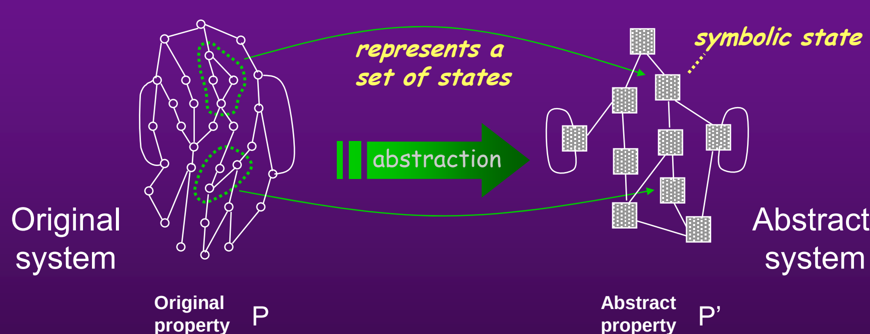
Abstraction of Source Code

(from Bandera lectures and talks)

Abstraction: the key to scaling up



Safety: The set of behaviors of the abstract system over-approximates the set of behaviors of the original system

Data Abstraction

- Data Abstraction
  - Abstraction proceeds component-wise, where variables are components

Data Type Abstraction

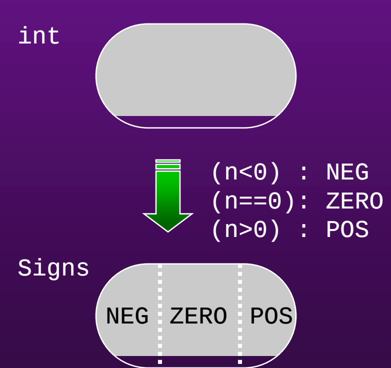
Collapses data domains via abstract interpretation:

Code

```
int x = 0;
if (x == 0)
  x = x + 1;
```

```
Signs x = ZERO;
if (Signs.eq(x,ZERO))
  x = Signs.add(x,POS);
```

Data domains



Hypothesis

Abstraction of data domains is necessary

Automated support for

- **Defining abstract domains (and operators)**
- **Selecting abstractions for program components**
- **Generating abstract program models**
- **Interpreting abstract counter-examples**

will make it possible to

- **Scale property verification to realistic systems**
- **Ensure the safety of the verification process**

Definition of Abstractions in BASL

```
abstraction Signs abstracts int
begin
  TOKENS = { NEG, ZERO, POS };

  abstract(n)
    begin
        n  <  0   -> {NEG};
        n  ==  0  -> {ZERO};
        n  >  0   -> {POS};
    end
```

Automatic Generation

```
operator + add
 begin
  (NEG , NEG)  -> {NEG}
  (NEG , ZERO) -> {NEG}  ;
  (ZERO, NEG)  -> {NEG}  ;
  (ZERO, ZERO) -> {ZERO} ;
  (ZERO, POS)  -> {POS}  ;
  (POS , ZERO) -> {POS}  ;
  (POS , POS)  -> {POS}  ;
  (_,_) -> {NEG,ZERO,POS};
  /* case (POS,NEG),(NEG,POS) */
 end
```

Example: Start safe, then refine: `+(NEG,NEG)={NEG,`~~ZERO,POS~~`}`

Proof obligations submitted to PVS...

Forall n1,n2: neg?(n1) and neg?(n2) implies not pos?(n1+n2)

Forall n1,n2: neg?(n1) and neg?(n2) implies not zero?(n1+n2)

Forall n1,n2: neg?(n1) and neg?(n2) implies not neg?(n1+n2)

Compiling BASL Definitions

```
abstraction Signs abstracts int
begin
  TOKENS = { NEG, ZERO, POS };

  abstract(n)
    begin
        n  <  0   -> {NEG};
        n  ==  0  -> {ZERO};
        n  >  0   -> {POS};
    end

 operator + add
  begin
   (NEG , NEG)  -> {NEG}  ;
   (NEG , ZERO) -> {NEG}  ;
   (ZERO, NEG)  -> {NEG}  ;
   (ZERO, ZERO) -> {ZERO} ;
   (ZERO, POS)  -> {POS}  ;
   (POS , ZERO) -> {POS}  ;
   (POS , POS)  -> {POS}  ;
   (_,_)-> {NEG, ZERO, POS};
   /* case (POS,NEG), (NEG,POS) */
  end
```

Compiled

```
public class Signs {
  public static final int NEG  = 0; // mask 1
  public static final int ZERO = 1; // mask 2
  public static final int POS  = 2; // mask 4

  public static int abs(int n) {
    if (n < 0) return NEG;
    if (n == 0) return ZERO;
    if (n > 0) return POS;
  }

  public static int add(int arg1, int arg2) {
    if (arg1==NEG  && arg2==NEG)  return NEG;
    if (arg1==NEG  && arg2==ZERO) return NEG;
    if (arg1==ZERO && arg2==NEG)  return NEG;
    if (arg1==ZERO && arg2==ZERO) return ZERO;
    if (arg1==ZERO && arg2==POS)  return POS;
    if (arg1==POS  && arg2==ZERO) return POS;
    if (arg1==POS  && arg2==POS)  return POS;
    return Bandera.choose(7);
    /* case (POS,NEG), (NEG,POS) */
  }
```

Interpreting Results

- **For an abstracted program, a counter-example may be infeasible because:**
  - **Over-approximation introduced by abstraction**
- **Example:**

  x = -2;   if(x + 2 == 0) then ...

  x = NEG; if(Signs.eq(Signs.add(x,POS),ZERO)) then ...

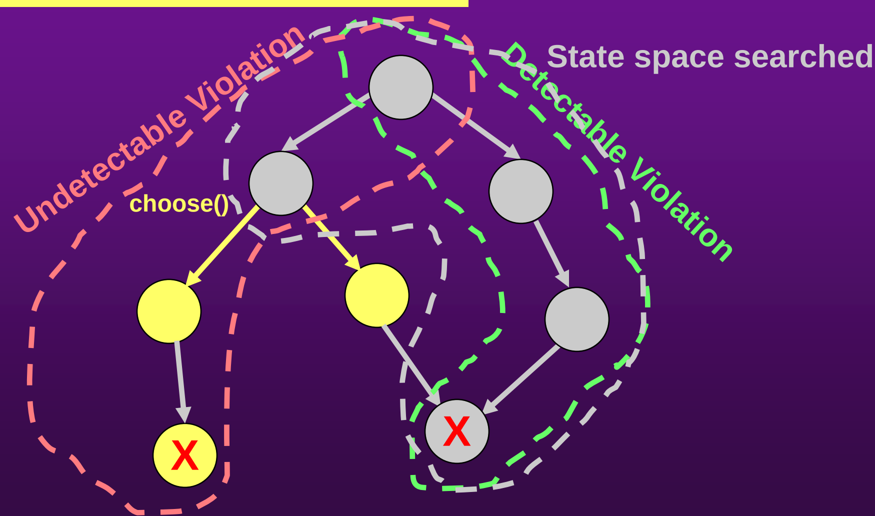
  {NEG,ZERO,POS}

Choose-free state space search

- **Theorem [Saidi:SAS'00]**

  Every path in the abstracted program where all assignments are deterministic is a path in the concrete program.
- **Bias the model checker**
  - **to look only at paths that do not include instructions that introduce non-determinism**
- **JPF model checker modified**
  - **to detect non-deterministic choice (i.e. calls to Bandera.choose()); backtrack from those points**

Choice-bounded Search



Counter-example guided simulation

- **Use abstract counter-example to guide simulation of concrete program**
- **Why it works:**
  - **Correspondence between concrete and abstracted program**
  - **Unique initial concrete state**

Example of Abstracted Code

Java Program:

```
   class App{
    public static void main(…) {
[1]  new AThread().start();
     …
[2]  int i=0;
[3]  while(i<2) {
       …
[4]    assert(!Global.done);
[5]    i++;
    }}}

   class Athread extends Thread {
    public void run() {
     …
[6]  Global.done=true;
    }}
```

Abstracted Program:

```
   class App{
    public static void main(…) {
[1]  new AThread().start();
     …
[2]  int i=Signs.ZERO;
[3]  while(Signs.lt(i,signs.POS)){
       …
[4]    assert(!Global.done);
[5]    i=Signs.add(i,Signs.POS);
    }}}

   class Athread extends Thread {
    public void run() {
     …
[6]  Global.done=true;
    }}
```

inism!
i=zero

Choose-free counter-example: 1 - 2 - 6 - 3 - 4

Example of Abstracted Code

Java Program:

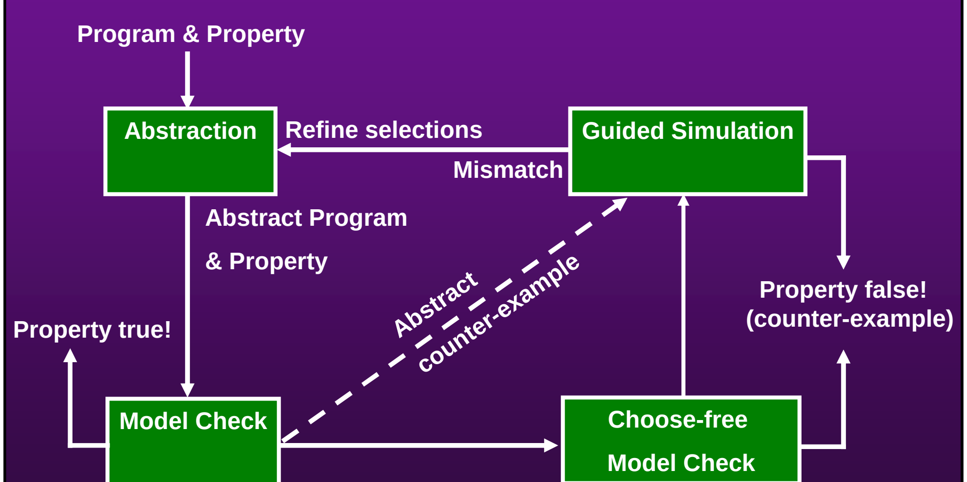
```
   class App{
    public static void main(…) {
[1]  new AThread().start();
     …
[2]  int i=0;                        i=0
[3]  while(i<2) {                    i=1
       …
[4]    assert(!Global.done);         i=0
[5]    i++;                          i=1
    }}}

   class Athread extends Thread {
    public void run() {
     …
[6]  Global.done=true;
    }}
```

Abstracted Program:

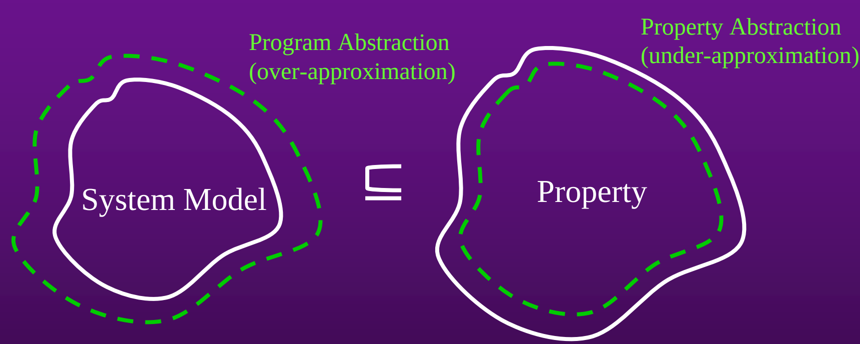
```
   class App{
    public static void main(…) {
[1]  new AThread().start();
     …
[2]  int i=Signs.ZERO;                         i=zero
[3]  while(Signs.lt(i,signs.POS)){             i=pos
       …
[4]    assert(!Global.done);                   i=zero
[5]    i=Signs.add(i,Signs.POS);               i=pos
    }}}

   class Athread extends Thread {
    public void run() {
     …
[6]  Global.done=true;
    }}
```

```
Abstract counter-example: 1 - 2 - 3 - 4 - 5 - 3 - 4 - 5 - 3 - 6 - 4
                                                      Mismatch
```

Hybrid Approach



Property Abstraction



If the *abstract property* holds on the *abstract system*, then the *original property* holds on the *original system*

Property Abstraction

- **Properties are temporal logic formulas, written in negational normal form.**
- **Abstract propositions under-approximate the truth of concrete propositions.**

Examples:

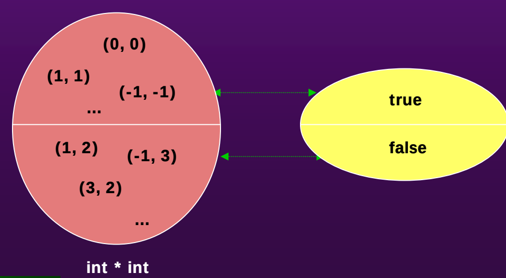
- **Invariance property:** $G\ (x > -1)$
- **Abstracted to:** $G\ ((x == zero) \lor (x == pos))$

- **Invariance property:** $G\ (x > -2)$
- **Abstracted to:** $G\ ((x == zero) \lor (x == pos))$

Predicate Abstraction

- Predicate Abstraction
  - Use a boolean variable to hold the value of an associated predicate that expresses a *relationship* between variables

predicate: x == y



An Example

```
Init:
  x := 0;  y := 0;  z := 1;
  goto Body;

Body:
  assert (z == 1);
  x := (x + 1);
  y := (y + 1);
  if (x == y) then Z1 else Z0;

Z1:  z := 1;
  goto Body;

Z0:  z := 0;
  goto Body;
```

- x and y are unbounded
- Data abstraction does not work in this case --- abstracting component-wise (per variable) cannot maintain the *relationship* between x and y
- We will use predicate abstraction in this example

Predicate Abstraction Process

- Add boolean variables to your program to represent current state of particular predicates
  - E.g., add a boolean variable `[x=y]` to represent whether the condition x==y holds or not
- These boolean variables are updated whenever program statements update variables mentioned in predicates
  - E.g., add updates to `[x=y]` whenever x or y or assigned

An Example

```
Init:
  x := 0;  y := 0;  z := 1;
  goto Body;

Body:
  assert (z = 1);
  x := (x + 1);
  y := (y + 1);
  if (x = y) then Z1 else Z0;

Z1:  z := 1;
  goto Body;

Z0:  z := 0;
  goto Body;
```

- We will use the predicates listed below, and remove variables x and y since they are unbounded.
- Don't worry too much yet about how we arrive at this particular set of predicates; we will talk a little bit about that later

Predicates	Boolean Variables
`p1: (x = 0)`	`b1: [(x = 0)]`
`p2: (y = 0)`	`b2: [(y = 0)]`
`p3: (x = (y + 1))`	`b3: [(x = (y + 1))]`
`p4: (x = y)`	`b4: [(x = y)]`

This is our new syntax for representing boolean variables that helps make the correspondence to the predicates clear

Transforming Programs

An example of how to transform an assignment statement

Predicates

- [(x = 0)]
- [(y = 0)]
- [(x = (y + 1))]
- [(x = y)]

Assignment Statement

```
x := 0;
```

The statement to the left is replaced the statements below

```
[(x=0)]       := true;

[(x=(y+1))]  := if [$(y=0)] then false
                  else top;

[(x=y)]       := if [$(y = 0)] then true
                  else
                    if ![$(y=0)] then false
                    else top;
```

Where:

- [$P] = prev. value of [P]
- top is a non-deterministic choice between true and false

Make a more compact representation using a helper function H (following SLAM notation)

```
[(x=0)]       := true;

[(x= (y+1))]  :=H(false, [$(y=0)]);

[(x=y)]  := H([$(y=0)], ![$(y=0)]);
```

Where:

$$H(e, e_2) = \begin{cases} \text{true, if } e_1 \\ \text{false, if } e_2 \\ \text{top, otherwise} \end{cases}$$

State Simulation

Given a program abstracted by predicates E_1, …, E_n, an abstract state simulates a concrete state if E_i holds on the concrete state iff the boolean variable [E_i] is true *and* remaining concrete vars and control points agree.

Concrete		*Abstract*
(n2,[x= 2, y =2, z =0]) (n2,[x =3, y = 3, z = 0])	*simulates*	(n2,[[x=0] = False, [y=0] = False, [x=(y+1)] = False, [x=y] = True, z = 0])
(n2,[x =1, y = 0, z = 1])	*simulates*	(n2,[[x=0] ! False, [y=0] ! True, [x=(y+1)] ! True, [x=y] ! False, z ! 1])
(n2,[x = 3, y = 3, z = 1])	*does not simulate*	

Computing Abstracted Programs

(n_1,σ_1) ⟸ simulates ⟸ $(n_1, [[P_1], [P_2], \ldots, [P_n]])$ — *Truth values of predicates on σ_1.*

↓ ↓

(n_2,σ_2) ⟸ simulates ⟸ $(n_2, [[P_1]', [P_2]', \ldots, [P_n]'])$ — *Truth values of predicates on σ_2.*

- For each statement *s* in the original program, we need to compute a new statement that describes how the given predicates change in value due to the effects of *s*.
- To do this for a given predicate P_i, we need to know if we should set $[P_i]$ to true or false in the abstract target state.
- Thus, we need to know the conditions at (n_1, σ_1) that guarantee that $[P_i]$ will be true in the target state and the conditions that guarantee that $[P_i]$ will be false in the target state. These conditions will be used in the helper function H.

$$[P\_i] := H(e, e_2) = \begin{cases} \text{true, if } e_1 \\ \text{false, if } e_2 \\ \text{top, otherwise} \end{cases}$$

Conditions that make $[P_i]$ true.

Conditions that make $[P_i]$ false.

Computing Abstracted Programs

Example

(n_1,σ_1) → $(n_1, [[P_1], [P_2], \ldots, [P_n]])$

a: x := 0;   [x=y] := H(…?…, …?…)

(n_2,σ_2) → $(n_2, [[P_1]', [P_2]', \ldots, [P_n]'])$

- What conditions have to hold before a is executed to guarantee that x=y is true (false) after a is executed?
  - Note: we want the least restrictive conditions
- The technical term for what we want is the "weakest pre-condition of a with respect to x=y"

Let's take a little detour to learn about weakest preconditions.

Floyd-Hoare Triples

$$\{F_1\}\ \underbrace{C}_{\text{Command}}\ \{F_2\}$$

Pre-condition (boolean formula) — F_1

Post-condition (boolean formula) — F_2

A triple is a logical judgement that holds when the following condition is met:

For all states *s* that satisfy F_1 (i.e., $s \models F_1$), if executing C on s terminates with a state s', then $s' \models F_2$.

Weaker/Stronger Formulas

- If $F' \Rightarrow F$ (F' implies F), we say that F is weaker than F'.
- Intuitively, F' contains as least as much information as F because whatever F says about the world can be derived from F'.
- Intuitively, stronger formulas impose more restrictions on states.

Thinking in terms of sets of states…

Let $S_{F'} = \{s \mid s \models F'\}$
$S_F = \{s \mid s \models F\}$

Note that $S_{F'} \subseteq S_F$ since F' imposes more restrictions than F

Question: what formula is the weakest of all? (In other words, *what formula describes the largest set of states? What formula imposes the least restrictions?*)

Weakest Preconditions

The *weakest precondition of C with respect to F_2* (denoted $WP(C,F_2)$) is a formula F_1 such that

$\{F_1\}\ C\ \{F_2\}$

and for all other F'_1 such that $\{F'_1\}\ C\ \{F_2\}$,

$F'_1 \Rightarrow F_1$ (F_1 is weaker then F'_1).

- This notion is useful because it answers the question: "what formula F_1 captures the fewest restrictions that I can impose on a state s so that when s' = [[C]]s then s' $\models F_2$?"
- WP is interesting for us when calculating predicate abstractions because for a given command C and boolean variable $[P_i]$, we want to know the least restrictive conditions that have to hold before C so that we can conclude that P_i is definitely true (false) after executing C.

Calculating Weakest Preconditions

Calculating WP for assignments is easy:

$WP(x := e, F) \quad = \quad F[x \leftarrow e]$

- Intuition: x is going to get a new value from e, so if F has to hold after x := e, then $F[x \leftarrow e]$ is required to hold before x := e is executed.

Examples

$WP(x := 0, x = y)$	=	$(x = y)[x \leftarrow 0]$	=	$(0 = y)$
$WP(x := 0, x = y + 1)$	=	$(x = y + 1)[x \leftarrow 0]$	=	$(0 = y + 1)$
$WP(x := x+1, x = y + 1)$	=	$(x = y + 1)[x \leftarrow x + 1]$	=	$(x + 1 = y + 1)$

Calculating Weakest Preconditions

Calculating WP for other commands (state transformers):

WP(skip, F)	= F
WP(assert e, F)	= $e \Rightarrow F$ $(\neg e \vee F)$
WP(assume e, F)	= $e \Rightarrow F$ $(\neg e \vee F)$

- Skip: since the store is not modified, then F will hold afterward iff it holds before.
- Assert and Assume: even though we have a different operational interpretation of assert and assume in the verifier, the definition of WP of these relies on the fact that we assume that if an assertion or assume condition is violated, it's the same as the command "not completing". Note that if *e* is false, then the triple *{$(\neg e \vee F)$} assert e {F}* always holds since the command never completes for any state.

Assessment

Intuition:

Source Program ***Abstracted Program***

Assignment to each boolean variable $[P_i]$ where each assignment has the form

But what's wrong with this?

$[P_i]$:= H(WP(C, P_i), WP(C, !P_i)).

Answer: the predicates P_i refer to concrete variables, and the entire purpose of the abstraction process is to remove those from the program. The point is that the conditions in the 'H' function should be stated in terms of the boolean variables $[P_i]$ instead of the predicates P_i.

Assessment

- In the case of x := 0 and the predicate x = y, we have

  WP(x := 0, x=y) = (0=y)
  WP(x := 0, !x=y) = !(0=y)

- In this case, the information in the predicate variables is enough to decide whether 0=y holds or not. That is, we can simply generate the assignment statement

```
[(x=y)] := H([$(y = 0)],![$(y=0)]);
```

Assessment

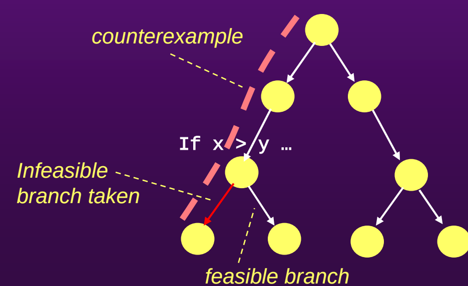
- In the case of x := 0 and the predicate x = (y+1), we have

  WP(x := 0, (x=y+1)) = (0=y+1)
  WP(x := 0, !(x=y+1)) = !(0=y+1)

- In this case, we don't have a predicate variable [0=y+1].
- We must consider combinations of our existing predicate variables that imply the conditions above. That is, we consider *stronger (more restrictive, less desirable but still useful)* conditions formed using the predicate variables that we have.

What Are Appropriate Predicates?

- In general, a difficult question, and subject of much research
- Research focuses on automatic discovery of predicates by processing (infeasible) counterexamples
- If a counterexample is infeasible, add predicates that allow infeasible branches to be avoided



Add the predicate x > y so that we have enough information to avoid the infeasible branch.

What Are Appropriate Predicates?

Some general heuristics that we will follow

- Use the predicates A mentioned in property P, if variables mentioned in predicates are being removed by abstraction
  - At least, we want to tell if our property predicates are true or not
- Use predicates that appear in conditionals along "important paths"
  - E.g., `(x=y)`
- Predicates that allow predicates above to be decided
  - E.g., `(x=0), (y=0), (x = (y + 1))`