Mechanized Formal Methods
Progress and Prospects

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Industrial Interest in Formal Methods—At Last

- FMCAD, Palo Alto November 1997, had over 170 attendees
  - With more than half coming from industry (e.g., IBM, Intel, HP, Cadence, Mentor, Synopsys)
- Three different groups in Intel (USA) are recruiting people in formal methods (for P7, P8, tool development)
  - Similar at other hardware companies
- Where did this interest come from?
  - Example of modern application of formal methods
  - Technological and methodological developments that got us to this point
- Can it be sustained?
  - Software applications
  - Research challenges

Mechanized Formal Methods Using Theorem Proving

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

Mechanized Formal Methods Using Model Checking

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

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

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

Mechanized Formal Methods: A Recent Example

- The much publicized Pentium FDIV bug cost Intel \$475 million

- And is the trigger for much of the recent interest in formal methods
  - Because it highlighted the rising cost of traditional verification
  - And its fallibility
  - And corresponding impact on time-to-market

- The bug was in the quotient-prediction table for a radix-4 SRT divider ($\Rightarrow$)

- So lots of formal methods groups have tackled SRT division

SRT Division—What Is It?

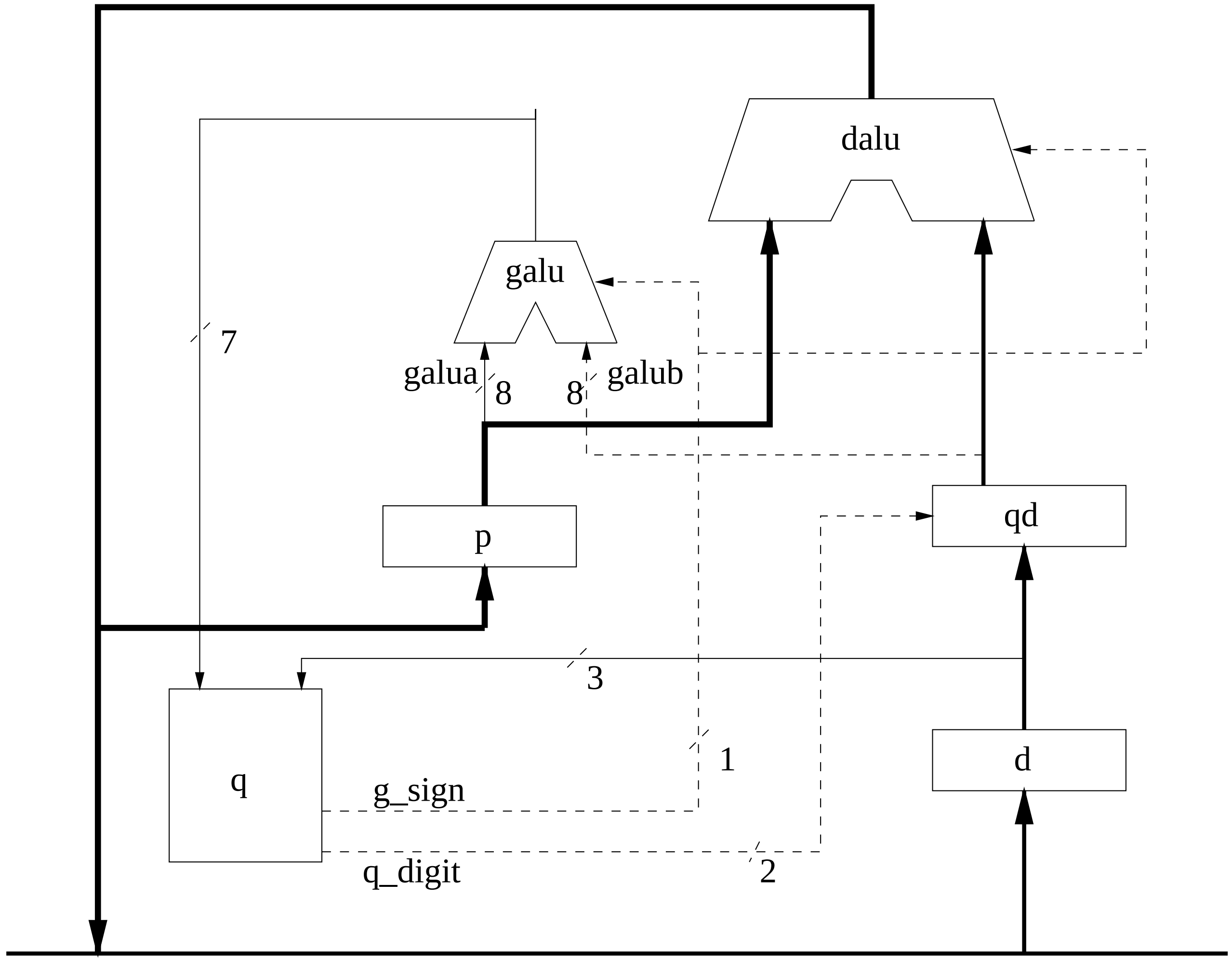
- A very fast method for division, used in many hardware floating point units
- Similar methods used for square root, other elementary functions
- The name comes from the initials of the three people who invented it in the 1950s

SRT Division—How Does It Work?

- It's a subtractive division method
  (like the longhand method learned in school)
  - Calculate the next quotient digit
  - Subtract the divisor that many times from the partial remainder
  - Shift the divisor and repeat

- But uses a redundant number system
  - E.g., for radix 4, use {-2, -1, 0, 1, 2}
  - Which allows some slop in the selection of quotient digits
  - So it can be fast (by table lookup, based on just a few significant digits from divisor and partial remainder)
  - And performed in parallel with the full precision subtraction

SRT Division: Datapath



SRT Division: 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  |  2 ||
      |00110|     |  2  |  2  |  2  |  2  |  2  |  2  |  2 ||
      |00101|  2  |  2  |  2  |  2  |  2  |  2  |  2  |  1 ||
      |00100|  2  |  2  |  2  |  2  |  c  |  1  |  1  |  1 ||
      |00011|  2  |  c  |  1  |  1  |  1  |  1  |  1  |  1 ||
      |00010|  1  |  1  |  1  |  1  |  1  |  1  |  1  |  1 ||
      |00001|  1  |  1  |  1  |  1  |  e  |  0  |  0  |  0 ||
      |00000|  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0 ||
      |11111|  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0 ||
      |11110| -1  | -1  |  d  |  d  |  0  |  0  |  0  |  0 ||
      |11101| -1  | -1  | -1  | -1  | -1  | -1  | -1  | -1 ||
      |11100|  a  |  b  | -1  | -1  | -1  | -1  | -1  | -1 ||
      |11011| -2  | -2  | -2  |  b  | -1  | -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: Verification

- Need to show that the quotient selection logic and lookup table perform division correctly

- This also requires showing the blank entries in the table are never accessed

- That was the FDIV bug—some of those entries were accessed and contained junk

SRT Division: Verification by BDDs

- BDDs (reduced, ordered Binary Decision Diagrams) are a canonical representation for Boolean formulas

- But grow exponentially for multipliers and dividers

- Nonetheless, Bryant verified one iteration of SRT using BDDs
  - Extracted invariant (an arithmetic property) by hand
  - Designed a checker circuit to compute it (4,300 gates vs. 1,100 for SRT itself)
  - Used BDDs to prove bit-level equivalence of functions computed by SRT and checker circuits

SRT Division: Verification by BMDs and Word-Level Model Checking

- BMDs (Binary Moment Diagrams) and variants allow efficient computation of properties of binary words (as opposed to single bits)

- Clarke, Khaira and Zhao used BMDs and word-level model checking to verify an invariant for SRT
  - Like Bryant's approach, but without need for a checker circuit

- Ho and others at Intel extended this to several algorithms from Weitek FPU and to IEEE-compliant rounding unit

SRT Division: Verification by Theorem Proving

- German and Clark used a theorem prover built on Mathematica to verify a particular SRT circuit and table

- Rueß, Shankar, and Srivas specified and verified the general theory of SRT division in PVS

- Then derived the conditions for the radix-4 lookup table

- And verified a particular table (similar to the Pentium's) due to Taylor (1981)

SRT Division: Correctness in PVS

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: PVS Verification I

- Proof is a simple case split on all possible values of indices `P` and `D` followed by a call to arithmetic 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: PVS 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 broad 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

AMD K5 Division in ACL2

- The AMD K5 processor uses a multiplicative division algorithm (based on Newton-Raphson) implemented in microcode

- Brock, Kaufmann, and Moore verified this algorithm against IEEE standard by theorem proving using ACL2

- Russinoff subsequently applied same techniques to square-root microcode, exposing a bug that had escaped testing

- Done in, or near, the design loop

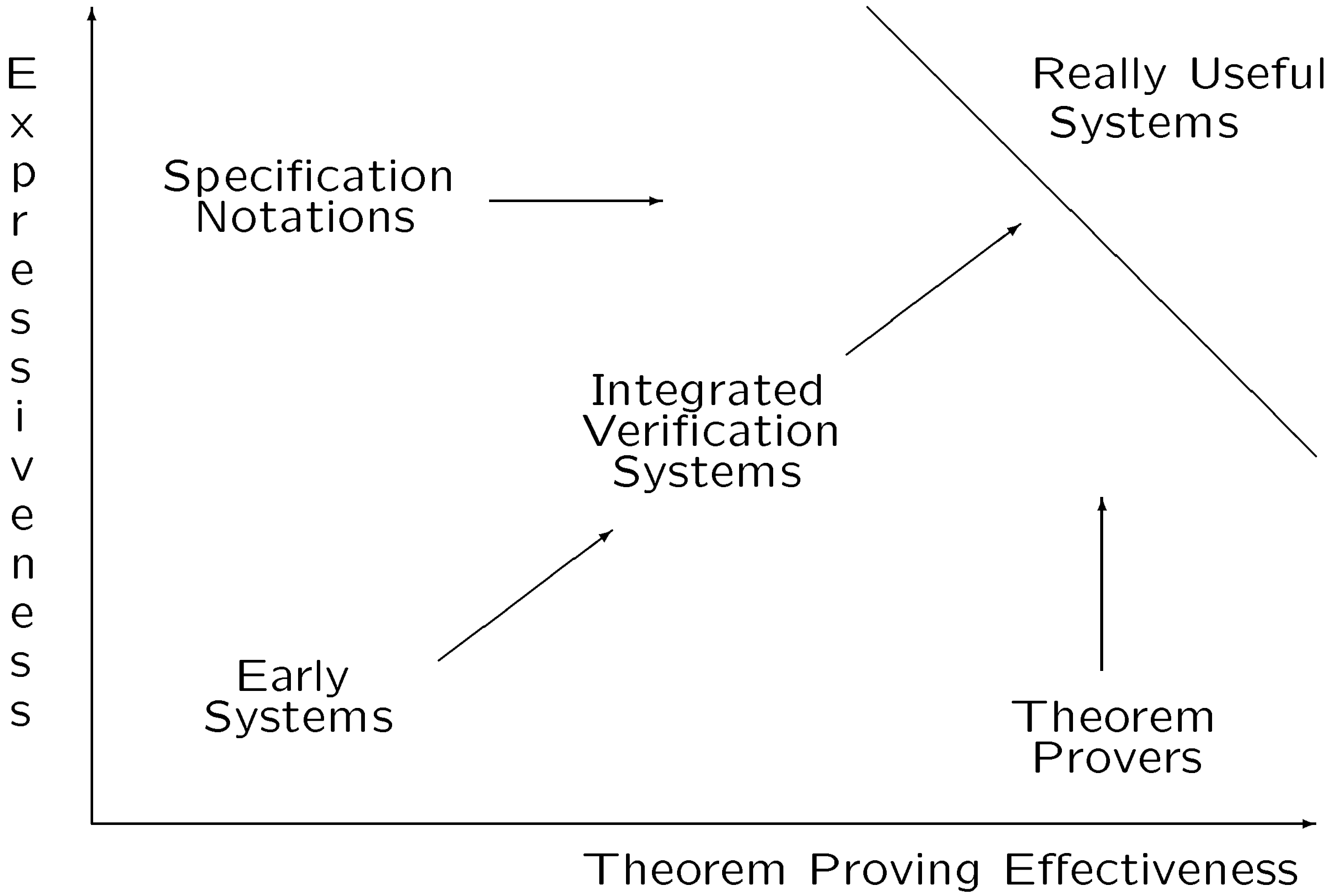
SRT Example: Comparison of Approaches

- Only the theorem proving approaches actually establish that SRT does division
- The "automatic" BDD and model checking methods required significant manual effort to transform the problem into a form they could handle
- Development and execution time comparable in all cases
- Designs and properties established by theorem proving methods are more general than those established by model checking
  - So more opportunity for reuse
- Model checking would come out much better on examples involving concurrency
- Wanted: integration of the two approaches
- But first, let's look at them individually

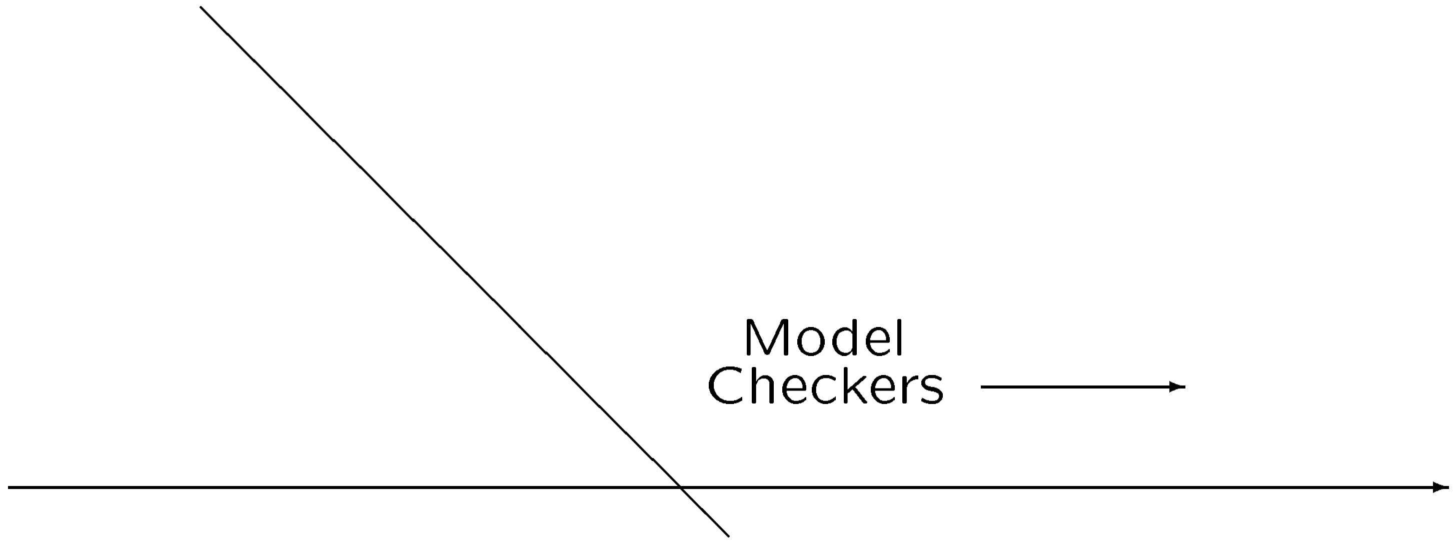
Mechanizing Formal Methods: The Specification Language

- 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

Mechanized Formal Methods: The Three Traditions



Then Along Came the Fourth



The Contribution of Model Checking

- From a forthcoming tutorial:

  > "With its dramatic success in automatically detecting design errors (mainly in hardware and protocols) model checking has recently rescued the reputation of formal methods"

- That may be going a little far, but those working with BDDs and model checking did great benefit to the field
  - Developed efficient and effective tools
  - Worked on real examples
  - Focussed on finding bugs, not correctness
  - Embraced partial specifications, and approximate and incomplete modeling
- Stimulated similar activities in theorem proving

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)

  These are done by hand
- Checking largely automatic (modulo BDD orderings)
- Produces counterexamples
- Generally used for debugging
- Find more problems by exploring *all* the behaviors of a downscaled system than by testing only *some* of the behaviors of full system

Theorem Proving: 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)
- Can deal with the whole problem
- 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 than formal methods
  - 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

Model Checking Approaches—I

- 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, traces)
  $\Box$: henceforth; $\Diamond$: eventually

Model Checking Approaches—II

- Variations (see CAV96 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
- 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 BDDs
- 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)

Integration

The power and utility of modern verification systems emerge from the tight integration of several elements

- Language and deduction
- Various theorem proving techniques
  (integration of decision procedures with each other and with rewriting)
- Theorem proving and model checking

Reconciling Language and Deduction

- 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 $\longrightarrow$ 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 (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
```

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`

Theorem Proving Techniques—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** $f(z)$
  - Bitvectors...
  - Need to decide the *combination* of these theories

Theorem Proving Techniques—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*

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

Loose Integration: 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

Tool and Method Integration: Example

- Havelund and Shankar abstracted bounded-retransmission protocol to finite-state
  - E.g., untransmitted portion of file is chosen from the uninterpreted enumeration `NONE`, `ONE`, and `MANY`
  - Cf. downscaled version: file is 1, 2, or 3 messages long
- Model-checked the abstracted protocol using the μ-calculus embedding
- Verified by theorem proving that abstraction preserved properties of interest
  - But required 45 invariants to establish this
- Need to integrate with additional tools for generating abstractions and invariants

Tool and Method Integration: Another Example

- 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 by abstraction plus induction
  - With the inductive step established by model checking

- Similar idea works, using LTL model checking, for cascaded handshake

The Future: Applications to Software

- More difficult than hardware
  - Fewer stereotypical problems
  - More dynamic structures

- Adapt what's worked for hardware
  - Go where the bugs are
    - ⋆ The hardest and most difficult problems (fault-tolerant, real-time, concurrent systems)
    - ⋆ The early lifecycle (requirements, designs, algorithms, not code)
  - Use powerful tools and a spectrum of methods

The Future: Local Advances

- Decision procedures benefit from massive speedups and better scaling (already demonstrated in Stanford's SVC)
- Also need decision procedures for additional theories (e.g., bitvectors, modular arithmetic)
- Need richer control of low-level automation (when can let rip, vs. when must hold on tight leash)
- And better proof description languages (cf. Mizar)
- Need new automation to deal with large problems (help with invariants, abstraction, use of model checking inside proofs)
- More use of hierarchical decomposition, refinement

The Future: Overall Structure—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 verification 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