CodePeer: Re-Engineering Abstract Interpretation

for Precise, Scalable Whole-Program Verification

Steve Baird and Tucker Taft
AdaCore Inc

Flight Software Workshop
JHU/Applied Physics Lab
October 2015

AdaCore
The GNAT Pro Company

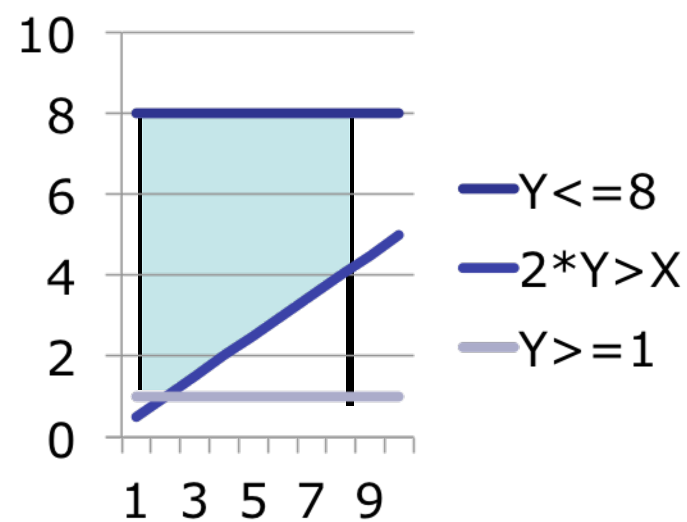
Where we are going

- Background
- Abstract Interpretation and alternatives
- Re-engineering with value-number approach in CodePeer Static Analyzer
- Inferring pre- and post- conditions in CodePeer
- Additional Roles for Abstract-Interpretation-Based Static Analysis

What is Abstract Interpretation?

- **Approximates the set of possible states of all variables at each point in a program, to allow proofs for safety, security, or correctness.**
- **Iterates until a fixed-point, then checks for violations.**
- ***Constructs* the set of possible values, rather than *searching* through them (handles large ranges).**
- **Represents relationships, e.g. 2*Y > X, using, e.g. polygons/polyhedrons**

X in 1..8, Y in 1..8, 2*Y > X:

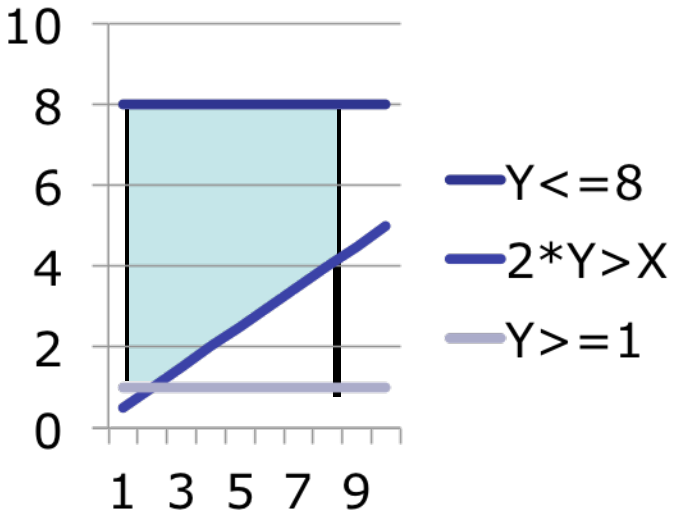


Alternative Program Proof Techniques

- **Model Checking (e.g. SPIN model checker)**
  - Searches through state space for states that violate desirable properties
  - State explosion is a challenge; may limit loop iterations
  - Symbolic Model Checking can help
- **Formal Proof (e.g. SPARK 2014 toolset)**
  - Constructs a series of Verification Conditions (VCs) that represent desired safety, security, or correctness properties that should hold at various points in the program
  - Use SMT Solver or equivalent to prove each Verification Condition
  - Use timeout to determine VC cannot be proved
  - Typically relies on programmer to provide pre/postconditions, loop invariants, etc.

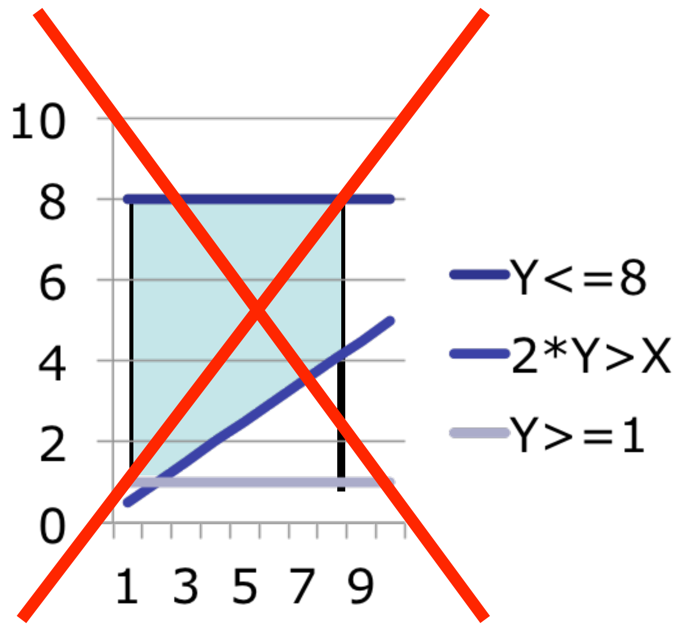
What is the problem with “classic” Abstract Interpretation?

- **Polyhedral representation of relationships between variables is fundamentally limiting (e.g. Y > B – X*Z/A)**
- **Many approaches exist (courtesy of WikiPedia):**
  - congruence relations on integers
  - convex polyhedra (high computational costs)
  - "octagons”
  - difference-bound matrices
  - linear equalities
- **Other issues:**
  - Initial value set for inputs may require exploring all paths that reach procedure
  - May require a driver or harness to provide realistic input values



What is the problem with "classic" Abstract Interpretation?

- **Polyhedral representation of relationships between variables is fundamentally limiting (e.g. Y > B – X*Z/A)**
- **Many approaches exist (courtesy of WikiPedia):**
  - congruence relations on integers
  - convex polyhedra (high computational costs)
  - "octagons"
  - difference-bound matrices
  - linear equalities
- **Other issues:**
  - Initial value set for inputs may require exploring all paths that reach procedure
  - May require a driver or harness to provide realistic input values



Can we Re-Engineer Abstract Interpretation to solve this?

Basic Trick used in CodePeer Static Analyzer:

- **Trick first learned in 1982 in optimizing Ada compiler**
- **Use "Value Numbers" to represent value of computing *interesting* expressions (e.g. "Y * 2 – X")**
- **Associate Value Sets (Vsets) with Value Numbers (VNs)**
- **Unlike variables, value numbers don't *change* in value over time**
  - *but what we know about them does*
- **Value set "*shrinks*" when we do a conditional jump**
  - e.g. **if** Y*2 > X **then** …
    - in **then** part we know "Y*2-X" in 1 .. +inf
    - in **else** part we know "Y*2-X" in –inf .. 0
- **Also *shrinks* when we do a check or an assertion**
  - e.g. **assert** X + Y > Z ➔ "X+Y-Z" in 1 .. +inf

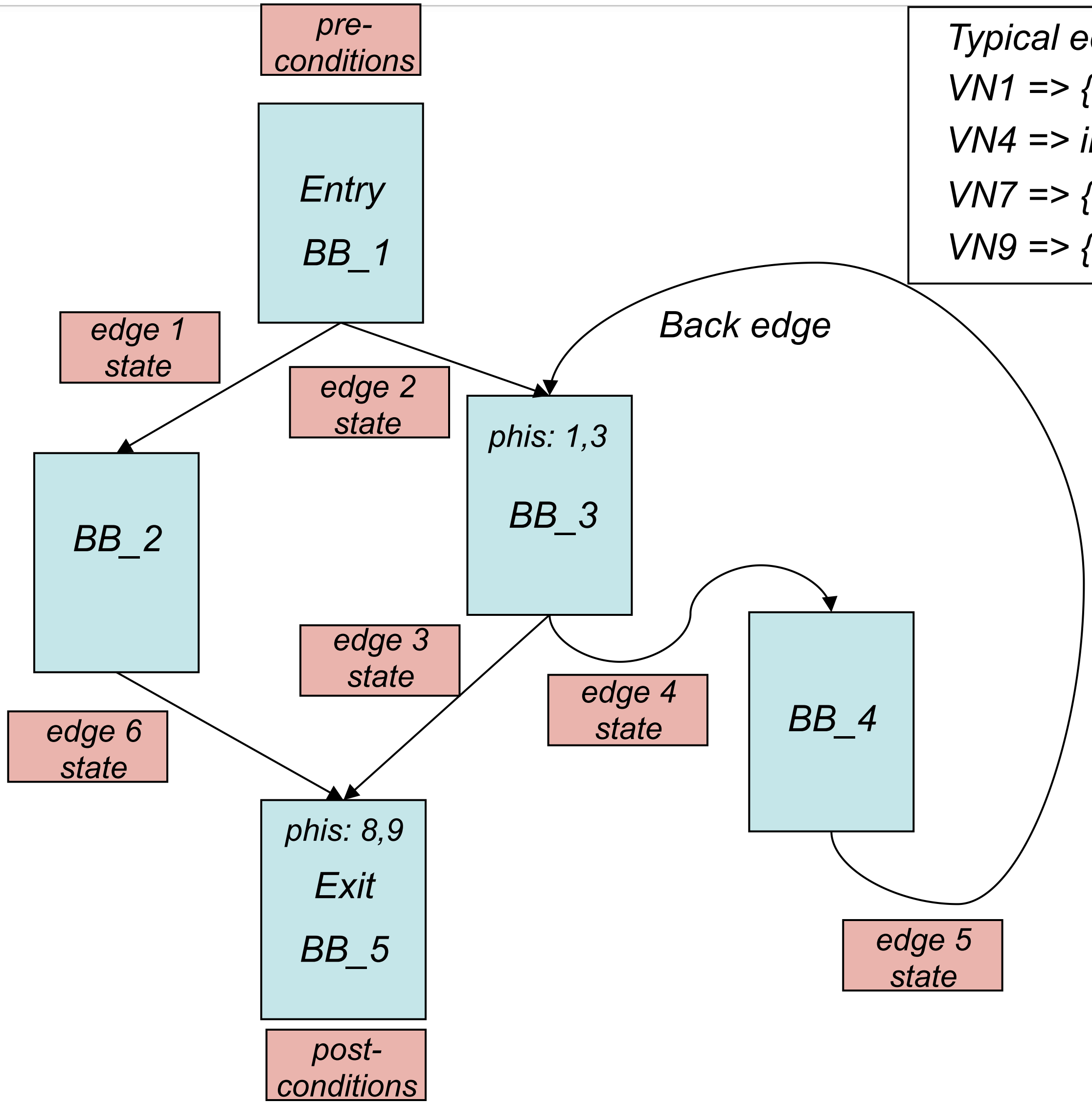
AdaCore
The GNAT Pro Company

How do Value Numbers simplify value-set determination?

- Only need to represent simple sets of integers, floats, or addresses for each value number (no polyhedrons!)
- Relationships between VNs are represented in value-number definition table (aka *computation table*)
- All variables/expressions with same value share a VN
- Each *basic block* and *edge* of Control Flow Graph (CFG) has its own *map* from VN to Value Set
- When one VN's Vset shrinks, we can efficiently *propagate* it to all related VNs in same map

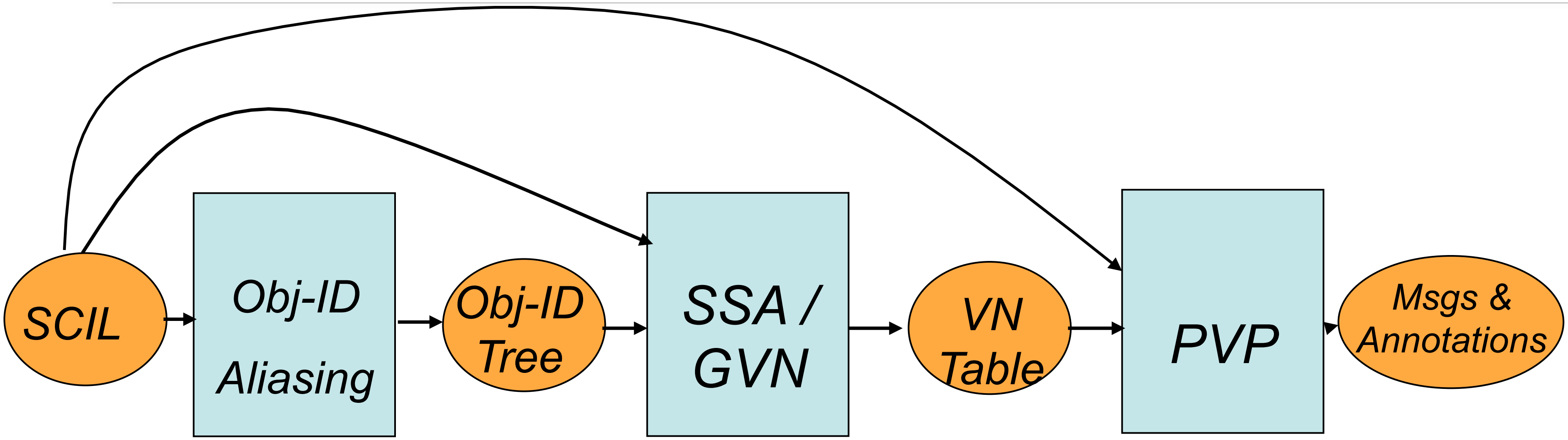
Typical VN=>Vset Map:
VN1 => {1..4}
VN4 => inverse{null}
VN7 => {0..+inf}
VN9 => {&obj2,&obj4}

Typical Control Flow Graph



Typical edge state:
VN1 => {1..4}
VN4 => inverse{null}
VN7 => {0..+inf}
VN9 => {&obj2,&obj4}

Overall 3-phase Structure of CodePeer Static Analyzer



Kinds of Annotations Inferred:

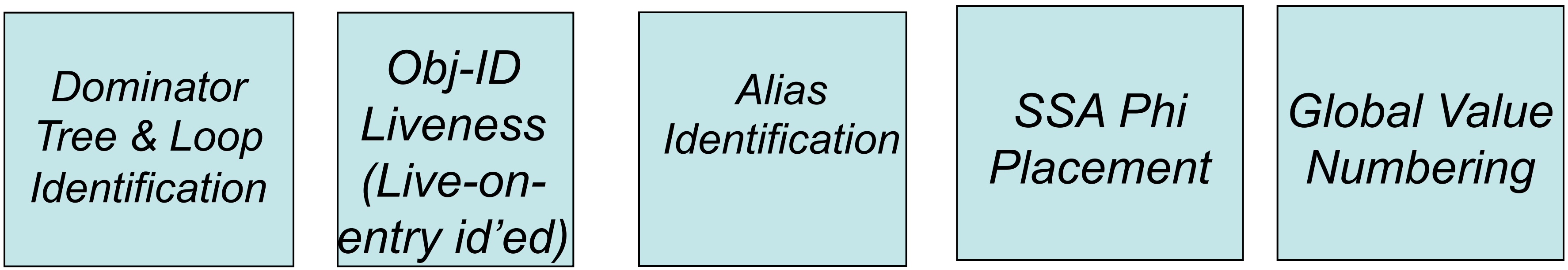
SSA
- *Inputs (Live-On-Entry)*

Obj-ID
- *Outputs (DMODs – Direct Modifications)*
- *New Objects (Escape Analysis),*

PVP
- *Preconditions, Postconditions, Presumptions*

AdaCore
The GNAT Pro Company

Static Single Assignment/Global Value Numbering (SSA/GVN) phase



- *Create a "Control Flow Graph" of basic blocks and find loops, etc.*
- *Assign a unique "value number" to every fetch of a variable and every computation*
- *Use "phi" value numbers at join points to represent alternative values*
  - *E.g. if X > Y then Max := X else Max := Y end if; Max == ? => PhiVN(X, Y)*
- *"Kappa" node introduced to represent value of potential alias after assignment*
  - *E.g. A[I] := 3; A[J] := 5; A[I] == ? => KappaVN(I == J? 5: 3)*

Goals of Possible Value Propagation (PVP) Phase

- **Compute Possible Value Set for every Value Number in every Basic Block**
  - Map of Value number to Value set
- **Check for failures of run-time checks, user assertions, and preconditions of called routines**
  - Initially assume checks will pass and thereby infer Pre/Postconditions
  - Iterate until a fixed point
  - Make final pass to report checks that still fail

Typical VN=>Vset Map:
VN1 => {1..4}
VN4 => inverse{null}
VN7 => {0..+inf}
VN9 => {&obj2,&obj4}

PVP Iterative Cycle

- **Works one basic-block (BB) at a time.**
- **Initializes the "checks" table for each BB**
  - Summary of all "implicit" and "explicit" checks performed in BB for each VN
- **Applies checks of BB and then propagates changes in VN value sets until they stabilize**
  - Propagate "down" to constituent VNs, then "up" to composite VNs
  - e.g. VN1 = VN2 * VN3
    - Shrink VN1, propagate "down" to VN2 and VN3;
    - Shrink VN3, propagate "up" to VN1.
- **Computes VN state for each BB and for each outgoing edge**
  - Saves edge states for later iterations
  - Saves exit-block state for pre/postconditions

Inferring Preconditions in PVP phase

- **Each Input** (parameter or global) **is given an "Input VN" to represent its** (unknown) ***initial*** **value**
- **Vset for Input VN is full possible range of type**
  - ***e.g.*** Inp_VN1 **in** -2^{31} .. $+2^{31}-1$
- **As we apply checks and assertions Vset for Input VN may shrink, eliminating the "bad" values.**
- **VNs corresponding to combinations of Inputs and Literals** (e.g. Inp_VN1 – Inp_VN2 * 2) **might also undergo checks/assertions and might shrink** (directly or indirectly)
- **At exit block, Vset of Input VN or combination thereof represents the "*good*" values** (those that survived the checks and assertions), **i.e. a *precondition***

e.g. Inp1 – Inp2 * 2 **in** 1 .. +inf;   Inp2 **in** –inf .. -1 | +1 .. +inf

→ ***Preconditions:*** Inp1 > Inp2 * 2;   Inp2 != 0

Inferring Postconditions in PVP phase

Same principle applies for Postconditions …

- **In Exit Block, Vset associated with a VN that represents the final value of some Output** (parameter, global, or function result) **or combination thereof, represents possible values upon completion, i.e. a *postcondition***
- ***Example:***

```
proc Incr(X : in out Integer) is
    X := X + 1
end proc Incr
```

initial value of X is Inp_VN1: — *Inp_VN1 in* $-2^{31} .. +2^{31}-1$

final value of X is VN2 = Inp_VN1 + 1: — *VN2 in* $-2^{31}+1 .. +2^{31}$

check that X + 1 doesn't overflow: — *VN2 in* $-2^{31}+1 .. +2^{31}-1$

propagates to: — *Inp_VN1 in* $-2^{31} .. +2^{31}-2$

➔ *precondition: X'Initial <=* $2^{31}-2$

➔ *postcondition: X'Final in* $-2^{31}+1 .. +2^{31}-1$*; X'Final = X'Initial+1*

CodePeer Screen shot showing Inferred Pre/Postconditions

```
P/P      --  Subp: fsw_example
         --
         --  Preconditions:
         --    N >= 1
         --
         --  Postconditions:
         --    A = One-of{1, 101, N - 1}
         --    A in (0..121 | 789..2^31-2)
         --    B = One-of{2, 102, N}
         --    B in (1..122 | 790..2^31-1)
         --    B = A + 1
         --
         --  Test Vectors:
         --    N: {123..456}, {457..789}, {1..122 | 790..2^31-1}
         --    A: {1}, {101}
         --    B: {2}, {102}
         --
      procedure Fsw_Example (N : Natural;
                             A, B : out Natural) is
      begin
       case N is
          when 123 .. 456 =>
             A := 1;
             B := 2;
          when 457 .. 789 =>
             A := 101;
             B := 102;
          when others =>
             A := N - 1;
             B := N;
       end case;
       pragma Assert (B - A = 1);
      end Fsw_Example;
```

Additional Roles for CodePeer Static Analyzer

- **Formal Prover for "Low-Hanging fruit"**
  - Verification conditions generated for every run-time check and every assertion, precondition, etc. in SPARK code
  - SMT Solvers can be slow on some verification conditions
  - CodePeer can quickly prove about 98% of checks cannot fail
  - Use CodePeer as pre-filter, so SMT solvers only need to deal with remaining 2% of verification conditions
- **Verifier for Model-Based Code Generation**
  - AdaCore QGen tool translates Simulink model into source code, either MISRA C or SPARK subset of Ada
  - Can use CodePeer to check for possible run-time errors in generated SPARK code
  - If CodePeer gives clean bill of health on SPARK code, it applies to generated MISRA C code as well

Summary

Abstract Interpretation is useful for whole-program verification

- **Advantages:**
  - Can handle large-range types
  - Avoids state-space explosion
  - Needs no user-provided pre/postconditions or loop invariants
- **Problems with classic approach:**
  - Limited ability to represent relationships between variables
  - May require top-down walk of all paths to provide value sets for inputs
  - May require driver or harness to provide realistic inputs
- **Re-engineered value-number based approach in CodePeer:**
  - Can represent arbitrary relationships between variables
  - Uses efficient mechanism to propagate information between value numbers
  - Can infer both numeric and symbolic pre/postconditions so no need for drivers/harnesses or top-down walk
- **Other Roles:** Filter Low-hanging fruit; Validate generated code for model