FSE4 Tutorial Introduction, October 1996

Theorem Proving and Model Checking for Software

John Rushby and N. Shankar

Computer Science Laboratory
SRI International
Menlo Park CA USA

Overview

- $\boxed{\text{The rationale for } \textit{mechanized} \text{ formal methods}}$

- Sketch of theorem proving and model checking

- Requirements on mechanized tools for formal methods

- Examples and applications

- Break

- Live demonstration using PVS

Formal Methods

- Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems (both hardware and software)

- 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

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

  (Terminology from the guidelines for avionics software, RTCA DO-178B/EUROCAE ED-12B [23])

- The value of calculation becomes more pronounced as the problems addressed become
  - Bigger
  - More important
  - More difficult and more complex

Mechanized Formal Methods

- To perform calculations efficiently and reliably on such problems, 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
- Without mechanization, formal methods are neither formal, nor methods

Corollaries

- Tools are not the most important thing about formal methods
  - They are the *only* important thing
- Building tools is an engineering enterprise
- The most important attribute of a tool is its effectiveness
  - If it cannot deal with problems of practical interest at reasonable cost, doesn't matter what other attributes it has
- To undertake formal calculations on real problems, you need automated tools (i.e., theorem provers and model-checkers) of *adequate* expressiveness and *massive* power and speed
  - Proof editors: off by 3–4 orders of magnitude
  - PVS: off by 1–2 orders of magnitude
  - Model checkers: space is more of a problem than time

Overview

- The rationale for *mechanized* formal methods
- Sketch of theorem proving and model checking
- Requirements on mechanized tools for formal methods
- Examples and applications
- Break
- Live demonstration using PVS

Theorem Proving Approaches

- 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

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

- Variation:
  - Prove that an implementation is a refinement of a specification
  - Classical commutative diagram, or theory interpretation

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*

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 generally 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

Model Checking Approaches–I

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

  $$\textit{system} \models \textit{requirement}$$

  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

Model Checking Approaches–II

- Dominant approach uses Computation Tree Logic (CTL)
  - Mechanized by SMV
- A branching time logic: characterizes properties of states in the tree representing 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)
- Linear time logic (mechanized by SPIN) characterizes properties of individual computations (i.e., sequences of states) $\Box$: henceforth; $\Diamond$: eventually

Model Checking Approaches–III

- Variations (see [1] for descriptions of all tools mentioned):
  - 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)

- Most 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"

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 be still 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

Overview

- The rationale for *mechanized* formal methods
- Sketch of theorem proving and model checking
- Requirements on mechanized tools for formal methods
- Examples and applications
- Break
- Live demonstration using PVS

Specification Languages for Mechanized Formal Methods

- General-purpose or specialized languages?
- Specialized languages can often be provided with correspondingly optimized mechanization
- E.g., tabular specification methods (Parnas, A7, SCR, RSML)
  - Automatically generate well-formedness conditions for disjointness and coverage of conditions
  - Simple cases can be discharged automatically by suitable decision procedures (e.g., SVC [2])
  - Model checking can examine invariants
- All these can also be done in a general-purpose system, but may not scale to really large examples

Specialized Languages

- Challenges posed by specialized languages are mostly a subset of those of general-purpose languages, but compounded by scale

- Costs of building effective mechanizations are so high that it's best to mechanize *mathematics* (i.e., a logic) rather than a specific formalism or model of computation

- Embed the specialized notation in the logic

- Or translate to it

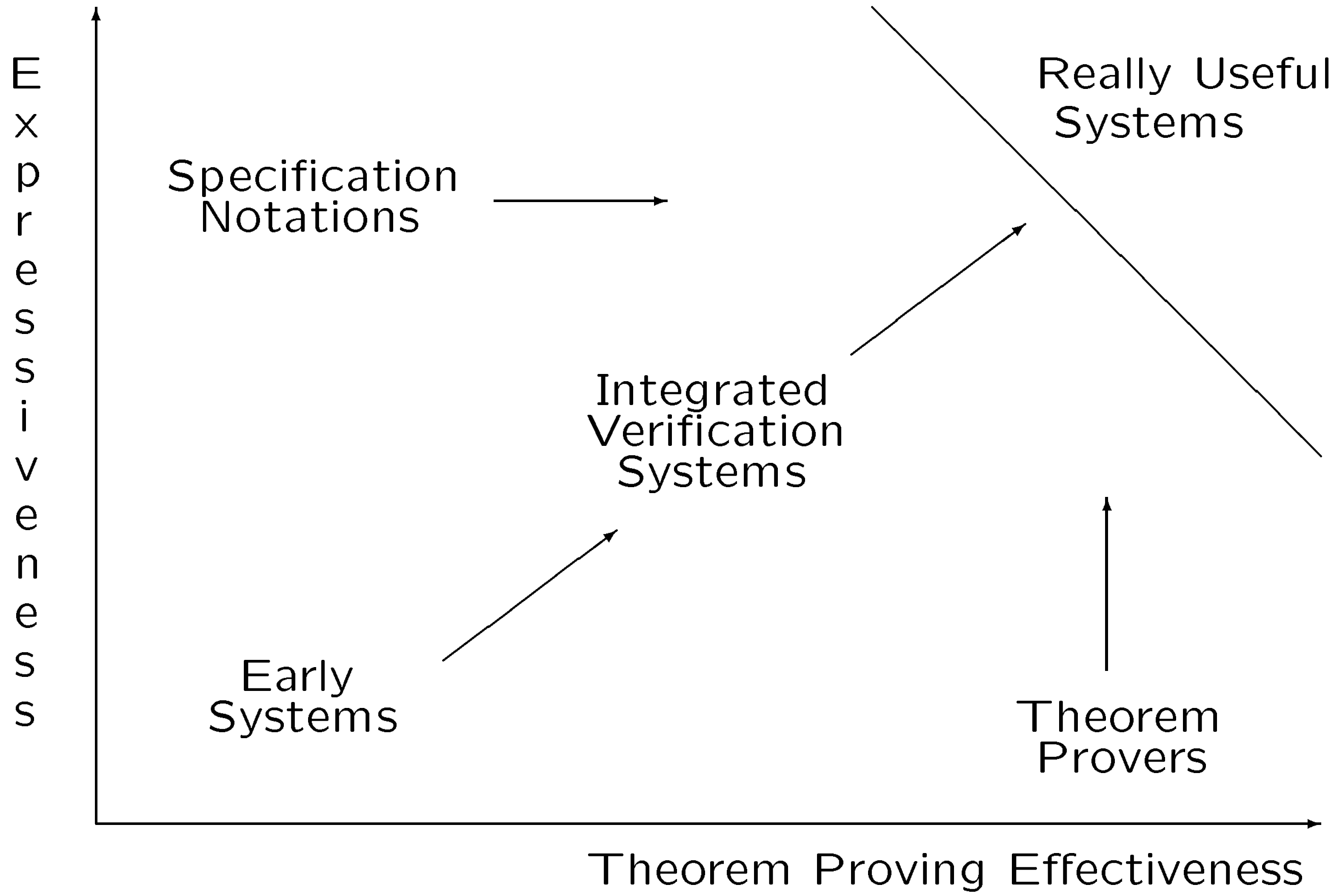
- Build customized system only when convinced of utility and necessity

- Will become easier as more open (component technology) theorem provers become available

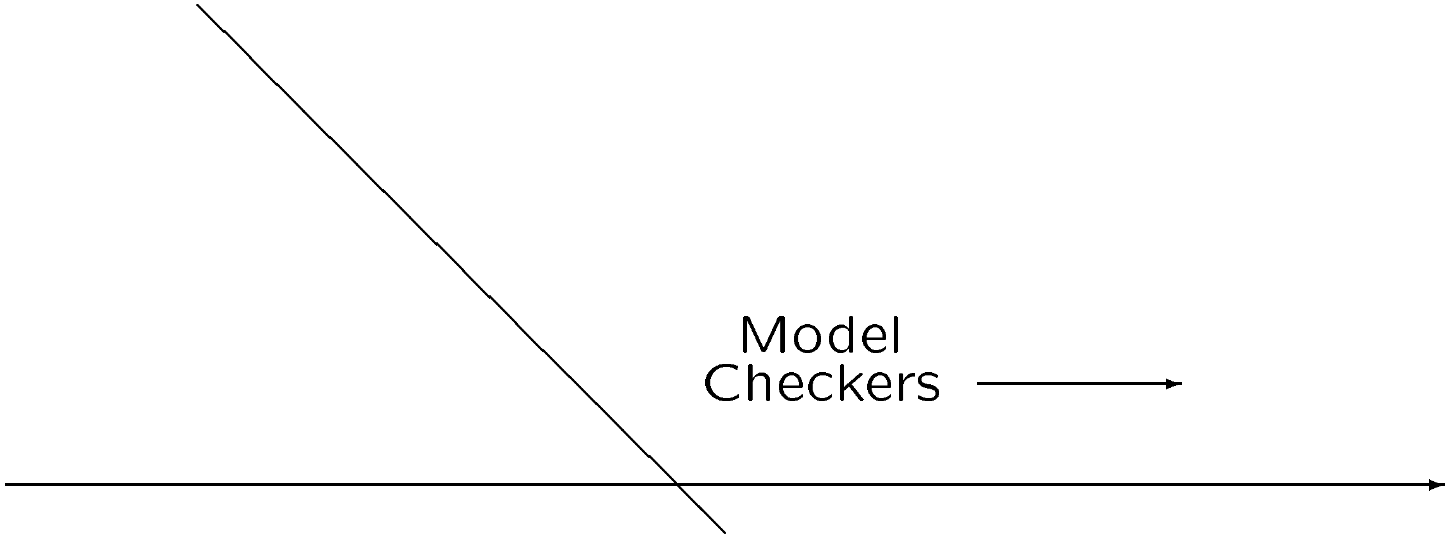
General Purpose Specification Languages

- 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 A Fourth



Reconciling the Conflicts: Foundations

- Traditional foundation for mathematics is first-order set theory
- And Hilbert-style presentation
- Designed by logicians for *studying* formal systems, not for actually *doing* formalization
  - Requires metamathematical assistance (schema) to specify many important concepts (e.g., induction)
  - And functions are inherently partial: inimical to efficient theorem proving
- For mechanized formal methods, we have different needs, and should therefore make different choices
  - Frameworks (LF, Calculus of Constructions etc.)
  - Higher-Order Logic (Simple Theory of Types)

Reconciling the Conflicts: Pragmatics

- You cannot design a specification language and then hope to add effective mechanization

- And you cannot build a theorem prover and hope to persuade very many people that its logic is a specification language

- The formal method, its specification language, and its tools have to be designed together
  - Tradeoffs and compromises are necessary
  - But new opportunities arise, too

- Example: Enhancing the error-detection of typechecking

Errors in Formal Specifications

- Most of the time spent with a formal methods tool goes in discovering and correcting errors
  - Misunderstanding the requirements
  - Flawed assumptions
  - Faulty design
  - Bugs in the formal specification
- Writing correct formal specifications is no easier than writing correct programs (for most people it's harder)
- Most formal specifications have errors; many are full of errors; a large number are meaningless (e.g., inconsistent)
- May be difficult to believe this, until you've experienced the personal shock of finding errors in your own specifications
- Tools should help you find errors early and cheaply

Typechecking Formal Specifications

- Many errors in specifications can be found by strict typechecking—just as in programming languages
- But unlike programming languages, type systems of specification languages need not be restricted to those that are trivially decidable
- Can assume a theorem prover is available and allow typechecking to become undecidable
- This eliminates many of the criticisms of those who consider strict typing too restrictive
- Allows a lot of the specification to be expressed in the types, so that typechecking becomes a very strong check on consistency

Example: Predicate Subtypes

- These are subtypes that are associated with a predicate
  Example (in PVS):

  even_nat: TYPE = $\{$i: nat | $\exists$ (j: nat): i = 2$\times$j$\}$

  half: [even_nat $\rightarrow$ nat]

  half_halves: AXIOM $\forall$ (e: even_nat): 2$\times$half(e) = e

- Expressions of the supertype allowed where the subtype is expected, provided defining predicate can be discharged in that context. For example,

  half_twice: LEMMA $\forall$ (i: nat): half(i+i) = i

- Generates the type-correctness condition (TCC)

  TCC1: OBLIGATION $\forall$ (i: nat): $\exists$ (j:nat): i+i = 2$\times$j

Example: Predicate Subtypes (Invariants)

- The specification of the transition function for a state machine will indicate how it computes the next state from the current state and input
- The "good" states of the state machine may be characterized by a predicate (e.g., `not stack_overflow(s)`)
- If the transition function is specified to have type `[good_state, input -> good_state]`
- Then typechecking will automatically generate all the proof obligations necessary to ensure that it is closed on the `good_states`

Example: Predicate Subtypes (Higher-Order)

- It often contributes clarity and precision to a specification if functions are identified as injections, surjections, etc.
- But how do we ensure that a purported surjection really has that property?
- Predicate subtypes! The surjections are a subtype of the functions associated with the following predicate

```
function_props[dom, rng: TYPE]: THEORY
  surjective?(f): bool = ∀ (r: rng): ∃ (d: dom): f(d) = r
```

So that

```
half: (surjective?[even_nat, nat]) = λ (e: even_nat): e/2
```

Generates the proof obligation

```
TCC2: OBLIGATION ∀ (n: nat): ∃ (e: even_nat): half(e) = n
```

Consistency of Formal Specifications

- It is disturbingly easy to write inconsistent axioms
- So some specification languages forbid (or strongly discourage) them, in favor of constructive definitions that are guaranteed conservative
  - But this is often too restrictive
    - ⋆ Eliminates property-oriented specifications
  - And sometimes inappropriate
    - ⋆ Want to describe the assumed environment, not implement it
- Formal methods tools should enable the consistency of axiomatic specifications to be demonstrated (e.g., exhibit a model)
- But should also provide a rich collection of definitional forms

Definitional Forms

- Can increase the variety and richness of definitional forms if prepared to use theorem proving to check them
- For example, recursive function definitions
  - Without theorem proving, may be restricted to the syntactic form of primitive recursion
  - With theorem proving, can accept definitions if arguments can be proved to decrease according to some well-founded ordering across recursive calls (note, system may need the sub-ϵ_0 ordinals if lexicographic orderings are to be accepted)
- Other desirable forms
  - Recursively defined abstract data types ("free types")
  - Inductively defined functions
- These structured definitional forms also help theorem proving

Example: Recursive Definition

- Here's the familiar factorial function

```
factorial(n: nat): RECURSIVE nat =
    IF n=0 THEN 1 ELSE n×factorial(n-1) ENDIF MEASURE n
```

  The termination TCC is what we expect:

```
TCC3: OBLIGATION ∀ (n: nat): n≠0 IMPLIES n-1 < n
```

- We also have a subtype TCC for `nat` from `n-1`:

```
TCC4: OBLIGATION ∀ (n: nat): n≠0 IMPLIES n-1 ≥ 0
```

  (The natural numbers are `nat: TYPE = { n: int | n ≥ 0 }`)

- Note how, together, these would identify the faulty termination condition `n=1`

Theorem Proving for Mechanized Formal Methods

- Interested in more than classical correctness
  - Animating specifications to *test* them
  - Challenging requirements to *validate* them
  - *Debugging* designs
  - *Deriving* test cases
  - *Pruning* and *sharpening* assumptions
  - *Adjusting* designs to changed requirements
  - *Exploring* alternative designs
- Hence, seldom proving true theorems
  - Need to return useful information from (rapid) failure
- Want to return more than "verified" on success
- And must cope with changes

Theorem Proving

- Tools for formal methods are built on theorem proving
- It is essential to make use of the insights and state-of-the-art techniques of that field
- But must recognize that the goals and assumptions of the theorem-proving community are not perfectly aligned with the needs of formal methods tools
  - Their goal is unassisted automation
  - They focus on proving true theorems
  - They automate raw logics, not specification languages

  Pure theorem provers are like dragsters; automating formal methods is more like Formula 1—many more challenges
  - But must go really fast when it *is* flat and straight
- Major choice: automatic or interactive theorem proving?

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 (ordered binary decision diagrams)
- 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
- 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
  - 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)

LCF-Style Tactics

- Given a repertoire of primitive inference steps
- Construct more powerful steps by writing programs (tactics) that use the primitive ones
- Soundness depends only on the primitive inferences
- "Pure" interpretation (HOL)
  - Basic steps should correspond to elementary rules of logic
  - Tactics use a general programming language (ML), reduce to large numbers of primitive inferences
- "Impure" interpretation (PVS)
  - Primitive steps should be chosen to provide the foundation for an efficient and powerful system
  - Tactics are simple combinations of primitive ones

The "Pure/Impure" Tradeoff With Tactics: 1

- Raw speed and power:
  - The pure systems must synthesize decision procedures from primitive inferences: much slower in practice than the built-in procedures of impure systems
  - And less powerful, because do not retain state, share data structures
- Effectiveness:
  - The pure systems have large numbers of higher-level tactics, hard to remember what they do, tend to be either very specialized or not very powerful

The "Pure/Impure" Tradeoff With Tactics: 2

- Soundness:
  - Admittedly more challenging to gain confidence in the soundness of powerful inferences, but not exactly hard (most of what they do is search: affects completeness and termination, not soundness)

- Overall:
  - Impure systems considerably more effective in practice
  - Can always find better uses for machine cycles during interactive proof development than grinding out primitive inferences
  - Use a "pure" second pass if paranoid
  - Or generate "proof objects"

The Theorem Proving Lifecycle

- There is a lifecycle to theorem proving in formal development
- 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–I

- 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

Loose Integration–II

- Both methods generally require downscaling
  - Unsound: success is not equivalent to verification
  - But errors discovered are likely to be real
- Tools should help with downscaling
- Use theorem proving on full system after downscaled system is debugged
- Can also apply to subgoals of ongoing proof as well as full conjectures

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 [20]

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 (and is in STeP and PVS)

Overview

- The rationale for *mechanized* formal methods
- Sketch of theorem proving and model checking
- Requirements on mechanized tools for formal methods
- $\boxed{\text{Examples and applications}}$
- Break
- Live demonstration using PVS

Using Formal Methods for Active Design Exploration

- Fault-tolerant clock synchronization algorithms are key elements in many safety-critical systems
- There are dozens of algorithms
- Schneider showed many can be given a uniform treatment
- Shankar formally verified this general treatment [24]
  - Establishes correctness of all algorithms in a certain class satisfying 11 assumptions
  - Found and corrected several small errors during verification
  - And showed interactive convergence satisfies the assumptions
- Paul Miner argued that one of the assumptions is very difficult to establish of a given algorithm
  - Almost as hard as proving synchronization

Active Design Exploration (ctd. 1)

- Miner showed troubling assumption could be discharged once and for all from suitably modified versions of the other ten

- And showed these are satisfied by the Lundelius-Lynch algorithm
  - Fault-tolerant midpoint: read the other clocks, exclude m slowest and m fastest, and set own clock to mid-point of remainder

- Then built a circuit to do it

- Incorporating a clever optimization due to Torres-Pomales
  - Run own clock at half speed after seeing $m+1$'st clock, until see $n-m$'th

Active Design Exploration (ctd. 2)

Formal methods applied to the optimized clock synchronization circuit [15]

- DDD (stream-based transformation system for hardware) used for routine design
- BDD-based tautology checker used for low-level optimizations
- PVS used for the clock-rate optimization
  - Needs fairly heavy arithmetic
  - Connection to DDD via streams (formalized as a co-inductive type)
  - Optimization verified as a bisimulation on streams

Integrated Theorem Proving and Model Checking

- Havelund and Shankar formally verified a bounded-retransmission communications protocol from Philips [12]
- 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-retransmisison 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
- Need to integrate with additional tools for generating abstractions and invariants

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 using LTL

Industrial Applications

- Applications to hardware have been more successful (in terms of bang for the buck and industrial takeup) than for software
  - Hardware applications rather stereotyped, so can invest in finding effective methods for given class of problems
- Boolean equivalence checking is incorporated in CAD tools
- Research groups in most processor design shops (AMD, Cadence, Fujitsu/HaL, HP, IBM, Intel, Motorola, Rockwell, SGI, Sun, TI)
  - Mostly model checking and BDD/BMD methods
  - Some theorem proving (Cadence, IBM, Intel, Rockwell)

Hardware Applications

- Routine applications for model-checking and BDD/BMDs
  - Cache coherence protocols (snoopy and directory) [6]
  - Floating point units (since Pentium bug) [5]
  - Pipeline correctness [4]
- Theorem proving is also able to handle these effectively
  - Flash cache-coherence (Park using PVS [18])
  - Subtractive division methods (including SRT), and IEEE compliance (Rueß, Shankar, Srivas [22], and Miner and Leathrum [16] using PVS)
  - Newton-Raphson division and square root methods, and IEEE compliance (Moore and Russinoff using ACL2 [3])
  - Pipeline correctness (Miller and Srivas using PVS [28])
- Superscalar is the current big challenge

Software Applications

- Until recently, many applications were to low-level code correctness or refinement (e.g., SACEM, B-Tool applications) where bang for the buck is very poor
  - Traditional processes are very effective
  - Specifications are large
  - Need to deal with imperative programming language issues
- Now, focus has shifted to requirements
  - Where most of the errors are
  - Existing processes are ineffective
- And to intrinsically hard algorithmic problems
  - E.g., distributed, concurrent, fault-tolerant, real-time, and cryptographic protocols
  - Where the rest of the errors are

Software Applications (ctd.)

- And to "simple theorems about big systems"
  - Very strong typechecking
  - Ensure absence of runtime errors
  - Completeness and consistency checking of tabular specifications

Example Software Applications

- Requirements, simple theorems
  - Space shuttle CRs (Crow and Di Vito using PVS [7])
  - TCAS (Czerny and Heimdahl using BDDs and PVS and SVC [13])
  - A7 (Atlee using SMV [26])
  - Z-like specifications (Jackson with Nitpick [14])

- Code, simple theorems
  - C-130J code (Praxis using Spark Examiner [8])
  - Extended Static Checking (Detlefs, Nelson and Sachs with Simplify [9])

Example Software Applications (ctd.)

- Intrinsically hard problems
  - Railroad switching (Groote using Stålmarck's method, BDDs [11])
  - Cryptographic protocols (Roscoe using FDR [21])
  - Communications protocols (notably some from Philips)—everyone
  - Fault-tolerant algorithms (SRI, NASA using PVS [17])
  - Distributed algorithms (MIT using Larch [19])
  - CAML garbage collector (Gonthier using TLP—22,000-line proof script! [10])

Challenges for the Future—Technical

- Tighter integration of theorem proving and model checking, and more cross-fertilization of techniques
- More powerful individual theorem proving methods
  - More general and faster decision procedures
  - Decision procedures for more theories
  - More effective quantifier instantiation in presence of decision procedures
- Deconstruction of monolithic systems
  - Theorem provers as open systems whose components can be used individually and composed in new ways

Challenges for the Future—Methodological

- More integration of analysis techniques
  - So that a graduated series of techniques can be applied, each finding more bugs, at correspondingly greater cost, than its predecessors

- Integration between mechanized formal methods and traditional methods
  - Testing
  - Object-oriented design
  - Efficient support for domain-specific languages and problems

Overview

- The rationale for *mechanized* formal methods
- Sketch of theorem proving and model checking
- Requirements on mechanized tools for formal methods
- Examples and applications
- $\boxed{\text{Break}}$
- Live demonstration using PVS

Overview

- The rationale for *mechanized* formal methods
- Sketch of theorem proving and model checking
- Requirements on mechanized tools for formal methods
- Examples and applications
- Break
- $\boxed{\text{Live demonstration using PVS}}$

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

To Learn More

- Browse papers and technical reports at
  `http://www.csl.sri.com/fm.html`
  - `tse95.html` is a good overview
  - `cav95.html` describes the model checker integration in PVS
  - Links to Stanford papers and to Murϕ there also
- 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 4.1.X or Solaris, or IBM AIX (Linux coming soon)
  - Need 32M memory, 100M swap space, Sparc 2 or better

References

[1] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[2] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Srivas and Camilleri [27], pages 187–201.

[3] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Srivas and Camilleri [27], pages 275–293.

[4] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[5] Yirng-An Chen, Edmund Clarke, Pei-Hsin Ho, Yatin Hoskote, Timothy Kam, Manpreet Khaira, John O'Leary, and Xudong Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In Srivas and Camilleri [27], pages 19–33.

[6] Edmund M. Clarke, Orna Grumberg, Hiromi Haraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of

the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[7] Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements: Four case studies. Submitted for publication, 1997.

[8] Martin Croxford and James Sutton. Breaking through the V and V bottleneck. In Marcel Toussaint, editor, *Ada in Europe*, volume 1031 of *Lecture Notes in Computer Science*, pages 344–354, Frankfurt/Main, Germany, October 1995. Springer-Verlag.

[9] David L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, San Diego, CA, January 1996. Association for Computing Machinery.

[10] Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In Alur and Henzinger [1], pages 462–465.

[11] J. F. Groote, J. W. C. Koorn, and S. F. M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *COMPASS '95 (Proceedings of the Tenth Annual Conference on Computer Assurance)*, pages 57–68, Gaithersburg, MD, June 1995. IEEE Washington Section.

[12] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME*

'96, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[13] Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, Niagara on the Lake, Canada, October 1996.

[14] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[15] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, September 1996. Springer-Verlag.

[16] Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Srivas and Camilleri [27], pages 64–78.

[17] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[18] Seungjoon Park and David L. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 1996.

[19] T. P. Petrov, A. Pogosyants, S. J. Garland, V. Luchangco, and N. A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques (FORTE) IX: Theory, Application and Tools*, pages 29–44, Kaiserslautern, Germany, October 1996. Chapman and Hall, on behalf of IFIP.

[20] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[21] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th Computer Security Foundations Workshop*, pages 98–107, Kenmare, Ireland, June 1995. IEEE Computer Society.

[22] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Alur and Henzinger [1], pages 123–134.

[23] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory,

SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification). Reprinted in [25, pp. 1–42].

[24] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

[25] Roger Shaw, editor. *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*, Bruges, Belgium, September 1995. Springer.

[26] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 77–88, Gaithersburg, MD, June 1996. IEEE Washington Section.

[27] Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.

[28] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen,

editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.