Fault-Tolerant Algorithms and the Design of PVS*

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

*PVS was built by Sam Owre and N. Shankar, with help from David Cyrluk, Pat Lincoln, Steven Phillips, Sreeranga Rajan, and Carl Witty. Its development was funded by SRI International

Overview

- Formal methods applied to fault-tolerant algorithms and redundancy management for digital flight control
- Lessons learned about how mechanized formal methods can contribute to systems engineering and analysis: there's more to it than just proving correctness
- Lessons learned about requirements for tools
- And how these influenced the design of PVS

Perspective

- 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 calculation (specifically, formal deduction)

- Thus 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 RTCA DO-178B/EUROCAE ED-12B)

Mechanization of Formal Methods

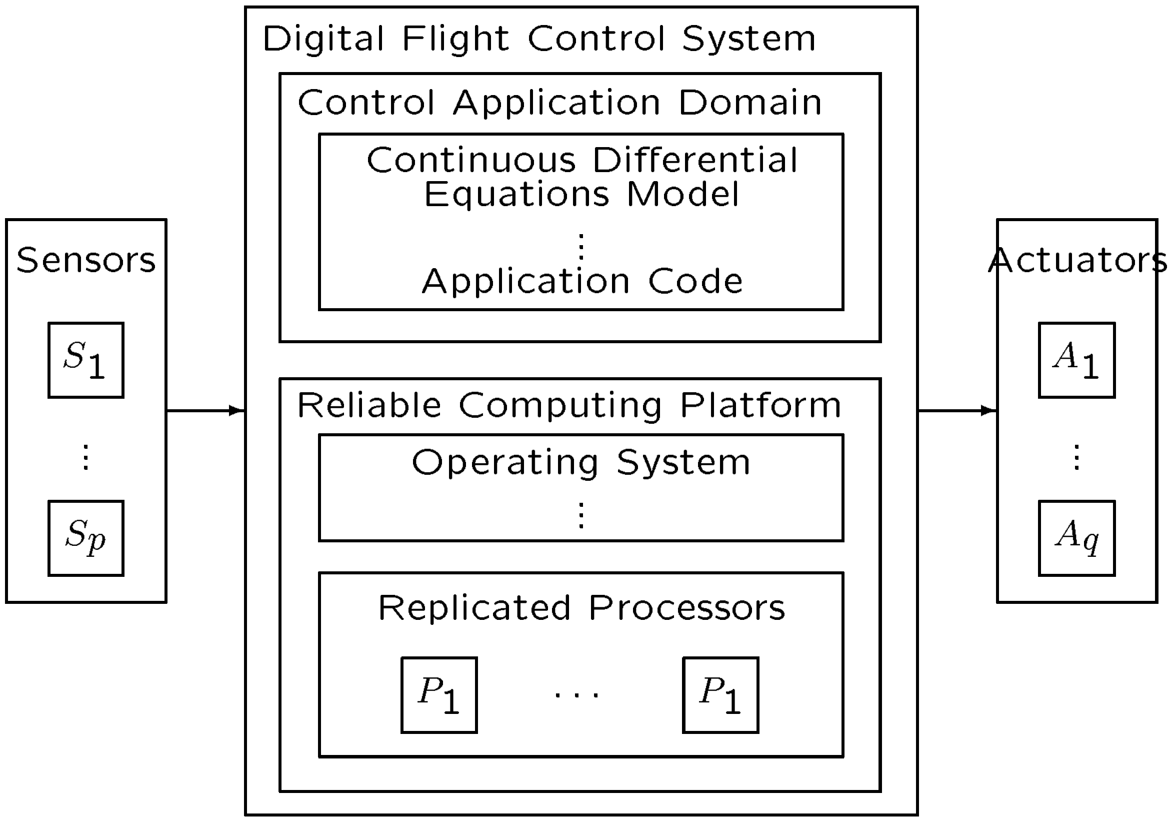
- Just like any other engineering calculations, it's tools that make formal calculations feasible in practice
- Tools are not the most important thing about formal methods
  - They are the *only* important thing
- Building tools is an engineering enterprise
- The most important attribute of a tool is its speed and power
- If it cannot deal with problems of practical interest at reasonable cost, doesn't matter what other attributes it has

Context

Goal: develop a formally verified architecture, algorithms, and implementations for a reliable computing platform for digital flight control based on state machine replication

Reliable Computing Platform: a distributed collection of processors, with associated sensors and actuators, and an "operating system" providing fault tolerance and redundancy management so that to an application task, it all looks like a single computer that never fails

Reliable Computing Platform



Context (ctd. 1)

Formally Verified: Construct a mathematical model of the system and its environment and prove a theorem of the form

if certain assumptions are true
  and there are enough nonfaulty processors
then overall system masks all faults

Assumptions: must be validated in the real world (e.g., nonfaulty processors can read each other's clocks with error at most ϵ)

Enough nonfaulty processors: must collect empirical data on individual failure rates, then use (Markov) modeling

Context (ctd. 2)

State machine replication:

- Keep processors synchronized $\surd$
- Distribute identical sensor data to all processors $\surd$
- Run identical workloads on all processors
- Perform exact-match majority voting on values sent to actuators (provides fault masking) $\surd$
- Incrementally refresh state of all processors with majority-voted versions (provides transient recovery) $\surd$
- Optionally, diagnose faulty processors and reconfigure (increases number of faults that can be tolerated) $\surd$

Context (ctd. 3)

Why do this?

Because redundancy management is among the most difficult problems in digital flight control

- Historically the *primary* source of system failure

Why state machine replication?

Because it has a rational justification

And because it can withstand arbitrary (aka. Byzantine) faults

Why does that matter?

Because then you don't have to provide evidence that faults outside your assumed class cannot occur

Why do formal verification if system is fault tolerant?

Because it's fault tolerant wrt. *hardware* faults; the algorithms and implementations of fault tolerance are single points of failure: they must be correct

Formal Verifications Performed: Clock Synchronization

Interactive Convergence Byzantine-fault-tolerant clock synchronization algorithm

- At the time (1988), one of the hardest computer-science verifications completed
- Over 200 lemmas (small theorems), 1700 lines of specification and proof commands (in EHDM)
- Found that in the journal presentation by Lamport and Melliar Smith (JACM), proof of the main theorem is flawed and 4 of the 5 lemmas are false
- Corrections required adjustment to assumptions
    - Also eliminated approximations
    - And developed more uniform proofs

Clock Synchronization (ctd. 1)

Our proof was subsequently repeated by Bill Young using Boyer-Moore prover

- He noted that our characterization of a good clock

$$|c(T_1) - c(T_2) - (T_1 - T_2)| < \frac{\rho}{2}|T_1 - T_2|$$

  is unsatisfiable when $T_1 = T_2$, and also excludes perfect clocks (those for which $c(T) = T$) [$<$ should be $\leq$]

- Shows that modeling must be validated very carefully

- Mechanization can help: show that defined predicates can be both satisfied and falsified, and that axioms are satisfied by intended models (also establishes their consistency)

Clock Synchronization (ctd. 2)

- Erwin Liu designed and formally verified a circuit to perform part of the algorithm
- Found the assumption that initial clock adjustments are all zero is unrealistic
- Deleted this assumption from the original verification and reran all proofs
- Found that proofs of some internal lemmas need adjusting, but main argument is unchanged

Clock Synchronization (ctd. 3)

- There are many clock synchronization algorithms
- Shankar verified general treatment due to Fred Schneider that includes almost of them (using EHDM)

  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)
- He showed that this assumption could be discharged once and for all from suitably modified versions of the other 10
- And did it for the Lundelius-Lynch algorithm

Formal Verifications Performed: Distributed Consensus

Oral Messages algorithm for distributed consensus (aka. Byzantine Agreement—distribution of consistent sensor values in the presence of faults)

- First verified by Bevier and Young using Boyer-Moore prover (they also verified its optimality and an implementation)
- We found a rather simpler treatment and used it as a test case in developing PVS (it's an order of magnitude simpler than clock synchronization)
- Then studied important variation (developed at Allied Signal for MAFT flight control computer) that uses "Hybrid" fault model

Hybrid Fault Models

- Classically, algorithms for state-machine replication consider components to be either faulty or nonfaulty
- No assumptions made about faulty components
- Tolerates any *kind* of fault, but requires a lot of redundancy, so cannot tolerate a large *number* of faults
(for a given level of replication)
- "Hybrid" fault models include "arbitrary" fault mode, but also consider some common, simple kinds of fault ("manifest" and "symmetric") and can tolerate more of these
- "Hybrid" algorithms are strictly superior to pure "Byzantine" ones (e.g., $n > 3a + 2s + m$ rather than $n > 3f$)
- But their analysis is much more complex

Distributed Consensus (ctd. 1)

- Algorithm Z, published with detailed proof of correctness by Thambidurai and Park, performs distributed consensus under hybrid fault model

- Lincoln and Rushby found a bug in the algorithm while attempting formal verification (implementation in MAFT is slightly different, and is OK)

- Developed modified algorithm and verified its correctness

  (Verified an *incorrect* variant on the way—bug was masked by an incorrect axiomatization of "hybrid majority vote")

- Subsequently adapted this algorithm to Draper FTP architecture and verified it in a couple of days

Clock Synchronization (Again)

Returned to clock synchronization to extend benefits of hybrid fault model to that problem

- Original verification had taken a couple of months, has over 200 lemmas
- Extension to hybrid case requires several small adjustments (e.g., faulty/nonfaulty dichotomy must be revised)
- Consideration of additional cases in Lemma 5
- And a more complicated counting argument in the main theorem
- Overall, a couple of day's work
- Algorithm is about the best known; also does well against link faults

Distributed Consensus (Again)

- Hybrid oral messages algorithm doesn't do well against link faults
- There is another class of distributed consensus algorithms that use authenticated communications: "lieutenants" cannot lie
- But they break if authentication is flawed
- Can also add authentication to Oral Message algorithms
- Simple worst-case analyses do not distinguish between the various candidate algorithms
- Cannot find good characterizations of behavior in presence of link faults

Exploring Design Alternatives with Formal Methods

- Murϕ is an explicit state exploration tool developed by David Dill at Stanford
  - Given a specification of a finite-state concurrent system, explores *all* behaviors by brute-force
- Used Murϕ to do exhaustive "symbolic fault injection" for five-processor configuration
- Rediscovered the bug in Algorithm Z
- But found that ZA works very well
- Seems a generally useful way to evaluate fault-tolerant algorithms

Formal Verifications Performed

Insufficient time to describe others:

- Verification of overall architecture and transient recovery
  - Done by Rick Butler, Ben DiVito and others at LaRC
  - The biggest verification done with EHDM (and probably the biggest by anyone using someone else's tools)
- Further properties of clock synchronization
  - Initial synchronization and transient recovery
  - Done by Paul Miner
- Circuit for clock synchronization
  - Verification used a combination of PVS, DDD, and a BDD-based tautology checker
  - Done by Paul Miner
- Diagnosis algorithms (done by Pat Lincoln in PVS)

Lessons Learned (in *this* domain)

- None of what we have done is "program verification" (i.e., proving a program correct wrt. its detailed specifications)

- In this domain, concern centers on the *hardest and most difficult problems of design*: the redundancy management architecture and algorithms that ensure continued safe operation despite hardware failures

- Intrinsically difficult problems: reasoning about distributed, concurrent, real-time systems operating in the presence of faults
  - Number of possible behaviors is vast

Lessons Learned (ctd. 1)

For intrinsically difficult problems and where systems can exhibit huge numbers of different behaviors

- Formal methods can add a lot of value: assurance through testing is totally infeasible, hand proofs are unreliable
- Need to understand quite a lot about the problem domain to apply formal methods usefully
- Need to be skilled at abstraction to apply formal methods effectively
- Can be accomplished by a few highly skilled people: don't need to train every programmer

Lessons Learned (ctd. 2)

Mechanized formal methods provide much value in addition to proofs of correctness

- Reveal errors in published proofs and algorithms—i.e., discovery of *in*correctness
- Expose all assumptions and assist in eliminating them and in sharpening their statements
- Support development of streamlined arguments and enhanced understanding that can lead to improvements in algorithm or architecture
- Allow inexpensive and reliable exploration of alternative designs
- And adaptation to changed requirements or assumptions

Lessons Learned (ctd. 3)

To achieve these benefits, mechanizations of formal methods must have certain attributes

- A civilized specification language
- Mechanisms for early error detection
- A *lot* of theorem-proving horsepower
  - That can be kept under control
  - And that provides more information than just "proved" and silence
- Mechanized deductive support in addition to theorem proving
- Support for changes and exploration

The Design of PVS

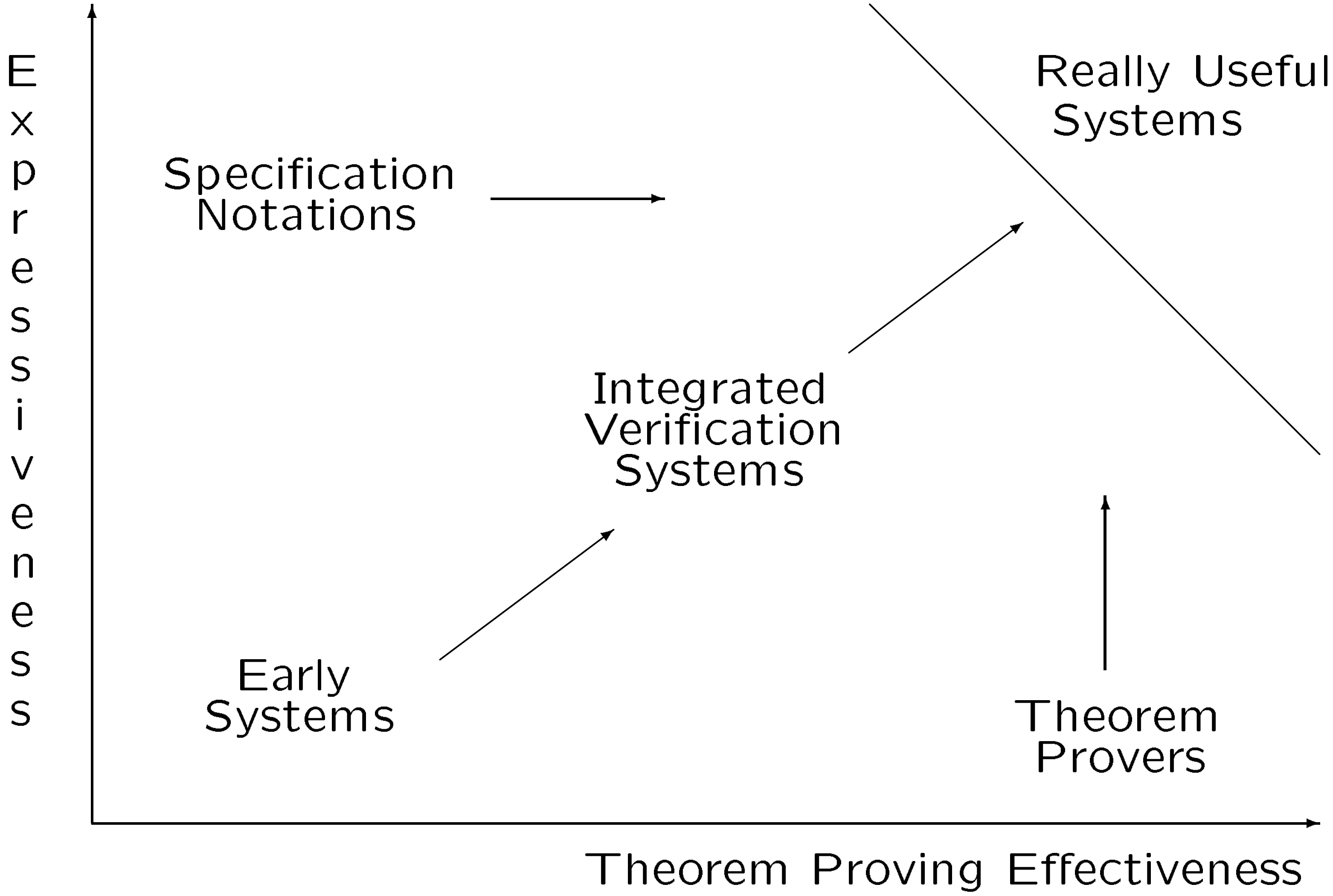
PVS had other influences besides these examples

- Hardware (Saxe pipeline, Tamarack, Cantu ALU, and AAMP5 examples)
- Requirements (Jet-Select example, A7 and SCR methods)
- Real-time (railroad crossing and other examples)
- Embeddings of other logics (Duration Calculus)
- Our previous systems (EHDM, Muse)
- Other systems (principally HOL, Boyer-Moore, and SMV)

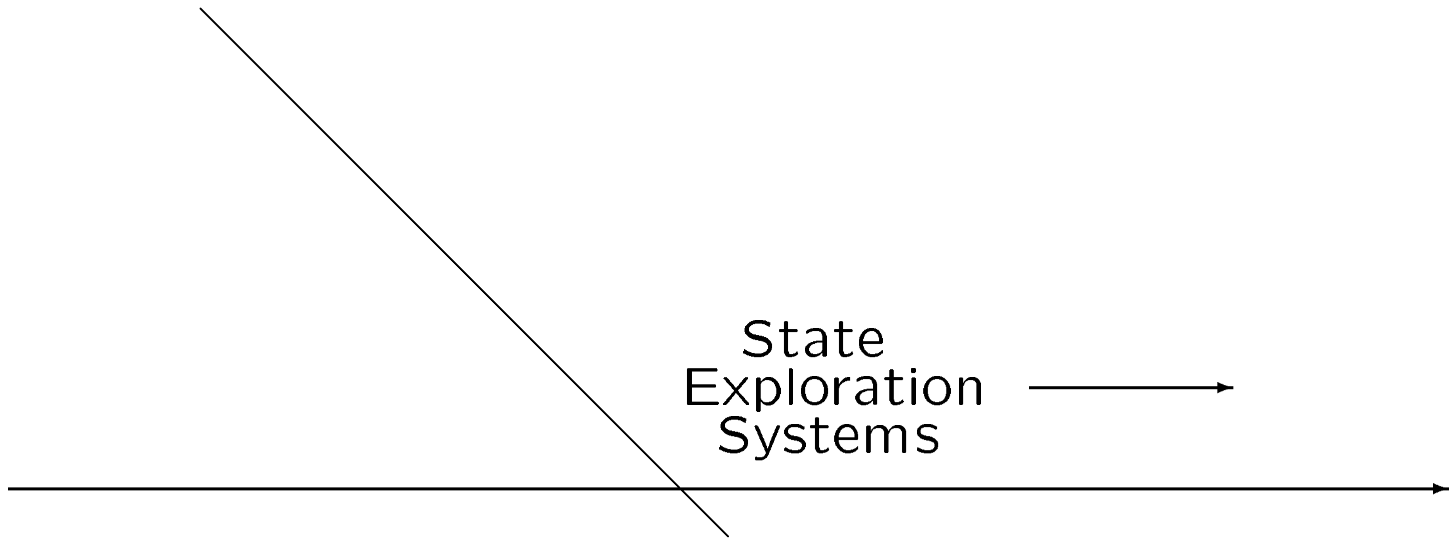
A Civilized Specification Language

- 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 | ∃ (j: nat): i = 2×j}
  half: [even_nat → nat]
  half_halves: AXIOM ∀ (e: even_nat): 2×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 ∀ (i: nat): half(2×i) = i
  ```

- Generates the type-correctness condition (TCC)

  ```
  TCC1: OBLIGATION ∀ (i: nat): ∃ (j:nat): 2×i = 2×j
  ```

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`

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

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
- Bit 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's 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
  - 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

The Theorem Proving Lifecycle During Formal Development

There is a lifecycle to theorem proving during 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

Design Choices in PVS Prover

- Theorem prover is interactive
  - Uses sequent calculus presentation
  - Decision procedures for ground linear arithmetic and equality
  - Hashing conditional rewriter integrated with these
  - Strategy language for defining higher-level proof procedures
- Theorem prover can call external BDD package
  - For simplification of large propositional structures
  - And to decide formulas in μ-calculus
    Used to automate CTL model checking (like SMV)
- Lemmas can be proved in any order
  (and introduced and modified on the fly)
- Proof dependency analysis keeps you honest
- Graphical displays of proof trees help comprehend big picture

Conclusions

- Verifications described were once *tours de force*, now routine
- Provided benefits over and above verification: debugging, exploration, adaptation
- Startup costs of applying formal methods to a new application area can be quite high
  - Challenge is usually in the modeling and abstraction, not the mechanization
  - Requires skilled people
- Subsequent exploitation inexpensive, with very high added value
  - Exploration of design alternatives, adapting to changes
  - Reuse in subsequent designs
- Mechanized formal methods are ready for industrial exploitation in suitable circumstances

To Learn More

- Browse papers and technical reports in `/pub/reports` on `ftp.csl.sri.com` (start with `readme.txt` and `abstracts.dvi.Z`), or use Mosaic or other WWW viewer on `http://www.csl.sri.com/sri-csl-fm.html`
  - CSL-95-1 is a chapter for the FAA Digital Systems Validation Handbook that discusses issues in formal methods and assurance
  - CSL-93-7 is a much longer and more technical treatment
- You can get PVS by anonymous FTP from `ftp.csl.sri.com` in `/pub/pvs` (also from WWW link at URL above)
  - Allegro Lisp for SunOS 4.1.3 (Solaris 2 available soon); Need 32M memory, 100M swap space, Sparc 2 or better
  - PVS2 is available for testing by current users; General release awaits completion of documentation