Software Model Checking

Aditya V. Nori
Microsoft Research India

Thanks to Tom Ball & Sriram Rajamani for material from their lectures
PROBLEM
Software validation problem

Does the software work?

- I hope some hacker cannot steal all my money, and publish all my email on the web!
- I hope it doesn’t crash!
- I hope it can handle my peak transaction load!
- I hope this version still interoperates with my other software!
How do we do software validation?

Testing:
• The “old-fashioned” way
• Run it and see if it works
• Fix it if it doesn’t work
• Ship it if it doesn’t crash!
What's wrong with testing?

program correctness. Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make
What's wrong with testing?

As a result of a long sequence of coincidences I entered the programming profession officially on the first spring morning of 1952 and as far as I have been able to trace, I was the first Dutchman to do so in my country. In retrospect the most amazing thing was the slowness with which, at least in my part of the world, the programming profession emerged, a slowness which is now hard to believe. But I am grateful for two vivid recollections from that period that establish that slowness beyond any doubt.
Program Verification

The algorithmic discovery of properties of a program by inspection of the source text

- Manna and Pnueli, "Algorithmic Verification"

Also known as: static analysis, static program analysis, formal methods, ....
Difficulties in program verification

• What will you prove?
  - Specification of a complex software is as complex as the software itself

• “Deep” specifications of software are hard to prove
  - State-of-art in tools and automation not good enough
Elusive triangle

Large programs

Deep properties

Automation

We will let go of this one!
Example properties

- Type safety
- Memory safety (absence of buffer overruns)
- Protocol conformance for APIs
- Race freedom
Defect detection / Verification
2a: something used in performing an operation or necessary in the practice of a vocation or profession
New generation of software tools

- **SLAM/SDV** (Windows Device Drivers)
- **SAL+PREfast** (Buffer overflow checking for C/C++)
- **Spec# & Boogie** (.NET)
- **ASTREE** (C, avionics software)
- **FindBugs** (Java, bug finder)
- **Saturn** (C, null deref bug finder)

and many more! ...
Other routes to reliability

- Test
- Don’t program in C 😊
- Debug
- Code inspection
- Modern languages (Java, C#, ML, …)
- Runtime checking
Worse is Better!

- *Worse is better*, also called the New Jersey style, is the name of a computer software design approach (or design philosophy) in which simplicity of both interface and implementation is more important than any other system attribute (including correctness, consistency, and completeness).

Worse is Better!

- *Worse is better*, also called the New Jersey style, is the name of a computer software design approach (or design philosophy) in which *simplicity* of both *interface* and implementation is more important than any other system attribute (including correctness, consistency, and completeness).

  http://en.wikipedia.org/wiki/Worse_is_Better
Outline

• **SLAM**: Software model checking via abstraction refinement
  – c2bp
  – bebop
  – newton

• **Synergy**: Property checking by combining static analysis and testing
Software Validation

• Large scale reliable software is hard to build and test.

• Different groups of programmers write different components.

• Integration testing is a nightmare.
Property Checking

• Programmer provides redundant partial specifications

• Code is automatically checked for consistency

• Different from proving whole program correctness
  – Specifications are not complete
Interface Usage Rules

- Rules in documentation
  - Incomplete, unenforced, wordy
  - Order of operations & data access
  - Resource management

- Disobeying rules causes bad behavior
  - System crash or deadlock
  - Unexpected exceptions
  - Failed runtime checks
Does a given usage rule hold?

• Checking this is computationally impossible!

• Equivalent to solving Turing’s halting problem (undecidable)

• Even restricted computable versions of the problem (finite state programs) are prohibitively expensive
Why bother?

Just because a problem is undecidable, it doesn’t go away!
Automatic property checking = Study of tradeoffs

- Soundness vs completeness
  - Missing errors vs reporting false alarms
- Annotation burden on the programmer
- Complexity of the analysis
  - Local vs Global
  - Precision vs Efficiency
  - Space vs Time
Broad classification

• Underapproximations
  – Testing
    • After passing testing, a program may still violate a given property

• Overapproximations
  – Type checking
    • Even if a program satisfies a property, the type checker for the property could still reject it
Current trend

- Confluence of techniques from different fields:
  - Model checking
  - Automatic theorem proving
  - Program analysis

- Significant emphasis on practicality

- Several new projects in academia and industry
Software Model Checking via Abstraction Refinement

- Model checking = exhaustive exploration of state space

- Challenge: realistic software has a huge state space?

- Approach: Abstraction-refinement
  - Construct an abstraction
    - a “simpler model” of the software that only contains the variables and relationships that are important to the property being checked
  - Model check the abstraction
    - easier because state space of the abstraction is smaller
  - Refine the abstraction
    - to reduce false errors
SLAM - Software Model Checking

SLAM models
- boolean programs: a new model for software

SLAM components
- model creation (c2bp)
- model checking (bebop)
- model refinement (newton)
SLIC

• Finite state language for stating rules
  – monitors behavior of C code
  – temporal safety properties (security automata)
  – familiar C syntax

• Suitable for expressing control-dominated properties
  – e.g. proper sequence of events
  – can encode data values inside state
State Machine for Locking

Locked

Unlocked

Error

Rel

Acq

Locking Rule in SLIC

state {
    enum {Locked, Unlocked}
    s = Unlocked;
}

KeAcquireSpinLock.entry {
    if (s==Locked) abort;
    else s = Locked;
}

KeReleaseSpinLock.entry {
    if (s==Unlocked) abort;
    else s = Unlocked;
}
The SLAM Process

- prog. P
- slic
- prog. P'
- SLIC rule
- c2bp
- bebop
- newton
- predicates
- boolean program
- path

Diagram:

1. prog. P → slic
2. slic → prog. P'
3. prog. P' → c2bp
4. c2bp → bebop
5. bebop → newton
6. newton → predicates
7. predicates → boolean program
8. boolean program → path
9. path → newton
10. newton → slic
Example

Does this code obey the locking rule?

do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
Example

Model checking boolean program (bebop)

do {
    KeAcquireSpinLock();
    if(*){
        KeReleaseSpinLock();
    }
} while (*);

KeReleaseSpinLock();

Example Model checking boolean program (bebop)

do {
    KeAcquireSpinLock();
    if(*){
        KeReleaseSpinLock();
    }
} while (*);

KeReleaseSpinLock();
Example

Is error path feasible in C program? (newton)

do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock();
Example

Add new predicate to boolean program (c2bp)

```
Example

Add new predicate to boolean program (c2bp)

do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets; b = true;
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++; b = b ? false : *;
    }
} while (nPackets != nPacketsOld); !b

KeReleaseSpinLock();
```
do {
    KeAcquireSpinLock();
    b = true;
    if(*){
        KeReleaseSpinLock();
        b = b ? false : *;
    }
} while ( !b );
KeReleaseSpinLock();
Example

Model checking refined boolean program (bebop)

```c
do {
    KeAcquireSpinLock();
    b = true;
    if(*){
        KeReleaseSpinLock();
        b = b ? false : *;
    }
} while (!b);
KeReleaseSpinLock();
```
Observations about SLAM

• **Automatic discovery of invariants**
  – driven by property and a finite set of (false) execution paths
  – predicates are *not* invariants, but observations
  – abstraction + model checking computes inductive invariants
    (boolean combinations of observations)

• **A hybrid dynamic/static analysis**
  – *newton* executes path through C code symbolically
  – *c2bp+bebop* explore all paths through abstraction

• **A new form of program slicing**
  – program code and data not relevant to property are dropped
  – non-determinism allows slices to have more behaviors
SLAM internals
(with some simplifications)
C-

Types \( \tau \) ::= void | bool | int | ref \( \tau \)

Expressions \( e \) ::= c | x | e_1 op e_2 | &x | *x

LExpression \( l \) ::= x | *x

Declaration \( d \) ::= \( \tau \) x_1,x_2,...,x_n

Statements \( s \) ::= skip | goto L_1,L_2,...L_n | L: s
| assume(e)
| l = e
| l = f(e_1,e_2,...,e_n)
| return x
| s_1; s_2; ...; s_n

Procedures \( p \) ::= \( \tau \) f(x_1: \tau_1,x_2: \tau_2,...,x_n: \tau_n) d s

Program \( g \) ::= d_1 d_2 ... d_n p_1 p_2 ... p_n
\textbf{C--}

Types \( \tau \) ::= void | bool | int

Expressions \( e \) ::= c | x | e_1 \text{ op } e_2

LExpression \( l \) ::= x

Declaration \( d \) ::= \( \tau \) \( x_1, x_2, \ldots, x_n \)

Statements \( s \) ::= skip | goto L_1, L_2 \ldots L_n | L: s
| assume(e)
| l = e
| f (e_1, e_2, \ldots, e_n)
| return
| \( s_1; \ s_2; \ldots; \ s_n \)

Procedures \( p \) ::= f (x_1: \tau_1, x_2: \tau_2, \ldots, x_n: \tau_n) \ d \ s

Program \( g \) ::= d_1 \ d_2 \ldots \ d_n \ p_1 \ p_2 \ldots \ p_n \)
Types \[ \tau ::= \text{void} \mid \text{bool} \]

Expressions \[ e ::= c \mid x \mid e_1 \text{ op } e_2 \]

LExpression \[ l ::= x \]

Declaration \[ d ::= \tau \ x_1, x_2, \ldots, x_n \]

Statements \[ s ::= \text{skip} \mid \text{goto } L_1, L_2 \ldots L_n \mid \text{L: } s \]
\[ \mid \text{assume}(e) \]
\[ \mid l = e \]
\[ \mid f(e_1, e_2, \ldots, e_n) \]
\[ \mid \text{return} \]
\[ \mid s_1; s_2; \ldots; s_n \]

Procedures \[ p ::= f(x_1: \tau_1, x_2: \tau_2, \ldots, x_n: \tau_n) d s \]

Program \[ g ::= d_1 d_2 \ldots d_n p_1 p_2 \ldots p_n \]
if (e)
   S1;
} else {
   S2;
}
S3;

goto L1, L2;
L1: assume(e);
   S1;
goto L3;
L2: assume(!e);
   S2;
goto L3;
L3: S3;
Example in C

```c
int g;

main(int x, int y){
    cmp(x, y);
    if (!g) {
        if (x != y)
            assert(0);
    }
}

void cmp (int a , int b) {
    if (a == b)
        g = 0;
    else
        g = 1;
}
```
Example in C--

```c
int g;

main(int x, int y){
  cmp(x, y);
  assume(!g);
  assume(x != y)
  assert(0);
}

void cmp(int a, int b) {
  goto L1, L2;
  L1: assume(a==b);
      g = 0;
      return;
  L2: assume(a!=b);
      g = 1;
      return;
}
```
c2bp: Predicate Abstraction for C Programs

Given
- \( P \): a C program
- \( F = \{e_1, \ldots, e_n\} \)
  - each \( e_i \) a pure boolean expression
  - each \( e_i \) represents set of states for which \( e_i \) is true

Produce a boolean program \( B(P,F) \)
- same control-flow structure as \( P \)
- boolean vars \( \{b_1, \ldots, b_n\} \) to match \( \{e_1, \ldots, e_n\} \)
- properties true of \( B(P,F) \) are true of \( P \)
c2bp Algorithm

- Performs modular abstraction
  - abstracts each procedure in isolation

- Within each procedure, abstracts each statement in isolation
  - no control-flow analysis
  - no need for loop invariants
void cmp (int a , int b) {
    goto L1, L2

    L1: assume(a==b);
        g = 0;
        return;

    L2: assume(a!=b);
        g = 1;
        return;
}

int g;
main(int x, int y){
cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

Preds:  \{x==y\}
        \{g==0\}
        \{a==b\}
int g;

main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

decl {g==0} ;

main( {x==y} ) {
Preds:  {x==y}
    {g==0}
    {a==b}
}
int g;

main(int x, int y){
cmp(x, y);
assume(!g);
assume(x != y)
assert(0);
}

decl {g==0} ;

main( {x==y} ) {
cmp( {x==y} );
assume( {g==0} );
assume( !{x==y} );
assert(0);
}

Preds: {x==y} {g==0} {a==b}
Suppose you are given an assignment $s$

- if $\text{Implies}_F(WP(s, e_i))$ is true before $s$ then
  - $e_i$ is true after $s$

- if $\text{Implies}_F(WP(s, !e_i))$ is true before $s$ then
  - $e_i$ is false after $s$

\[
\{e_i\} = \text{Implies}_F(WP(s, e_i)) \ ? \text{true} : \text{Implies}_F(WP(s, !e_i)) \ ? \text{false} : *;
\]
Abstracting Expressions via $F$

- $F = \{ e_1, \ldots, e_n \}$

- $\text{Implies}_F(e)$
  - *weakest* boolean function over $F$ that implies $e$

- $\text{ImpliedBy}_F(e)$
  - *strongest* boolean function over $F$ implied by $e$
  - $\text{ImpliedBy}_F(e) = \neg \text{Implies}_F(\neg e)$
Implies\(_F(e)\) and ImpliedBy\(_F(e)\)
Computing Implies\(_F(e)\)

- \( F = \{ e_1, \ldots, e_n \} \)

- **Minterm**: \( m = d_1 \land \ldots \land d_n \)
  - where \( d_i = e_i \) or \( d_i = \neg e_i \)

- **Implies\(_F(e)\)**
  - disjunction of all minterms that imply \( e \)

- **Naïve approach**
  - generate all \( 2^n \) possible minterms
  - for each minterm \( m \), use decision procedure to check validity of each implication \( m \Rightarrow e \)

- Many optimizations possible
do {
  KeAcquireSpinLock();
  nPacketsOld = nPackets; b = true;
  if(request){
    request = request->Next;
    KeReleaseSpinLock();
    nPackets++; b = b ? false : *;
  }
} while (nPackets != nPacketsOld); !b

KeReleaseSpinLock();
Assignment Example

Statement in P:  
y = y+1;

Predicates in F:  
{x==y}

Weakest Precondition:  
WP(y=y+1, x==y) = (x==y+1)

Implication in F:  
Implies_F( x==y+1 ) = false 
Implies_F( x!=y+1 ) = (x==y)

Abstraction of assignment in B:  
{x==y} = {x==y} ? false : *;
Abstracting Assumes

• \( WP(\text{assume}(e),Q) = e \Rightarrow Q \)

• \text{assume}(e) \text{ is abstracted to: }
  \text{assume}(\text{ImpliedBy}_F(e))

• Example:
  \( F = \{x==2, \ x<5\} \)
  \text{assume}(x<2) \text{ is abstracted to: }
  \text{assume}(\{x<5\} \&\& !\{x==2\}) \)
Abstracting Procedures

• Each predicate in $F$ is annotated as being either global or local to a particular procedure

• Procedures abstracted in two passes:
  – a signature is produced for each procedure in isolation
  – procedure calls are abstracted given the callees’ signatures
Abstracting a procedure call

- **Procedure call**
  - a sequence of assignments from actuals to formals
  - see assignment abstraction

- **Procedure return**
  - **NOP** for C -- with assumption that all predicates mention either only globals or only locals
  - with pointers and with mixed predicates:
    - Most complicated part of c2bp
```c
int g;

main(int x, int y) {
  cmp(x, y);
  assume(!g);
  assume(x != y)
  assert(0);
}

decl {g==0};

main( {x==y} ) {
  cmp( {x==y} );
  assume( {g==0} );
  assume( !{x==y} );
  assert(0);
}

void cmp (int a, int b) {
  Goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

  L2: assume(a!=b);
      g = 1;
      return;
}

void cmp ( {a==b} ) {
  Goto L1, L2

  L1: assume( {a==b} );
      {g==0} = T;
      return;

  L2: assume( !{a==b} );
      {g==0} = F;
      return;
}
```
The SLAM Process

- **prog. P**
- **slic**
- **SLIC rule**
- **prog. P'**
- **c2bp**
- **bebop**
- **newton**
- **predicates**
- **boolean program**
- **path**

Diagram:
- **prog. P** -> **slic** -> **prog. P'**
- **c2bp** -> **bebop** -> **newton**
- **predicates**
- **boolean program**
- **path**
Bebop

- Model checker for boolean programs
- Based on CFL reachability
- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- Generation of hierarchical error traces
Bebop: single procedure case

do {
    KeAcquireSpinLock();
    b = true;
    if(*){
        KeReleaseSpinLock();
        b = b ? false : *;
    }
} while (!b);

KeReleaseSpinLock();
Bebop: interprocedural case

- Based on CFL reachability or Sharir-Pnueli’s “functional” approach

- Key idea: instead of computing fixpoints over “states”, compute fixpoints over two kinds of “edges”:
  - Path edges
  - Summary edges
Domains

\[ g \in \text{Global} \]
\[ l \in \text{Local} \]
\[ f \in \text{Frame} \]
\[ s \in \text{Stack} = \text{Frame}^* \]
\[ \text{State} = \text{Global} \times \text{Local} \times \text{Stack} \]

Transitions:

\[ T \subseteq (\text{Global} \times \text{Local}) \times (\text{Global} \times \text{Local}) \]
\[ T^+ \subseteq \text{Local} \times (\text{Local} \times \text{Frame}) \]
\[ T^- \subseteq (\text{Local} \times \text{Frame}) \times \text{Local} \]
Transition Relation

\[ g \in \text{Global} \]
\[ l \in \text{Local} \]
\[ f \in \text{Frame} \]
\[ s \in \text{Stack} = \text{Frame}^* \]
\[ \text{State} = \text{Global} \times \text{Local} \times \text{Stack} \]

Transitions:
\[ T \subseteq (\text{Global} \times \text{Local}) \times (\text{Global} \times \text{Local}) \]
\[ T^+ \subseteq \text{Local} \times (\text{Local} \times \text{Frame}) \]
\[ T^- \subseteq (\text{Local} \times \text{Frame}) \times \text{Local} \]

**STEP**
\[ T (g, l, g', l') \]
\[ (g, l, s) \rightarrow (g', l', s) \]

**PUSH**
\[ T^+(l, l', f) \]
\[ (g, l, s) \rightarrow (g', l', s.f) \]

**POP**
\[ T^-(l, f, l'') \]
\[ (g, l, s.f) \rightarrow (g', l', s) \]
Naïve model checking

Start with initial state: \((g_0, l_0, \varepsilon)\)

Find all the states \(s\) such that \((g_0, l_0, \varepsilon) \rightarrow^* s\)

Even if Globals and Locals are finite, the set of reachable states could be infinite since the stack can grow infinite

No guarantee for termination

Still, assertion checking is decidable

Need to use a different algorithm \((CFL\ reachability)\)
CFL reachability

\[ P \subseteq (\text{Global} \times \text{Local}) \times (\text{Global} \times \text{Local}) \]
\[ \text{Sum} \subseteq (\text{Global} \times \text{Local}) \times \text{Frame} \times (\text{Global} \times \text{Local}) \]

CFL-INIT

\[ P(g_0, l_0, g_0, l_0) \]

CFL-STEP

\[ P(g_1, l_1, g_2, l_2) \rightarrow T(g_2, l_2, g_3, l_3) \]

\[ P(g_1, l_1, g_3, l_3) \]

CFL-PUSH

\[ P(g_1, l_1, g_2, l_2) \rightarrow T^+(l_2, l_3, f) \]

\[ P(g_2, l_3, g_2, l_3) \]

CFL-SUM

\[ P(g_1, l_1, g_2, l_2) \rightarrow T^-(l_2, f, l_3) \]

\[ \text{Sum}(g_1, l_1, f, g_2, l_3) \]

CFL-POP

\[ P(g_1, l_1, g_2, l_2) \rightarrow T^+(l_2, l_3, f) \]

\[ \text{Sum}(g_2, l_3, f, g_3, l_4) \]

\[ P(g_1, l_1, g_3, l_4) \]

[Sharir-Pnueli 81] [Reps-Sagiv-Horwitz 95]
decl g;
void main()
decl u,v;

[1] u := !v;

[2] equal(u,v);

[3] if (g) then
    R: skip;
    fi
[4] return;

void equal(a, b)

[5] if (a = b) then
[6]    g := 1;
    else
[7]    g := 0;
    fi
[8] return;
Symbolic CFL reachability

- Partition path edges by their “target”
  - $\text{PE}(v) = \{ <d1,d2> \mid <\text{entry},d1> \rightarrow <v,d2> \}$

- What is $<d1,d2>$ for boolean programs?
  - A bit-vector!

- What is $\text{PE}(v)$?
  - A set of bit-vectors

- Use a BDD (attached to $v$) to represent $\text{PE}(v)$
decl g;
void main()
decl u,v;

[1] u := !v;

[2] equal(u,v);

[3] if (g) then
   R: skip;
   fi

[4] return;

void equal(a, b)

[5] if (a = b) then
[6]   g := 1;
   else
[7]   g := 0;
   fi

[8] return;
Bebop: summary

- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- Generation of hierarchical error traces

**Complexity:** $O(E \times 2^{O(N)})$
- $E$ is the size of the CFG
- $N$ is the max. number of variables in scope