Theorem Proving and Hardware Verification

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Theorem Proving: What Is It?

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

  as formulas in a single logic
- And prove that

$$background + assumptions + system \vdash requirement$$

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

Model Checking: What Is It?

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

$$system \models requirement$$

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

Theorem Proving: Compared with Model-Checking

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

So Why The Bad Rap? — Wrong Focus

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

- Irrelevance
  - Not driven by real examples

So Why The Bad Rap? — Technical Shortcomings

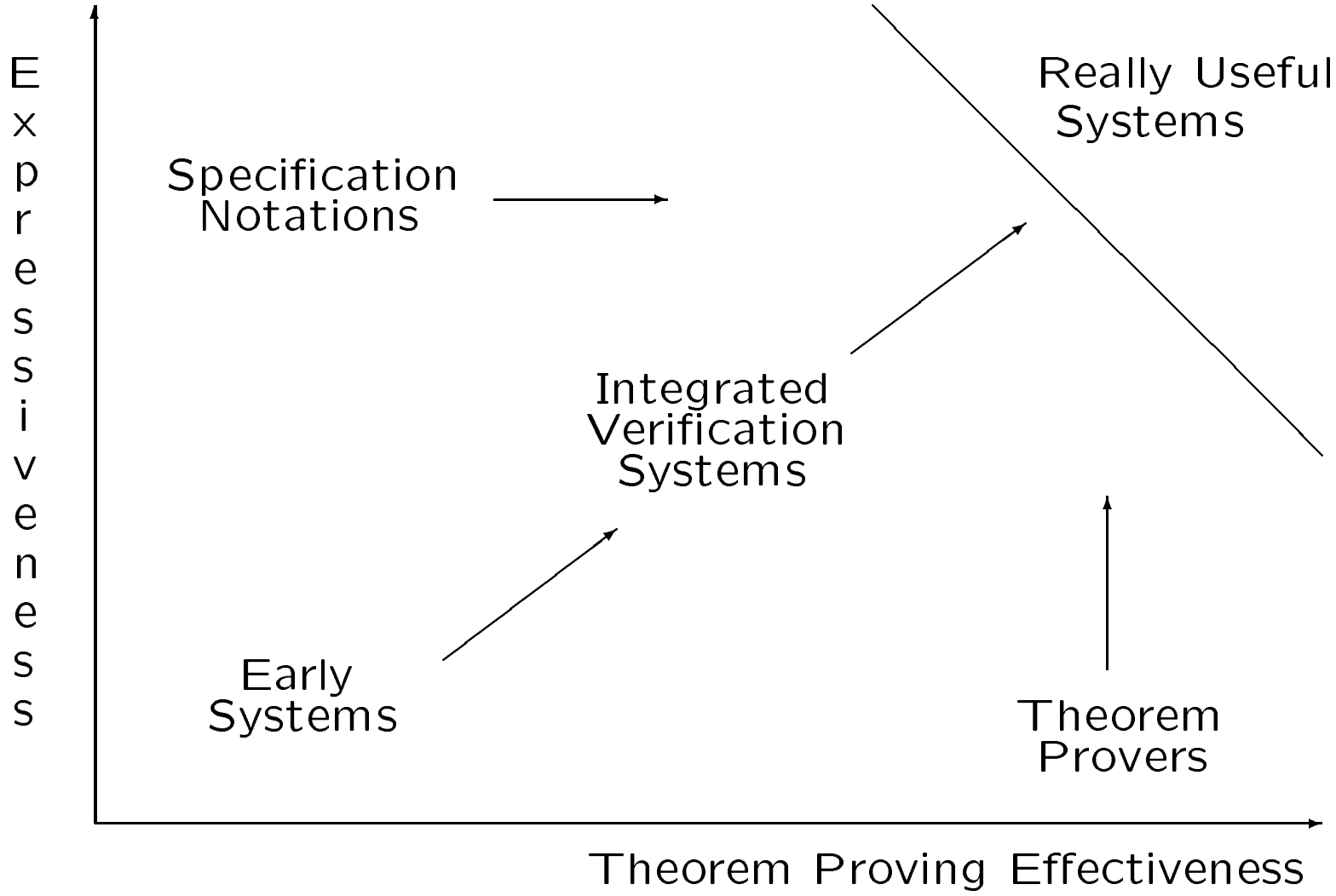
- The field of automated deduction provides important techniques, but has a different focus
  - Its goal is unassisted automation
  - It focuses on proving true theorems
  - It automates raw logics, not specification languages
  - Its applications are math and AI

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

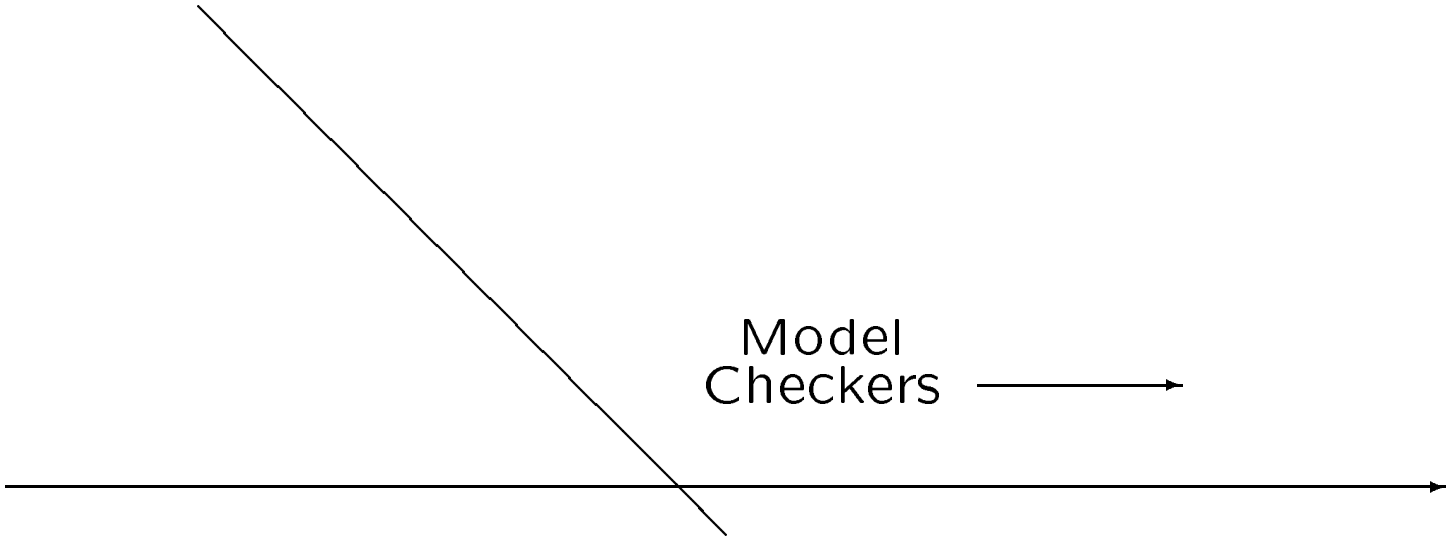
Specification Languages and Mechanization

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

The Three Traditions



And A Fourth



Automatic Theorem Proving: The Downside

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

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

Automatic Theorem Proving: The Upside

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

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

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

Interactive Proof Checking: The Downside

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

Interactive Proof Checking: The Upside

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

Design Choices in the 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 the 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
- Retains state over separate invocations
- 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

Design Choices in the 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

Example SRT Division: Derivation and Specification

- The famous Pentium FDIV bug was in the quotient lookup table of a radix-4 SRT divider
- SRT now something of a test-piece for formal verification
- BDDs grow exponentially for multipliers and dividers
- Nonetheless, Bryant verified one iteration of SRT by comparison against a simpler divider circuit
- Clarke, Khaira and Zhao used BMDs and word-level model checking to verify a (manually constructed) invariant for SRT
- Ho et al extended this to other algorithms of Weitek FPU and to rounding unit
- German and Clark used a theorem prover built on Mathematica to verify a particular SRT circuit and table

SRT: Derivation and Specification in PVS

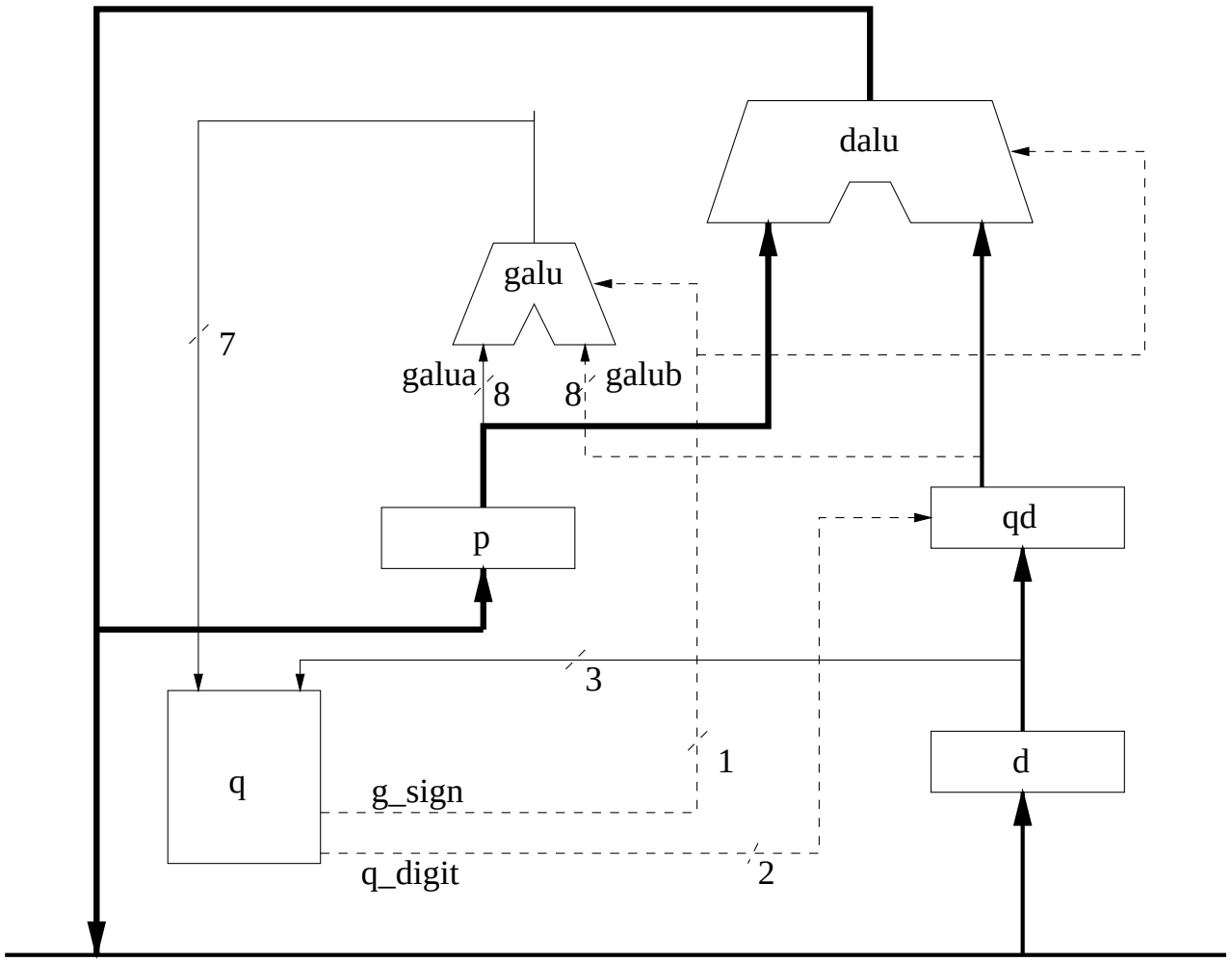
- PVS treatment specifies and verifies the general theory of SRT division and derives the conditions for the radix-4 lookup table

- Then verifies a particular table (similar to the Pentium's) due to Taylor (1981)
  - Need to show blank entries are inaccessible
    - This was the FDIV bug
    - PVS specification uses the `TABLE` construct, which automatically generates proof obligations (TCCs) for blank entries
  - And that other entries have the right values

Quotient Lookup Table

```
q(D: bvec[3], (P: bvec[7] | estimation_bound?(valD(D), valP(P)))): subrange(-2, 2) =
LET a = -(2 - P(1) * P(0)), b = -(2 - P(1)), c = 1 + P(1), d = -(1 - P(1)), e = P(1),
    Dp:nat = bv2pattern(D), Ptruncp:nat = bv2pattern(P^(6,2))
IN TABLE Ptruncp, Dp
                |[ 000| 001| 010| 011| 100| 101| 110| 111]|
              %-----------------------------------------%
              |01010|    |    |    |    |    |    |    2 ||
              |01001|    |    |    |    |  2 |  2 |  2 ||
              |01000|    |    |    |  2 |  2 |  2 |  2 ||
              |00111|    |    |  2 |  2 |  2 |  2 |  2 ||
              |00110|    |  2 |  2 |  2 |  2 |  2 |  2 ||
              |00101|  2 |  2 |  2 |  2 |  2 |  2 |  1 ||
              |00100|  2 |  2 |  2 |  2 |  c |  1 |  1 ||
              |00011|  2 |  c |  1 |  1 |  1 |  1 |  1 ||
              |00010|  1 |  1 |  1 |  1 |  1 |  1 |  1 ||
              |00001|  1 |  1 |  1 |  1 |  e |  0 |  0 ||
              |00000|  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
              |11111|  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
              |11110| -1 | -1 |  d |  d |  0 |  0 |  0 ||
              |11101| -1 | -1 | -1 | -1 | -1 | -1 | -1 ||
              |11100|  a |  b | -1 | -1 | -1 | -1 | -1 ||
              |11011| -2 | -2 | -2 |  b | -1 | -1 | -1 ||
              |11010| -2 | -2 | -2 | -2 | -2 | -2 |  b | -1 ||
              |11001| -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 ||
              |11000|    |    | -2 | -2 | -2 | -2 | -2 | -2 ||
              |10111|    |    |    | -2 | -2 | -2 | -2 | -2 ||
              |10110|    |    |    |    |    | -2 | -2 | -2 ||
              |10101|    |    |    |    |    |    | -2 | -2 ||
              %-----------------------------------------%
    ENDTABLE
```

SRT Division: Datapath



SRT Division: Correctness

Need to show quotient selection logic and lookup table satisfy the predicate

```
lookup_legitimate?(q,D,(P:rational | estimation_bound?(D, P))): bool =
  COND
   q =  2 -> 4/3 * (D + delta) <= P,
   q =  1 -> 1/3 * (D + delta) <= P & (P + eps) <= 5/3 * D,
   q =  0 -> -2/3 * D <= P & (P + eps) <= 2/3 * D,
   q = -1 -> -5/3 * D <= P & (P + eps) <= -1/3 * (D + delta),
   q = -2 -> (P + eps) <= -4/3 * (D + delta)
  ENDCOND
```

SRT Division: Verification I

- Proof is a simple case split on all possible values of indices `P` and `D` followed by a call of decision procedures.
- Generates 256 subgoals and proves all but one of them automatically
- Remaining subgoal has a nonlinearity and needs an extra case split on the remaining bits `P!1(1)` and `P!1(0)`
- Proofs originally took several hours, now down to minutes using new optimizations, and decision procedures from Stanford

SRT Division: Verification II

- The PVS `table` construct automatically generates the TCCs (proof obligations) to show that the blank areas in the table are unreachable
- These 23 TCCs are proved in same way as correctness theorem
- Injection of an error similar to that in the Pentium leads to a failed TCC proof whose final sequent is a counterexample that highlights the error

```
  P!1(6) = 1, P!1(5) = 1, P!1(4) = 0,
  P!1(3) = 0, P!1(2) = 0
  estimation_bound?(...)
  D!1(2) = 0, D!1(1) = 1, D!1(0) = 0
 |-------
  lookup_legitimate?(...)
```

Counterexample: `D = 010, P = 11000`

SRT Division: Generalization in PVS

- Miner and Leathrum extended the PVS treatment to encompass the broader class of subtractive division algorithms
- Presented a methodology that enables specific algorithms to be debugged and verified quite easily
- Can be used as a design tool: demonstrated by developing and verifying new SRT tables in under three hours each
- Also formalized IEEE 854 in PVS
- And verified IEEE-compliance of rounding unit

Technology Transfer

- SRI worked with Collins Commercial Avionics to apply formal methods to Collins avionics processors
- Projects sponsored by NASA Langley Research Center
- Goal: transfer SRI technology to Collins (and for SRI to learn how our technology scaled to real examples)
- First project (two years): specification and partial verification of microcode of (complex) AAMP5 microprocessor
  - SRI in the lead
- Second project (in progress): complete specification and verification of (simpler) AAMP-FV microprocessor
  - Collins in the lead

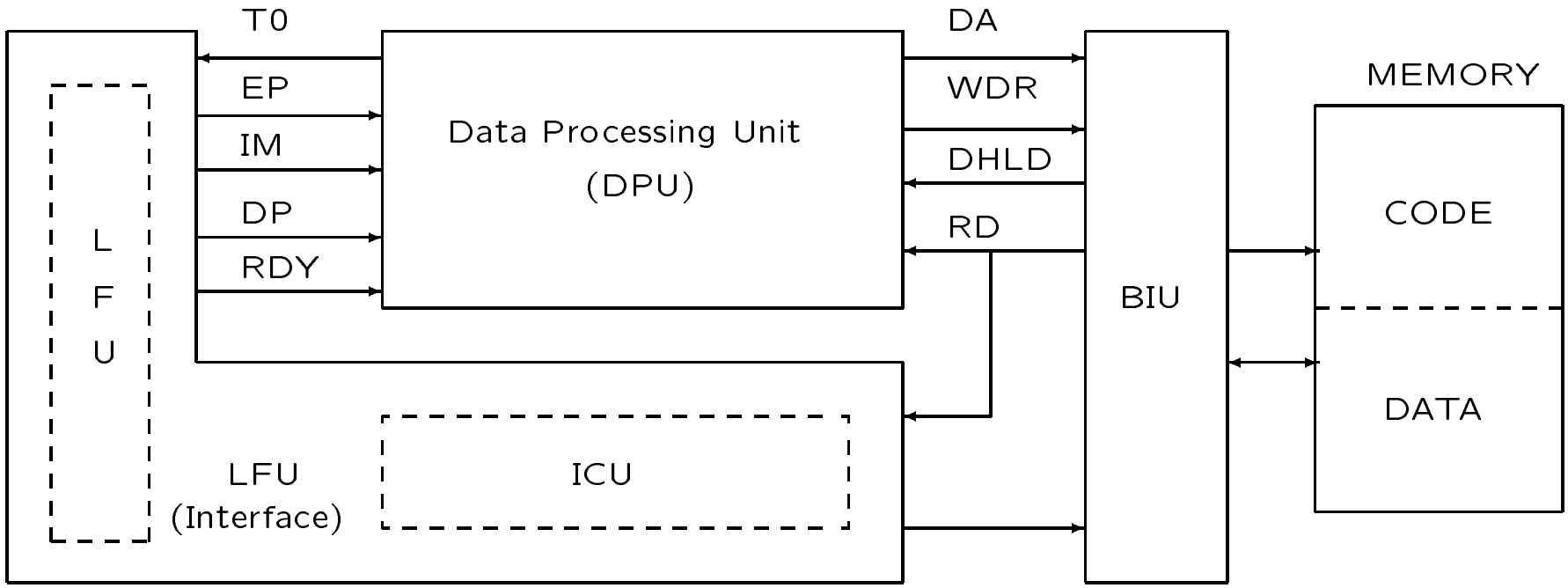
AAMP5 Microprocessor

- Not designed for formal methods, but to provide fourfold performance improvement over AAMP2
- Numerous applications—e.g., 30 AAMP2s on each 747-400
- Used for critical, but not ultra-critical, applications (e.g., instrumentation and displays, not autoland)
- Derived from CAPS architecture, dating back to 1974
  - When hardened memories were very small—needed high density for code and data
- So uses stack architecture (many zero-address instructions)
- Intended for embedded applications and code generated by high-level languages (Ada)—provides hardware support for runtime environment and real-time executive (209 instructions)
- Very different from RISC architectures

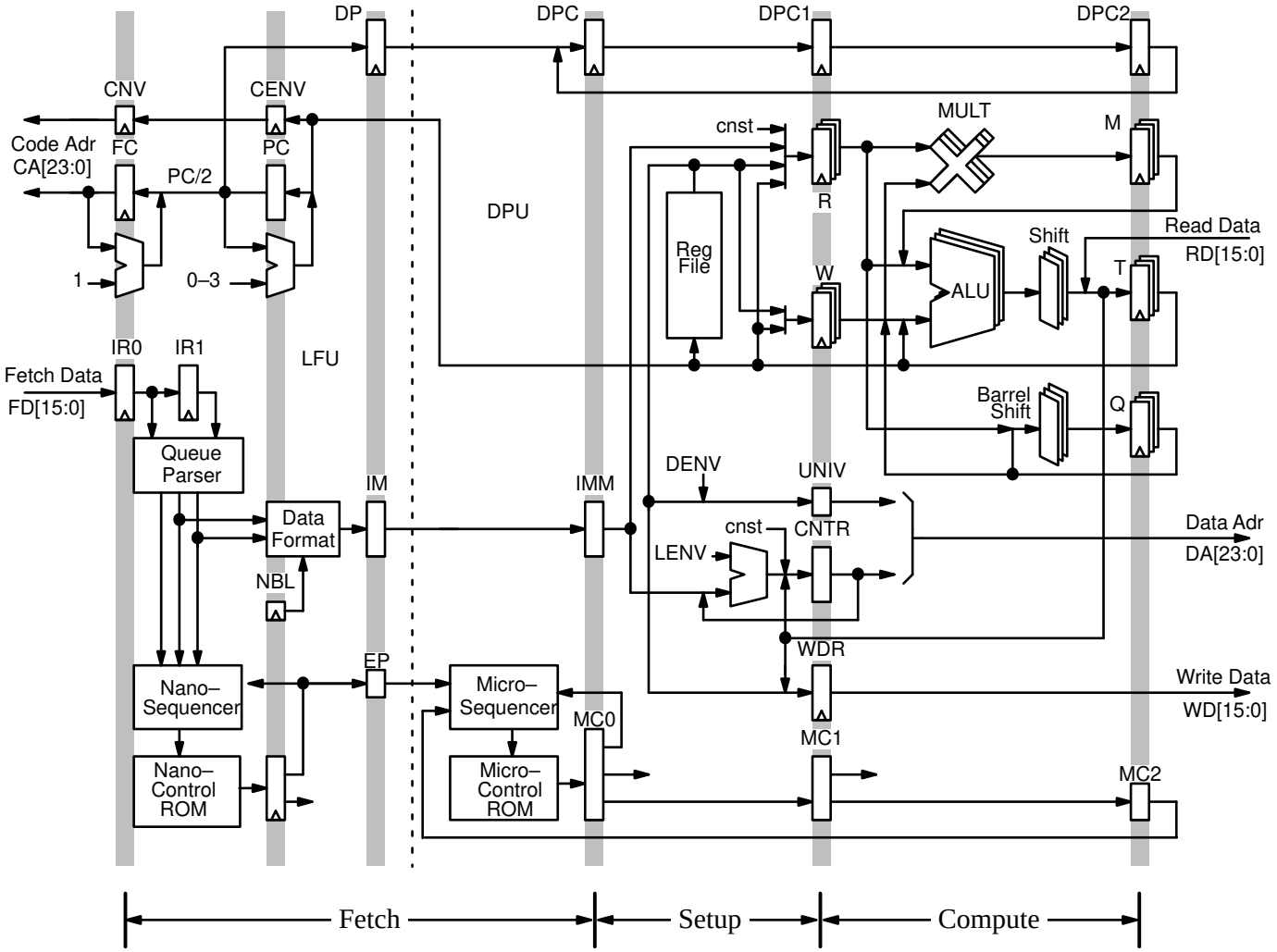
AAMP5 Microprocessor: Complexity

- Microcoded implementation
- Variable length instructions
- Also multiple word sizes
  - Complex bitvector operations outside ALU
- The upper portion of the stack is cached in processor registers
  - Automatically adjusted prior to each instruction–takes variable amount of time
- Three-stage pipeline with data and control hazards
- Semi-autonomous subunits for asynchronous instruction and data prefetch, and memory and bus transfers
- Implementation has 500,000 transistors

AAMP5 Block Diagram



AAMP5 Datapath



AAMP5 Microprocessor: Comparison with RISC

- The AAMP verifications are technically challenging
  - Variable instruction timings
  - Stack cache not completely transparent
  - Complex instructions, microcode
  - Complex bitvector operations outside ALU
- A RISC processor (single pipe) would actually be easier
  - Our automation would deal with the deeper pipe
- Microcode accounts for 30% of bugs in Intel designs

Technical Goals

- Formally specify as much of the AAMP5 macroarchitecture and instruction set as feasible (actually did 108 of 209 instructions)

- Formally specify the microarchitecture of the AAMP5 DPU at the RTL level
  - Abstract the DPU environment: interaction with LFU and BIU characterized by rely-guarantee assertions

- Formally verify a core set of 11 instructions representing the major instruction classes
  - I.e., prove that execution of microcode by microarchitecture achieves specified behavior for that instruction
  - Collins seeded subtle errors in microcode for two instructions (unknown to SRI)

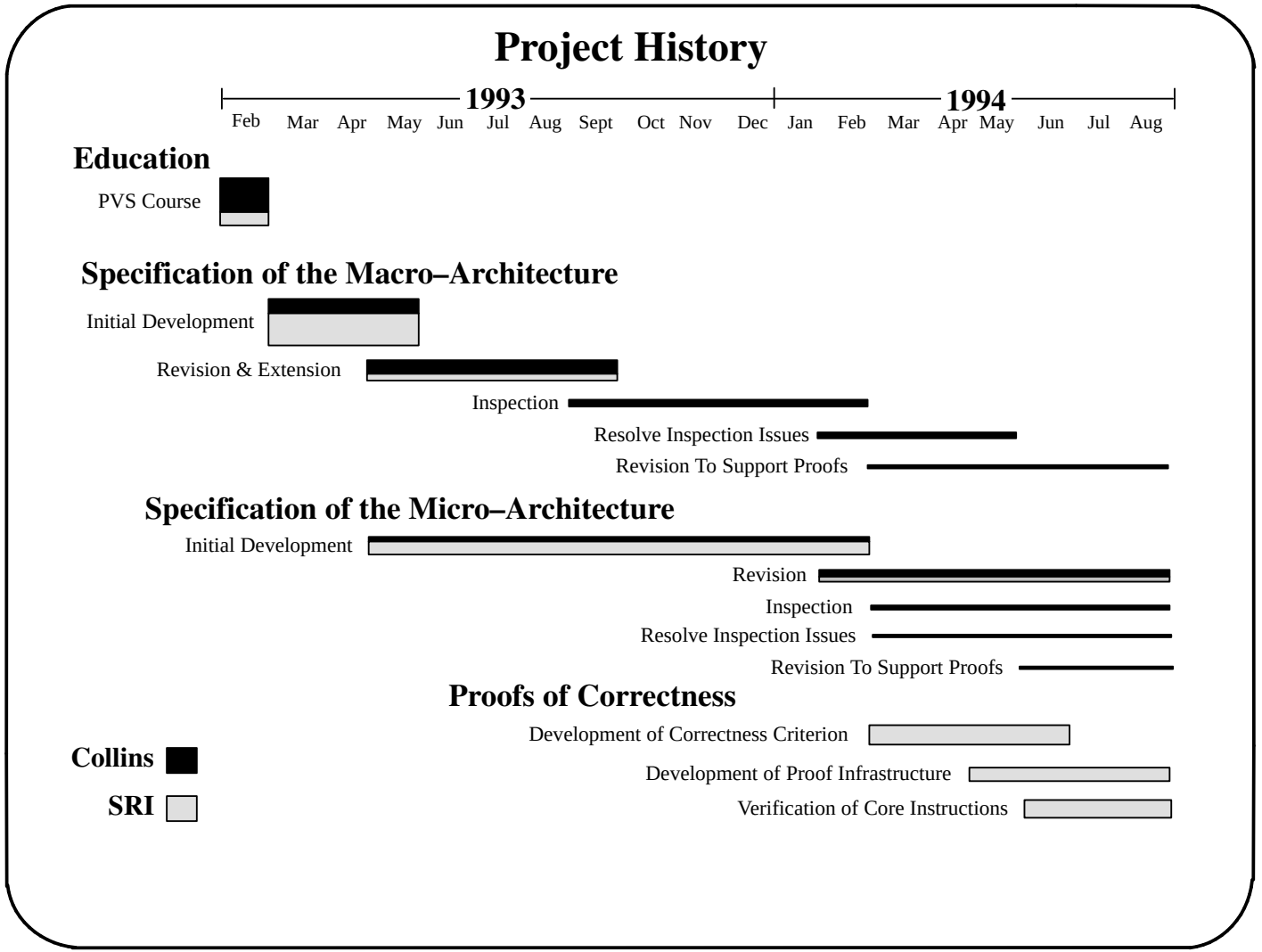
Programmatic Goals

- Assess feasibility of formal verification for industrial use
- Determine cost and value
- Learn how to integrate with traditional process
- Transfer technology to Collins' engineers
- Learn what was needed to achieve true "industrial strength" if formal hardware verification

Process

- Run as a "shadow" project
- Undertaken incrementally in order to involve Collins' engineers and to provide feedback to traditional development
  - SRI would make first pass at each step in the process
  - Collins would polish it, and then do more of the same
- Collins subjected formal specifications to their traditional inspection process

Timeline and Effort



Formal Specifications

Macroarchitecture: 480 hours by SRI, 675 by Collins

- Initial development done by SRI
  - 1,595 Lines of PVS and comments in 25 theories
- Revised and extended by Collins
  - Specified most of the executive service routines
  - Completed 108 of the AAMP's 209 instructions
  - 2,550 lines of PVS and comments in 48 theories

Microarchitecture: 640 hours by SRI, 479 by Collins

- Initially done by SRI, revised by Collins
- 2,679 lines of PVS and comments organized into 20 theories

Bitvector library: Initially by SRI, taken over by NASA Langley

- 2,030 lines of PVS and comments in 31 theories

Inspections of Specifications

- Formal inspections performed at Collins by between 3 and 5 practicing electrical engineers
- Inspectors had little trouble reading PVS (learned by sitting in)
  - Average preparation rate 150 lines of PVS and comments per hour
  - Inspection rate about 330 lines per hour
- Performed 11 inspections of macroarchitecture specification
  - Found 53 style errors, 28 correctness errors
- Performed 10 inspections of microarchitecture specification
  - Found 64 style errors, 19 correctness errors

Microcode Errors Found During Formal Specification

- Discovered two new microcode errors
- Found while specifying behavior of the AAMP in unusual cases
- One was due to a missing requirement
  - Would have been found during Ada validation testing
- Other was a coding error
  - Required specific stack configuration, an improperly sized stack, and a specific instruction sequence
- Both specific to the AAMP5 and corrected before first fabrication
- Collins had not completed traditional verification at that time

Formal Verification

- Performed entirely by SRI
- 800 hours
- Developed reusable proof infrastructure
- Formally verified 11 instructions representing the major instruction classes
- Found errors in specifications that had escaped inspections
- Also found the two seeded bugs
  - First was designed to be difficult to detect
  - Second was an actual error that had escaped detection by simulations and walkthroughs; discovered by Collins while running application code on an early prototype of AAMP5
- Discovery was very systematic and SRI was able to explain how to correct both errors

Follow-On: AAMP-FV

- Collins were enthusiastic about formal specification
- But initially skeptical of formal verification
- The way the seeded bugs were found persuaded them it was feasible and had real value
- But decided to undertake new project rather than push through with AAMP5 verification
  - Better suited Collins business case
  - Move tech transfer to a new stage: Collins engineers take the lead, SRI researchers serve as consultants (also do the complex instructions)
  - With startup and learning costs written off to AAMP5, clean start would allow better estimate of costs on real project

AAMP-FV

- Simpler processor, closer to one for ultra-critical applications (e.g., autolanders, 777 autopilot)
- No pipelining, nor autonomous subunits
- 80 instructions
  - 55 have straightline microcode
  - 5 are complex (very long microcode–OS functions etc.)
  - 20 are challenging (iterative microcode for shifts, divide, complex intermediate states for multiply, etc.)
- About 100,000 transistors

AAMP-FV Status

- Micro- and macroarchitecture specification done by Collins
- Formal specifications are the principal design documentation
- Formal verification is the principal debugging method
- 50 (of 55) straightline instructions verified so far, by Collins following strategies developed by SRI
- Several challenging instructions verified by SRI while developing suitable strategies
- Level of effort (computed after 57 instructions)
  - Specification 80+130 hrs. (cf. over 1,100 for AAMP5)
  - Verification: 1800 hrs (32 hrs./instruction)
  - 10,000 lines of PVS

Lessons Learned: Programmatic

- It takes a long time to develop mutual understanding and trust
- On the job training better than courses in formal methods
- Target problem has to be big and real to interest engineers
- Likely to require methodology development, tool enhancements
- Must do this concurrently with tech-transfer
- High risk: difficult to predict schedule and outcome
- Governm't funding essential to get through this difficult period
- And a committed champion in the receiving organization
- Initial skepticism about feasibility of proof is a big barrier
- But eventually discover mechanized proof is the primary source of benefit

Lessons Learned: Technical

- General design decisions in PVS worked well
  - Collins engineers accept PVS specifications
  - Able to do large proofs, if approach is well-developed
- However process still requires too much skill and patience
  - Proofs are far too slow (5–15 minutes)
  - Far too much effort goes into controlling size of intermediate terms
  - Insufficiently powerful automation makes challenging instructions too hard
- Need to gain two orders of magnitude speed improvement and scaling in low-level automation
  - Building decision procedures for bitvectors to provide this
  - Will evaluate on Java machine (uses AAMP-FV core)

Loose Integration of Theorem Proving and Model Checking

- Published specification of Sparc V9 RMO memory model is an axiomatic one
- Seungjoon Park developed executable description in Murϕ
- Allows exhaustive testing of e.g., synchronization code
- Also built a translator from language of Murϕ to PVS
- Then used PVS to prove consistency between the published axiomatic specification and executable description

Another Example

- Havelund and Shankar formally verified a bounded-retransmission communications protocol from Philips (FME 96)
- 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!

Tighter Integration of Theorem Proving and Model Checking

- Real system of interest is usually infinite state
- Conventionally, downscale or abstract to finite state by hand
- Then model check the reduced system
  - Model checking eliminates need for invariants (hmm...)
- Design exploration performed on reduced system
- Theorem proving could justify and automate the reduction

Integrating Theorem Proving and Model Checking

- Need to be able to describe a temporal logic, its models, and satisfaction relation in the logic of the theorem prover
- Model checker is then a decision procedure for a finite-state fragment
- 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

Tight Integration: Example

- Havelund and Shankar 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 μ-calculus embedding
- Verified by theorem proving that abstraction preserved properties of interest
  - But needed 45 invariants to establish this

Systematic and Automatic Generation of Invariants

- Experimental tools to do this integrated with PVS by Bensalem and others at Verimag
- New methods developed by Su and Park at Stanford
- Using "aggregation," Park and Dill verified FLASH cache-coherence protocol with PVS

The Future: Local Advances

- Decision procedures need massive speedups and better scaling (already demonstrated in Stanford's SVC)
- Also need decision procedures for additional theories (e.g., bitvectors—currently being integrated in PVS)
- Need richer control of low-level automation (when can let rip, vs. when must hold on tight leash)
- Need new automation to deal with large problems (help with invariants, abstraction, use of model checking inside proofs)
  - N-processor mutual exclusion uses induction plus CTL model-checking for base and inductive steps
  - N-processor cascaded ack uses induction plus LTL model-checking for base and inductive steps

The Future: A Cooperative Federation

- Theorem proving and automated verification provide many techniques that are impressively effective
  - When applied to suitable problems
- Few applications in formal methods and hardware verification coincide exactly with capabilities of individual techniques
- The need is for integrated systems that bring several techniques together
  - Integration needs to be tight
  - Requires major system-building effort
- Overall: a federation of powerful and efficient individual techniques (decision procedures, model checking)
  - That work together
  - Coordinated by a theorem prover

To Learn More About PVS

- 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