JPL, 10 am Feb 6, 1998

Modern Formal Methods

"The virtue of a logical proof is not that it compels belief but that it suggests doubts" [Lakatos]

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Formal Methods: Old and New

Old: Designing software is a mathematical exercise

- Stresses notation, method
- Tools are paper and pencil (or "proof assistants")

New (last 5–8 years):

You can calculate properties of software and hardware designs

- Only reliable way to examine complex behaviors of modern systems
- Construct a mathematical model of some aspect of a system, and calculate how it will perform
- So that you can do design exploration, find and fix problems, provide assurance that it will work
- Before you build it
- Because it's cheaper, faster, and better that way
- Just like in other engineering disciplines

Engineering Calculations

- Find an appropriate mathematical domain
  - For many physical systems, it's some branch of calculus
  - For algorithmic designs it's mathematical logic
- Develop methods and supporting tools that compute efficiently in that domain
  - For aerodynamics: Navier-Stokes and numerical wind tunnels
  - For mathematical logic: automated deduction using theorem provers and model checkers
- Learn how to model reality within that domain in a way that is both accurate and tractable
- Develop methodology to best use the new capabilities created

Formal Calculations

- Most problems in continuous mathematics can be approximated in polynomial time: typically n^2 or n^3
- All problems in automated deduction are at least NP-hard, most are superexponential (2^{2^n}), nonelementary ($2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}\}n$), or undecidable
- Why? Have to search a massive space of discrete possibilities
- Which exactly mirrors why it's so hard to provide assurance for algorithmic systems
  - Have to consider vast number of different behaviors
  - Absence of continuity means extrapolation from finite testing is unreliable
- And which is why formal calculations pay off
  - Practical way to examine all possibilities

Making Formal Calculations Tractable

- Try to use mathematical domains where decision problems are "only" exponential or superexponential
  - Theories of equality with uninterpreted functions, linear arithmetic for integers and reals, bitvectors etc.
- Each is rather limited, so develop methods to decide their combination (cooperating decision procedures)
- Exploit the power of symbolic representation to treat similar cases uniformly (i.e., the power of "forall")
- This is a bit like doing algebra rather than arithmetic
  - E.g., showing $(x^2 - y^2) = (x - y)(x + y)$
    rather than $(5^2 - 3^2) = (5 - 3)(5 + 3)$
  - Hard, but has the same benefit of considering *all* cases

Making Formal Calculations Tractable (ctd. 1)

- Use human insight to short-circuit some of the search
  - This is "interactive theorem proving"
  - Challenge is to create effective balance between automation and interaction
- Note: the goal is not to reproduce textbook proofs

Making Formal Calculations Tractable (ctd. 2)

- Certain problems can be formulated as "model checking"
  - Complexity is linear in number of states
  - But number of states grows as product of size of data structures, and is exponential in number of components
- Hence, keep size of data structures small, number of interacting components few
  - Instead of unbounded buffer pool, limit to 2 or 3
  - Instead of n caches, m addresses, use 3 and 2.
- Requires insight to "downscale" the model appropriately
- Experience is that you learn more by examining all possibilities of simplified model than by probing some of the possibilities of the full thing (as by simulation or testing)

Example: Priority Inversions

- Consider a real-time operating system with dynamic scheduling based on fixed priorities
  - Rate-monotonic: tasks with highest iteration rates get highest priorities
- Tasks become eligible to run periodically
- The eligible task of highest priority gets to run
  - Interrupting and suspending the current task if necessary
- Can show all tasks meet their deadlines if processor utilization is less than about 68%

Example: Priority Inversions–II

- Now suppose some of the tasks share resources (e.g., both use the same device, or one provides data to another)
- Cannot just suspend low priority task that's tying up a resource needed by higher priority task, since the resource will still be unavailable
- So need to let the low priority task finish its work
- This is a priority inversion, because the low priority task is running ahead of the high priority one
- However, since all tasks must have known execution time, the inversion is of bounded duration, and performance is still predictable

Example: Priority Inversions–II

- A task is said to be blocked if a task of lower priority is using a resource it needs
- The eligible, unblocked, task of highest priority gets to run
- Is there any pattern of resource sharing and rates of execution that can lead to unbounded priority inversions?
- Let's code it up in Murϕ
    - An explicit-state "model-checker" (actually a state exploration tool) from David Dill's group at Stanford
    - And let Murϕ examine all scenarios with three tasks and one resource

Priority Inversions—Murϕ Model

```
 Const
  N: 3;
Type
  procs: 1..N;
Var
  pending: array[procs] of boolean;
  delay: array[procs] of 0..N+1;
  mutex : array[procs] of array[procs] of boolean;

Function canrun(i: procs): boolean;
Begin
  Return pending[i] &
    Forall j:procs Do !(j<i & mutex[j][i] & pending[j]) Endforall;
End;

Function Canqueue(I: Procs): Boolean;
Begin
  Return Forall J:procs Do !(j>i & mutex[i][j] & pending[j]) Endforall;
End;

Function blockedby(i: procs): procs;
  Var j: procs;
Begin
  j:=1;
  While !(mutex[j][i] & pending[j]) Do j:=j+1; Endwhile;
  Return j;
End;

Startstate
Begin
  For i: procs Do
    For j: procs Do mutex[i][j] := false; Endfor;
    delay[i] := 0;
    pending[i] := false;
Endfor;
End;
```

Priority Inversions–Murϕ Model (ctd.)

```
Ruleset i: procs do
Ruleset k: procs do
Rule "assign mutexes" forall j: procs do !pending[j] endforall
     & i < k  ==>
Begin
  For r: procs Do
    For s: procs Do mutex[r][s] := false; Endfor;
  Endfor;
  mutex[i][k] := true
End;
Endruleset;
Endruleset;

Rule "run" exists i: procs do canrun(i) endexists ==>
  Var j: procs;
Begin
  j:=N;
  While !canrun(j) Do
    If pending[j] Then delay[j] := delay[j]+1;
       Assert delay[j] < N+1;
    Endif;
    j:=j-1;
  Endwhile;
  pending[j] := false;
End;

Ruleset i: procs do
Rule "try" !pending[i] & canqueue(i) ==>
Begin
  pending[i] := true;
  delay[i] := 0;
End;
Endruleset;
```

Priority Inversions: Murϕ Analysis

```
Result:
        Assertion failed: inversions.m, line 60.
State Space Explored:
        86 states, 210 rules fired in 0.90s.

Startstate Startstate 0 fired.
pending[1..3]:false
delay[1..3]:0
mutex[1..3][1..3]:false
----------
Rule assign mutexes, i:1, k:3 fired.
mutex[1][3]:true
----------
Rule try, i:1 fired.
pending[1]:true
----------
Rule try, i:2 fired.
pending[2]:true
----------
Rule try, i:3 fired.
pending[3]:true
----------
Rule run fired.
pending[2]:false
delay[3]:1
----------
Rule try, i:2 fired.
pending[2]:true
----------
Rule run fired.
pending[2]:false
delay[3]:2
----------
Rule try, i:2 fired.
pending[2]:true
----------
Rule run fired.
pending[2]:false
delay[3]:3
----------
Rule run fired.
The last state of the trace (in full) is:
pending[1]:true
pending[2]:false
pending[3]:true
delay[1]:0
delay[2]:0
delay[3]:4
mutex[1..3][1..3]:false except
mutex[1][3]:true
----------
End of the error trace.
```

Priority Inversions: Familiar Scenario?

- On Mars Pathfinder:
  - Bus management task ran at high priority
  - Meteorological data gathering task ran at low priority
    - ⋆ Bus was a shared resource between these two tasks
  - Communications task ran with medium priority
- The scenario found by Murϕ is the same as that which caused the total system resets on Pathfinder
- According to David Wilner, Chief Technical Officer of Wind River Systems, which makes VxWorks (OS on Pathfinder)

Solving the Priority Inversion Problem

- Priority Inversions are a well-understood problem
- They are solved by Priority Inheritance
  - Task holding a resource that's blocking a task of higher priority temporarily inherits the priority of the blocked task
- Solution confirmed by Murϕ model
  (for three tasks and one resource)

```
Status:
        No error found.
 State Space Explored:
        182 states, 655 rules fired in 1.02s.
```

- Priority Inheritance is available in VxWorks, but was consciously not used for the Bus Management task

What's New

- Massive technical advances in decision procedures and model checking over the last few years
- Learned how to integrate techniques
  - Multiple decision procedures
  - Decision procedures and first order reasoning
  - Theorem proving and model checking
  - Automation and interaction
- Learned how to provide attractive specification languages that also support efficient calculation
- Learned how best to target the techniques
  - Hardest and most critical problems
  - Early lifecycle

Successes

- Protocols: a major application of model checking
  - Debugging, conformance checking, test generation
  - Recent very successful application to crypto protocols
  - Stunning debugging of real-time protocols
- Synchronous fault-tolerant systems (as in fly/drive by wire): found, fixed bugs in important algorithms, then verified them
  - Significantly increased resilience in practical cases
  - Because calculation handles the extra case analysis
- Hardware design:
  - Boolean equivalence checking is standard
  - Model checking is routine in processor, cache design
  - Theorem proving for floating-point units, pipelines, μ-code
- Requirements: consistency and completeness checking for tabular specifications becoming industrialized

Requirements Analysis with Formal Methods

- Many aspects of requirements engineering demand human thought, insight, communication, and discussion
- These precious human resources should be applied where they are truly necessary, and not expended on tasks that can be reduced to rote calculations
- Formal methods provide ways of documenting requirements so that some important questions concerning consistency, completeness, and utility can be examined by calculation
  - Faster, cheaper, better than by unaided human review
- Don't waste thought when calculation is better

Example Applications: Space Shuttle CRs

- Space Shuttle flight software development process is generally considered exemplary (CMM Level 5)
- Requirements for modifications are documented in Change Requests (CRs)
- Subjected to extensive review by Requirements Analysts
- Team from JPL, Lockheed-Martin, NASA Langley, SRI applied a variety of formal methods to some recent CRs in parallel with traditional process

Example Application: Space Shuttle GPS

- GPS (Global Positioning System) navigation is being added to the shuttle
- GPS data loaded as if state-vector updates from ground radar
- Therefore CR (110 pages) has extensive interfaces to existing components
- Two principal functions were formally specified in PVS (3,300 lines) and updated through three sets of revisions
- Anomalies ("issues") were discovered through the process of formalization and typechecking

Shuttle GPS: Issues Discovered through Formal Methods

- Part way through, numbers of issues discovered were

  Formal methods: 30 (7 major)

  Human review: 4 (1 major)

- At completion, issues discovered through formal methods

Issue Severity	Mod B	Mod D/E	Mod F/G	Totals
High Major	1	0	0	1
Low Major	7	3	0	10
High Minor	19	40	6	65
Low Minor	8	0	2	10
Totals	35	43	8	86

How Is This Different To Prototyping/Pseudocode/CASE?

- It's not all that different
- But, liberated from the need to be executable, specification languages allow you to be *precise*, yet abstract
- And without premature concern for implementation details
- Another great benefit of true specification languages is the very rich type systems, and correspondingly exacting typechecking, that they can provide

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
- Typechecking generates proof obligations—called TCCs in PVS
- Allows a lot of the specification to be expressed in the types, so that typechecking becomes a very strong check on consistency

Example: Tables

- PVS has (one- and two-dimensional) `TABLE` constructs for specification by cases

```
sign_htable(x: int): int = TABLE
                     %-------------------%
                     |[ x<0 | x=0 | x>0 ]|
                     %-------------------%
                     |  -1  |  0  |  1  ||
                     %-------------------%
  ENDTABLE
```

- Cases are required to be exhaustive and mutually disjoint (checked as TCCs)

- Semantics then correspond to nested if-then-else

Table TCCs

- Sign example table generates the following TCCs

```
% Disjointness TCC
  sign_cond_TCC2: OBLIGATION  (FORALL (x: int):
    NOT (x < 0 AND x = 0)
     AND NOT (x < 0 AND x > 0)
      AND NOT (x = 0 AND x > 0));

% Coverage TCC
  sign_cond_TCC3: OBLIGATION (FORALL (x: int): x < 0 OR x = 0 OR x > 0);
```

- Both of these are discharged automatically by PVS's default strategy for TCCs using decision procedures for linear arithmetic
- The variant specification that uses an `ELSE` in place of the condition `x > 0` generates a simpler disjointness TCC (just the first of the three conjuncts) and no coverage TCC.

Example Application: Space Shuttle HAC

- Critical descent maneuver is initial bank onto the “Heading Alignment Cylinder” (HAC)
- CR (20 pages) for improved display of HAC flight information to Shuttle pilots documents key portions of the required control logic in tabular form
- These were formalized using PVS tables

Shuttle HAC: Small Example Table

```
switch_position: TYPE = {low, medium, high}
major_mode:      TYPE = {mm301, mm302, mm303, mm304, mm305, mm602, mm603}
iphase:          TYPE = {n: nat | n <= 6} CONTAINING 0

ADI_error_inputs: TYPE =
    [# mode: major_mode,
       switch_position: switch_position,
       iphase: {p: iphase | (mode = mm602 => p >= 4) AND
                            ((mode = mm305 OR mode = mm603) => p <= 3)},
       wowlon: {b: bool | b => (mode = mm305 OR mode = mm603)} #]
```

Shuttle HAC: Small Example Table (ctd.)

```
ADI_error_scale_deflection(A: ADI_error_inputs) : [real, real, real] =
 LET mode = mode(A), switch_position = switch_position(A),
    iphase = iphase(A), wowlon = wowlon(A) IN
 TABLE  %  Result is of form: [roll error, pitch error, yaw error]
                    ,                  switch_position
                    %---------------------------------------------------%
                    |[     high      |     medium      |      low      ]|
 %------------------------------------------------------------------------%
 | mode = mm301 OR
   mode = mm302 OR
   mode = mm303     | (10, 10, 10)   | (5, 5, 5)       | (1, 1, 1)      ||
 %------------------------------------------------------------------------%
 | mode = mm304 OR
  (mode = mm602 AND
    (iphase = 4 OR
     iphase = 6))   | (25, 5, 5/2)   | (25, 2, 5/2)    | (10, 1, 5/2)   ||
 %------------------------------------------------------------------------%
 | mode = mm602 AND
   iphase = 5       | (25, 5/4, 5/2) | (25, 5/4, 5/2) | (10, 1/2, 5/2) ||
 %------------------------------------------------------------------------%
 | (mode = mm305 OR
   mode = mm603) AND
    NOT wowlon      | (25, 5/4, 5/2) | (25, 5/4, 5/2) | (10, 1/2, 5/2) ||
 %------------------------------------------------------------------------%
 | wowlon           | (20, 10, 5/2)  | (5, 5, 5/2)     | (1, 1, 5/2)    ||
 %------------------------------------------------------------------------%
ENDTABLE
```

Shuttle HAC: Issues Discovered through Formal Methods

- Attempts to prove TCCs revealed need to document implicit "domain knowledge"
  - For example: `Major mode = 305` or `603` implies $\texttt{IPHASE} \leq 3$
- Domain knowledge iteratively incorporated into the specification using dependent predicate subtyping
  - Allows automated proof
- Conditions for several rows in one table still overlapped
- Led to identification of missing conjunct in some cases
- Explicit formulation of domain knowledge considered valuable in its own right

Beyond Consistency and Coverage Checking

- Requirements flaws often due to difficulty anticipating complex interactions among components, and between system and environment (especially in the presence of faults)
- Space of possible behaviors is vast, some get overlooked
- Requirements reviews are unreliable (say RAs)
    - No systematic methodology
    - Too dependent on skills and diligence of reviewers
    - No completion criteria
- Simulation, prototyping and testing are unreliable
    - Visit only small fraction of the possible behaviors
    - Hard to distinguish properties of the requirements from those of the prototype or simulation
- Need ways to predict behavior under *all* circumstances

Example Application: Space Shuttle JS

- Jet Select (JS) is a low-level control function that chooses which reaction control jets to fire in order to accomplish a given maneuver
- CR proposed change in algorithm from "max dot-product" to "min angle"
- Existing (max dot-product) requirements (70 pages) formalized in PVS
- Twelve formal challenges were proposed
- Eleven were proved
- Unable to prove the challenge "no failed jet is ever selected for firing"
- Exposed major issue in mature requirements

Example Application: Space Shuttle 3E/O

- Three-Engines Out (3E/O) is an abort maneuver invoked when all three main engines fail during ascent
- Complex interaction with environment (70 pages)
- Used Murϕ to explore behavior of 3E/O interacting with highly nondeterministic model of the environment
- 3E/O results

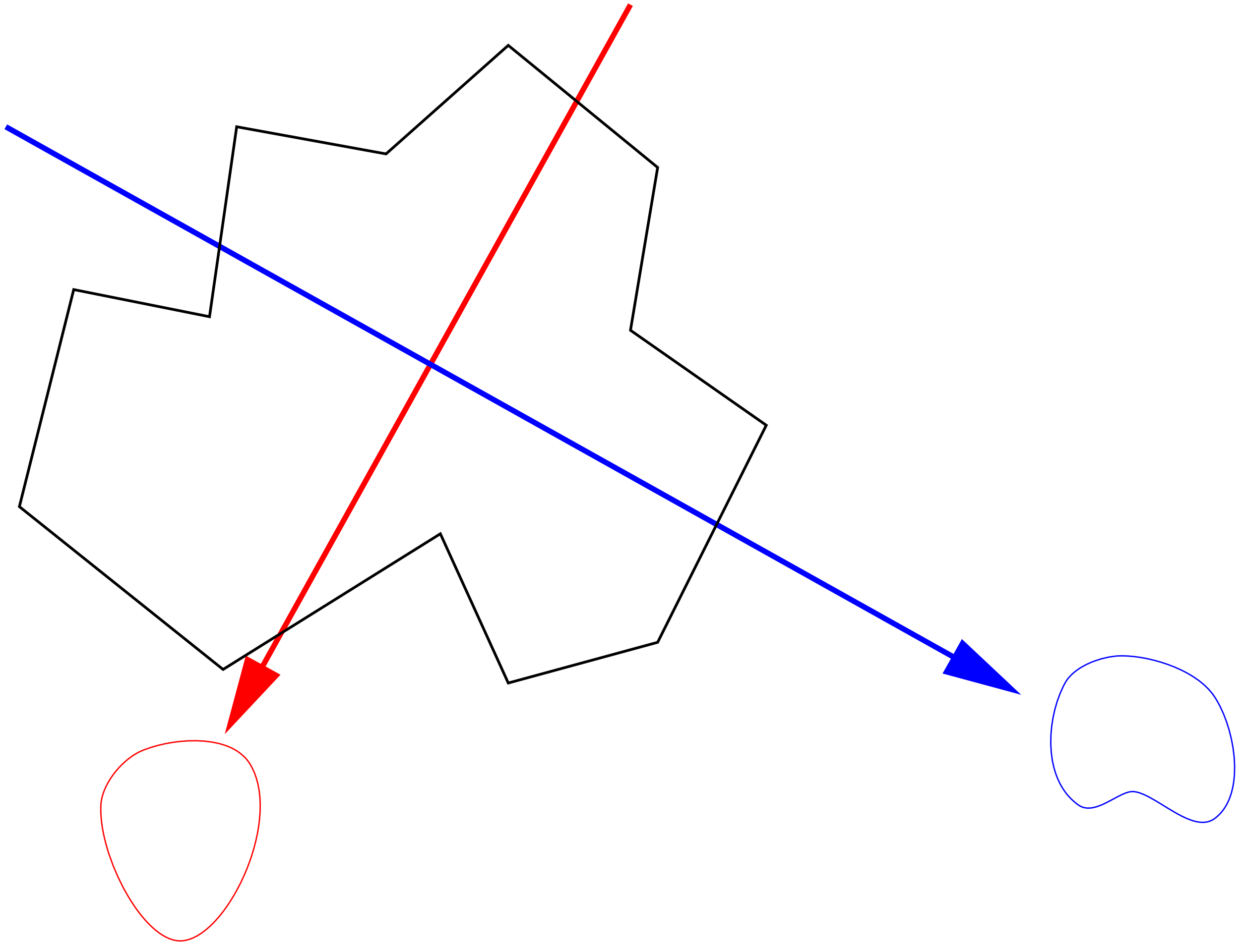
Error Type	Number
undocumented assumption	3
inconsistent or anomalous terminology	10
redundant calculation	2
missing initialization	1
interface anomaly	2
logical error	1

Only the logical error was also found by human review

The Current Challenge

- Scaling to industrial applications is still the major challenge
  - Often overwhelmed by state explosion (model checking), or formula size and invariant generation (theorem proving)
- The big idea: instead of scaling up the tools to fit the applications, cut the applications down to size
  - Property preserving abstraction
  - Currently done by hand, or with tool support for simple cases (cf. slicing)
  - New techniques emerging that can automate the construction of abstracted descriptions, permit multiple tools to work on different facets of a common problem

Automated Abstraction



Claims for Formal Methods

1. Enable faults (of assumptions, requirements, design) to be detected *earlier* than otherwise
   - Due to greater precision and explicitness early in the lifecycle
2. Enable faults to be detected with greater *certainty* than otherwise
   - Because they replace reviews (consensus) by analyses (calculations)
   - And can provide *total* coverage (of selected, modeled properties)
3. *Guarantee* absence of specified faults (subject to accuracy of modeling employed)
   - Because the calculations (proofs) can be checked mechanically (by a theorem prover)

Other Benefits of Formal Calculations

Calculations can be a tool of discovery, not just of certification

- *Discovering*, *pruning* and *sharpening* assumptions
- Challenging requirements to *validate* them
- *Adjusting* designs to changed requirements
- *Debugging* designs
- *Exploring* alternative designs
  - Like CFD, formal methods can allow the design space to be explored more completely and more cheaply than prototyping
- Providing assurance for *negative* and *nonfunctional* properties
- *Deriving* test cases

When to Use Formal Methods?

- When existing methods are inadequate, expensive, absent
- Existing methods are adequate for *most* late-lifecycle processes
    - Very disciplined development and coding methods are used
    - Fagan-style inspections, systematic unit-testing methods, standard, reused components
      (hardware below RTL level is similar)
    - No serious failures in flight software traced to coding bug
    - Formal methods are at their most expensive
      (though methods are very well understood)
    - But there is a case for applying formal methods to especially novel, difficult and/or crucial implementations
      (concurrent, distributed, single-points of failure)

When to Use Formal Methods? (ctd.)

- Existing methods are barely adequate for *many* early-lifecycle processes
  - Accidents are generally the result of multiple failures, unanticipated interactions, improperly understood requirements
  - Few other systematic or analytic methods are available early in the lifecycle
  - Formal methods are at their most effective and economical (though developing effective approaches to specific problems is often a research problem)
  - “Lite” methods can go a long way, can be completely automated, and are acceptable to practitioners (e.g., “Tablewise” checks well-formedness and internal consistency of tables for mode switching logic)

When to Use Formal Methods? (ctd.)

- Existing methods largely inadequate for design issues involving concurrency, complex interactions, fault-tolerance
- Formal methods can provide substantial benefits
  - Benefits come from calculation
  - So heavyweight tools essential
  - Tools are often at their most effective on these kinds of problems
- Issues are generally hard and critical, so require great skill and experience
  - However, such issues are usually relatively localized and few in number
  - So can accomplish a lot with a few highly skilled people

To Learn More About PVS

- Browse general papers and technical reports at `http://www.csl.sri.com/fm.html`
  - `tse95.html` is a good overview of PVS
  - `pvs-bib.html` links to a bibliography with over 180 entries
- Detailed Information about PVS is available from `http://www.csl.sri.com/pvs.html`
  - `http://www.csl.sri.com/pvs/examples` gets you to a directory of tutorials and their supporting specification files
- You can get PVS from `ftp://ftp.csl.sri.com/pub/pvs`
  - Allegro Lisp for SunOS, Solaris, IBM AIX, or Linux
  - Need 64M memory, 100M swap space, Sparc 20 or better (Best is 64MB 200 MHz P6, costs under $2,000)