FORTE/PSTV, 9–12 Tuesday 18 November 1997, Osaka

Specification, Proof Checking, and Model Checking for Protocols and Distributed Systems with PVS

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, California, USA

Overview

- Goal

  Practitioners: gain an appreciation of the utility and characteristics of theorem proving and model checking, and of their integration

  Researchers: see something of the research challenges in this area

- Agenda
  - Motivation for mechanized formal methods
  - Brief outline of theorem proving and model checking
  - Their strengths, weaknesses, and how they can be used together
  - Thumbnail sketch of PVS
  - Example: simple cache-coherence protocol
  - Demonstration of PVS on the example

Formal Methods

- Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems
- But the primary benefit from specifically *formal* methods is that they allow certain questions about a design to be answered by symbolic calculation
  (e.g., by theorem proving or model checking)
- These symbolic calculations can be used for debugging and design exploration as well as post-hoc verification
- Comparable to the way computational fluid dynamics is used in the design of airplanes and jet engines
- Especially valuable when behavior is complex, as with distributed, concurrent systems—i.e., protocols

Formal Methods and Calculation

- Using calculations, some activities that are traditionally performed by *reviews*
  - Processes that depend on human judgment and consensus

  can be replaced or supplemented by *analyses*
  - Processes that can be repeated and checked by others, and potentially so by machine
- Good for assurance and certification
- The singular benefit of formal techniques such as model checking and theorem proving over simulation, animation, prototyping, testing, is that they can examine *all* behaviors (of a possibly simplified model)
- Good for debugging

Formal Methods and Tools

- To perform formal calculations efficiently and reliably, you need automated tools
- And the tools must be sufficiently powerful that they can deal with problems of significant interest
- Just like any other engineering calculations, it's tools that make formal calculations feasible in practice
- Tools are not the most important thing about formal methods
  - They are the *only* important thing
- The principal techniques underlying formal methods tools are model checking and theorem proving

Model Checking: What Is It?

- Specify
  - The system as a finite labeled transition system
  - A required property as a formula in a temporal logic
- Establish that the system is a model of the formula

  $$\mathit{system} \models \mathit{requirement}$$

  Accomplished by a search over the state space
- Usually no background theories (sometimes limited arithmetic)
- Usually, the only assumptions concern interpretation of liveness properties (branching vs. linear time, fairness)
  - Built in to choice of logic and selection of operators
- Works best on control-intensive systems w/small state spaces

Model Checking (continued 1)

- Dominant approach uses Computation Tree Logic (CTL)
  - Mechanized by SMV
- A branching time logic: characterizes properties of states in the tree that represents possible paths in a computation
  - $AG(p)$: p is true everywhere along each path (invariant)
  - $AF(p)$: p is true somewhere along each path
  - $EG(p)$: p is true everywhere along some path
  - $EF(p)$: p is true somewhere along some path

  Also "next" and "until" operators, plus "fair" variants (include only paths where some specified formula is true infinitely often)

Model Checking (continued 2)

- Linear time logic (mechanized by SPIN) characterizes properties of individual computations (i.e., sequences of states)
  $\Box$: henceforth; $\Diamond$: eventually
- Variations (see CAV'96 for descriptions):
  - ω-regular language inclusion (Cospan)
  - Bisimulation in process algebras (Concurrency Workbench)
  - Refinement in CSP (FDR)
  - Reachability analysis (Petri nets, Murϕ, Lotos)
  - Hybrid systems (HiTech, Uppaal)

Model Checking (continued 3)

- Most approaches can be represented in μ-calculus
- Two Variants
  - Park's: quantified boolean logic with predicate and relation symbols plus least and greatest fixpoint operators
  - Kozen's: modal logic with fixpoint operators

Model Checking Issues

- Synchronous or asynchronous transition systems?
  - Synchronous: all components make their transitions in lockstep (natural for hardware with common clock)
  - Asynchronous: transitions of different components are interleaved arbitrarily (natural for distributed algorithms)
  - Each can represent the other
- Which logic?
  - Model checking for branching time logics can be linear in length of the formula and the size of the state space
  - For linear time, it's exponential in length of the formula
  - But the formula is usually short, and many liveness properties cannot be stated in branching time logics
- Main issue is size of state space: the "state explosion problem" (exponential in number of components)

Model Checking Techniques: Explicit State Enumeration

- Explicit state enumeration systems can generally build state space incrementally ("on the fly") and halt when error found
- Breadth-first search produces short counterexamples
- Can use hashing ("supertrace" in SPIN) to avoid storing previously explored states
  - But may fail to explore full state space (due to collisions)
- Can use partial order methods to avoid enumerating computations that differ only in order of interleaving of independent operations
- And symmetry reductions to avoid enumerating states that differ only in irrelevant ways (e.g., which processor has failed, when all that matters is how many have failed)
- Can typically explore millions of states in a few hours

Model Checking Techniques: Symbolic Methods

- "Symbolic model checking" (e.g., SMV) represents states symbolically using Ordered Binary Decision Diagrams (OBDDs)
- Counterexamples can still be produced
- Many explicit states may have same symbolic representation
- Can allow huge state spaces to be examined (10^{120})
- Finding good variable ordering is crucial (can take an hour, may be no good ordering)

Model Checking: Pros and Cons

- Often cannot express full requirements
  - Instead, check several smaller properties
- Few real systems have sufficiently small state space to allow direct checking
  - Must generally abstract them
  - Or "downscale" them (aggressive simplification)
- Largely automatic (modulo BDD orderings)
- Produces counterexamples
- Generally used for debugging
- Find more problems by exploring *all* the behaviors of a downscaled system than by testing only *some* of the behaviors of full system

Theorem Proving: What Is It?

- Specify
  - The system (at some suitable level of abstraction)
  - A required property
  - The assumptions
  - Necessary background theories

  as formulas in a single logic
- And prove that

$$\textit{background} + \textit{assumptions} + \textit{system} \vdash \textit{requirement}$$

- Variation:
  - Prove that implementation is refinement of specification (i.e., commutative diagram, or theory interpretation)
- Works best on data-intensive systems

Theorem Proving Techniques and Issues—I

- How rich a logic?
  - Easier to automate restricted logics (e.g., unquantified)
  - But can make for awkward specifications
- Interactive guidance vs. automation
  - You need both; the issue is *balance*
- Automation: decision procedures
  Essential for ubiquitous theories
  - Propositional calculus
  - Equality over uninterpreted function symbols
  - Linear arithmetic over rationals and integers
  - Function updates (aka. overriding, arrays, stores)
    i.e., f **with** $[(x) := y](z) \stackrel{\text{def}}{=}$ **if** $z = x$ **then** y **else** $f(z)$
  - Need to decide the *combination* of these theories

Theorem Proving Techniques and Issues—II

- Automation: Rewriting
  - Provides decision procedures if rules are finite terminating and confluent
  - More often used heuristically
  - Essential
- Automation: Heuristics
  - First order: resolution, etc.
  - Induction: Boyer-Moore, rippling, etc.
  - Nice to have, hard to control, difficult to interpret failure
- Automation: combination
  - Everything needs to work *together*
- Beyond theorem proving: automated invariant generation, abstraction

Theorem Proving: Pros and Cons

- Specialized methods can be very effective on a limited domain (e.g., boolean equivalence checking in CAD tools)
- General methods can usually state and prove any true property given enough time, skill and patience
- Often require too much time, skill, patience
- Many theorem provers are poorly matched to the requirements of formal methods
  1. Lack adequate automation (proof "assistants")
  2. Don't support civilized specification language, nor theories needed for computer science (classical theorem provers)
  3. Do not fail gracefully
     - False conjecture or inadequate heuristic?
     - Do not help pinpoint error

Theorem Proving: Compared with Model-Checking

- The system is not restricted to finite-state (nor to a transition system)
- Can leave details uninterpreted
- The required property is not restricted to invariance or eventuality properties that can be expressed in CTL (for example)
- The assumptions are not restricted to fairness constraints, but can be arbitrary theories
- And arbitrary mathematical theories can be supplied as background (not built-in as, for example, linear constraints are in hybrid model checking)
- The price of this generality is less automation
  - Usually—but not always

So Why The Bad Rap? — Wrong Focus

- Fundamentalism
  - "Proof" means emulating pencil-and-paper arguments
    - ⋆ Heavily interactive; rules out real automation
    - ⋆ Leads to inappropriate foundations
  - Reductionist approach to soundness
    - ⋆ Reduce everything to elementary rules of logic
    - ⋆ Rules out decision procedures and other heavy horsepower
  - Total correctness
    - ⋆ Prove everything from top to bottom
    - ⋆ Doesn't focus on where the benefits are
- Irrelevance
  - Not driven by real examples

So Why The Bad Rap? — Technical Shortcomings

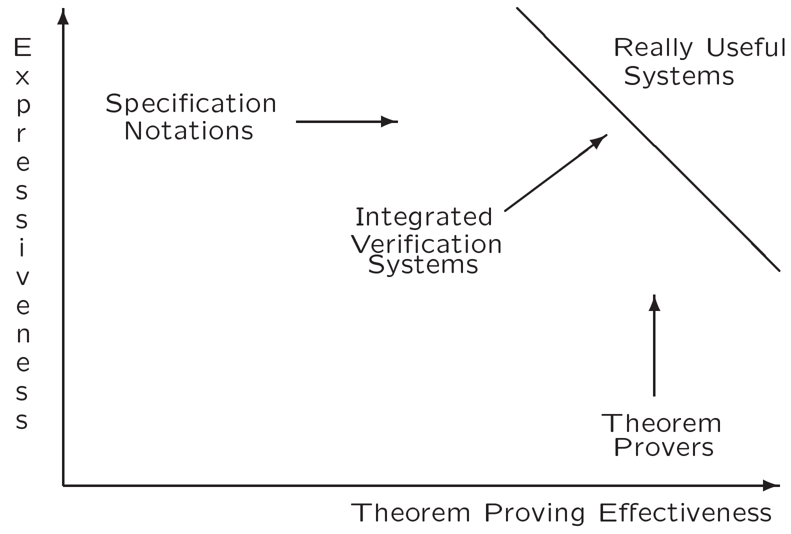
- The field of automated deduction provides important techniques, but has a different focus
  - Its goal is unassisted automation
  - It focuses on proving true theorems
  - It automates raw logics, not specification languages
  - Its applications are math and AI

  Pure theorem provers are like dragsters; mechanizing formal methods is more like Formula 1—many more challenges
  - But must go really fast when it *is* flat and straight
- Nonetheless, most essential techniques were known by 1980
  - Didn't know how best to combine them
  - Didn't know how to balance automation vs. interaction
  - Nor mechanization vs. expressiveness

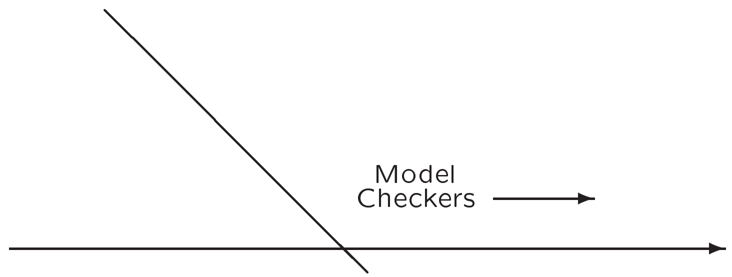
Specification Languages and Mechanization

- A specification language provides some of the features and conveniences of a programming language (e.g., declarations, packaging in modules, parameterization)
- And computer-science data types (e.g., numbers, tuples, records, functions, sets, lists, trees)
- And should use a reasonably familiar syntax
- But it must also support deduction—i.e., it must be a logic
- We want that logic to be expressive, and we also want deduction to be automated very effectively
- Conflict: expressiveness and effectively automated deduction are usually inversely related
- Reconciling these disparate requirements is the fundamental challenge in building tools for formal methods

The Three Traditions



And The Fourth



Automatic Theorem Proving: The Downside

- There are no fully automatic theorem provers
- All require human guidance in indirect forms, such as
  - Weighting of literals, function symbols
  - Variable ordering
  - Strategy selection
  - Orientation of equations
  - Invention and ordering of lemmas
  - Induction hints

  These have more to do with the prover than the proof
- User has to experiment to get them right
- Little feedback; little help discovering falsehoods
- And when proof is done, neither man nor machine is any wiser

Automatic Theorem Proving: The Upside

There are a number of proven ideas and techniques that work remarkably well on specialized problem domains

- Propositional logic: BDDs, Davis-Putnam
- Pure equality: congruence closure
- Linear arithmetic: integer/linear programming (loop residue, SUP-INF)
- Datatypes, algebra: rewriting
- Modal/Temporal logics: model checking (μ-calculus)
- Predicate calculus: resolution, unification
- Induction: Boyer-Moore heuristics, rippling
- Higher-Order: restricted forms of higher-order unification

Seldom encounter pure problems in practice: the engineering challenge is integrating these methods to solve mixed problems

Interactive Proof Checking: The Downside

- Interactive guidance is an alternative to heroic automation
- Many interactive proof checkers are all interaction and no automation
  - Often associated with fundamentalism
  - Enormous human investment to prove small theorems
  - Infeasible to prove large or hard theorems
- Use many lemmas, large numbers of commands
  - Hard for user to focus on the big picture during the proof
  - Essence of the proof is lost in the details
  - Proofs are not robust in the face of even small changes
- The users's concentration is the most precious resource: no excuse for wasting human time on trivial details if they can be blown away by automation

Interactive Proof Checking: The Upside

- Interaction sustains the illusion of progress
- And puts the user in control
- Easier to learn and to master
  - Provide there aren't too many different proof commands
- Can provide helpful feedback from failed proofs
- Can be customized with tactics (programs or macros that build larger proofs steps from the basic ones)

The Formal Analysis Lifecycle

- There is a lifecycle to formal analysis
- Need different capabilities for each stage
  - Exploration: must fail quickly, be able to stay in the loop
    Specialized methods are very useful
    - Animation (direct execution/simulation)
    - State exploration (reachability analysis)
    - "Backwards execution" (fault-tree analysis)
    - Efficient methods for specialized cases (model checking)
  - Development: efficiency!
    Plus ability to comprehend overall structure
  - Presentation: want a real proof
  - Maintenance and Generalization: robustness
- Need to integrate many different techniques

Loose Integration Of Theorem Proving and Other Methods

- Use "lite" methods prior to theorem proving
- Animation (simulation, direct execution)
  - Idea is to evaluate specification on particular cases for testing or validation
    - I know what I want; have I specified it correctly?
    - Assuming I've specified it correctly, is this what I want?
  - Specifications having the form of functional programs, or sets of equations, can often be executed by rewriting, plus evaluation of ground terms
  - Finite quantification can be reduced to iteration or search
- State exploration
  - Explore all reachable states to check invariants
  - Mechanized by model checking and related methods

Tight Integration—I

- Use a model checker as a component of a theorem prover
- Need to be able to describe a temporal logic, its models, and satisfaction relation in the logic of the theorem prover
- Example: CTL, transition relations, and the satisfaction relation between them can be specified in PVS in terms of Park's μ-calculus (QBL with least and greatest fixed points)
  - $\mathbf{EX}(N, f)(u) : \texttt{bool} = \exists v : (f(v) \wedge N(u, v))$
  - $\mathbf{EG}(N, f)(u) : \texttt{bool} = \nu(\lambda Q : (f \wedge \mathbf{EX}(N, Q)))$
- A BDD-based validity checker for the Boolean fragment of Park's μ-calculus is available as a backend to PVS
- Combination provides CTL model checking on finite-state systems within PVS

Tight Integration—II

- When model checking is integrated with theorem proving, can
  - Use induction to prove n-process properties, with the inductive step discharged by model checking
  - Use theorem proving to justify an abstraction to finite state, then model check the finite state abstraction
- Justification of abstraction usually requires developing suitably strong invariants
  - This needs to be automated also

Integrated Theorem Proving and Model Checking

- Havelund and Shankar formally verified a bounded-retransmission communications protocol from Philips
- First "downscaled" (aggressively simplified) it to finite-state
  - Restricted size of data structures to explicit small values
  - Adequate for refutation, not for verification
- Checked safety properties by state exploration using Murϕ
- Then started verification of full protocol in PVS
- Desired invariant needed strengthening by additional conjuncts
  - Plausible conjuncts found by examining failed proof state
  - Checked with Murϕ
  - Added to invariant *without quitting proof*
  - Iterated the above—57 times!

Integrated Theorem Proving and Model Checking (ctd.)

- Next, they abstracted bounded-retransmission protocol to finite-state
  - E.g., untransmitted portion of file is chosen from the uninterpreted enumeration `NONE`, `ONE`, and `MANY`
  - Cf. downscaled version: file is 1, 2, or 3 messages long
- Model-checked the abstracted protocol using the CTL embedding
- Verified by theorem proving that abstraction preserved properties of interest
  - But needed 45 invariants to establish this
- Subsequently done automatically by Graf and Saïdi using mechanized abstraction and invariant generation

Integrated Theorem Proving and Model Checking: Additional Examples

- N-process mutual exclusion can be recursively reduced to two-way
  - Abstract first $N-1$ processes to a single process
  - Establish mutual exclusion between that and N'th process
- Proof in PVS by abstraction plus induction
  - With the inductive step established by CTL model checking
- Similar methods used to establish liveness of cascaded acknowledgments (e.g., multimedia protocols) using LTL

PVS: What Is It?

- It's a *verification system*
  - A general purpose specification language integrated with a theorem prover, model checker, and ancillary tools
- Freely distributed under license by SRI
  - Get it from `http://www.csl.sri.com/pvs.html`
  - Runs on SunOS, Solaris, IBM AIX, or Linux
  - Need 64M memory, 100M swap space, Sparc 20 or better (Best is 64MB 200 MHz P6, costs under $3,000 in USA)
  - Uses Emacs and Tcl/Tk for its interface
- Strengths: rich specification language, powerful prover, large libraries, fully-supported system
- Used at several hundred sites worldwide
- Applications: safety-critical systems, hardware, distributed algorithms, mathematics

Design Choices in PVS Specification Language

- Specification language is a higher-order logic
  - With parameterized theories
  - Computer-science type-constructors (records, lists etc.) Also tables with checks on row/column well-formedness
  - Predicate subtypes and dependent types
  - Definitional forms include recursive functions, abstract data types, inductive definitions
- Built-in "prelude" and loadable "libraries" of domain-specific theories
- Emacs interface and extensive browsing and reporting capabilities
- LaTeX-prettyprinter

Design Choices in PVS Prover

- Automate everything that is decidable
  - Propositional calculus, equality, linear arithmetic, function updates, bitvectors (soon), finite-state model checking
- Provide heuristic automation for the obvious cases of everything else
  - Hashing conditional rewriter integrated with the decision procedures
  - Instantiation, induction
- Leave the human free to concentrate on, and to direct, the steps that require real insight
  - Interaction uses sequent calculus presentation
  - Tactic language for defining higher-level proof strategies

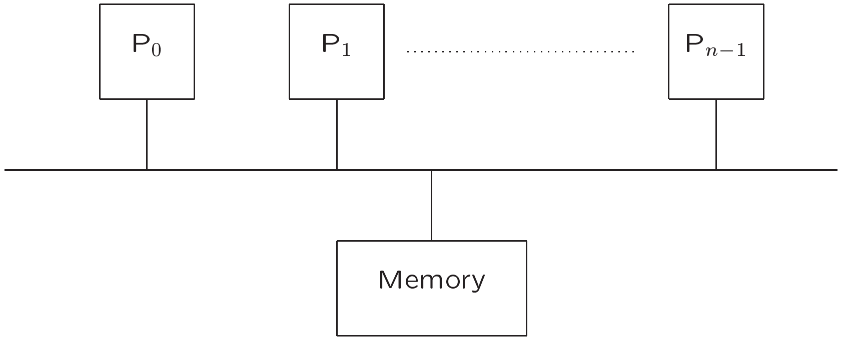
Design Choices in PVS Prover: Interaction

- Formulas manipulated and presented by prover retain the syntactic form of the user's specification
- Browsing facilities locate and display definitions, and help find formulas that reference a given name
- Proof replay, stepping, and editing assist proof maintenance
- Graphical displays of proof trees help comprehend big picture
- Lemmas can be proved in any order
  - And introduced and modified on the fly
- Proof chain (dependency) analysis keeps you honest

The Example: Cache Coherence

- Simplification of single-bus cache-coherence from Futurebus+ (without split transactions)
- Memory and several processors, each with a cache (i.e., a fast local memory), are attached to a bus
- Goal is to maintain the behavior of a single shared memory, while using the caches to speed access and improve concurrency
- In order to maintain appearance of shared memory, caches must reflect changes made by other processors (i.e., they must be "coherent")
- Caches do this by observing ("snooping") all bus transactions and loading ("snarfing") values into their local states appropriately

System Structure



Informal System Description

- Memory is organized into *lines*—contiguous addresses treated as a unit
- Each cache can hold the values of several lines
- Each cache marks each line with a *status* as follows:

status	meaning
invalid	line not present in this cache
shared	line is present for read-only access
exclusive	line is present for read-write access

Informal System Description (continued 1)

- Processor can read value of an address directly from its cache if corresponding line is *shared* or *exclusive*
- And can write it if it is *exclusive*
- Otherwise, must first get cache into appropriate state by issuing one of three transactions

  read-shared: get a shared copy of the line

  read-modified: get an exclusive copy of the line

  write-back: give up exclusive copy
- Desired property:

  Sequentially consistent memory model: behavior seen by individual processors is same as that given by some sequential interleaving of their reads and writes

Informal System Description (continued 2)

The three types of transaction are implemented as follows

read-shared:

- Any cache with an exclusive copy supplies the line and changes its own status to *invalid*
- Otherwise, the memory supplies the line

read-modified:

- First get *shared* copy as described above
- Change it to *exclusive*
- Other caches with a copy of line change it to *invalid*

write-back:

- Value of an exclusive copy is written back to memory
- Status is changed to *invalid*

Formal Modeling and Analysis

- Build a mathematical model of the protocol
- Analyze it for some desired property
- Must choose how much detail to include in the model
  - Too much detail: analysis may be infeasible
  - Too little detail: analysis may be inaccurate (i.e., fail to detect bugs, or report spurious ones)
  - Must also choose a modeling style that supports intended form of analysis
- Requires judgment to do this

Modeling the Example System and its Properties: Accuracy and Level of Detail

- If we wish to examine the sequential consistency property for our protocol, we will need to model the reading and writing of memory, and specify the memory model itself—which is quite challenging
- Instead (or at first), we can focus on an "internal" property of the protocol

  At most one exclusive: at most one cache may have the *exclusive* status for any given line

  This seems a necessary (and sufficient?) property for sequential consistency and will allow us to use a simpler model (we don't need to model the values read and written)

A Further Simplification to the Model

- Observe that the cache lines are treated independently
- Simplify the model still further by assuming just a single line
- Notice that the caches are not independent: we may lose something if we reduce them to just one or two (see later)

Formalizing the Model

- There are two well-established methods for formalizing distributed computations

  Event based: CSP, CCS, Lotos, and other process algebras

  State based: SPL, TLA, and other methods based on temporal logic
- PVS has been used for both
- Here we'll use the state-based approach

Formalizing the Model (continued)

- We will need to model a system state comprising
  - Some (not necessarily bounded or finite) set of processors ("processor" includes its cache)
  - The status recorded for the cache line by each processor
  - The current transaction on the bus, and who initiated it (the "bus-master")
- Then we model the (possibly nondeterministic) transitions in the system state produced by each kind of transaction
- And check that the desired property is always preserved

Specifying the Protocol State in PVS

```
cache_array[N : posnat]: THEORY
BEGIN
  processor : TYPE = below(N)

  line_status : TYPE = {invalid, shared, exclusive}

  transactions : TYPE = {idle, read_shared,
                         read_modified, write_back}

  protocol_state : TYPE =
     [# cache : [processor -> line_status],
        transaction : transactions,
        bus_master : processor #]
...
```


Specifying the System State Transitions in PVS

```
ps, ps0, ps1: VAR protocol_state

p_next(ps0, ps1): bool =
  LET p = bus_master(ps0), t=transaction(ps0) IN
       (idle?(t) AND do_idle(ps0, ps1))
    OR (read_shared?(t) AND do_read_shared(ps0, ps1)(p))
    OR (read_modified?(t) AND do_read_modified(ps0, ps1)(p))
    OR (write_back?(t) AND do_write_back(ps0, ps1)(p))

do_idle(ps0, ps1): bool = (ps1 = ps0)
```


The read_shared Transaction

```
p, q, r: VAR processor

do_read_shared(ps0, ps1)(p): bool  =
  invalid?(cache(ps0)(p)) AND shared?(cache(ps1)(p))
  AND ((EXISTS q: exclusive?(cache(ps0)(q))
          AND NOT exclusive?(cache(ps1)(q))
          AND (FORALL r: r/=q IMPLIES
                 exclusive?(cache(ps1)(r)) =
                 exclusive?(cache(ps0)(r))))
       OR (FORALL q: NOT exclusive?(cache(ps0)(q))
                 AND NOT exclusive?(cache(ps1)(q))))
```

The read_modified and write_back Transactions

```
do_read_modified(ps0, ps1)(p): bool =
  shared?(cache(ps0)(p))
  AND (FORALL q: cache(ps1)(q) =
          IF p = q THEN exclusive
          ELSIF shared?(cache(ps0)(q)) THEN invalid
          ELSE cache(ps0)(q) ENDIF)

do_write_back(ps0, ps1)(p): bool =
   exclusive?(cache(ps0)(p))
   AND cache(ps1) = cache(ps0) WITH [(p) := invalid]
```


Specifying Initial Constraints And Required Property in PVS

```
p_init(ps): bool = FORALL p: invalid?(cache(ps)(p))

p_safe(ps): bool =
  FORALL p, q:
    exclusive?(cache(ps)(p)) AND exclusive?(cache(ps)(q))
        IMPLIES p = q

invariant: THEOREM (p_init(ps) IMPLIES p_safe(ps))
  AND (p_safe(ps0) AND p_next(ps0, ps1) IMPLIES p_safe(ps1))
```


Analyzing the Specification

- We could try to establish the invariant by theorem proving
- This will be good if the protocol does satisfy it
- But can be an expensive way to discover bugs if it does not
- Best to use cheaper methods first to shake out any simple bugs
- For example
  - Simulation (not directly supported by PVS at the moment)
  - Model checking

Analyzing the Specification Using Model Checking

- Model checking requires
  - A (small) finite state space
  - A property expressed in a suitable (temporal) logic
- Then uses brute-force search over the state space to establish that the transition system is a (Kripke) model of the property
  - It's automatic (sort of)
  - And can often provide a counterexample when a bug is found
- PVS provides model checking for properties expressed in Computation Tree Logic (CTL)

Making the Specification Suitable for Model Checking

- We need to make the state space finite
- Use drastic simplification ("downscaling")
  - Set the number of processors, N, to 2
- This may miss some bugs, but will provide a useful check
  - Can then try N=3, 4,... until the search too slow
- Also need to specify the property in CTL

```
invariant_ctl: THEOREM
        p_init(ps) IMPLIES AG(p_next, p_safe)(ps)
```

AG is the CTL operator for "always" (invariance)

Unfortunately...

The PVS model checker does not currently handle numeric subranges (as in the processor type), so have to modify the specification by manually expanding out all quantifications over this type

Results of Model Checking

- The PVS proof commands

```
(expand "p_next")
(model-check)
```

  Verify invariant_ctl in a couple of seconds
- Sometimes properties are true for the wrong reason
- It is prudent to introduce a bug and make sure it is detected before declaring victory

Analyzing the Specification Using Theorem Proving

- Now we have some confidence the property is true, we can attempt to verify it for general N by theorem proving
- Starting the PVS theorem prover gives us this *sequent*

```
invariant:
  |-------
{1} (FORALL (ps, ps0, ps1: protocol_state):
      (p_init(ps) IMPLIES p_safe(ps))
        AND (p_safe(ps0) AND p_next(ps0, ps1)
                IMPLIES p_safe(ps1)))
Rule?
```

First Proof: Step 2

- The proof commands (skosimp) and (ground) eliminate the quantification and perform propositional simplification
- We are then presented with the first of two cases

```
This yields 2 subgoals:
invariant.1:
{-1}  p_init(ps!1)
  |-------
{1}    p_safe(ps!1)
Rule?
```

This is discharged by the proof command (grind), which expands definitions and performs obvious deductions

First Proof: Step 3

- `This completes the proof of invariant.1.`

```
{-1}    p_safe(ps0!1)
{-2}    p_next(ps0!1, ps1!1)
  |-------
{1}    p_safe(ps1!1)
Rule?
```

- We expand the definition of p_next and divide the proof into cases for each of the four kinds of transactions using the commands (expand "p_next") and (ground).

First Proof: Step 4

- Applying propositional simplification

```
 and decision procedures
this yields 4 subgoals:
invariant.2.1:
{-1}    idle?(transaction(ps0!1))
{-2}    do_idle(ps0!1, ps1!1)
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)
Rule?
```

- This is proved by (grind)

First Proof: Step 5

- We'll skip the second and third cases temporarily
- `invariant.2.4:`

```
{-1}    write_back?(transaction(ps0!1))
{-2}    do_write_back(ps0!1, ps1!1)(bus_master(ps0!1))
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)
rule?
```

- This is dispatched by the command (grind :if-match all)

First Proof: Step 6

- Back to the second case
- invariant.2.2:

```
{-1}    read_shared?(transaction(ps0!1))
{-2}    do_read_shared(ps0!1, ps1!1)(bus_master(ps0!1))
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)
Rule?
```

- This requires an interactive proof (it should be automatable)

First Proof: Step 6 (continued)

- Here is the proof of the second case
-

```
    (GRIND :IF-MATCH NIL)
     (("1"
       (INST -8 "q!2")
       (ASSERT)
       (INST -9 "q!1")
       (ASSERT)
       (INST -9 "q!2")
       (ASSERT))
      ("2" (INST -6 "p!1") (ASSERT)))
```


First Proof: Step 7

- Finally, we are presented with the second case
- invariant.2.3:

```
{-1}    read_modified?(transaction(ps0!1))
{-2}    do_read_modified(ps0!1, ps1!1)(bus_master(ps0!1))
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)
Rule?
```

- As with the previous case, we start with `(GRIND :IF-MATCH NIL)`
- This gives us the following sequent...

First Proof: Step 7 (continued)

```
invariant.2.2:
{-1} q!1 < N
{-2} p!1 < N
[-3] read_modified?(transaction(ps0!1))
{-4} shared?(cache(ps0!1)(bus_master(ps0!1)))
{-5} FORALL (q: processor):
      cache(ps1!1)(q)
       = IF bus_master(ps0!1) = q
           THEN exclusive
         ELSIF shared?(cache(ps0!1)(q))
           THEN invalid
         ELSE cache(ps0!1)(q) ENDIF
{-6} FORALL (p: processor):
      exclusive?(cache(ps0!1)(p))
       IMPLIES FORALL (q: processor):
         exclusive?(cache(ps0!1)(q)) IMPLIES p=q
{-7} exclusive?(cache(ps1!1)(p!1))
{-8} exclusive?(cache(ps1!1)(q!1))
  |-------
{1} p!1 = q!1

Rule?
```

First Proof: Aha!

Contemplation, or experimentation with the prover, should convince you that this case is not provable from the information provided

- On a `read_modified` transaction, the `bus_master` changes from `shared` to `exclusive`
- All other processors with `shared` status drop back to `invalid`
- But what if there is a processor already in the `exclusive` status?
- The specification says nothing about this circumstance
- So that processor could retain its status, thereby violating the `p_safe` property

Strengthening the Invariant

- The situation just described cannot occur
- Only obtain `exclusive` status from the `shared` status
- The first processor to gain `exclusive` status causes all other processors with `shared` status to revert to `invalid`
- The problem is not that `p_safe` is untrue, but that it is not *inductive*
  - It does not provide a strong enough antecedent to support the proof of its own invariance
- Solution is to prove a stronger property than the one we are really interested in

  Strict exclusive: if any processor has `exclusive` status, all others must be `invalid`

Specifying the New Property in PVS

- Here's the natural statement

```
strong_p_safe(ps): bool =
  FORALL p: exclusive?(cache(ps)(p)) IMPLIES
         FORALL q: q /= p IMPLIES invalid?(cache(ps)(q))

strong_invariant: THEOREM
  (p_init(ps) IMPLIES strong_p_safe(ps))
     AND (strong_p_safe(ps0) AND p_next(ps0, ps1)
         IMPLIES strong_p_safe(ps1))
```

- And here's the form for model checking

```
strong_invariant_ctl: THEOREM
   p_init(ps) IMPLIES AG(p_next, strong_p_safe)(ps)
```


Results of Model Checking and Proof for New Property

- Model checking succeeds as before
- And this time the proof goes through
- More complex protocols can require dozens of strengthening steps to produce an invariant that is inductive
- Automatic generation of suitable auxiliary invariants is a major research topic

A Tighter Integration Of Model Checking and Theorem Proving

- We've seen that model checking a downscaled instance is a useful prelude to theorem proving the general case
- But a more interesting integration is to use model checking as part of a proof for the general case
- One approach is to create a finite state *property-preserving abstraction* of the original protocol
  - Use model checking to ensure the abstraction satisfies the property
  - Use theorem proving to show the abstraction preserves the property

Conditions for Property-Preserving Abstraction

- Want an abstracted state type `abstract_state`
  - And corresponding transition relation `a_next`
  - And initiality predicate `a_init`
  - Together with abstracted safety property `a_safe`
- And an abstraction function `abst` from `protocol_state` to `abstract_state`, such that following properties hold

```
init_simulation: THEOREM
   p_init(ps) IMPLIES a_init(abst(ps))
next_simulation: THEOREM
   p_next(ps0, ps1) IMPLIES a_next(abst(ps0), abst(ps1))
safety_preserved: THEOREM a_safe(abst(ps)) IMPLIES p_safe(ps)
abs_invariant_ctl: THEOREM
   a_init(as) IMPLIES AG(a_next, a_safe)(as)
```


Abstracted Model

- Doesn't matter for properties we are interested in exactly how many caches have `exclusive` status
- Only whether there are 0, 1, or many
- And whether or not there are any with `shared` status
- So introduce abstracted status type

```
abstract_status: TYPE =
           {all_invalid, some_shared, one_exclusive,
            many_exclusive, exclusive_shared}
```

- Each value of type `abstract_status` abstracts the status of N concrete caches

Interpretation of Abstract State Values

Abstract Status	Interpretation for N Caches
`all_invalid`	all `invalid`
`some_shared`	some `shared`, no `exclusive`
`one_exclusive`	one `exclusive`, no `shared`
`many_exclusive`	many `exclusive`, no `shared`
`exclusive_shared`	some `exclusive`, some `shared`

Abstraction Function in PVS

```
abst_cache(c: [processor -> line_status]): abstract_status =
  IF (EXISTS p: exclusive?(c(p)))
  THEN IF (EXISTS q: shared?(c(q)))
       THEN exclusive_shared
       ELSIF (EXISTS p:
           (FORALL q: exclusive?(c(q)) IMPLIES p = q))
       THEN one_exclusive
       ELSE many_exclusive ENDIF
  ELSIF (EXISTS p: shared?(c(p)))
  THEN some_shared
  ELSE all_invalid ENDIF
```


More of the Abstracted Specification

```
abstract_state: TYPE = [# cache: abstract_status,
                          transaction: transactions #]

abst(ps): abstract_state =
   (# cache := abst_cache(cache(ps)),
      transaction := transaction(ps) #)

as0, as1: VAR abstract_state

a_next(as0, as1): bool =
  LET t = transaction(as0) IN
       (idle?(t) AND do_idle(as0, as1))
    OR (read_shared?(t) AND do_read_shared(as0, as1))
    OR (read_modified?(t) AND do_read_modified(as0, as1))
    OR (write_back?(t) AND do_write_back(as0, as1))
```


Abstracted read_shared Transaction

- Following looks plausible

```
do_read_shared(as0, as1): bool = LET x=cache(as0) IN
  (all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))
   AND some_shared?(cache(as1))
```

- However, also need to account for the case where a transaction is issued in an inappropriate state (i.e., one with the many many_exclusive or exclusive_shared status)
- Do not care about the subsequent state in this case, but must allow some transition, so disjoin the negation of the desired antecedent state

```
do_read_shared(as0, as1): bool = LET x=cache(as0) IN
  ((all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))
    AND some_shared?(cache(as1)))
 OR NOT (all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))
```


Abstracted read_shared Transaction

- Transition specification has the abstract form `(a AND b) OR NOT a`
- Which is equivalent to `a IMPLIES b`
- Also, the expression `(all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))` is the safety property that we wish to establish
- Hence, final specification is the following

```
a_safe(as): bool =
  LET x = cache(as) IN
   all_invalid?(x) OR some_shared?(x) OR one_exclusive?(x)

do_read_shared(as0, as1): bool =
  (a_safe(as0) IMPLIES some_shared?(cache(as1)))
```

The read_modified and write_back Transactions

```
do_read_modified(as0, as1): bool =
  LET x = cache(as0), y = cache(as1) IN
      (some_shared?(x) OR exclusive_shared?(x))
        AND IF some_shared?(x) THEN one_exclusive?(y)
             ELSE many_exclusive?(y) ENDIF

do_write_back(as0, as1): bool =
  LET x = cache(as0), y = cache(as1) IN
        (one_exclusive?(x)
      OR many_exclusive?(x)
      OR exclusive_shared?(x))
    AND (one_exclusive?(x) IMPLIES all_invalid?(y))
```


Results of Model Checking and Proof For Abstracted Specification

- `init_simulation` is proved by `(grind)`
- `safety_preserved` requires a simple interactive proof (described in the notes)
- `a_invariant_ctl` is established by model checking in a few seconds
- `next_simulation` requires rather a long interactive proof

Trying It Yourself

- The specifications and proofs described can be downloaded from `http://www.csl.sri.com/forte97.html`
- So can the slides and the notes
- You can load the specifications and proofs into PVS and step through the proofs (the notes explain how)
- Then try constructing your own proofs for some formulas or cases
- Next, try modifying the specifications and proofs to explore variant systems and properties

Suggested Exercises

- The initial condition specifies that all processors should start with the `invalid` status. Investigate weaker conditions (e.g., `NOT exclusive`, or `p_safe`). Test them first by model checking in the downscaled instance, then try the corresponding invariant and abstraction proofs
- Repeat the verification and model checking for the approach based on property-preserving abstraction, but with the `strong_p_safe` property
- The `write_back` transaction causes the `bus_master` to change from `exclusive` to `invalid` status. Investigate allowing this processor to retain `exclusive` status, or to change to `shared` status. What if other processors snarf the value (i.e., change from `invalid` to `shared` status) during `write_back` under these conditions?

More Suggested Exercises

- Introduce a bug into the specification. Investigate how the bug comes to light in the different forms of analysis. Introduce a bug that is only manifested when three or more processors are considered
- Only the memory or a processor with `exclusive` status can supply the line to a `read_shared` transaction. Examine the case where a processor with `shared` status can also respond. Does this not potentially allow multiple processors to respond, causing collisions on the bus? Does the formal model expose this? If not, how can it be extended to do so?
- (This is an advanced exercise) Show that the protocol provides sequential consistency

Summary

Have seen three methods of formal analysis

- Model checking a downscaled instance
  - Fast, easy (at least for small examples), good at finding bugs
  - Drastic downscaling detracts from verification
  - Research issues include various ways of reducing state explosion, compositional approaches
- Theorem proving the full protocol
  - Less automated, need to strengthen invariants
  - Provides full verification, can be used to find bugs
  - Research issues include more automation, automated invariant generation, compositional methods

Summary (continued)

- Property-preserving abstraction with combined theorem proving and model checking
  - Combines some of the strengths of each, Looks like the way forward
  - Research issues include automated generation of abstractions

Overall: only just beginning to realize the full potential of automated formal methods in the analysis of protocols and distributed algorithms

Likely to see dramatic advances in integrated approaches during the next 10 years

Formal methods of analysis will become the dominant technology

To Learn More About PVS

- Browse general papers and technical reports at `http://www.csl.sri.com/fm.html`
  - `tse95.html` is a good overview of PVS
  - `pvs-bib.html` links to a bibliography with over 140 entries
- Detailed Information about PVS is available from `http://www.csl.sri.com/pvs.html`
  - `http://www.csl.sri.com/pvs/examples` gets you to a directory of tutorials and their supporting specification files
- You can get PVS from `ftp://ftp.csl.sri.com/pub/pvs`
  - Allegro Lisp for SunOS, Solaris, IBM AIX, or Linux
  - Need 64M memory, 100M swap space, Sparc 20 or better (Best is 64MB 200 MHz P6, costs under $3,000 in USA)