Calculating With Requirements

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Don't Waste Thought When Calculation Is Better

- 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

Formal Methods

- Just as some properties of bridges and aerofoils can be modeled mathematically
- So some properties of requirements, designs, programs, circuits can be modeled mathematically
- *Descriptive formal methods*: mathematical concepts provide good ways of thinking about and documenting systems
  - Internal consistency, and some other properties, can be checked by calculation
  - For example, typechecking
- *Analytic formal methods*: you can use calculations to predict the properties of a system from its description
  - Using theorem proving, or model checking, for example

Formal Methods and Calculation

- 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 symbolic calculation

- These symbolic calculations can be used for debugging and design exploration as well as post-hoc verification

- Comparable to the way computational fluid dynamics is used in the design of airplanes and jet engines

Formal Methods, Reviews, and Analyses

Using calculations, some activities that are traditionally performed by *reviews*

- Processes that depend on human judgment and consensus

can be replaced or supplemented by *analyses*

- Processes that can be repeated and checked by others, and potentially so by machine

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

Mechanized Formal Methods

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

- To perform calculations efficiently and reliably on such problems, you need automated tools

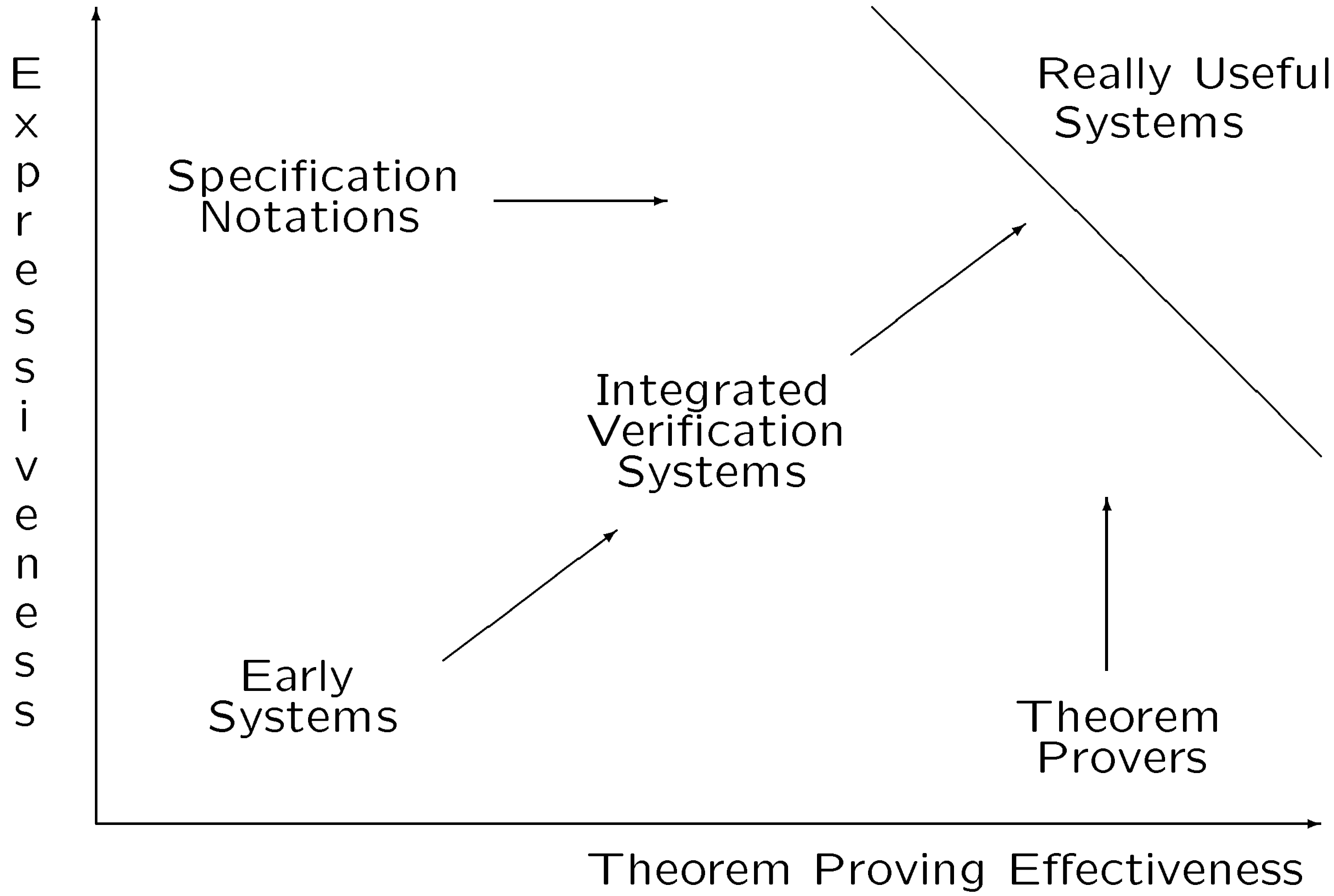
- And the tools must be sufficiently powerful that they can deal with problems of significant interest

- Just as with any other engineering calculations, it's tools that make formal calculations feasible in practice

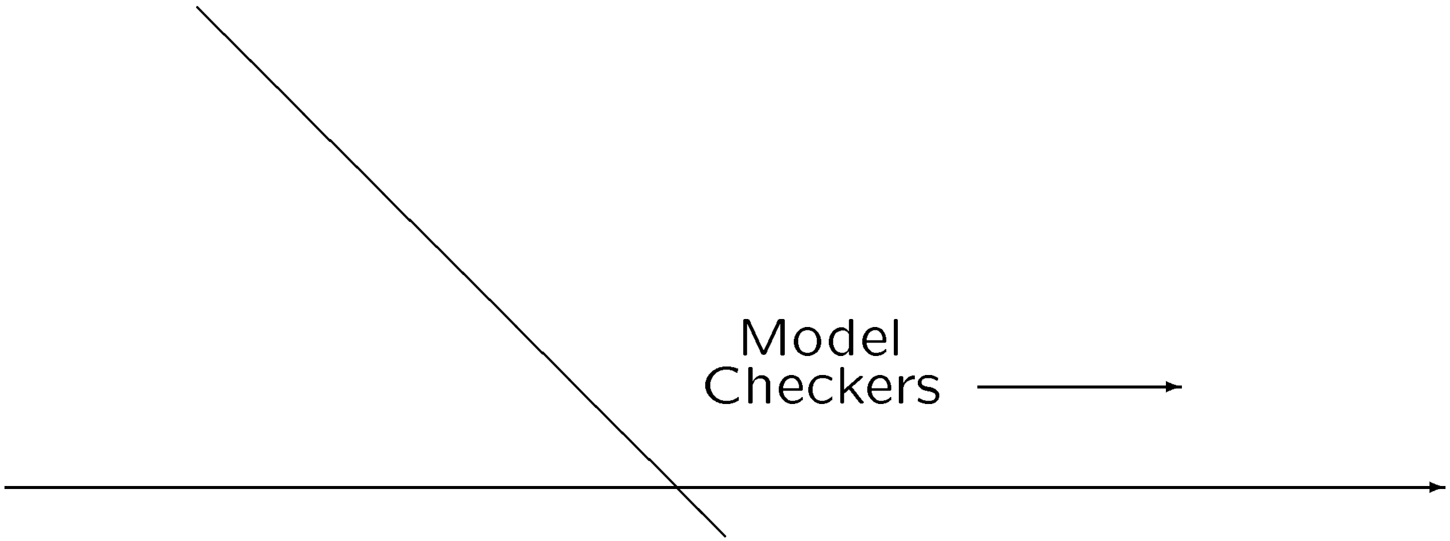
Formal Methods Tools

- Tools are not the most important thing about formal methods
  - They are the *only* important thing
- Without mechanization, formal methods are neither formal, nor methods

Mechanized Formal Methods: The Three Traditions



Then Along Came the Fourth



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

Errors in Formal Specifications

- Writing correct formal specifications is no easier than writing correct programs (for most people it's harder)

- May be difficult to believe this, until you've experienced the personal shock of finding errors in your own specifications

- Tools can help you find errors early and cheaply

- Example: Z/Eves tool performs "domain analysis"
  - Has found flaws in every Z specification examined so far

Finding Errors In Formal Specifications

Tools for formal methods should provide a collection of graduated techniques to help in the detection of errors

- Syntax: parser
- Reviews: prettyprinter, cross-reference generator
- Formal consistency: typechecker
- Structural consistency, completeness: tables, stereotypes
- Sanity check: direct execution ("animation")
- Deeper sanity check: formal challenges
- Dark corners: state exploration/model checking
- Hazard Analysis/Safety: backwards execution
- Correctness: formal verification

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

- 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`

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)} #]
```

```
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

Mechanized Formal Methods

- One way to predict behavior of a system is to construct a mathematical model and *calculate* it
- Model must be reasonably accurate
- And calculation must be performed without error
- For continuous systems, use well-developed mathematical theories (e.g., Navier-Stokes equations for aerodynamics)
- For computer systems, must use discrete mathematics (logic, set theory) and build our own theories
- And proofs of theorems, or model checking, take the place of numerical calculation
- That is what analytic formal methods are about
- For speed, scalability, reliability, repeatability, they need to be mechanized—just like other engineering calculations

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

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

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

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

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

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

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

Issues and Prospects

- Lightweight or heavyweight analyses?
  - Both (and points in between)
  - Use lightest tool that will do the job (i.e., still find errors)
- Model checking or theorem proving?
  - Both, separately and integrated
  - Model check conjectures before you try to prove them
  - Model check finite case; prove it's an abstraction
  - Prove n-process systems by induction; model check cases
- Specialized or general-purpose tools?
  - Both, but built on common components
  - New tools needed also (e.g., invariant and lemma generation, test-case generation etc.)
  - Theorem prover ties everything together

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