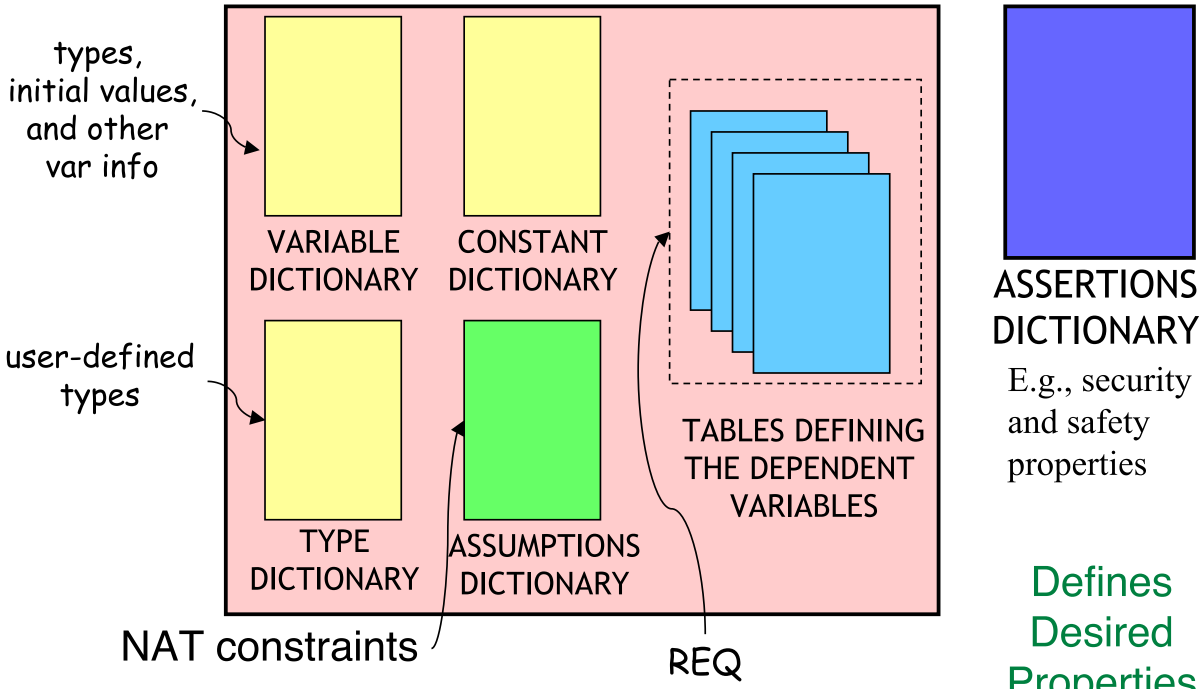
VERIFY SPECS USING MODEL CHECKING

Check critical application properties

VERIFY SPECS USING THEOREM PROVING

As we move down the chain, we increase assurance in the spec

AN SCR REQUIREMENTS SPEC: A COLLECTION OF TABLES



Defines Desired Properties

Defines the Requirements Model

CONSISTENCY CHECKING

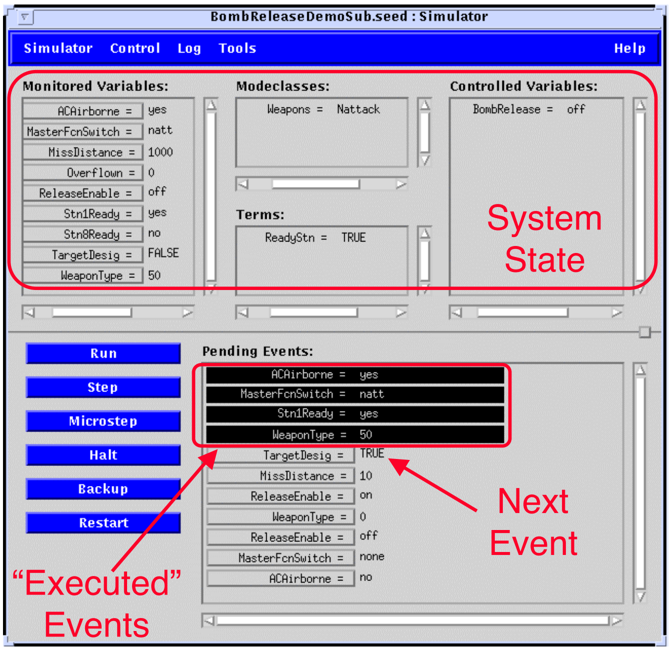
- Checks *well-formedness* of the spec
  - Does the spec satisfy the formal model?
  - CC checks spec for *application-independent* properties, including properties required of the tables
  - Is the spec syntax-correct, type-correct, …?
- Analyzing Disjointness and Coverage
  - Check that certain logical formulas defined on conditions and events are **tautologies**; e.g., given a condition table

    Disjointness: Check that the entries c_1 and c_2 in each pair of cells in each row satisfy $\boxed{c_1 \wedge c_2 = false}$

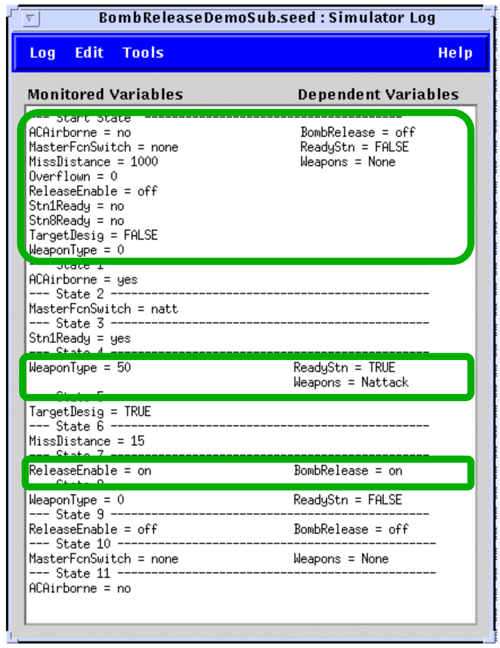
    Coverage: Check that the entries in each row satisfy $\boxed{c_1 \vee c_2 \vee \cdots \vee c_n = true}$

USING THE SIMULATOR FOR VALIDATION

Simulator Display



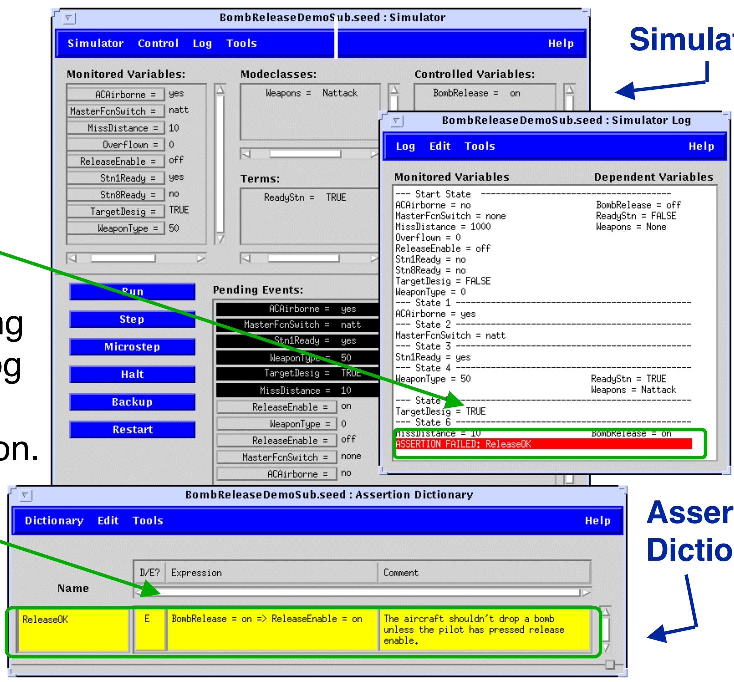
Simulator Log



Monitored Vars Dependent Vars

CHECKING ASSERTIONS WITH THE SIMULATOR

At step 4, the simulator has detected a violation of an assertion. Clicking on the warning msg. in the log highlights the failed assertion.



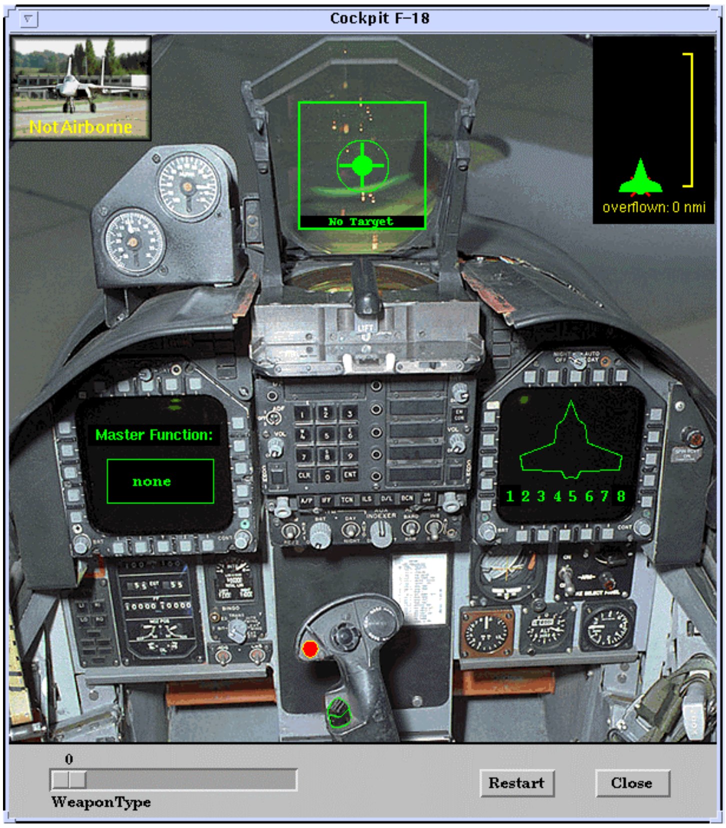
Simulator Display

Simulator Log

Assertion Dictionary

ASSERTION: **BombRelease = on ⇒ ReleaseEnable = on**

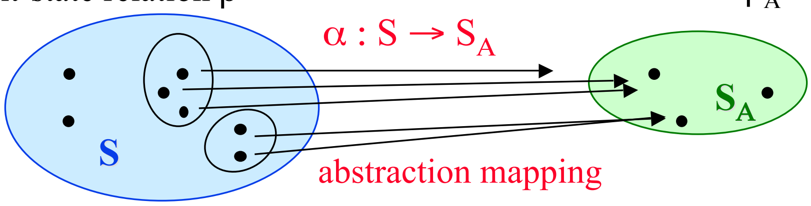
A FRONT-END FOR THE SIMULATOR



MODEL CHECKING SCR SPECS

State machine Σ
state space S
initial state predicate Θ
next-state relation ρ

Abstract machine Σ_A
state space S_A
initial state predicate Θ_A
next-state relation ρ_A



property q

property q_A

- Given a property q, we want the following to hold:
  - q_A is an invariant of Σ_A implies that q is an invariant of Σ *(soundness)*
  - q is an invariant of Σ if q_A is an invariant of Σ_A *(completeness)*
    (thus, a counterexample to q_A is found in Σ_A implies q is not an invariant of Σ)
- Two kinds of invariants of interest ← ALWAYS PROPERTIES
  - properties of each reachable state (*state invariants*)
  - properties of each pair of reachable states in relation ρ (*transition invariants*)

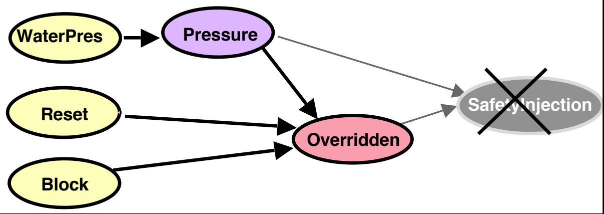
NEXTIME PROPERTIES →

THREE AUTOMATABLE ABSTRACTION METHODS

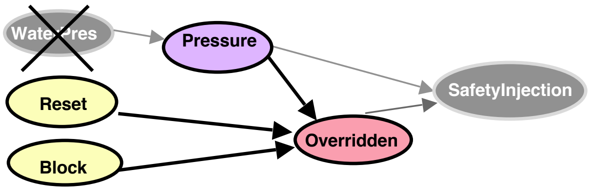
Reset = On ∧ Pressure ≠ High ⇒ ¬Overridden

Opening the vent valve shall be prevented unless the differential pressure is within safe limits

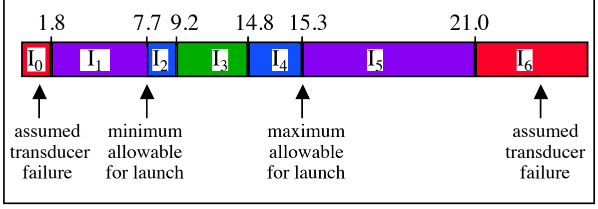
ABSTRACTION METHOD 1: REMOVE IRRELEVANT VARIABLES



ABSTRACTION METHOD 2: USE EXISTING DATA ABSTRACTIONS



ABSTRACTION METHOD 3: CREATE NEW DATA ABSTRACTIONS



- Eliminate variables irrelevant to the validity of the property
- Remove unneeded detail

STATE INVARIANTS

Definition of a state invariant: a property that holds in every reachable state of a state machine model

Form of the state invariants that our algorithms generate

$$\boxed{v = a_i \Rightarrow q_i}$$

v is any dependent variable in the spec

Mode invariants are a special case

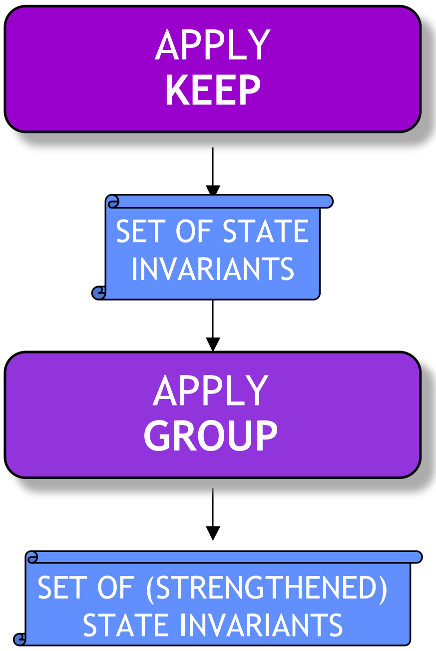
$$\boxed{M = m_i \Rightarrow q_i}$$

M is a mode class

TWO ALGORITHMS FOR CONSTRUCTING STATE INVARIANTS

Apply KEEP, a fixpoint algorithm, to construct initial set of invariants

Use this set of invariants and other information generated by KEEP in applying GROUP



MODE TRANSITION TABLE FOR AUTOMOBILE CRUISE CONTROL

Row	Old Mode	Event	New Mode
1	Off	@T(IgnOn)	Inactive
2	Inactive	@F(IgnOn)	Off
3	Inactive	@T(Lever=const) WHEN EngRunning AND NOT Brake	Cruise
4	Cruise	@F(IgnOn)	Off
5	Cruise	@F(EngRunning)	Inactive
6	Cruise	@T(Brake) OR @T(Lever=off)	Override
7	Override	@F(IgnOn)	Off
8	Override	@F(EngRunning)	Inactive
9	Override	@T(Lever=resume) WHEN NOT Brake OR @T(Lever=const) WHEN NOT Brake	Cruise

Initially: M=Off AND NOT IgnOn and NOT EngRunning

Exit Off
Enter Off

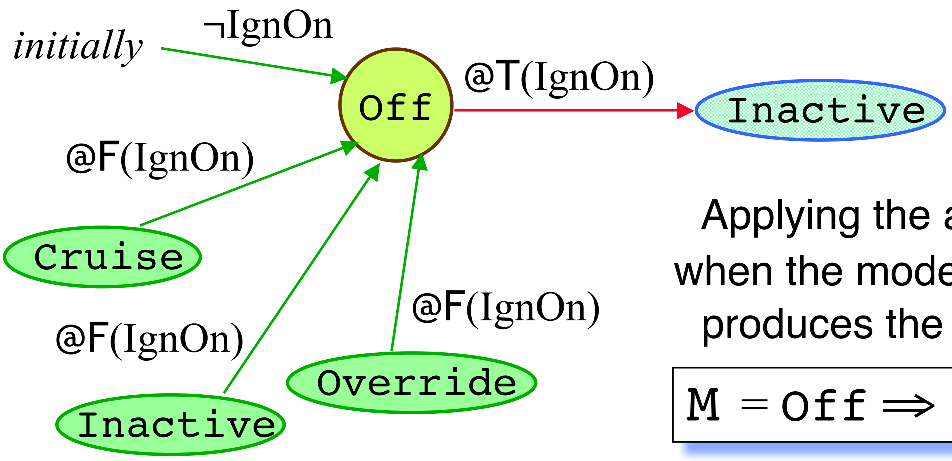
PROBLEM: Find a mode invariant of mode Off

BASIC RULE FOR GENERATING MODE INVARIANTS

BASIC RULE

q is a *mode invariant* of mode m if

1) *q* is true upon entry into mode m (*q* is also true initially if m is an initial mode)
2) Occurrence of event @F(*q*) forces unconditional exit from m



Applying the algorithm when the mode `M` is `Off` produces the invariant

$$\texttt{M} = \texttt{Off} \Rightarrow \neg\texttt{IgnOn}$$

KEEP, AN ALGORITHM FOR GENERATING MODE INVARIANTS

Compute Mode Entry Conditions:
C is the disjunction $c_1 \vee c_2 \vee \ldots \vee c_m$ of conditions true when mode entered

Compute Unconditional Exit Set:
Set $\{d_1, d_2, \ldots, d_n\}$ of simple Boolean conditions whose falsification causes unconditional exit from mode

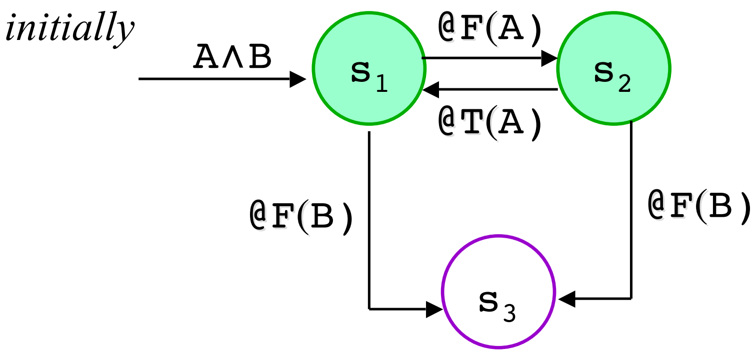
To strengthen invariant, use
- *initial state predicate*
- *environmental constraints*
- *invariants computed on earlier passes*

Compute Mode Invariant
Keep just the d_i found in C

REPEAT THESE THREE STEPS UNTIL A FIXPOINT IS REACHED

Jeffords and Heitmeyer, *FSE98*

APPLYING GROUP: A SIMPLE EXAMPLE

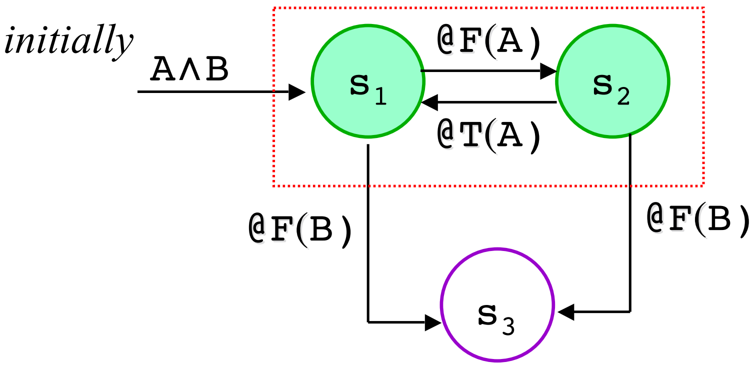


Applying **KEEP**, it is easy to show that

- A is an invariant of s_1
- ¬ A is an invariant of s_2

BUT **KEEP** does not produce any interesting results about B

APPLYING GROUP: A SIMPLE EXAMPLE



Applying **GROUP**, we consider sets of states

If we consider the set of states G = $\{s_1, s_2\}$

- The uncond. exit set of G is B
- The mode entry condition of G is A∧B
- Hence B is an invariant of s_1 and s_2

INVARIANTS CONSTRUCTED BY KEEP

APPLY
KEEP

M=Off ⇒ ¬IgnOn
M =Cruise ⇒ ¬Brake ∧ Lever ≠ Off
M =Override ⇒ *true*
M =Inactive ⇒ *true*

GROUP STRENGTHENS INVARIANTS CONSTRUCTED BY KEEP

APPLY
KEEP

M=Off ⇒ ¬IgnOn
M =Cruise ⇒ ¬Brake ∧ Lever ≠ Off
M =Override ⇒ *true*
M =Inactive ⇒ *true*

APPLY
GROUP

M=Off ⇒ ¬IgnOn
M =Cruise ⇒ ¬Brake ∧ Lever ≠ Off ∧ IgnOn ∧ EngRunning
M =Override ⇒ IgnOn ∧ EngRunning
M =Inactive ⇒ IgnOn

APPLYING THE SCR TOOLS IN PRACTICE