Opportunities for Industrial Applications of Formal Methods

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA
Formal Methods

- These are ways for exploring properties of computational systems for all possible executions
- As opposed to testing or simulation
  - These just sample the space of executions
- Formal methods use symbolic methods of calculation, e.g.,
  - Abstract interpretation
  - Model checking
  - Theorem proving
- Cf. $x^2 - y^2 = (x - y)(x + y)$ vs. $5*5-3*3 = (5-3)*(5+3)$
Practical Formal Methods

- Symbolic calculations have high computational complexity
  - NP Hard or worse, often superexponential, sometimes undecidable
- So to make them practical we have to compromise
  - Accept some wrong answers
    - *Incompleteness* (false alarms)
    - *Unsoundness* (undetected bugs)
  - Consider only very simple properties (not full correctness)
  - Focus on models of the software, not the actual code
  - Use human guidance
- Let’s look at some of these
Bug Finding by Static Analysis

- Many commercial tools are available for this
  - E.g., Coverity, KlocWork, CodeSonar,
    FindBugs, Lint
  - These work on C, C++, Java

- Most are tuned to reduce the number of false alarms

- Even at the cost of missing some real bugs (i.e., unsound)

- Because the main market is in bug finding
Example: **Bug Finding by Static Analysis**

```c
unsigned int X, Y;
while (1) {
    /* ... */
    B = (X == 0);
    /* ... */
    if (B) {
        Y = 1 / X
    }
    /* ... */
}
```

```c
int x, y, z;
y = 1;
while (1) {
    if (x > 0) {
        y = y+x
    } else {
        y = y-x
    }
    z = 1 / y
}
```

A simple static analyzer will find the **bug on the left**, but will probably give a **false alarm** for the **correct program on the right**.

- Or else fail to find the bug when `y` is initialized to `0`
Verification by Static Analysis

- Some tools are tuned the other way
- Mostly for safety-critical applications
- Guarantee to find all bugs in a certain class (i.e., sound)
- Possibly at the cost of false alarms
- For example
  - Spark Examiner: guarantee absence of runtime errors (e.g., divide by zero) in Ada
  - Astrée guarantee no over/underflow or loss of precision in floating point calculations (in C generated from SCADE)
Example: **Verification** by Static Analysis

We abstract integers by their **signs**

```c
int x, y, z;  // x, y in {neg, zero, pos}
y = 1;         // y is pos
while (1) {
    if (x > 0) {
        y = y+x  // x is pos; y ← pos ⊕ pos; i.e., pos
    } else {
        y = y-x  // x ∈ {zero, neg}; y ← pos ⊕ {zero, neg},  // i.e., pos
    }
    z = 1 / y  // division is ok
}
```

This is an example of **data abstraction**; other methods include **predicate abstraction**, and **abstract interpretation**
Model Checking

- Most static analyzers consider only simple properties
  - Often the properties are built-in and fixed
  - E.g., range of values each variable may take

- Model checking is more versatile

- User can specify property

- There are model checkers for C and Java

- But most work on more abstract models of software
  (typically state machines)

- We’ll do an example
Car Door Locking Example

- Highly simplified from an example by Philipps and Scholz
- **Controller for door locks**
  - To keep it simple, we’ll have just one door
- The *lock* can be in one of four states:
  - locking, unlocking, locked, unlocked
  - Starts in the *unlocked* state
- At each time step it takes an input with one of three values: open, close, idle
  - And asserts a signal *ready* when it is locked or unlocked
- The *controller* receives the *ready* signal from the *lock*, a crash signal from the *airbag*, and a command from the user open, close, idle
- **Safety requirement:**
  - Door is *unlocked* following open command, or crash

John Rushby, SRI
Car Door Locking Example (ctd)

- The lock is given, it behaves as follows
  - When it receives a close input:
    - Does nothing if already locked
    - If it is unlocked, goes to the intermediate locking state
    - If it is locking, goes to locked
    - If it is unlocking, nondeterministically continues to unlocked, or reverses to locking
  - Mutatis mutandis for open input
  - See state machine on next page

- Our task is to design the controller
  - Lock may still be performing a previous action
  - Only visibility into the lock’s state is the ready signal
  - Which it sees with one cycle delay
Lock and Controller

Lock (given)

Controller (designed)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>crash</td>
<td>open</td>
</tr>
<tr>
<td>open</td>
<td>open</td>
</tr>
<tr>
<td>close</td>
<td>close</td>
</tr>
<tr>
<td>idle &amp; ready</td>
<td>idle</td>
</tr>
<tr>
<td>else</td>
<td>repeat last</td>
</tr>
</tbody>
</table>

Output *ready* in green states

John Rushby, SRI

CAL Inauguration Seminar: 11
Model Checking the Car Door Locking Example

- Typically, we would specify this in a statecharts-like graphical formalism (e.g., StateFlow)
- But I will use the textual input to the SAL model checkers so we can see more of what is going on
- It’s fairly easy to build translators and GUIs from engineering notations to the raw notation of a model checker
The Car Door Locking Example: Model Checker Input

Ideally, use an integrated front end; here we look at raw model-checker input

Integrated front-end development environment

- AADL, UML2, Matlab
- TOPCASED, SSIV etc.

Evidential Tool Bus (ETB)

This example

- SAL
- PVS
- Yices

John Rushby, SRI
Begining of the lock Module in SAL

lock: MODULE =
BEGIN
INPUT
  action: lockaction
OUTPUT
  ready: BOOLEAN
LOCAL
  state: lockstate
INITIALIZATION
  state = unlocked
DEFINITION
  ready = (state = locked OR state = unlocked);
TRANSITION
  [locking:
   action = close AND state = unlocked -- state’ = locking;
  []
reverse_unlocking:
   action = close AND state = unlocking --
   state’ IN {s: lockstate | s = locking OR s = unlocked}
Rest of the `lock` Module in SAL

[]
lock:
   `state = locking --> state' = locked;`
[]
unlocking:
   `action = open AND state = locked --> state' = unlocking;`
[]
reverse_locking:
   `action = open AND state = locking --> state' IN {s: lockstate | s = unlocking OR s = locked}
[]
unlock:
   `state = unlocking --> state' = unlocked;`
[]
   `ELSE -->`
]
END;
Beginning of the controller Module in SAL

controller: MODULE =
BEGIN
INPUT
    user: lockaction,
    ready: BOOLEAN,
    crash: BOOLEAN
OUTPUT
    action: lockaction
INITIALIZATION
    action = idle;
Rest of the controller Module in SAL

TRANSITION
[
crash:
  crash --> action’ = open;
[]
open:
  user = open --> action’ = open;
[]
close:
  user = close --> action’ = close;
[]
return_to_idle:
  user = idle AND ready --> action’ = idle;
[]
  ELSE -->
]
END;

John Rushby, SRI

CAL Inauguration Seminar: 17
Specifying The System and a Property

- The system is the synchronous composition of the two modules:
  \[ \text{system: MODULE = lock || controller;} \]
- Inputs and outputs with matching names (i.e., lockaction and ready) are automatically “wired up”
- Now we’ll check a property: whenever the user gives an open input, then the state will eventually be unlocked
  - We need to be careful that the user doesn’t immediately cancel the open with a close
  - So we’ll require that there are no close inputs following the open
- We could have GUI for specifying properties, but here we’ll use Linear Temporal Logic (LTL) which is the raw input to a model checker
A Formal Analysis

- We specify the property in LTL as follows:
  \[
  \text{prop1: LEMMA system |-} \\
  \quad \text{G(user=\text{open} \ \text{AND} \ X(G(user \neq \text{close})))} \implies \text{F(state=\text{unlocked})};
  \]

- In LTL, \( \text{G} \) means always, \( \text{F} \) means eventually, and \( \text{X} \) means next state
  - These are sometimes written \( \square \), \( \Diamond \), and \( \circ \), respectively

- We put all the SAL text into a file `door.sal`

- Then we can ask the SAL symbolic model checker to check the property `prop1`:
  \[
  \text{sal-smc -v 3 door prop1}
  \]

- In a fraction of a second it says: proved

- Unlike a simulation, this has considered all possible scenarios satisfying the hypothesis (e.g., whether lock is ready or not).
More Analyses

- We can check that the door eventually always stays unlocked
  \[\text{prop1a: LEMMA system } \vdash \]
  \[G(\text{user} = \text{open AND } X(G(\text{user} /= \text{close})) \]
  \[\Rightarrow F(G(\text{state} = \text{unlocked}));\]

- And we can sharpen eventually to four steps
  \[\text{prop1b: LEMMA system } \vdash \]
  \[G(\text{user}=\text{open AND } X(G(\text{user}/=\text{close})) \]
  \[\Rightarrow XXXX(G(\text{state} = \text{unlocked}));\]
  \[(XXXX \text{ is a macro for four applications of } X)\]

- We can check that four is the minimum by trying three
  \[\text{prop1c: LEMMA system } \vdash \]
  \[G(\text{user}=\text{open AND } X(G(\text{user}/=\text{close})) \]
  \[\Rightarrow XXX(\text{state} = \text{unlocked});\]

- Sure enough, SAL says invalid
Counterexamples

- But it also gives us a counterexample
  
  user : close open idle idle
  action: close open idle idle
  state : unlocked unlocked locking locked unlocking unlocked

- Push-button proof is nice, but counterexamples are a major additional benefit of model checking: when a property is invalid, we get a trace that manifests its invalidity

- For example, let’s check that the crash input always results in the door becoming unlocked

- We’ll start by assuming the user does no close inputs when the crash occurs

- prop2: LEMMA system |-
  
  G(crash AND G(user /= close) => F(state = unlocked));
Another Counterexample

- SAL says invalid and the counterexample shows that the crash input occurs when the door is locked and the guard on the return_to_idle transition is enabled... and the system chooses to take the latter transition.

- We need to add NOT crash to the guard for the return_to_idle transition to ensure it cannot occur when crash is enabled.

- Now prop2 is proved.
Yet Another Counterexample

- Next, let's check whether we can allow a close input when the crash occurs

- prop3: LEMMA system \(|-\)
  \[ G(\text{crash AND } X(G(\text{user }\neq \text{ close}))) \implies F(\text{state }= \text{ unlocked}) \];

- We get another counterexample!

- A fix is to add NOT crash to the guard for the close transition, too
Counterexamples And Test Case Generation

- We can generate test cases by providing deliberately false assertions.
- The counterexample is a test case.
- To get a test case that drives the system to a state where property $P$ is true, use the property $G(\neg P)$.
- Example: test case to get the system into the unlocking state.
  \[ \text{test1: LEMMA system |- } G(\text{state} \neq \text{unlocking}); \]
- The test case is the input sequence close, open.
Model Checking Technology

- Technically, a model checker tests whether a system specification is a Kripke model of a property expressed as temporal logic formula.

- The simplest kind of property is an invariant \(G(p)\) in LTL.
  - i.e., one that is true in every reachable state.

- So the simplest kind of model checking is reachability analysis.

- Construct every reachable state of the system and check that desired properties (invariants) hold.
  - Feasible if all state variables are finite.
  - May require abstraction to achieve this.

- Simplest method: explicit state reachability analysis.
  - E.g., SPIN.
Explicit State Reachability Analysis and Model Checking

- Imagine a simulator for some system/environment model
- Keep a set of all states visited so far, and a list of all states whose successors have not yet been calculated
  - Initialize both with the initial states
- Pick a state off the list and calculate all its successors
  - i.e., run all possible one-step simulations from that state
  - Throw away those seen before
- Add new ones to the set and the list
- Check each new state for the desired properties
- Iterate to termination, or some state fails the property
  - Or run out of memory, time, patience
- On failure, counterexample (backtrace) manifests problem
- Extend to model checking of general LTL properties using Büchi automata

John Rushby, SRI
Symbolic Model Checking

- Explicit state model checkers run out of power around 10-100 million reachable states

- But that’s only around 25 state bits

- Can often represent states more compactly using symbolic representation

- E.g., the infinite set of states
  \{ (0, 1), (0, 2), (0, 3), \ldots (1, 2), (1, 3), \ldots (2, 3), \ldots \} can be symbolically represented as the finite expression \{ (x, y) | x < y \}

- Symbolic model checkers use such symbolic representations

  - E.g. NuSMV, sal-smc
Symbolic Model Checking (ctd)

- **Compile the model to a Boolean transition relation** $T$
  - i.e., a circuit

- **Initialize the Boolean representation of the stateset** $S$ **to the initial states** $I$

- **Repeatedly apply** $T$ **to** $S$ **until a fixpoint**
  - $S' = S \cup \{ t \mid \exists s \in S : T(s, t) \}$
  - Final $S$ is a formula representing all the reachable states

- **Check the property against final** $S$

- **Mechanized efficiently using** BDDs
  - Reduced ordered Binary Decision Diagrams

Commodity software, honed by competition (CUDD)
Bounded Model Checking

- Modern symbolic model checkers can handle 600 state bits before special tricks are needed
- Seldom get beyond 1,000 state bits
- Bounded model checkers are specialized to finding counterexamples
- Sometimes can handle bigger problems than SMC
  - E.g, NuSMV, sal-bmc
Bounded Model Checking

- Is there a counterexample to $P$ in $k$ steps or less?
- Find assignments to states $s_0, \ldots, s_k$ such that
  \[
  I(s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land \cdots \land T(s_{k-1}, s_k) \land \neg(P(s_1) \land \cdots \land P(s_k))
  \]
- Given a Boolean encoding of $I$, $T$, and $P$ (i.e., circuit), this is a propositional satisfiability (SAT) problem
- SAT is the quintessential NP-Complete problem
- But current SAT solvers are amazingly fast
- Commodity software, honed by competition (MiniSAT, Siege, zChaff, Berkmin)
- BMC uses same representation as SMC, different backend
**Verification with BMC**

- BMC was originally developed for refutation (bug finding)
- But can be used for verification of invariants via \( k \)-induction

**1-induction; ordinary inductive invariance (for \( P \)):**

**Basis:** \( I(s_0) \supset P(s_0) \)

**Step:** \( P(r_0) \land T(r_0, r_1) \supset P(r_1) \)

**Extend to induction of depth \( k \) (cf. strong induction):**

**Basis:** No counterexample of length \( k \) or less

**Step:** \( P(r_0) \land T(r_0, r_1) \land P(r_1) \land \cdots \land P(r_{k-1}) \land T(r_{k-1}, r_k) \supset P(r_k) \)

These are close relatives of the BMC formulas

- Induction for \( k = 2, 3, 4 \ldots \) may succeed where \( k = 1 \) does not
  - Can also use lemmas

- Note that counterexamples help debug invariant

John Rushby, SRI
SAT Solving

- Find satisfying assignment to a propositional logic formula
- Formula can be represented as a set of clauses
  - In **CNF**: conjunction of disjunctions
  - Find an assignment of truth values to variable that makes at least one literal in each clause TRUE
  - Literal: an atomic proposition $A$ or its negation $\overline{A}$
- Example: given following 4 clauses
  - $A, B$
  - $C, D$
  - $E$
  - $A, D, E$

One solution is $A, C, E, \overline{D}$

($A, D, E$ is not and cannot be extended to be one)

- Do this when there are **1,000,000s** of variables and clauses
SAT Solvers

- SAT solving is the quintessential NP-complete problem
- But now amazingly fast in practice (most of the time)
  - Breakthroughs (starting with Chaff) since 2001
    - Building on earlier innovations in SATO, GRASP
  - Sustained improvements, honed by competition
- Has become a commodity technology
  - MiniSAT is 700 SLOC
- Can think of it as massively effective search
  - So use it when your problem can be formulated as SAT
- Used in bounded model checking and in AI planning
  - Routine to handle $10^{300}$ states
SAT Plus Theories

- SAT can encode operations on **bounded** integers
  - Using bitvector representation
  - With adders etc. represented as Boolean circuits
  And other **finite** data types and structures
- But cannot do not **unbounded** types (e.g., reals), or **infinite** structures (e.g., queues, lists)
- And even bounded arithmetic can be **slow** when large
- There are fast **decision procedures** for these theories
- But their basic form works only on **conjunctions**
- General propositional structure requires case analysis
  - Should use efficient search strategies of SAT solvers
    That’s what a solver for Satisfiability Modulo Theories does
  - **SMT solvers**: e.g., Barcelogic, CVC, MathSAT, Yices
  - Sustained improvements, **honored by competition**
Decidable Theories

- Many useful theories are decidable (at least in their unquantified forms)
  - Equality with uninterpreted function symbols
    \[ x = y \land f(f(f(x))) = f(x) \supset f(f(f(f(f(y))))) = f(x) \]
  - Function, record, and tuple updates
    \[ f \text{ with } [(x) := y](z) \overset{\text{def}}{=} \text{if } z = x \text{ then } y \text{ else } f(z) \]
  - Linear arithmetic (over integers and rationals)
    \[ x \leq y \land x \leq 1 - y \land 2 \times x \geq 1 \supset 4 \times x = 2 \]
  - Special (fast) case: difference logic
    \[ x - y < c \]

- Combinations of decidable theories are (usually) decidable
  - \( e.g., 2 \times \text{car}(x) - 3 \times \text{cdr}(x) = f(\text{cdr}(x)) \supset \)
    \[ f(\text{cons}(4 \times \text{car}(x) - 2 \times f(\text{cdr}(x)), y)) = f(\text{cons}(6 \times \text{cdr}(x), y)) \]

  Uses equality, uninterpreted functions, linear arithmetic, lists

John Rushby, SRI
SMT Solving

- Individual and combined decision procedures decide conjunctions of formulas in their decided theories.

- SMT allows general propositional structure
  - e.g., \((x \leq y \lor y = 5) \land (x < 0 \lor y \leq x) \land x \neq y\)
    - ...possibly continued for 1000s of terms

- Should exploit search strategies of modern SAT solvers.

- So replace the terms by propositional variables
  - i.e., \((A \lor B) \land (C \lor D) \land E\)

- Get a solution from a SAT solver (if none, we are done)
  - e.g., \(A, D, E\)

- Restore the interpretation of variables and send the conjunction to the core decision procedure
  - i.e., \(x \leq y \land y \leq x \land x \neq y\)
SMT Solving by “Lemmas On Demand”

- If satisfiable, we are done
- If not, ask SAT solver for a new assignment
- But isn’t it expensive to keep doing this?
- Yes, so first, do a little bit of work to find fragments that explain the unsatisfiability, and send these back to the SAT solver as additional constraints (i.e., lemmas)
  - \( A \land D \subseteq \bar{E} \) (equivalently, \( \bar{A} \lor \bar{D} \lor \bar{E} \))
- Iterate to termination
  - e.g., \( A, C, E, \bar{D} \)
  - i.e., \( x \leq y, x < 0, x \neq y, y \not\leq x \) (simplifies to \( x < y, x < 0 \))
  - A satisfying assignment is \( x = -3, y = 1 \)
- This is called “lemmas on demand” (de Moura, Ruess, Sorea) or “DPLL(T)”; it yields effective SMT solvers
Infinite Bounded Model Checking

- These are bounded model checkers that use SMT solvers
  - E.g., sal-inf-bmc
- Allow analysis of models with infinite state spaces
  - E.g., real-time, other continuous variables
Model Checking for Hybrid Systems

- Often need plant models with continuous dynamics
  - i.e., differential equations

- Hybrid systems mix discrete and continuous behavior
  - As in Simulink/StateFlow
  - Timed systems are a special case

- There are specialized model checkers for hybrid systems
  - E.g., Checkmate
  
  Seldom get beyond 5 or 6 continuous variables

- Another approach uses automated theorem proving to abstract hybrid systems to conservative discrete approximations
  - E.g., hybrid-sal

  Can sometimes handle 25 continuous variables
The Ecosystem of Formal Methods Tools

- **Underlying technology** is highly competitive, specialized
  - Abstract interpreters, BDDs, SAT, SMT solvers, general theorem proving
- **Next level** is well-understood, established incumbents
  - Static analyzers, model checkers, full theorem provers
- **The action** is in automation of the outer loop
  - Counterexample-guided abstraction refinement, interpolants

And **specialized combinations**
- Mixed concrete and symbolic (*concolic*) execution
- Combinations of methods
  - Static analysis generates lemmas for model checker
- **The opportunities** are in enabling these combinations
  - **Tool buses**: open up the tools, make them scriptable
Integration Example: LAST

- **LAST** (Xia, DiVito, Muñoz) generates **MC/DC tests** for avionics code involving **nonlinear arithmetic** (with **floating point numbers**, **trigonometric functions** etc.)
- Applied it to Boeing autopilot simulator
- **Generated tests to (almost) full MC/DC coverage in minutes**
- It’s built on **Blast** (Henzinger et al)
  - A software model checker, itself built of components
  - Including CIL and CVC-Lite
- But extends it to handle nonlinear arithmetic using **RealPaver** (a numerical nonlinear constraint unsatisfiability checker)
  - Added 1,000 lines to **CIL** front end for MC/DC
  - Added 2,000 lines to integrate **RealPaver** with **CVC-Lite**
  - Changed 2,000 lines in **Blast** to tie it all together
- **Toolbus goal is to simplify this kind of construction**

John Rushby, SRI
Opportunities for Applications of Formal Methods

- The ability of formal methods to consider all possible executions creates powerful opportunities
- Exploration of properties in early-lifecycle models
- Thorough analysis of detailed design models
- Guaranteed detection of certain classes of errors in implementations
- Automated generation of test cases
Vee Diagram Tightened with Formal Methods

Example: Rockwell-Collins

John Rushby, SRI

CAL Inauguration Seminar: 44
Industrial Applications of Formal Methods

• Need to integrate formal methods in the development tool-chain
  ○ Interfacing different notations
  ○ Automating/assisting abstraction and lemma generation

• Do so in an open-ended way that allows new tools

• And combinations of tools

• Get in early
  ○ Pick the low-hanging fruit
    Ride the wave of increasing power as the technology matures

• Good luck!

John Rushby, SRI