Software Verification with Satisfiability Modulo Theories

Nikolaj Bjørner
Microsoft Research
SSFT 2014, Menlo Park
Contents

A primer on SMT with Z3

SMT & Verification by Assertion Checking
- Boogie GC, Quantifiers, Theories

SMT & Verification by Assertion Inference
- Symbolic Software Model Checking, Horn Clauses
Verifying Compilers

Annotated Program \( \rightarrow \) Verification Condition \( F \)

- pre/post conditions
- invariants
- and other annotations
Program 1.2.1 A recursion-free program with bounded loops and an SSA unfolding.

```c
int Main(int x, int y)
{
    if (x < y)
        x = x + y;
    for (int i = 0; i < 3; ++i) {
        y = x + Next(y);
    }
    return x + y;
}

int Next(int x) {
    return x + 1;
}
```

```c
int Main(int x0, int y0)
{
    int x1;
    if (x0 < y0)
        x1 = x0 + y0;
    else
        x1 = x0;
    int y1 = x1 + y0 + 1;
    int y2 = x1 + y1 + 1;
    int y3 = x1 + y2 + 1;
    return x1 + y3;
}
```

\[
\exists x_1, y_1, y_2, y_3 \left( (x_0 < y_0 \implies x_1 = x_0 + y_0) \land (\neg(x_0 < y_0) \implies x_1 = x_0) \land y_1 = x_1 + y_0 + 1 \land y_2 = x_1 + y_1 + 1 \land y_3 = x_1 + y_2 + 1 \land \text{result} = x_1 + y_3 \right)
\]
Verifying Compilers

http://rise4fun.com/Boogie

http://rise4fun.com/Dafny
A Verified GC
A more sophisticated collector

// Copyright (c) Microsoft Corporation. All rights reserved.

//
// Verified mark-sweep garbage collector
//
// medium term goal: support more Bartok array-of-struct and vector-of-struct object layouts
// long term goal: support various other features: threads, pinning, stack markers, etc.

// Imports:
// - Trusted.bpl
// - VerifiedBitVectors.bpl
// Includes:
// - VerifiedCommon.bpl

// \Spec\bin\Boogie.exe /noinfer Trusted.bpl VerifiedBitVectors.bpl VerifiedCommon.bpl VerifiedMarkSw
Boogie Command language

- \( x := E \)
  - \( x := x + 1 \)
  - \( x := 10 \)

- \text{havoc } x

- \( S ; T \)

- \text{assert } P

- \text{assume } P

- \( S \square T \)
### Reasoning about execution traces

- **Hoare triple:** `{ P } S { Q }
  - Starting in P, either S diverges, or
  - Terminates safely in a state satisfying Q

- **Weakest precondition:**
  - `{ wp(S, Q) } S { Q }, and
  - If `{ P } S { Q } then P \implies wp(S, Q)"
## Weakest preconditions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{wp}( x := E, Q ) ) =</td>
<td>( Q[ E / x ] )</td>
</tr>
<tr>
<td>( \text{wp}( \text{havoc} \ x, Q ) = )</td>
<td>( (\forall x \cdot Q) )</td>
</tr>
<tr>
<td>( \text{wp}( \text{assert} \ P, Q ) = )</td>
<td>( P \land Q )</td>
</tr>
<tr>
<td>( \text{wp}( \text{assume} \ P, Q ) = )</td>
<td>( P \Rightarrow Q )</td>
</tr>
<tr>
<td>( \text{wp}( S ; T, Q ) = )</td>
<td>( \text{wp}( S, \text{wp}( T, Q ) ) )</td>
</tr>
<tr>
<td>( \text{wp}( S \parrow T, Q ) = )</td>
<td>( \text{wp}( S, Q ) \land \text{wp}( T, Q ) )</td>
</tr>
</tbody>
</table>
if E then S else T end =

assume E; S

assume ¬E; T
While loop with loop invariant

while E
  invariant J
do
  S
end

= assert J;
  havoc x; assume J;
  ( assume E; S; assert J; assume false
    □ assume ¬E
  )

check that the loop invariant holds initially

where x denotes the assignment targets of S

“fast forward” to an arbitrary iteration of the loop

check that the loop invariant is maintained by the loop body
Verification conditions: Structure

∀ Axioms (non-ground)

BIG and-or tree (ground)

Control & Data Flow
procedure Chunker.NextChunk(this: ref where $IsNotNull(this, Chunker)) returns ($result: ref where $IsNotNull($result, System.String));

// in-parameter: target object
free requires $Heap[this, $allocated];
requires ($Heap[this, $ownerFrame] == $PeerGroupPlaceholder || !(Heap[$Heap[this, $ownerRef], $inv] <: $Heap[this, $ownerFrame])) ||

// out-parameter: return value
free ensures $Heap[$result, $allocated];
ensures ($Heap[$result, $ownerFrame] == $PeerGroupPlaceholder || !(Heap[$Heap[$result, $ownerRef], $inv] <: $Heap[$result, $ownerFrame])) ||

// user-declared postconditions
ensures $StringLength($result) <= $Heap[this, Chunker.ChunkSize];
// frame condition
modifies $Heap;
free ensures $BeingConstructed == null;
free ensures (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv && $f != $FirstConsistentOwner && (!IsStaticField($f) || !IsDirectlyModifiableField($f)) && $o != null && old($Heap[$o, $allocated] && old($Heap[$o, $ownerFrame] == $PeerGroupPlaceholder ||
    old($Heap[old($Heap[$o, $ownerRef], $inv] <: old($Heap[$o, $ownerFrame])) || old($Heap[old($Heap[$o, $ownerRef], $localinv] == $BaseClass(old($Heap[$o, $ownerFrame])) && old($o != this || !(Chunker <: DeclType($f)) || !$IncludedInModifiesStar($f)) && old($o != this || $f != $exposeVersion) ==> old($Heap[$o, $f] == $Heap[$o, $f]);

// boilerplate
free ensures $BeingConstructed == null;
free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } $o != null && old($Heap[$o, $allocated] && $Heap[$o, $allocated] == $typeof($o));
free ensures (forall $o: ref :: { $Heap[$o, $FirstConsistentOwner] } old($Heap[old($Heap[$o, $FirstConsistentOwner], $exposeVersion] ==
            $Heap[old($Heap[$o, $FirstConsistentOwner], $exposeVersion] ==> old($Heap[$o, $FirstConsistentOwner] == $Heap[$o, $FirstConsistentOwner]);
free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } old($Heap[$o, $allocated] ==> old($Heap[$o, $inv] == $Heap[$o, $inv] &&
            old($Heap[$o, $localinv] == $Heap[$o, $localinv] == $Heap[$o, $localinv]);
free ensures (forall $o: ref :: { $Heap[$o, $allocated] } old($Heap[$o, $allocated] ==> $Heap[$o, $allocated]) && (forall $ot: ref :: { $Heap[$ot, $ownerFrame] } { $Heap[$ot, $ownerRef] } old($Heap[$ot, $allocated] && old($Heap[$ot, $ownerFrame] != $PeerGroupPlaceholder ==> old($Heap[$ot, $ownerRef] == $Heap[$ot, $ownerRef] && old($Heap[$ot, $ownerFrame] == $Heap[$ot, $ownerFrame]) &&
            old($Heap[$BeingConstructed, $NonNullFieldsAreInitialized] == $Heap[$BeingConstructed, $NonNullFieldsAreInitialized];
Equality-Matching

\[ p(\forall \ldots) \wedge a = g(b, b) \wedge b = c \wedge f(a) \neq c \wedge p(\forall x \ldots) \rightarrow f(g(c, b)) = b \]

\( g(c, x) \) matches \( g(b, b) \) with substitution \([x \mapsto b]\) modulo \( b = c \)

[de Moura, B. CADE 2007]
struct cell{
    int data;
    cell* next;
};

void zero(cell * c) {
    while(c){
        c->data = 0;
        c = c->next;
    }
    assert (∀d ∈ c_{old} → next . d = null ∨ d → data = 0);
}
void zero(cell * c) {
    while(c){ c = c -> data = 0; c = c -> next; }

    assert (\forall d \in c_{old} \rightarrow next . d = null \lor d \rightarrow data = 0); 
}

Classical memory model:

Next: Cell \rightarrow Cell
Data: Cell \rightarrow \textbf{int}

wp(c = c \rightarrow next, Q) := Q[Next(c)/c]

Next*: Cell \times Cell \rightarrow \textbf{Bool} := TC(Next)
void zero(cell * c) {
    while(c){ c → data = 0; c = c → next; }

    assert (∀ d ∈ c_{old} → next . d = null ∨ d → data = 0);
}

Memory model based on Next*

Next*: Cell × Cell → Bool
Data: Cell × int → Bool

Next* is Transitive, Reflexive, Linear, Anti-symmetric for acyclic lists
Next^+(c, d) := c ≠ d ∧ Next*(c, d)
Next^1(c, d) := Next^+(c, d) ∧ ∀e. Next^+(c, e) → Next*(d, e)

wp(d = c → next, Q) := ∀e Next^1(c, e) → Q[e/d]
Reachability and EPR

Next* is Transitive, Reflexive, Linear, Anti-symmetric

\[ \text{Next}^+(c, d) := c \neq d \land \text{Next}^*(c, d) \]

\[ \text{Next}^1(c, d) := \text{Next}^+(c, d) \land \forall e. \text{Next}^+(c, e) \rightarrow \text{Next}^*(d, e) \]

\[ \text{wp}(d = c \rightarrow \text{next}, Q) := \]

\[ \forall e. \text{Next}^1(c, e) \rightarrow Q[e/d] \land \text{alloc}(c) \land c \neq \text{null} \]

\[ \text{wp}(c \rightarrow \text{next} = \text{null}, Q) := \]

\[ Q[\lambda ab. \text{Next}^*(a, b) \land (\text{Next}^*(a, c) \rightarrow \text{Next}^*(b, c))]/\text{Next}^* \]

\[ \text{wp}(c \rightarrow \text{next} = d, Q) := \]

\[ Q[\lambda ab. \text{Next}^*(a, b) \lor (\text{Next}^*(a, c) \land \text{Next}^*(d, b))]/\text{Next}^* \]

Assuming \( c \rightarrow \text{next} = d \); is preceded by \( c \rightarrow \text{next} = \text{null} \)
Reachability and EPR

• Verification

  – _Python exercise:_ implement _wp_ for _Next*_

• Synthesizing Inductive Invariants

  – [Itzhaky et.al CAV 14] uses Predicate Abstraction for EPR.
Verification by Assertion
Inference
Horn Clauses

\[ mc(x) = x - 10 \quad \text{if } x > 100 \]
\[ mc(x) = mc(mc(x + 11)) \quad \text{if } x \leq 100 \]

assert (\( x \leq 101 \) \( \rightarrow mc(x) = 91 \))

\[ \forall X. \ X > 100 \rightarrow mc(X, X - 10) \]
\[ \forall X, Y, R. \ X \leq 100 \land mc(X + 11, Y) \land mc(Y, R) \rightarrow mc(X, R) \]
\[ \forall X, R. \ mc(X, R) \land X \leq 101 \rightarrow R = 91 \]

Solver finds solution for \( mc \)

[Hoder, B. SAT 2012]
Transition System

- $V$ - program variables
- $\text{init}(V)$ - initial states
- $\text{step}(V, V')$ - transition relation
- $\text{safe}(V)$ - safe states
Safe Transition System

\[ \exists \text{Inv.} \]

- \[ \forall V. \text{init}(V) \rightarrow Inv(V) \]
- \[ \forall V, V'. Inv(V) \land step(V, V') \rightarrow Inv(V') \]
- \[ \forall V. \text{safe}(V) \rightarrow Inv(V) \]

-- [Rybalchenko et.al. PLDI 2012, POPL 2014] Termination and reactivity are also handled in framework of solving systems of logical formulas.
Recursive Procedures

Formulate as Horn clauses:

\[ \forall X. \ X > 100 \rightarrow \text{mc}(X, X - 10) \]
\[ \forall X, Y, R. \ X \leq 100 \land \text{mc}(X + 11, Y) \land \text{mc}(Y, R) \rightarrow \text{mc}(X, R) \]
\[ \forall X, R. \ \text{mc}(X, R) \land X \geq 101 \rightarrow R = 91 \]

Solve for \text{mc}
Recursive Procedures

**Formulate as Predicate Transformer:**

\[ \varphi_{mc}(X,R) = \begin{cases} \text{true} & \text{if } X > 100 \land R = X - 10 \\ \lor \text{false} & \text{if } X \leq 100 \land \exists Y. \ mc(X + 11, Y) \land \ mc(Y, R) \end{cases} \]

Check: \( \mu \varphi_{mc}(X,R) \land X \geq 101 \rightarrow R = 91 \)
Instead of computing $\mu F (mc)(X,R)$, then checking $\mu F (mc)(X,R) \land X \leq 101 \rightarrow R = 91$

Suffices to find post-fixed point $mc_{post}$ satisfying:

$\forall X, R. \quad F (mc_{post})(X, R) \rightarrow mc_{post}(X, R)$

$\forall X, R. \quad mc_{post}(X, R) \land X \leq 101 \rightarrow R = 91$
Program Verification as SMT
- aka
A Crusade for Hornish Satisfaction

Program Verification (Safety)

as Solving fixed-points

as Satisfiability of Horn clauses

[Bjørner, McMillan, Rybalchenko, SMT workshop 2012]
Hilbert Sausage Factory: [Grebenshchikov, Lopes, Popeea, Rybalchenko, PLDI 2012]
A model checking Example

Program 1.4.1 Processing requests using locks.

```c
    do {
        lock ();
        old_count = count;
        request = GetNextRequest ();
        if (request != NULL) {
            ReleaseRequest (request);
            unlock ();
            ProcessRequest (request);
            count = count + 1;
        }
    } while (old_count != count);
    unlock ();
```
Abstraction as Boolean Program

Program 1.4.2 Processing requests using locks, abstracted.

```plaintext
1  do {
2      lock();
3      b = true;
4      if (*) {
5          unlock();
6          if (b) {
7              b = false;
8          }
9          else {
10             havoc b;
11          }
12      }
13  }
14  while (!b);
15  unlock();
```

b := count == old_count

[SLAM, BLAST, Graf & Saidi, Uribe, ..]
(Predicate) Abstraction/Refinement

- SMT solver used to synthesize (strongest) abstract transition relation $F$:

$$\rho(\bar{x}, \bar{x}') \Rightarrow F(b_1(\bar{x}), ..., b_n(\bar{x}), b_1(\bar{x}'), ..., b_n(\bar{x}'))$$
Control as Horn Clauses

(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (Loop count old_count lock_state) (WhileTest count count true))))

; Loop with if-test
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (and (not (= old_count count)) (WhileTest count old_count lock_state))
        (Loop count old_count lock_state)))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (and (= old_count count) (= lock_state true))
        (WhileTest count old_count lock_state)))

(check-sat)
(get-model)
Solving Horn Clauses

Pre-processing

\[ \text{HornClauses} \rightarrow \text{HornClauses}' \]

Search

– Find model \( M \) such that \( M \models \text{HornClauses} \)

Or

– Find refutation proof \( \pi: \text{HornClauses} \vdash_{\pi} \bot \)
Pre-processing

- Cone of Influence
- Simplification
- Subsumption
- Inlining
- Slicing
- Unfolding
Cone of Influence – top down

\[ P(x) \land Q(y) \rightarrow \text{false} \]

\[ R(x) \land x > 0 \rightarrow P(x) \]
\[ R(x) \land x < 0 \rightarrow P(x) \]

\[ x = 2y \rightarrow R(x) \]

\[ Q(y) \land y \leq x \rightarrow Q(x) \]

\[ P(x) \rightarrow S(x) \]
\[ T(x) \rightarrow S(x) \]

\[ S \text{ is not used} \]

\[ S(x) := \text{true} \]
Cone of Influence – bottom up

\[ P(x) \land Q(y, 0) \rightarrow false \]

\[ R(x) \land x > 0 \rightarrow P(x) \]
\[ R(x) \land x < 0 \rightarrow P(x) \]

\[ x = 2y \rightarrow R(x) \]

\[ Q(y, z) \land y \leq x \rightarrow Q(x, 1) \]

There is no “rule to produce \(Q(x, 1)\)”

\[ Q(x, y) := y = 1 \]
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
  (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  => (and (Loop count old_count lock_state) (not (= count count))
    (Loop count count true))))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  => (and (Loop count old_count lock_state) (= count count)
    (= true true)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  => (and (Loop count old_count lock_state) (not (= (+ 1 count) count))
    (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  => (and (Loop count old_count lock_state) (= (+ 1 count) count)
    (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  => (Loop count old_count lock_state)
  (= lock_state false)))

(check-sat)
(get-model)
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
   (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (and (Loop count old_count lock_state) (not (= count count)))
   (Loop count count true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (and (Loop count old_count lock_state) (= count count)
            (= true true)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (and (Loop count old_count lock_state) (not (= (+ 1 count) count))
            (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (and (Loop count old_count lock_state) (= (+ 1 count) count)
            (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (Loop count old_count lock_state)
       (= lock_state false)))

(check-sat)
(get-model)

(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
   (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (Loop count old_count lock_state)
       (= true true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (Loop count old_count lock_state)
       (Loop (+ 1 count) count false)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (Loop count old_count lock_state)
       (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
   => (Loop count old_count lock_state)
       (= lock_state false)))

(check-sat)
(get-model)
Simplification

(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
 (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count))
             (Loop count count true))))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count)
             (= true true)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count ) count ))
         (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count ) count )
             (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
       (= lock_state false)))

(check-sat)
(get-model)
Cone of Influence

(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (and (Loop count old_count lock_state) (not (= count count)))
        (Loop count count true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (and (Loop count old_count lock_state) (= count count))
        (= true true)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (and (Loop count old_count lock_state) (not (= (+ 1 count) count))
        (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (and (Loop count old_count lock_state) (not (= (+ 1 count) count))
        (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
    (=> (Loop count old_count lock_state)
        (= lock_state false)))

(check-sat)
(get-model)
Result

(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
  (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count)))
       (Loop count count true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count))
       (= truetrue)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count) count))
          (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count) count)
           (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
       (= lock_state false))))

(check-sat)
(get-model)
IC3/PDR: Property Directed Reachability

The IC3 Algorithm for Symbolic Model Checking by Aaron Bradley

Procedures

- Regular vs. Push Down systems
- As a Conflict-driven solver for recursive Horn clauses

Beyond

- Linear Real Arithmetic
- Timed Automata Decision Procedure
- Interpolants from models

[ SAT 2012. Kryštof Hoder & Nikolaj Bjørner ]
PDR – the algorithm

Objective is to solve for $R$ such that

$\neg F(R)(X) \rightarrow R(X), \quad R(X) \rightarrow Safe(X), \quad \forall X$

Key elements of PDR algorithm:

Over-approximate reachable states

$R_0 := F(false), R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_N := true$

Propagate back from $\neg Safe$

Resolve conflicts

Strengthen/propagate using induction
PDR – the algorithm

Objective is to solve for $R$ such that

$$
\mathcal{F}(R)(X) \rightarrow R(X), \quad R(X) \rightarrow Safe(X), \quad \forall X
$$

Initialize:

- $Safe_R \leftarrow true$
- $R_0 := \mathcal{F}(false)$
- $\mathcal{F}(R_0)$

Main invariant:

- $Safe \leftarrow \mathcal{F}(R_i)$
- $R_{i+1}$
A digression
Dualities – Recurring Theme

- Core DPLL(T) engine
- Fixed Points engine
- Nonlinear solver
- Linear Integer solver
Core Engine in Z3: Modern DPLL/CDCL

- **Initialize**
  \[ \epsilon \models F \]
  \( F \) is a set of clauses

- **Decide**
  \[ M \models F \Rightarrow M, \ell \models F \]
  \( \ell \) is unassigned

- **Propagate**
  \[ M \models F, C \lor \ell \Rightarrow M, \ell^C \lor \ell \models F, C \lor \ell \]
  \( C \) is false under \( M \)

- **Sat**
  \[ M \models F \Rightarrow M \]
  \( F \) true under \( M \)

- **Conflict**
  \[ M \models F, C \Rightarrow M \models F, C \models C \]
  \( C \) is false under \( M \)

- **Learn**
  \[ M \models F \models C \Rightarrow M \models F, C \models C \]

- **Unsat**
  \[ M \models F \models \emptyset \Rightarrow Unsat \]

- **Backjump**
  \[ MM' \models F \models C \lor \ell \Rightarrow M, \ell^C \lor \ell \models F \]
  \( \neg \ell \in M' \), \( M' \cap \neg C = \emptyset \)

- **Resolve**
  \[ M \models F \models C' \lor \neg \ell \Rightarrow M \models F \models C' \lor C \]
  \( \ell^C \lor \ell \in M \)

- **Restart**
  \[ M \models F \Rightarrow \epsilon \models F \]

- **Forget**
  \[ M \models F, C \Rightarrow M \models F \]
  \( C \) is a learned clause

[Nieuwenhuis, Oliveras, Tinelli J.ACM 06] customized
DPLL(\(T\)) solver interaction

**T- Propagate**
\[
M \mid F, C \vee \ell \Rightarrow M, \ell^{\wedge C \ell} \mid F, C \vee \ell \quad \text{C is false under } T + M
\]

**T- Conflict**
\[
M \mid F \Rightarrow M \mid F \mid \neg M' \quad \text{M' \subseteq M and M' is false under } T
\]

**T- Propagate**
\[
a > b, b > c \quad \mid F, a \leq c \vee b \leq d \Rightarrow
\]
\[
a > b, b > c, b \leq d^{a \leq c \vee b \leq d} \quad \mid F, a \leq c \vee b \leq d
\]

**T- Conflict**
\[
M \mid F \Rightarrow M \mid F, a \leq b \vee b \leq c \vee c < a
\]
\[
\text{where } a > b, b > c, a \leq c \subseteq M
\]
Search: Mile-high perspective

Modern SMT solver

Decisions: Assignments

Conflict Resolution

Conflict Clauses

Fixedpoint solver

Bad $\rightarrow$ WP(Bad)

Conflict Resolution

$(\text{Init}) \leftarrow \text{Init}$
Conflict resolution with arithmetic

Initially $y_1 := y_2 := 0$;

\[ P_1 := \left[ \begin{array}{l}
\ell_0 : y_1 := y_2 + 1; \\
\ell_1 : \text{await } y_2 = 0 \lor y_1 \leq y_2; \\
\ell_2 : \text{critical}; \\
\ell_3 : y_1 := 0;
\end{array} \right] \]

\[ || P_2 := \left[ \begin{array}{l}
\ell_0 : y_2 := y_1 + 1; \\
\ell_1 : \text{await } y_1 = 0 \lor y_2 \leq y_1; \\
\ell_2 : \text{critical}; \\
\ell_3 : y_2 := 0;
\end{array} \right] \]

\[ R(0,0,0,0). \]
\[ T(L, M, Y_1, Y_2, L', M', Y_1', Y_2') \land R(L, M, Y_1, Y_2) \rightarrow R(L', M) \]
\[ R(2,2, Y_1, Y_2) \rightarrow \text{false} \]

Step($L, L', Y_1, Y_2, Y_1'$) $\rightarrow$ $T(L, M, Y_1, Y_2, L', M, Y_1', Y_2)$
Step($M, M', Y_2, Y_1, Y_2'$) $\rightarrow$ $T(L, M, Y_1, Y_2, L, M', Y_1, Y_2')$

Step($0,1, Y_1, Y_2, Y_2+1$).
(Y$_1 \leq Y_2 \lor Y_2 = 0$) $\rightarrow$ Step($1,2, Y_1, Y_2, Y_1$).
Step($2,3, Y_1, Y_2, Y_1$).
Step($3,0, Y_1, Y_2, 0$).

\[ \ell_0 : y := \hat{y} + 1; \text{goto } \ell_1 \]
\[ \ell_1 : \text{await } \hat{y} = 0 \lor y \leq \hat{y}; \text{goto } \ell_2 \]
\[ \ell_2 : \text{critical} ; \text{goto } \ell_3 \]
\[ \ell_3 : y := 0; \text{goto } \ell_0 \]
Search: Mile-high perspective
PDR(T): Conflict Resolution

Initially \( y_1 := y_2 := 0; \)

Loop forever do
- \( \ell_0: y_1 := y_2 + 1; \)
- \( \ell_1: \text{await } y_2 = 0 \lor y_1 \leq y_2; \)
- \( \ell_2: \text{critical}; \)
- \( \ell_3: y_1 := 0; \)

\[ P_1 : \]

\[ \text{conflict resolution} \]

\[ L = 0 \]
\[ M = 0 \]
\[ Y_2 = 0 \]
\[ Y_1 = 0 \]

\[ L = 0 \]
\[ M = 1 \]
\[ Y_2 = 0 \]

\[ Y_2 \geq Y_1 + 1 \land Y_1 \geq 0 \]

Conflict

\[ L = 1 \]
\[ M = 1 \]
\[ Y_1 = 1 \]
\[ Y_2 = 0 \]

\[ L = 1 \]
\[ M = 2 \]
\[ Y_2 = 0 \]

\[ Y_2 \leq 0 \]

Resolution

Get Generalization from Farkas Lemma

Eg., resolve away blue internal variables
PDR(T): Conflict Resolution

initially $y_1 := y_2 := 0$;

$\begin{align*}
\text{loop forever do } \\
\ell_0 : y_1 := y_2 + 1; \\
\ell_1 : \text{await } y_2 = 0 \lor y_1 \leq y_2; \\
\ell_2 : \text{critical}; \\
\ell_3 : y_1 := 0;
\end{align*}$

$\begin{align*}
\text{loop forever do } \\
\ell_0 : y_2 := y_1 + 1; \\
\ell_1 : \text{await } y_1 = 0 \lor y_2 \leq y_1; \\
\ell_2 : \text{critical}; \\
\ell_3 : y_2 := 0;
\end{align*}$

Conflict Resolution

Conflict Propagation

Conflict Resolution

Conflict Propagation

Conflict Propagation

$L = 0$

$M = 0$

$Y_2 = 0$

$Y_1 = 0$

$M = 1$

$Y_2 \geq 1$

$L = 1$

$M = 1$

$Y_2 \geq 1$

$L = 1$

$M = 1$

$Y_2 \geq 1$

$L = 2$

$M = 2$
IC3/PDR – some observations

Interpolation \(\cong\) Solution to Horn Clauses [Rybalchenko]
- \(\forall x, y. A[x, y] \Rightarrow I(x), \forall x, z. I(x) \Rightarrow B[x, z]\)
- Instead of mining interpolants from proofs, PDR uses models and cores

Timed push-down systems \(\cong\) PDR for difference arithmetic

Property Directed Polyhedral Abstraction \(\cong\) PDR + Cute Interpolants
[Ongoing with Arie Gurfinkel]

Shape analysis \(\cong\) PDR with EPR + Predicate Abs/Zipper Interpolants
[Ongoing: Gurfinkel, Itzhaky, Korovin, Lahav, Reps, Talur, Sagiv]

Property + Reachability Directed [CAV 14, Komuravelli, Chaki, Gurfinkel]
High-level Takeaways

• Program Analysis as Solving Logical Formulas

  – I presented some samples of \textit{encoding} analysis problems into logic.

  – I gave a taste of \textit{solving} algorithms for some classes of logical formulas.
SMT SOLVING
DPLL(T) BASED APPROACH
SMT : Basic Architecture

- Equality + UF
- Arithmetic
- Bit-vectors
- ...

SAT + Theory Solvers = SMT

Case Analysis
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \quad y = x + 1, \quad (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \quad p_2, \quad (p_3 \lor p_4) \quad p_1 \equiv (x \geq 0), \quad p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \quad p_4 \equiv (y < 1) \]
SAT + Theory solvers

**Basic Idea**

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \; y = x + 1, \; (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \; p_2, \; (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \; p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \; p_4 \equiv (y < 1) \]

Assignment

\[ p_1, \; p_2, \; \neg p_3, \; p_4 \]

\[ x \geq 0, \; y = x + 1, \]
\[ \neg (y > 2), \; y < 1 \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

Assignment

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver

\[ p_1, \ p_2, \neg p_3, \ p_4 \]

\[ x \geq 0, \ y = x + 1, \neg (y > 2), \ y < 1 \]

Unsatisfiable

Theory Solver

\[ x \geq 0, \ y = x + 1, \ y < 1 \]
SAT + Theory solvers

**Basic Idea**

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]

\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

**SAT Solver**

**Assignment**

\[ p_1, \ p_2, \neg p_3, \ p_4 \]

\[ x \geq 0, \ y = x + 1, \neg (y > 2), \ y < 1 \]

**Theory Solver**

New Lemma

\[ \neg p_1 \lor \neg p_2 \lor \neg p_4 \]

Unsatisfiable

\[ x \geq 0, \ y = x + 1, \ y < 1 \]
SAT + Theory solvers

New Lemma
\( \neg p_1 \lor \neg p_2 \lor \neg p_4 \)

Unsatisfiable
\( x \geq 0, y = x + 1, y < 1 \)

AKA Theory conflict

Theory Solver