SC(R)^3: Towards Usability of Formal Methods

Marsha Chechik
Department of Computer Science
University of Toronto

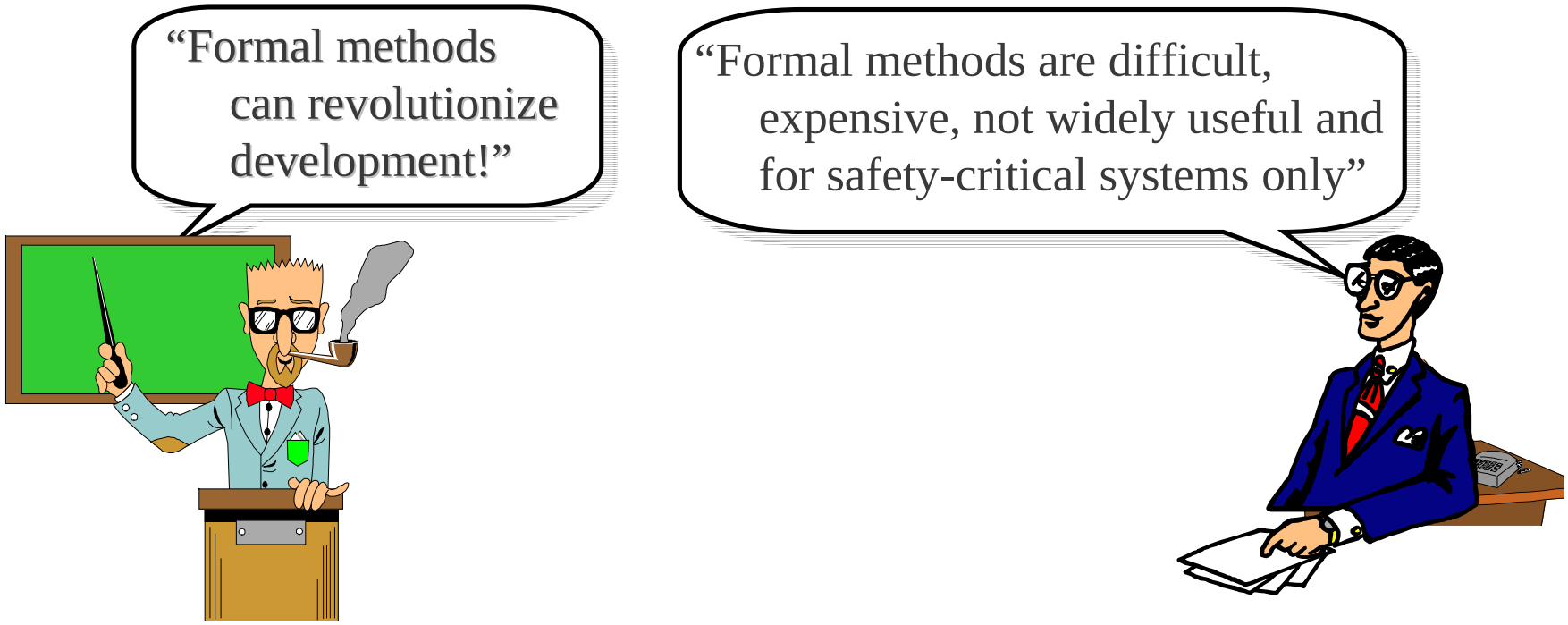
December 2, 1998
CASCON'98

Formal Methods

"The use of mathematical modeling, calculation and prediction in the specification, design, analysis, construction and assurance of computer systems and software."

- Engineering - partial differential equations to model variations in physical quantities over time or space.
- Software - model discrete quantities. Need concepts like sets, graphs, partial-orders, finite-state machines.
- "Calculation" is in terms of formal logic rather than numerical computation.

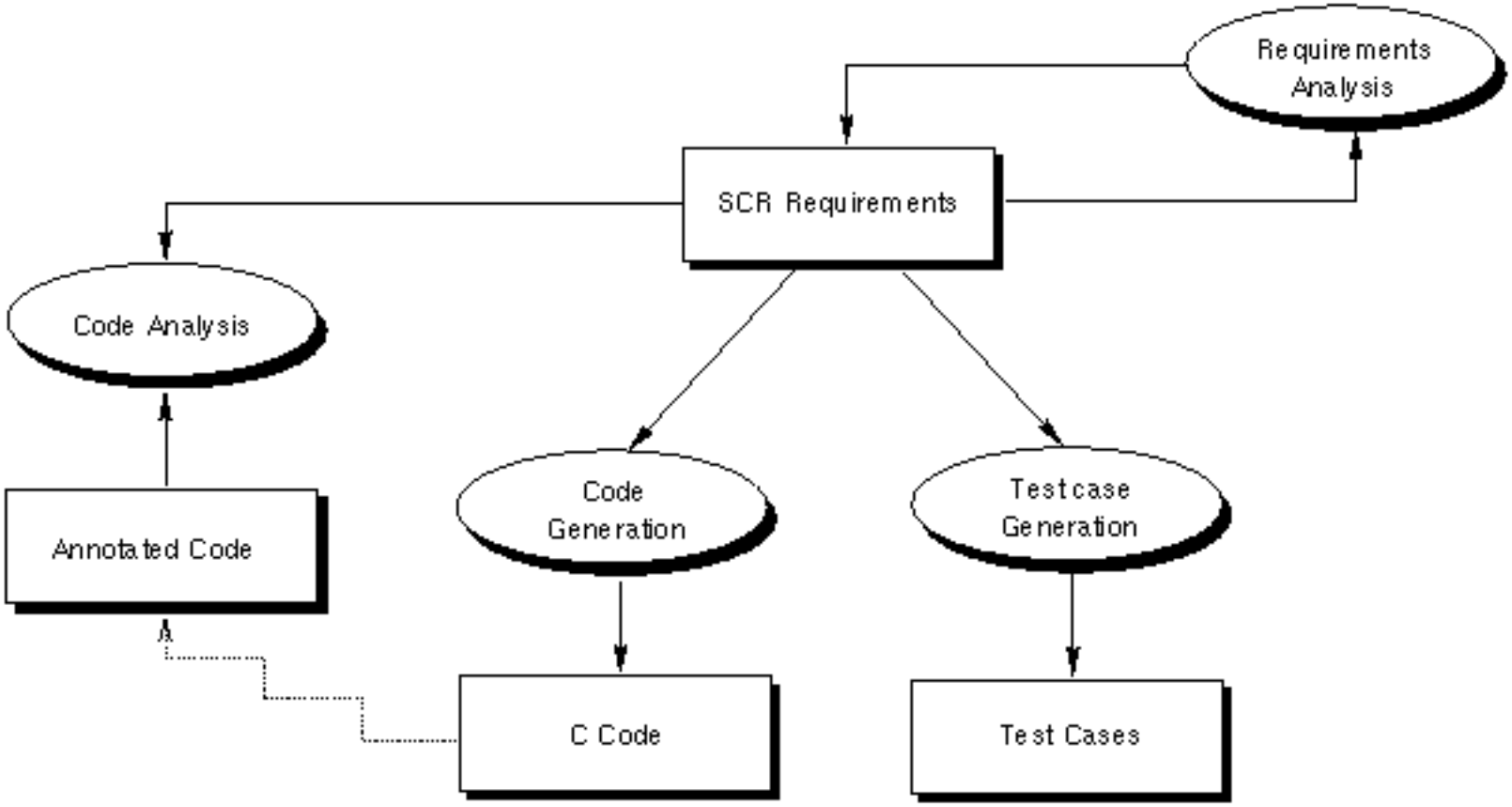
Why aren't FMs used more?



Need to improve *Usability* of Formal Methods

- $$$ - Amortize the cost of creating formal documentation throughout software lifecycle
- Use (create?) easy to read and review notations
  - state machines, tables
- Decrease analysis cost through automation
  - create computer tools requiring minimum user-guidance
- Adopt existing technology wherever possible

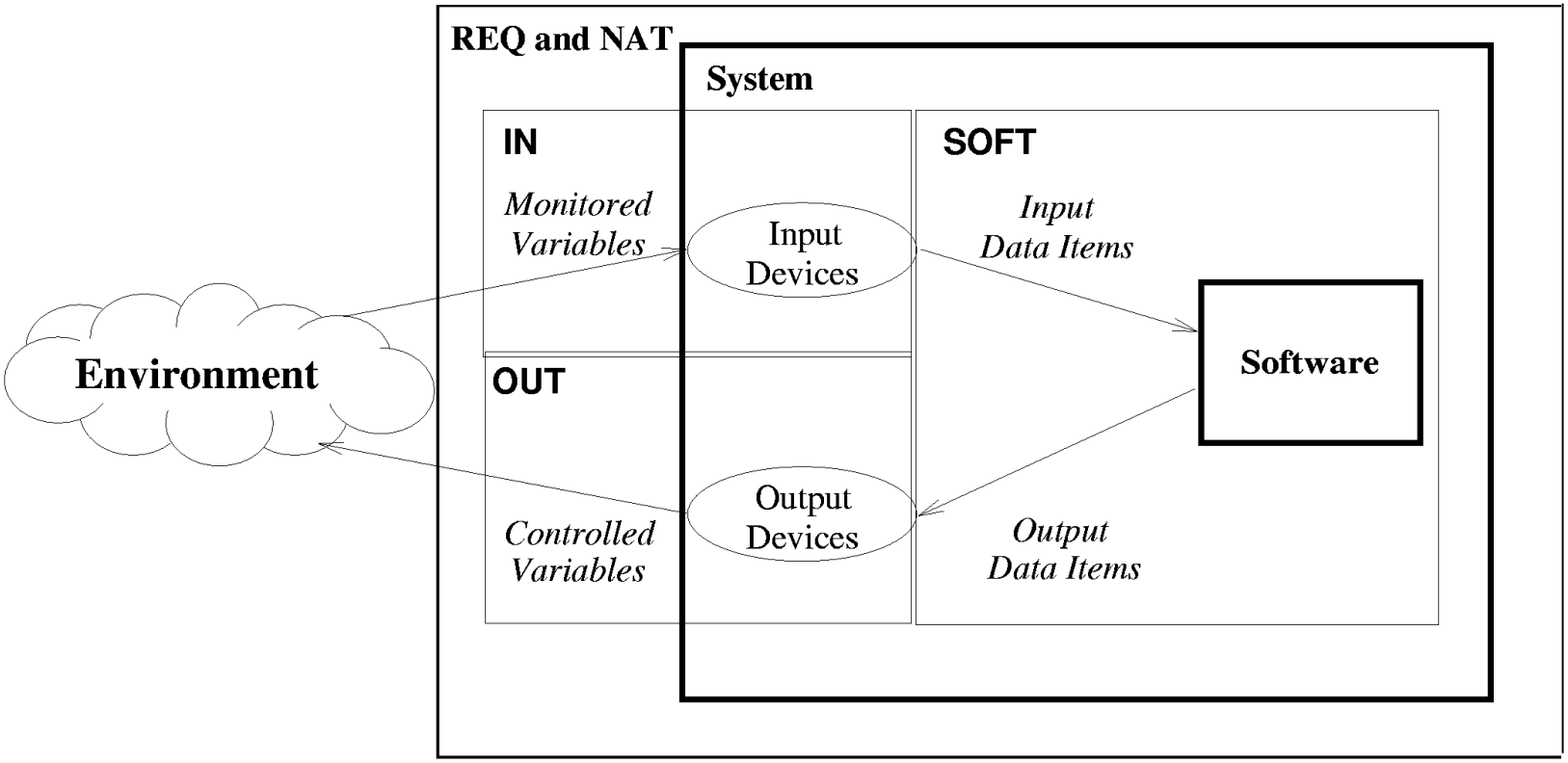
Overview of the Approach



Overview of the Rest of the Talk

- Specifications: SCR and global properties
- Running example: Cruise Control System
- Requirements analysis
  - Translating specifications
  - Verification and counter-example
- Code analysis
  - Annotations
  - Analysis
- Summary and future work

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]

Cruise Control System

- Specified by Jim Kirby in 1988
- Activated by driver when at desired speed
- Speed is maintained until
  - The driver presses the brake pedal
  - The driver presses the gas pedal
  - The car's speed becomes uncontrollable
  - The engine stops running
  - The driver turns the ignition off
  - The driver turns the cruise control off

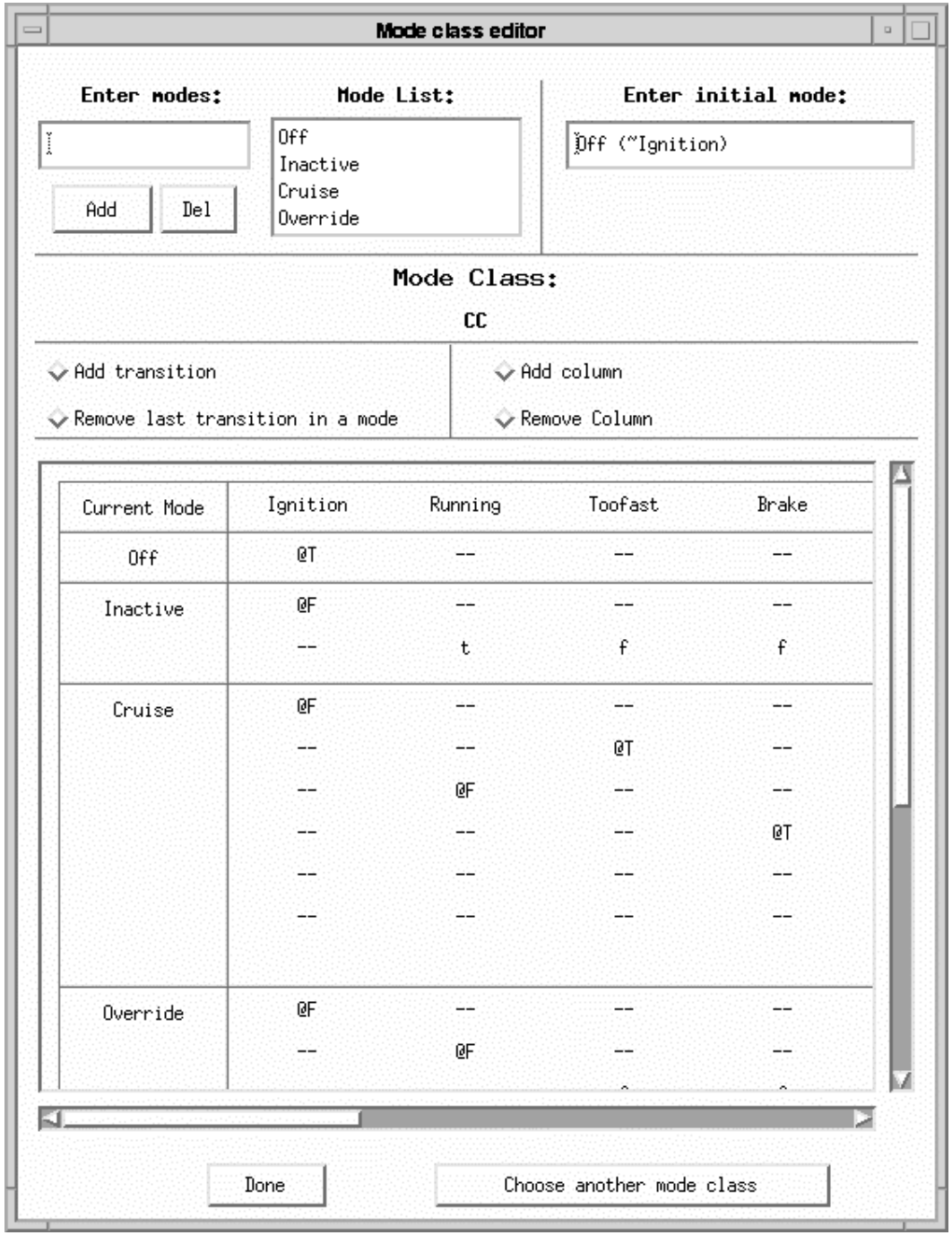
Driver can reactivate system by pressing "Resume" button

SCR Table for Cruise Control System

Current Mode	*Ignition*	*Running*	*Toofast*	*Brake*	*Accel*	*b_Cruise*	*b_Off*	**New Mode**
Inactive	@T	--	--	--	--	--	--	Off
	--	true	false	false	false	@T	--	Cruise
Cruise	@F	--	--	--	--	--	--	Off
	--	--	@T	--	--	--	--	Inactive
	--	@F	--	--	--	--	--	Inactive
	--	--	--	@T	--	--	--	Inactive
	--	--	--	--	@T	--	--	Inactive
Override	@F	--	--	--	--	--	--	Off
	--	@F	--	--	--	--	--	Off
	--	--	false	false	false	--	@T	Inactive
	--	--	false	false	false	@T	--	Cruise
Off	@T	--	--	--	--	--	--	Inactive

Initial: Off (~Ignition)

SC(R)^3 mode class editor



Controlled Variables for Cruise Control System

Mode	True	False
Cruise	@T(speed_slow)	@F(Inmode)
Cruise	@T(Inmode) WHEN [b_Resume & speed_slow]	@F(speed_slow)

Initial: False

Assumptions

Assumptions specify constraints on the values of conditions, imposed by laws of nature or by other mode classes in the system

- The engine is running only if the ignition is on

```
Running ->> Ignition
```

- The car can be going "too fast" only if it is running

```
Toofast ->> Running
```

- Variables relating vehicle's speed are connected by enumeration

```
speed_slow / speed_ok / speed_fast
```

Properties for Cruise Control System

- Expressed in CTL - Computational Tree Logic + SCR events
- Invariant properties

```
AG(CC=Override -> (Ignition & Running)
  AG(ThrottleOff | ThottleMaintain |
     ThrottleAccel | ThrottleDecel)
```

- "Progress" properties - system behaves according to specifications

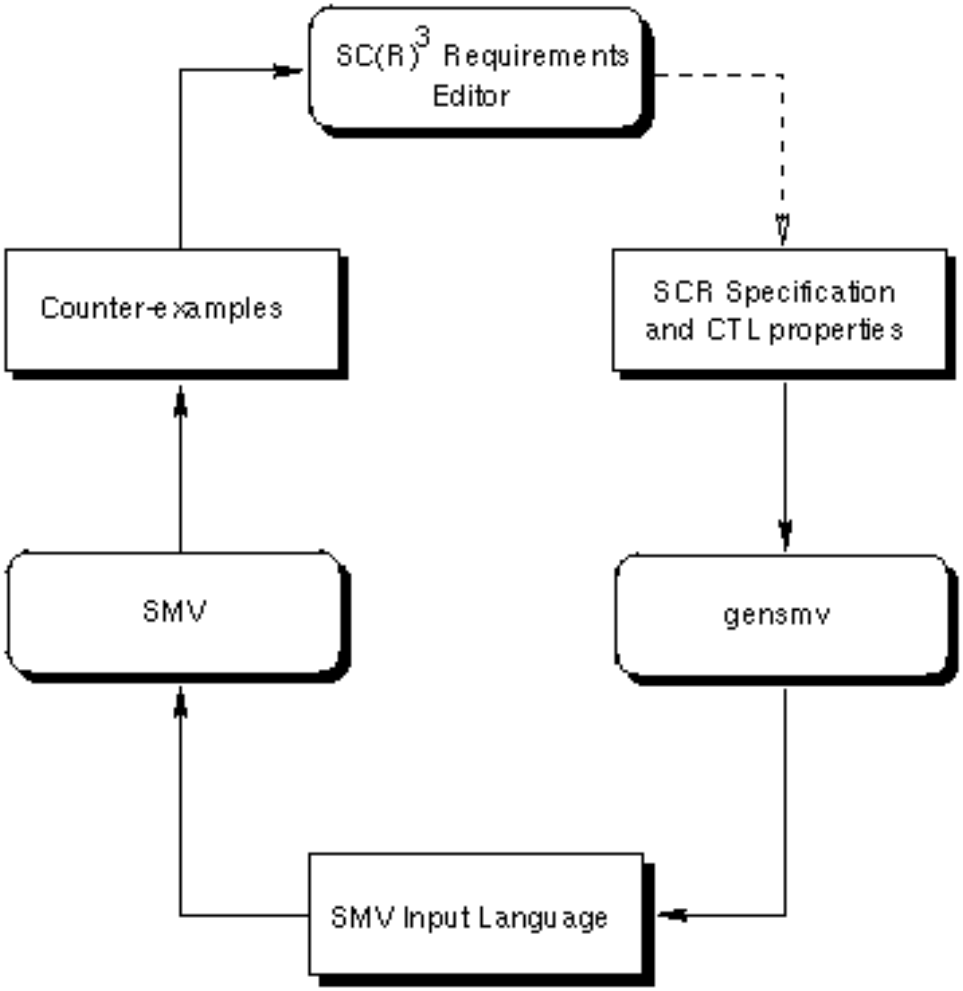
```
AG(CC=Cruise ->
   AX(@F(Ignition) -> CC=Off)
```

Model-Checking

- Model-checker SMV [McMillan93]
  - Takes CTL formulas and a model
  - Uses state exploration
  - Verifies properties or gives counter-examples

- Need to create input to SMV, translate properties and interpret results

Requirements Analysis



Translating SCR Specifications

- Tool `gensmv` [Atlee,Sreemani 96]
  - Detail mode transition tables with information derived from environmental assumptions

    ```
    Running ->> Ignition
    ```

  - Translate into `SMV`. Express @T and @F by introducing additional variables to store previous values

    ```
    AG(CC=Cruise ->
     AX((Pignition & ~Ignition)-> CC=Off)
    ```

  - We extended `gensmv` to reason about controlled variables

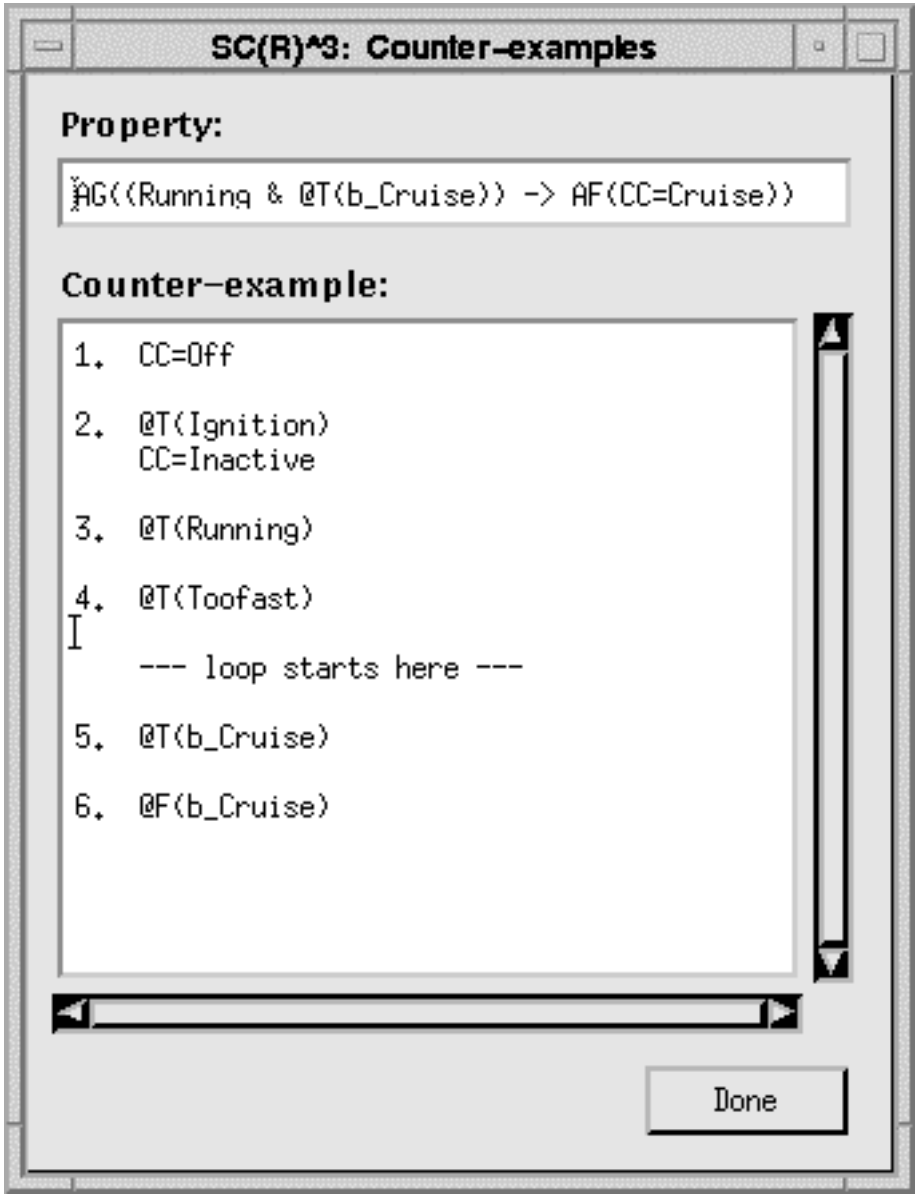
Counter-Example Facility

- Want to make SMV completely transparent for users
- Capture SMV's counter-examples, translate into SCR

Example property:

```
AG(Running &
@T(b_Cruise)) ->
AF(CC=Cruise)
```

Property fails when the car is running too fast



Checking Cruise Control System

- Checking done on SPARCstation 20 (2 x 75 MHz processors), 256 megabytes of RAM, using SMV version 2.4
- Size of system:
  - 22 monitored and 13 controlled variables
  - translated into 47 SMV variables
- To make the analysis more tractable:
  - "Slicing" - remove variables that do not appear in any properties and that others do not depend on (by hand)
  - Variable reordering
    - option `-reorder` (default in version 2.5 of SMV)
  - Building state-space incrementally (`-f`, `-inc`)

Checking Cruise Control System

Test	User time	System time	BDD nodes	Total number of vars
Full specification				
-reorder	26487.25s	420.31s	34315278	47
Throttle properties				
Base	5.67s	0.2s	112687	26
-f -inc	11.06s	0.07s	18714	26
-reorder	5.69s	0.15s	75356	26
-f -inc -reorder	12.28s	0.1s	18830	26
All but throttle				
Base	8102.76s	50.59s	9780674	43
-f -inc	18783.8s	263.46s	3408360	43
-reorder	4237.56s	7.37s	4425504	43
-f -inc -reorder	13599.5s	12.54s	3408360	43

- Use both options on a fast machine with limited memory
- Use reorder alone on a slower machine

Errors in Controlled Variable Tables

- [Orig. Spec.]: `OilOn` is evaluated when we enter any mode other than `Off`

  `@T(Inmode) WHEN [Omiles_low]`

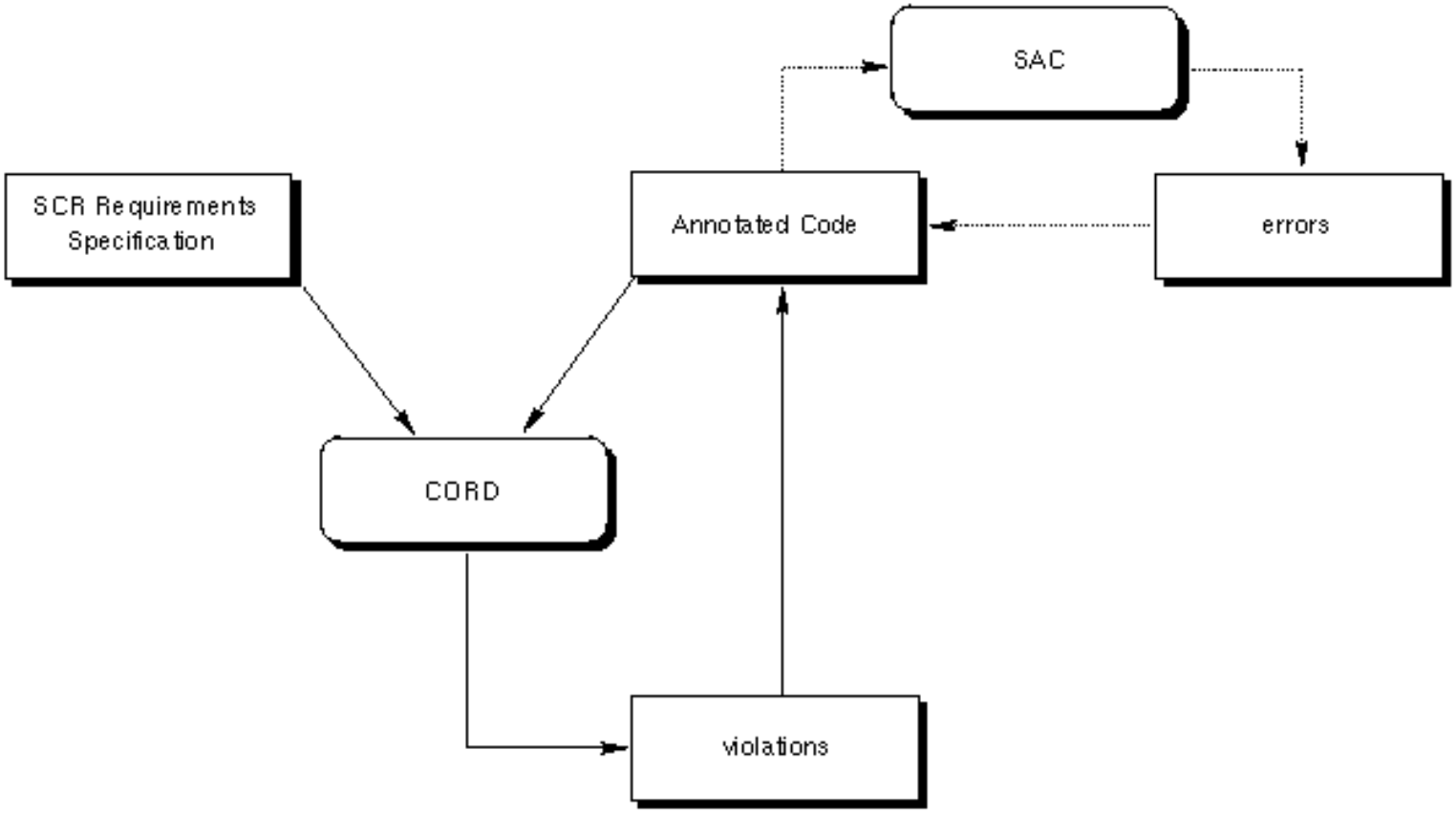
- [Orig.Spec.] `Throttle` never becomes `Off`

Throttle Setting				
Mode	Condition			
Cruise	speed_slow	speed_ok & ~Accel	speed_fast & ~Accel	Accel & ~ speed_slow
Throttle	Accel	Maintain	Decel	Off

  - Property

    `~Throttle = Off -> AG(~Throttle=Off)`

Checking Correctness of Code



Annotation Language

- Goal: annotate code with statements which describe changes in system state on the requirements level.
- Statements involve local (rather than invariant) info
- Language constructs:
  - Initial: assigns values to variables in starting state
  - Update: assigns values to variables indicating a state change
  - Assert: documents developer assumptions about the system at a given point and helps make analysis more precise

Annotations (Cont'd)

```
If (vIgnition) {
    @@ Assert ~Ignition;
     vIgnition = true;
    @@ Update Ignition;
    vCC = mInactive;
    @@ Update CC=Inactive;
}
else {
    @@ Assert Ignition;
    vIgnition = false;
    @@ Update ~Ignition;
    vCC = mOff;
    @@ Update CC=Off;
}
```

Checking Consistency of Annotations

- "Analysis is as good as annotations"
- Code maintenance greatly reduces correspondence between code and annotations
- Tool `sac`
  - Create a list of correspondences between req. and code variables

```
correspondences:
{Ignition} -> {vIgnition}
{CC} -> {vCC}
```

  - `sac` ensures that an assignment (or check) of a code variable is always followed by an Update (an Assert) of an appropriate requirements variable

Analysis by `cord`

- Check that transitions in the code are exactly the same as specified in requirements:
  - The code and spec start out in the same initial state
  - The code implements *all* specified transitions (ALT properties)

  E.g., "there is a transition from `Cruise` to `Off` on event `@F(Ignition)`"
  - The code implements *only* specified transitions (OLT properties)

  E.g., "The only transitions into `Off` are from modes `Inactive`, `Cruise` or `Override` on event `@F(Ignition)`"

Summary of Analysis

- Static analysis (similar to constant propagation) on annotations and control-flow statements in code
- Aggressive state-folding strategy aimed at minimizing the number of states (bound above by #annotations + # control-flow contracts)
- Automatically generate OLT and ALT properties (involving 2-3-state fragments => efficient checking)
- Also can check invariants and reachability properties and violations of environmental assumptions

Summary of Analysis (Cont'd)

Verification is inexact: `cord` computes more behaviors then can be present in code

- OLT props and invariants are checked *pessimistically* (more errors are found then present)
- ALT and reachability properties are checked *optimistically* (some errors can not be reported)

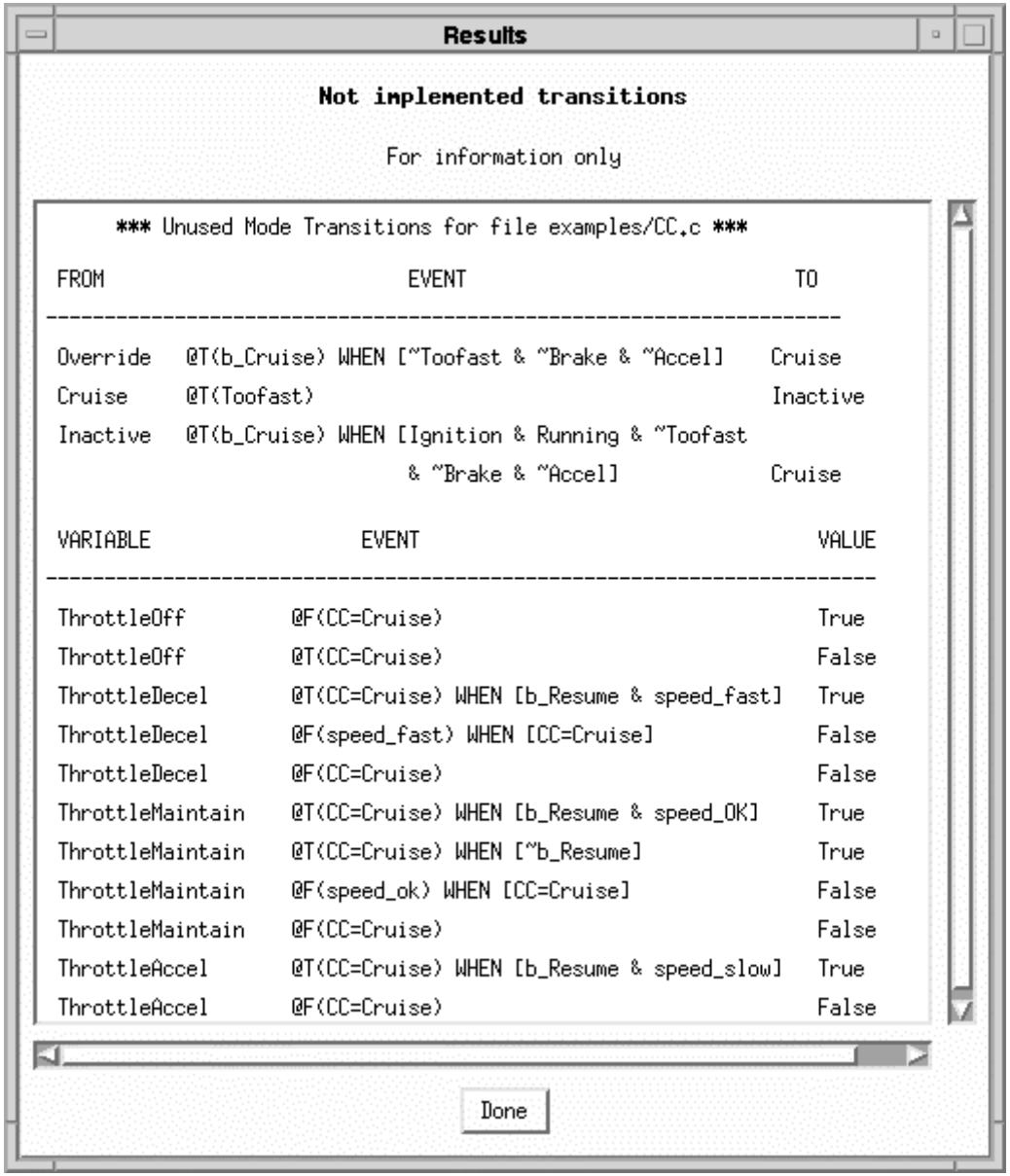
Thus, use `cord` as a debugging rather than verification tool

- So far - booleans and enumerated types, but current work is on *abstract interpretation* of code (i.e. not annotations)
  - approximation of infinite types
  - operations like addition and comparison

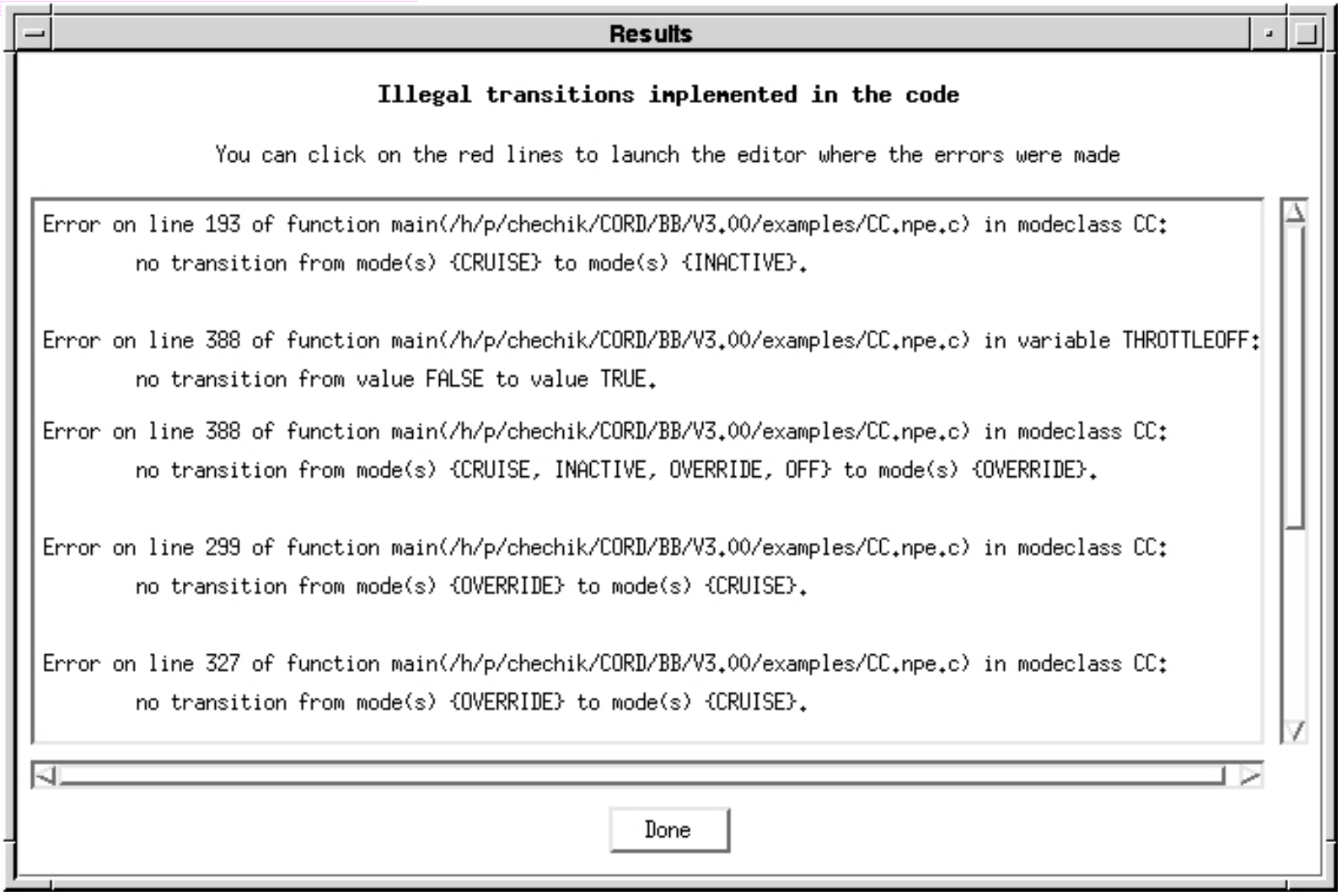
Checking Cruise Control System

- Implementation in C and Xlib, 675 lines of code (380 for GUI)
- Code annotated in 40 minutes with 37 Update, 25 Assert and 1 Initial annotation
- Analysis done in 7.52 sec (3.45 user, 0.74 system) on SPARCstation-4 (85 MHz processor) with 64 Mb of RAM
- Resulted in 14 ALT and 5 OLT violations:
  - Incorrect annotation (overlooked by `sac`)
  - Incorrect transition in code (to `Override` instead of `Cruise`)
  - Condition missing in code
  - One false negative

Violations of ALT Properties



Violations of OLT Properties



Summary

- SC(R)^3 - an integrated toolset for specification and reasoning about tabular requirements
- Check requirements for correctness
- Analyze consistency between annotated code and requirements
- Current work on test-case generation and generation of "usable" code

Step towards increasing usability of formal methods:

- Replace free-form logic by easy-to-read/review tables
- Amortize the cost of creating formal requirements over multiple automated analysis activities

New Process

- (Human) requirements designer provides specification and properties
- SC(R)^3 does model-checking of specification against properties
- SC(R)^3 generates an implementation of mode logic and stubs for interface code
- Using spec, SC(R)^3 generates test cases (for interface testing)
- Implementation analyzed for performance and (optionally) manually fine-tuned
- Code is checked for consistency with specification

Future Work

- Requirements analysis
  - model-checking (SMV) vs theorem-proving (PVS)
  - timing properties, data vs control, infinite-domain
- Code analysis
  - More expressive annotation language
  - Infinite-domain variables
  - Combining abstract interpretation with theorem-proving
- Test-case generation, code generation