Advanced Summer School on Formal Methods, Beijing 18–22 October 1999

Proof Checking, Model Checking, And Integrated Approaches: The Bakery Example in PVS

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

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 Solaris or Linux
  - 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 Bakery Algorithm for Distributed Mutual Exclusion

- Idea is based on the way people queue for service in US delis and bakeries
- A machine dispenses tickets printed with numbers that increase monotonically
- People who want service take a ticket
- The unserved person with the lowest numbered ticket is served next
  - Safe: at most one person is served (i.e., is in the "critical section") at a time
  - Live: each person is eventually served
- Preserve the idea without centralized ticket dispenser

Informal Protocol Description

- Works for $n \geq 1$ processes
- Each process has a counter, initially zero
- When it wants to enter its critical section, a process sets its counter greater than that of any other process
- Then it waits until its counter is smaller than that of any other process with a nonzero counter
- At which point it enters its critical section
- Resets its counter to zero when it exits its critical section
- Show that at most one process is in its critical section at any time (i.e., "mutual exclusion")

Formal Modeling and Analysis

- Build a mathematical model of the protocol
- Analyze it for a 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

- The protocol uses shared memory and is sensitive to the exact "memory model" used (on multiprocessors with relaxed memory models, reads and writes from different processors may be reordered)
- If we wish to examine the mutual exclusion property of a particular implementation of the protocol, we will need to model the memory model employed—which will be quite challenging
- Instead (or at first), we may prefer to focus on the behavior of the protocol in an ideal environment with sequentially consistent memory

Modeling the Example System and its Properties (ctd.)

- Also, although the protocol is suitable for n processes, we may prefer to focus on the important (and simplest nontrivial) special case $n = 2$
- And although each process will perform activities other than the given protocol, we will abstract these details away and assume each process is in one of three phases

  idle: performing work outside its critical section

  trying: to enter its critical section

  critical: performing work inside its critical section

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 shared state (memory) and temporal logic

- PVS has been used for both

- Here we have an intrinsically shared-memory protocol, so we'll use the state-based approach

Formalizing the Model (continued)

- We will need to model a system state comprising
  For each process:
  - The value of its counter, which is a natural number
  - The phase it is in—recorded in its "program counter" which takes values `idle`, `trying`, `critical`
- Then we model the (possibly nondeterministic) transitions in the system state produced by each protocol step
- And check that the desired property is always preserved

More Formal Protocol Description (Unity-like)

```
Type
    phase: enum {idle, trying, critical};
    counter: nat;
Var
    pc1, pc2: phase;
    t1, t2: counter;
Initially
    t1 := 0;        t2 := 0;
    pc1 := idle;    pc2 := idle;
Rules
    pc1 = idle ==>  t1 := t2+1; pc1 := trying;
    pc2 = idle ==>  t2 := t1+1; pc2 := trying;

    pc1 = trying & (t1 < t2 | t2 = 0) ==> pc1 := critical;
    pc2 = trying & (t1 > t2 | t1 = 0) ==> pc2 := critical;

    pc1 = critical ==> t1 := 0;   pc1 := idle;
    pc2 = critical ==> t2 := 0;   pc2 := idle;
Invariant
    !(pc1=critical & pc2=critical);
```

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 simple bugs
- For example
  - Simulation, animation
    - ⋆ PVS has a high-performance "ground evaluator"
    - ⋆ Better for animating sequential algorithms than simulating concurrent ones
  - Model checking
    - ⋆ PVS has a symbolic model checker, but it does not provide counterexamples
    - ⋆ Better for verification than for debugging

Analyzing the Specification Using Model Checking

- We'll use the Murϕ system from Stanford
  - An efficient explicit-state reachability analyzer
  - Can only check invariant properties
    (A separate version can do liveness)
- In general, 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 provide a counterexample when a bug is found

Making the Specification Suitable for Model Checking

- We need to make the state space finite
- Use drastic simplification (“downscaling”)
  - We’ve already done this to some extent, by fixing the number of processors, `n`, as 2
  - We also need to set an upper bound (say 3) on the counters
- This may miss some bugs, but will provide a useful check
  - Can then raise the limit to 4, 5, . . . until the search becomes too slow
  - But note that we have to modify the protocol to bound the counters
  - So it’s not the same protocol

The Bounded Specification in Murϕ

```
Const
    max: 3;
Type
    pc: enum {idle, trying, critical};
    counter: 0..max;
Var
    pc1, pc2: pc;
    t1,t2: counter;

Startstate
Begin
    t1 := 0;          t2 := 0;
    pc1 := idle;      pc2 := idle;
End;
```

The Bounded Specification in Murϕ (ctd)

```
Rule "P1 tries" pc1 = idle & t2 < max ==>
    Begin  t1 := t2+1; pc1 := trying; End;
Rule "P2 tries" pc2 = idle & t1 < max ==>
    Begin  t2 := t1+1; pc2 := trying; End;

Rule "P1 enters" pc1 = trying & (t1 < t2 | t2 = 0) ==>
    Begin pc1 := critical; End;
Rule "P2 enters" pc2 = trying & (t1 > t2 | t1 = 0) ==>
    Begin pc2 := critical; End;

Rule "P1 quits" pc1 = critical ==>
    Begin t1 := 0;  pc1 := idle; End;
Rule "P2 quits" pc2 = critical ==>
    Begin t2 := 0;  pc2 := idle; End;

Invariant !(pc1=critical & pc2=critical);
```

Results of Model Checking

- Murϕ reports

```
Status:

        No error found.

State Space Explored:

        21 states, 30 rules fired in 1.83s.
```

- Sometimes properties are true for the wrong reason
- It is prudent to introduce a bug and make sure it is detected before declaring victory
- Here, if we remove the +1 adjustment to the counters, we get

```
Result:

        Invariant "Invariant 0" failed.

State Space Explored:

        9 states, 14 rules fired in 1.83s.
```

The Backtrace

```
Startstate Startstate 0 fired.
pc1:idle    pc2:idle
t1:0        t2:0
----------
Rule P2 tries fired.
pc2:trying
----------
Rule P2 enters fired.
pc2:critical
----------
Rule P1 tries fired.
pc1:trying
----------
Rule P1 enters fired.
The last state of the trace (in full) is:
pc1:critical   pc2:critical
t1:0           t2:0
----------
End of the error trace.
```

Another Check

- We can check that the counters are capable of increasing indefinitely by adding the invariant

  `Invariant t1 < max;`

  (After undoing the deliberate errors just introduced)

- `Result:`

```
        Invariant "Invariant 1" failed.
State Space Explored:
        15 states, 18 rules fired in 1.86s.
```

- The pattern is: `P1 tries`, then the following sequence repeats `P1 enters`, `P2 tries`, `P1 quits`, `P2 enters`, `P1 tries`, `P2 quits`

Analyzing the Specification Using Theorem Proving

- Now we have some confidence the property is true, we can attempt to verify it for unbounded counters by theorem proving
- First we need to specify it in PVS
- PVS is a logic, it does not have a notion of state, nor of programs, built in—we must directly embed the transition relation semantics of Murϕ

```
bakery: THEORY
BEGIN
  phase : TYPE = {idle, trying, critical}

  state: TYPE = [# pc1, pc2: phase, t1,t2: nat #]

  s, pre, post: VAR state
```

The Transitions in PVS

```
P1_transition(pre, post): bool =
 IF pre`pc1 = idle
   THEN post = pre WITH [(t1) := pre`t2 + 1, (pc1) := trying]
 ELSIF pre`pc1 = trying AND (pre`t2 = 0 OR pre`t1 < pre`t2)
   THEN post = pre WITH [(pc1) := critical]
 ELSIF pre`pc1 = critical
   THEN post = pre WITH [(t1) := 0, (pc1) := idle]
 ELSE post = pre
 ENDIF

P2_transition(pre, post): bool = ... similar

transitions(pre, post): bool =
  P1_transition(pre, post) OR P2_transition(pre, post)
```

Initialization and Invariant in PVS

```
init(s): bool = s`pc1 = idle AND s`pc2 = idle
                  AND s`t1 = 0 AND s`t2 = 0

safe(s): bool = NOT(s`pc1 = critical AND s`pc2 = critical)

INV, INIT: var pred[state]
R: VAR pred[[state,state]]

invariant(INV, INIT, R): bool =
 FORALL s: INIT(s) => INV(s)
  AND FORALL pre,post: INV(pre) AND R(pre,post) => INV(post)

first_try: LEMMA invariant(safe, init, transitions)
```

First Attempted Proof: Step 1

- Starting the PVS theorem prover gives us this sequent

```
first_try :
  |-------
{1}   invariant(safe, init, transitions)
Rule?
```

- The proof commands `(EXPAND "invariant")` and `(GROUND)` open up the definition of `invariant` and split it into cases

- We are then presented with the first of the two cases

```
This yields  2 subgoals:
first_try.1 :
  |-------
{1}   FORALL s: init(s) => safe(s)
```

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

First Attempted Proof: Step 2

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

```
first_try.2 :
  |-------
{1}   FORALL pre, post:
          safe(pre) AND transitions(pre, post) => safe(post)
```

- The commands `(SKOSIMP)`, `(EXPAND "transitions")`, and `(GROUND)` eliminate the quantification and split `transitions` into separate cases for processes 1 and 2

```
first_try.2.1 :
{-1}  P1_transition(pre!1, post!1)
[-2]  safe(pre!1)
  |-------
[1]   safe(post!1)
```

- `(EXPAND "P1_transition")` and `(GROUND)` split the proof into four cases according to the kind of step made by the process

First Attempted Proof: Step 3

- The first one is discharged by `(GRIND)`, but the second is not and we are presented with the sequent

```
first_try.2.1.2 :
[-1]  pre!1`t2 = 0
{-2}  trying?(pre!1`pc1)
[-3]  post!1 = pre!1 WITH [(pc1) := critical]
[-4]  safe(pre!1)
{-5}  critical?(pre!1`pc2)
  |-------
```

- When there are no formulas below the line, a sequent can only be true if there is a contradiction among those above the line
- Here, Process 1 enters its critical section because Process 2's ticket is zero—but Process 2 is already in its critical section

First Attempted Proof: Aha!

- This situation is impossible because Process 2 must have incremented its ticket (making it nonzero) when it entered its `trying` phase
- But contemplation, or experimentation with the prover, should convince you that this fact is not provable from the information provided
- Similarly for three other unprovable subgoals
- The problem is not that `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

Second Attempted Proof

- Solution is to prove a stronger property than the one we are really interested in

```
strong_safe(s): bool = safe(s)
   AND ((s`pc1 = trying OR s`pc1 = critical) => s`t1 /= 0)
   AND ((s`pc2 = trying OR s`pc2 = critical) => s`t2 /= 0)

second_try: LEMMA invariant(strong_safe, init, transitions)
```

- It's a good idea to check this first by model checking

Second Attempted Proof: Aha! Again

- The stronger formula deals with the case we just examined, and the symmetric case for Processor 2, but still has two unproved subgoals; here's one of them

```
second_try.2 :
{-1}  trying?(pre!1`pc1)
{-2}  pre!1`t1 < pre!1`t2
{-3}  post!1 = pre!1 WITH [(pc1) := critical]
{-4}  critical?(pre!1`pc2)
  |-------
{1}    (pre!1`t1 = 0)
```

- The situation here is that Process 1 is entering its critical section even though Process 2 is already in its critical section
- But this cannot happen because Process 2 must have had the smaller ticket when it entered (because Process 1 has a nonzero ticket), contradicting formula -2

Third Attempted Proof

- Again we need to strengthen the invariant to carry along this fact
- ```
  inductive_safe(s):bool = strong_safe(s)
   AND ((s`pc1 = critical AND s`pc2 = trying) => s`t1 < s`t2)
   AND ((s`pc2 = critical AND s`pc1 = trying) => s`t1 > s`t2)

  third_try: LEMMA
   invariant(inductive_safe, init, transitions)
  ```
- Finally, we have a invariant that is inductive—and is proved with `(GRIND)`
  ```

Observations

- Model checking was easy, but required downscaling and yielded little insight
- Proof required strengthening the invariant
  - Each strengthening was suggested by a failed proof
  - Straightforward but tedious
  - More complex protocols can require dozens of strengthening steps to produce an invariant that is inductive
- Model checking was useful for testing proposed strengthenings
- Automatic generation of suitable auxiliary invariants is a major research topic

A Tighter Combination 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_trans`
  - And initiality predicate `a_init`
  - Together with abstracted safety property `a_safe`
- And an abstraction function `abst` from `state` to `abstract_state`, such that following properties hold

```
init_simulation: THEOREM
 init(s) IMPLIES a_init(abst(s))
trans_simulation: THEOREM
 transitions(pre,post) IMPLIES a_trans(abst(pre),abst(post))
safety_preserved: THEOREM a_safe(abst(s)) IMPLIES safe(s)
abs_invariant_ctl: THEOREM
 a_init(as) IMPLIES AG(a_trans, a_safe)(as)
```

Abstracted Model

- It doesn't matter to the protocol what the actual values of the tickets are
- All that matters is whether or not each of them is zero, and whether one is less than the other
- We can use Booleans to represent these relations
- So introduce the abstracted (finite) state type

```
abstract_state: TYPE =
  [# pc1, pc2: phase, t1_is_0, t2_is_0, t1_lt_t2: bool #]

  as, a_pre, a_post: VAR abstract_state
```

Abstraction Function
And Abstracted Properties in PVS

```
abst(s): abstract_state =
  (# pc1 := s`pc1, pc2 := s`pc2,
     t1_is_0 := s`t1 = 0, t2_is_0 := s`t2 = 0,
     t1_lt_t2 := s`t1 < s`t2 #)

a_init(as): bool =
   as`pc1 = idle AND as`pc2 = idle
     AND as`t1_is_0 AND as`t2_is_0

a_safe(as): bool =
   NOT (as`pc1 = critical AND as`pc2 = critical)
```

More of the Abstracted Specification

```
a_P1_transition(a_pre, a_post): bool =
 IF a_pre`pc1 = idle
   THEN a_post = a_pre WITH [(t1_is_0) := false,
                             (t1_lt_t2) := false,
                             (pc1) := trying]
 ELSIF a_pre`pc1 = trying
         AND (a_pre`t2_is_0 OR a_pre`t1_lt_t2)
   THEN a_post = a_pre WITH [(pc1) := critical]
 ELSIF a_pre`pc1 = critical
   THEN a_post = a_pre WITH [(t1_is_0) := true,
                             (t1_lt_t2) := NOT a_pre`t2_is_0,
                             (pc1) := idle]
 ELSE a_post = a_pre
ENDIF
```

The Rest of the Abstracted Specification

```
a_P2_transition(a_pre, a_post): bool =
 IF a_pre`pc2 = idle
   THEN a_post = a_pre WITH [(t2_is_0) := false,
                             (t1_lt_t2) := true,
                             (pc2) := trying]
 ELSIF a_pre`pc2 = trying
         AND (a_pre`t1_is_0 OR NOT a_pre`t1_lt_t2)
   THEN a_post = a_pre WITH [(pc2) := critical]
 ELSIF a_pre`pc2 = critical
   THEN a_post = a_pre WITH [(t2_is_0) := true,
                             (t1_lt_t2) := false,
                             (pc2) := idle]
 ELSE a_post = a_pre
ENDIF
```

Proofs to Justify The Abstraction

- The conditions `init_simulation` and `safety_preserved` are proved by `(GRIND)`
- And `abs_invariant_ctl` is proved by `(MODEL-CHECK)`
- But `trans_simulation` results in two unproved cases—here's the first

```
trans_simulation.2.6.1 :
[-1]  post!1 = pre!1
{-2}  idle?(pre!1`pc1)
  |-------
{1}   critical?(pre!1`pc2)
[2]   pre!1`t1 = 0
[3]   pre!1`t1 > pre!1`t2
{4}   idle?(pre!1`pc2)
{5}   pre!1`t1 < pre!1`t2
```

The Problem Justifying The Abstraction

- The problem here is that when the two tickets are equal but nonzero, the concrete protocol drops through to the `ELSE` case and requires the pre and post states to be the same
- But in the abstracted protocol, this situation can satisfy the condition for Process 2 to enter its critical section
  - Because `NOT a_pre‘t1_lt_t2` abstracts `pre‘t1 >= pre‘t2` rather than `pre‘t1 > pre‘t2`
- But this situation can never arise, because each ticket is always incremented to be strictly greater than the other
- We can prove this as an invariant

```
not_eq(s): bool = s‘t1 = s‘t2 => s‘t1 = 0

extra: LEMMA invariant(not_eq, init, transitions)
```

- This is proved with `(GRIND)`

A Justification of the Abstraction

- A stronger version of the simulation property allows us to use a known invariant to establish it

```
strong_trans_simulation: THEOREM
   invariant(not_eq, init, transitions)
    AND not_eq(pre) AND not_eq(post)
    AND transitions(pre, post)
      IMPLIES a_trans(abst(pre), abst(post))
```

- This is proved by

```
(SKOSIMP)
(EXPAND "transitions")
(GROUND)
(("1" (EXPAND "P1_transition")
      (APPLY (THEN (GROUND) (GRIND))))
 ("2" (EXPAND "P2_transition")
      (APPLY (THEN (GROUND) (GRIND)))))
```

Integration: Automated Abstraction and Model Checking

- Rather than seeking to justify an abstraction constructed by hand (whose own proof may require further invariants) we can use automated theorem proving to construct an abstracted transition relation
- Start with universal transition relation on the abstract state; then for each arc
  - Generate the VC that allows it to be removed
  - Leave it in if cannot prove the VC
- This is automated in InVeSt (an adjunct to PVS developed in conjunction with Verimag)

Automated Abstraction in PVS

- A different method (due to Saïdi and Shankar) for automatically constructing an abstracted specification is provided by the PVS `(ABSTRACT)` command
- The abstraction is specified in the proof command by listing the predicates of the concrete specification that are to be abstracted as Booleans
- ```
  auto_abstract: THEOREM
     init(s) IMPLIES AG(transitions, safe)(s)
  ```
- This is proved by

```
 (ABSTRACT-AND-MC
  ( "lambda(s) :s`t1=0"
    "lambda(s) :s`t2=0"
    "lambda(s) :s`t1 < s`t2 "))
```
```

Summary

Have seen four 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 suitable for model checking justified by theorem proving
  - Combines some of the strengths of each
  - Still need invariants to justify abstraction
  - Research issues include automated invariant generation
- Automated construction of a property-preserving abstraction that can be model checked
  - Looks like the way forward
  - Research issues include use of abstractions to help develop invariants, and iterated use of these methods

Projections

- 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
- We are working on a prototype Symbolic Analysis Laboratory (SAL) that has automated abstraction and invariant generation as its main focus
- Formal methods of analysis will become the dominant technology debugging and verification

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
  - `spin99.html` points to papers on abstraction
  - `pvs-bib.html` links to a bibliography with over 250 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 specification files
- You can get PVS from `ftp://ftp.csl.sri.com/pub/pvs`
  - Allegro Lisp for Solaris and Linux
  - Need 64M memory, 100M swap space, 200Mhz or better (Best is 256MB 500 MHz PIII, under $3,000 in USA)