Tutorial, CPS PI Meeting, DC 3–5 Oct 2013
Formal Verification Technology

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA
Overview

- A tour of the landscape
- Some topics for the future/close to my heart
Formal Analysis: The Basic Idea

- **Symbolic evaluation**...

- Instead of evaluating, say, \((5 - 3) \times (5 + 3)\) and observing that this equals \(5^2 - 3^2\)

- We evaluate \((x - y) \times (x + y)\)

- And get some big symbolic expression
  \[ x \times x - y \times x + x \times y - y \times y \]

- And we use automated deduction
  - The laws of (some) logic
  - And of various theories, e.g., arithmetic, arrays, datatypes

  To establish some properties of that expression
  - Like it always equals \(x^2 - y^2\)

- The symbolic evaluation can be over computational systems expressed as hardware, programs, specifications, etc.
Formal Analysis: Relation to Engineering Calculations

- This is just like the calculations regular engineers do to examine properties of their designs
  - Computational fluid dynamics
  - Finite element analysis
  - And so on

- In each case, build models of the artifacts of interest in some appropriate mathematical domain

- And do calculations over that domain

- Useful only when mechanized
Formal Analysis: The Difficulty

• For calculations about computational systems, the appropriate mathematical domain is logic

• Where every problem is at least NP Hard

• And many are exponential, superexponential \(2^{2^n}\), nonelementary \(\left(2^{2^{2^{\cdots}}}\right)^n\), or undecidable

• Hence, the worst case computational complexity of formal analysis is extremely high

• So we need clever algorithms that are fast much of the time

• But we also need to find ways to simplify the problems
Formal Analysis: The Benefit

- Can examine all possible cases
  - Relative to the simplifications we made

- Because finite formulas can represent infinite sets of states
  - e.g., $x < y$ represents \{(0,1), (0,2), \ldots (1,2), (1,3)\ldots\}

- Massive benefit: computational systems are (at least partially) discrete and hence **discontinuous**, so no justification for extrapolating from examined to unexamined cases

- In addition to providing **strong assurance**

- Also provides effective ways to **find bugs**, **generate tests**

- And to **synthesize** guaranteed designs
Basic Technology: BDDs

- For **finite state** systems (or approximations that are)
- We can grind everything down to Booleans and represent the system as essentially a circuit
  - Reduced Ordered Binary Decision Diagrams (BDDs) and variants provide **canonical forms with efficient operations**
  - Use these to **calculate the reachable states** by composing BDD representing current set of states with BDD representing the system until a fixed point is reached
  - Check desired properties are **true in all reachable states**
    - Desired properties can be represented as a synchronous observer, or a formula in a temporal logic (CTL, LTL, etc.), eventuality properties require Buchi automata
  - Can also go backwards from a set of states where property is violated to see if an initial state can be reached
- This is **Symbolic Model Checking**: SMV etc.
- Good for up to 300–1,000 state bits
Basic Technology: SAT

- Can alternatively ask if a property is violated in $k$ or less steps, where $k$ is a specific number, like 37
- Given system specified by initiality predicate $I$ and transition relation $T$ on states $S$, and desired property $P$
- Find assignment to states $s_0, \ldots, s_k$ satisfying
  \[I(s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land \cdots \land T(s_{k-1}, s_k) \land \neg(P(s_1) \land \cdots \land P(s_k))\]
- Given a Boolean encoding of $I$, $T$, and $P$ (i.e., circuit), this is a propositional satisfiability (SAT) problem
- SAT solvers have become amazingly effective recently, and continue to improve (annual competition)
  - 100,000s of variables and formulas
- This is called Bounded Model Checking (BMC): NuSMV etc.
- Can also perform verification rather than refutation by slight adjustment that performs $k$-induction (may need invariants)
Basic Technology: Decision Procedures and SMT

- Suppose we don’t want to grind everything down to circuits
- Many useful theories are **decidable** (e.g., linear arithmetic, equality with uninterpreted functions)
- **Decision procedures** work on **conjunctions** of formulas
- Combine these with SAT solving to handle **propositionally complex** formulas over **combinations** of decided theories
- This yields solvers for **Satisfiability Modulo Theories (SMT)**
  - Biggest advance in 20 years
- Which in turn yields **infBMC** and **inf-k-induction**
  - **Inf** because some of the theories are **infinite**
Basic Technology: Beyond SMT

• All SMT solvers employ heuristics for performance
  ○ On multicore, run different heuristics/strategies in parallel
  ○ Called a portfolio

• Beyond SMT, there’s nonlinear arithmetic and other hard theories, quantifiers (\exists, \forall, first and higher order), and lemma generation (especially loop invariants)
  ○ Active areas; lots of recent progress

• That’s the basic technology
  ○ I’m going to describe some others later

But how do we use them?

• Remember even these stunningly powerful methods are typically not polynomial, and do not scale (much)
Dealing With Computational Complexity

• Use human guidance
  ◦ Even with automation, often need user-supplied invariants
  ◦ Or interactive theorem proving—e.g., PVS

• Use approximate models, incomplete search
  ◦ model checkers are often used this way

• Aim at something other than verification
  ◦ E.g., bug finding, test case generation

• Verify weak properties
  ◦ That’s what static analysis typically does

• Give up soundness and/or completeness
  ◦ That’s what commercial static analysis typically does

• Concentrate on small, high criticality components
  ◦ For example, monitors
Approximations, Simplifications, Abstractions (1)

- These can be **sound** or **unsound**
  - **Sound** means if no errors found, then there are none

- **Unsound**: **downscaling**
  - Just chop things down
  - e.g., replace 32 bit integers by 2 bits, limit size of data structures, omit entire parts of the system

- Works for bug finding
  - Exploring all behaviors of an **approximation** finds more bugs than sampling some of the behaviors of the real thing
Approximations, Simplifications, Abstractions (2)

- Sound: data abstraction, abstract interpretation
- Instead of computing on integers, say, compute on \{negative, zero, positive\}
- And many more sophisticated domains
- Iterate to fixed point
  - Need widening and other methods to force convergence
- Can be effective for weak properties
  - Absence of runtime exceptions
  - e.g., Microsoft system (Clousot)
- A lot of engineering, and/or annotation needed to reduce false alarms
  - e.g., Astrée (avionics floating point)
- Can deliver invariants useful to other methods
Approximations, Simplifications, Abstractions (3)

- Sound: predicate abstraction

- Instead of individual variables, focus on their relations

- e.g., eliminate $x$ and $y$, track $x < y$ (i.e., a Boolean)

- Use the relations appearing in conditionals, loops
CEGAR Loops

- Use aggressive, sound approximation
- Get a counterexample to desired property
- Is this due to overapproximation, or because the property really is false?
- Try to evaluate the counterexample on original problem
- If it works, we are done (property is false)
- If not, mine it to find source of overapproximation
  - Craig Interpolation often used for this
- Counter-Example-Guided Abstraction Refinement: CEGAR
Software (As Opposed to State Machines)

- There’s a *program counter*
- Inefficient to represent it as just another state variable
- Need *Abstract Reachability Tree (ART)*, etc.
- Yields *Software Model Checking* (Blast, CMBC, CPA Checker)
- Alternatively, focus on the *abstract data types* (e.g., Alloy)
- Or *generate test cases* using deliberate counterexamples
- Can interleave *symbolic* and *concrete* evaluation to force tests to all reachable control locations
  - *Concolic testing* (Dart etc.)
Software, Again

• Software model checking, interactive program verification, even static analysis often need user-supplied invariants, and other annotations

• **Difficult to obtain**, even a spec can be difficult to obtain

• Powerful type systems can help
  ◦ Predicate subtypes, dependent types

• But software engineering is rarely concerned with creating **truly new code**, mostly it is **modifying existing code**: fixing bugs, adding APIs or functionality, refactoring

• The new code should be the **same** as the old code, except for what was changed

• This is **equivalence checking**

• Tractable to SMT without annotations

• E.g., **SymDiff** (Microsoft)
Cyber Physical Systems

• We have realtime

• And a controlled plant
  ◦ Typically described by differential equations

• These yield timed automata, hybrid automata etc.

• Verification problems are harder, but the payoffs greater
  ◦ Because testing seldom encounters critical cases

• A lot of progress recently

• Some of it direct automation: UPPALL, SpaceEX

• Some of it abstractions to problems solved by SMT
  ◦ Timeout automata, relational abstractions etc.
State of the Art

- Few off the shelf tools for std. programming environments
  - Some sound, often specialized, static analysis: Astrée
  - Mostly unsound: Coverity, Code Sonar, PRQA etc.
- Quite good tools for some CPS environments
  - Design Verifier for Stateflow/Simulink
  - Similar for SCADE, Statemate
- Many good backend tools (model checkers), tool components (SMT)
- SOA applications often employ many of these in ad-hoc toolchains with a lot of glue code and engineering
- Sometimes starting from standard languages, sometimes from specialized ones (SAL, Charon etc.)
- What’s needed is an ecosystem of components and a tool bus
- We are building one (ETB)
Interim Summary

- There's a **lot of backend power** available (SMT)

- And a lot of good ideas, experimental tools, components

- Most of the work is **building toolchains** that start from **something acceptable** to the shop concerned

- And that does **something valuable** while **limiting annotation and user interaction** to a level acceptable to the shop concerned

- It need not be **full verification**
So what? Verification and Safety

- Even if it is full verification, it is not an unequivocal guarantee of properties like safety.

- Safety often concerns attributes of the plant.

- Like the hazards that it poses.

- Verification may establish that each hazard is adequately eliminated or managed.

- But how do we know we’ve identified all the hazards?
Safety/Assurance Cases

• The intellectual foundation of all methods of system assurance is that we have
  ◦ Claims about safety (or other critical attribute)
  ◦ Evidence about our system (tests, reputation of developers, prior systems, formal assurance)
  ◦ Arguments that justify the claims, based on the evidence

• In standards-based approaches, claims and argument are implicit, the standard specifies what evidence to produce

• But there is a notion of Safety (or more generally) Assurance Case that makes the CAE structure explicit
  ◦ That’s why our tool bus is an Evidential Tool Bus (ETB)

• Standards work well in slow-moving, uniform fields (aircraft)

• Safety Cases may be best where there is a lot of innovation and diversity (medical devices)
Epistemic and Logic Vulnerabilities in Safety Cases

- In civil aircraft, all accidents and incidents caused by software are due to flaws in the system requirements specification or to gaps between this and the software specification
  - i.e., none are due to coding errors
  - Because their verification is pretty good, albeit manual
- Verification is wrt. assumptions, requirements, knowledge of the system and its environment
- These are all about epistemology: what you know
  - Can get these wrong: e.g., overlooked hazard
- So there are two sources of vulnerability in safety cases
  - Epistemic (flawed knowledge): new ideas needed here
    - Maybe moving formal modeling upward
  - Logic (flawed reasoning): verification can fix this
    - Subject to epistemic concerns about its own soundness
- Cf. validation and verification in traditional V&V
A Conundrum

- Cannot eliminate failures with certainty (because the environment is uncertain), so top-level safety claims about systems are stated quantitatively
  - E.g., no catastrophic failure in the lifetime of all airplanes of one type

- And these lead to probabilistic requirements for software-intensive subsystems
  - E.g., probability of failure in civil flight control $< 10^{-9}$ per hour

- To assure this, do lots of verification and validation (V&V)

- But V&V is all about showing correctness

- And for stronger claims, we do more V&V
  - Or more intensive V&V: e.g., formal verification

- So how does amount of V&V relate to probability of failure?
Useful Small Systems: Monitors

- These are particularly interesting in safety critical applications, where you need extreme reliability
  - One operational “channel” does the business
  - Simpler monitor channel can shut it down on error

- Used in airplanes (ARP 4754)

- Turns malfunction and unintended function into loss of function
  - Which is dealt with OK by higher-level fault handling
  - Also prevents transitions into bad states

- Monitors against system requirements, not software requirements

- Can be simple because it only need observe, rather than generate, behavior

- Can be formally verified or synthesized
Reliability of Monitored Systems (1)

• The most critical aircraft software needs failure rates below $10^{-9}$ per hour sustained for 15 hours

• Suppose the failure rate of the operational system is $10^{-4}$ and that of the monitor is $10^{-5}$, does that give us $10^{-9}$?

• No! Failures may not be independent
  ○ Failure of one channel probably indicates a hard demand

• No good way forward
  ○ Need “covariance of the difficulty function”
Reliability of Monitored Systems (2)

• But the monitor is simple enough that it can be formally verified or synthesized

• Claim is not that it is reliable but that it is perfect... probably
  ○ Perfection means will never have a failure in operation
  ○ Failure is defined wrt. system requirements, not software requirements, hence differs from correctness

• Attach subjective probability to likelihood of perfection

• Theorem: probability of failure of monitor alone is related to its probability of perfection: \( p_{fd} = p_{np} \times p_f|_{np} \)

• Theorem: probability of perfection of the monitor is conditionally independent of the failure rate of the primary

• So if the monitor has probability of imperfection of \( 10^{-5} \), we do get \( 10^{-9} \) overall!
Reliability of Monitored Systems (3)

- Lots of technical details omitted here
- This analysis is aleatoric, need the epistemic assessment
- And is $10^{-5}$ credible as a probability of imperfection?
- Monitor may go off when it should not (Type 2 failure)
- But the basic idea is sound
  - IEEE TSE Spotlight Paper September/October 2012
- Idea is that you monitor the system specification
  - Get this right by assumption synthesis etc.
- Whereas the operational system is built to the software requirements specification
- Recall, all aircraft incidents due to problems precisely here
- So this approach precisely addresses most vulnerable point
Finally, A Thought Experiment

• Suppose that at some point in a system development I discern the need to make some part of it fault tolerant

• I must choose the types and numbers of faults that it should tolerate (this is called the fault model)

• Suppose I choose a “simple” fault model
e.g., “crash” faults, and no more than two of them

• Then that might enable me to design a correspondingly simple algorithm to perform the fault tolerance

• Thus, I might have very few doubts about whether my algorithm is correct (wrt. its fault model)
  ○ i.e., little logic doubt

• But I might have considerable doubts about whether the fault model will be valid in the real context of its deployment
  ○ i.e., large epistemic doubt
Alternatively

• I could make very few assumptions about the faults
  ◦ That is, a weak fault model

• But then the mechanisms to tolerate those faults might take me into the world of complex adaptive systems

• So here I reduce my epistemic doubt at the price of larger logic doubt

• Traditionally, in critical systems, we have favored reducing logic doubt at the expense of epistemic doubt
  ◦ e.g., no adaptive systems in flight control

• Resilience is about tipping the balance in the other direction

• But without too much logic doubt

• This is the verification challenge of the future
Summary

- There’s a lot of verification technology available
- Off the shelf toolchains for weak properties
- For strong properties, still need to roll your own
- Emerging ecosystem of components, standardized intermediate representations, APIs, tool buses
- Beyond the science and technology, big issues are integration
  - Into industrial workflows and toolchains
  - Into totality of an assurance case
- New opportunities
  - Synthesis rather than verification: $\exists \forall$ SMT solvers
    - $\exists A, B, C : \forall x, y : A \times x + B \times y = C$
  - Resilience: possibly move the verification to runtime
    - Adaptive systems, online synthesis