AUTOMATED SOFTWARE ANALYSIS USING SAT SOLVING

Nazareno AGUIRRE

LOGIC FOR SOFTWARE VERIFICATION

Nazareno AGUIRRE & Diego GARBERVETSKY

AGENDA

- A well-known Computer Science problem
  - Software Correctness
- Two approaches
  - Deductive Verification
  - Automated Bounded Verification

SOFTWARE CORRECTNESS

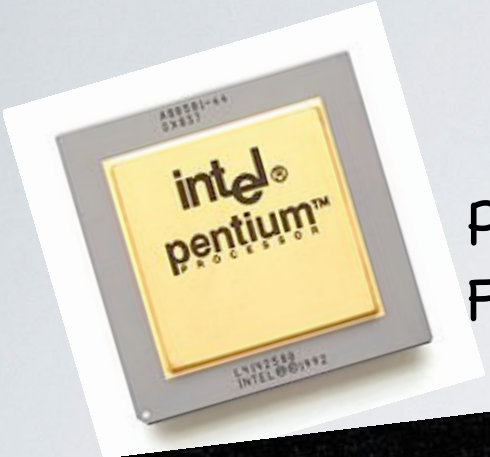
- What is it?

  the problem of guaranteeing that a software system (or an algorithm) behaves as intended, i.e., it correctly solves the problem it has been built to solve.

- Why should we care?

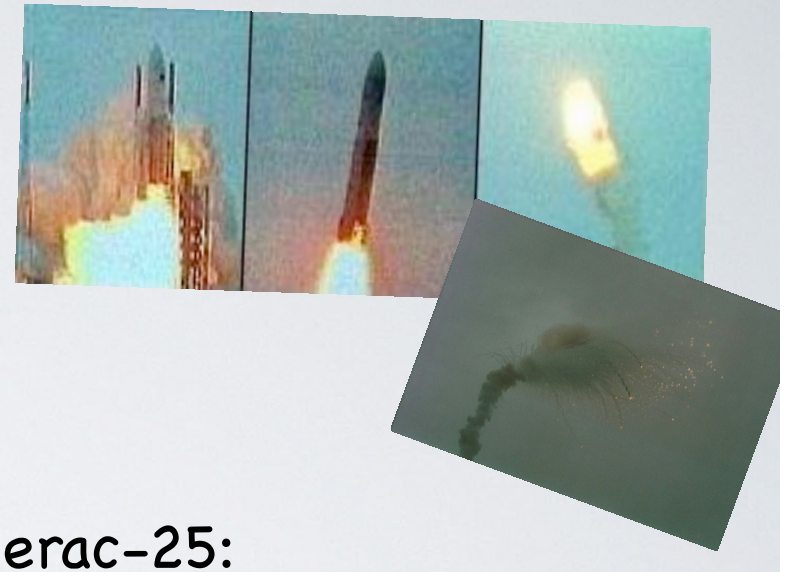
  many activities depend on software and some are critical. Their failure can have catastrophic consequences (e.g., automated driving, control of medical devices, financial applications, …)

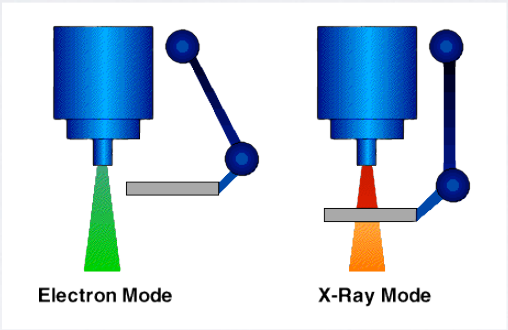
SOME FAMOUS BUGS



Pentium:
FDIV

Ariane 5:
64 bits fp
vs 16 bits int





Therac-25:
one man job

Mars Climate Orbiter:
Metric vs Imperial

HMS Sheffield:
Friendly Exocet

Electronic voting:
Integrity/Confidentiality

https://en.wikipedia.org/wiki/List_of_software_bugs

PROBLEM

How can we tell if a program is correct?

CLEARLY INCORRECT PROGRAMS

```c
#include <stdio.h>

int main {

    int n;
    scanf("%d;;

    printf(N)
```

```c
#include <stdio.h>

int main() {

    int *p = NULL;
    printf("%d", *p);

    return 0;
}
```

These programs *cannot* be correct

First syntactically incorrect. Second leads to a runtime failure.

WHAT ABOUT THIS ONE?

```
int f(int n, int m) {
    if (n > m)
        return n;
    else
        return m;
}
```

We cannot tell whether the program is correct or incorrect without knowing what is its **intended behavior**

PROGRAM SPECIFICATION

A description of the intended behavior of a program.

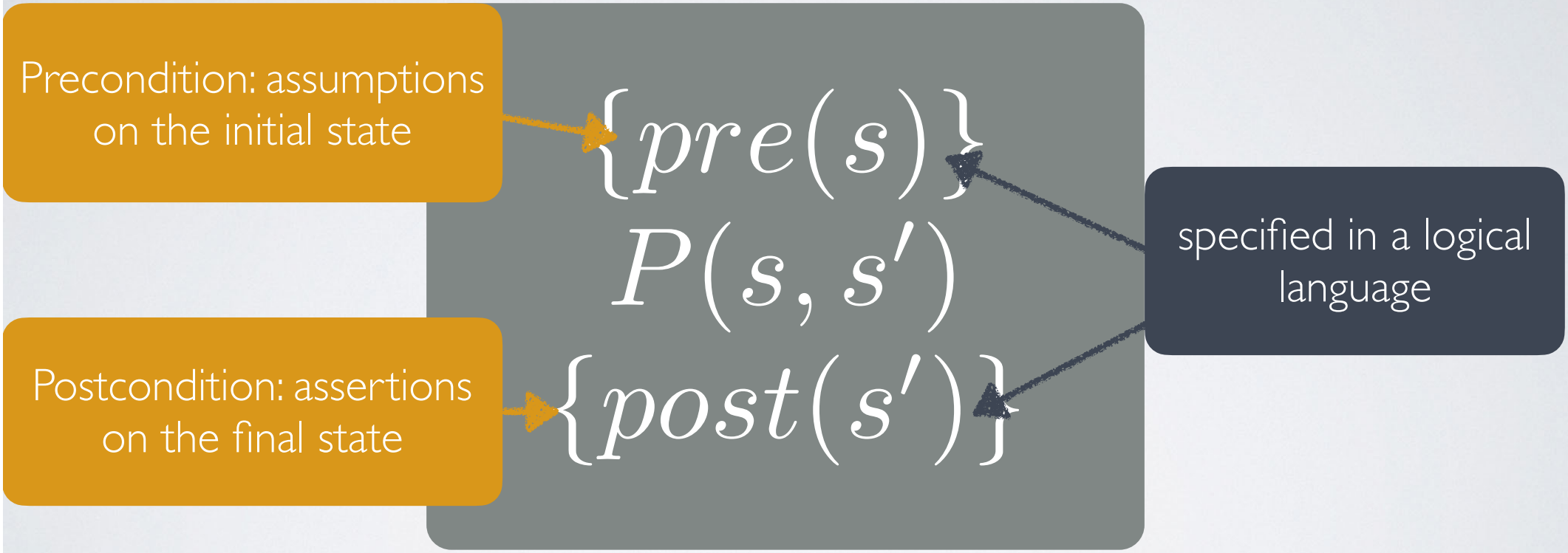
```
/*
 * Computes the maximum of two integers.
 */
int f(int n, int m) {
    if (n > m)
        return n;
    else
        return m;
}
```

PROGRAM SPECIFICATIONS IN NATURAL LANGUAGE

- **Most widely adopted form of specification**
  - Software documentation
  - Program comments
- Natural language is **inherently ambiguous**
  - Inadequate for rigorous correctness analysis
    - **cannot accurately contrast software behavior with software specification**

FORMAL PROGRAM SPECIFICATIONS

Gives rise to the use of **correctness assertions** to specify program behavior:



Assertion holds iff **every execution of P that starts in a state that satisfies pre, terminates and does so in a state that satisfies post**.

EXAMPLE

$$\{true\}$$

```
int result = 0;
if (n > m) {
    result = n;
}
else {
    result = m;
}
```

$$\{(\text{result} = \text{n} \vee \text{result} = \text{m}) \wedge \text{result} \geq \text{n} \wedge \text{result} \geq \text{m}\}$$

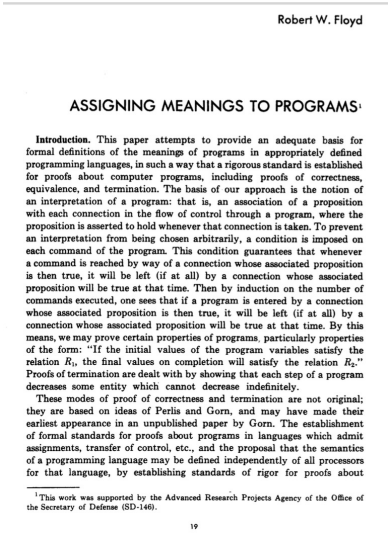
Approaches for software verification

- **Verification**: To prove the program is right
  - **Deductive Verification**
  - Abstract interpretation
  - ModelChecking (using abstractions)
- **Falsification**: To show the program is wrong
  - Testing/Fuzzing
  - Symbolic Execution
  - Monitoring
  - ModelChecking
  - **Automated Bounded Verification**

Deductive Program Verification

The task of **mathematically checking the correctness of a program with respect to a formal specification.**

Originally proposed by R. W. Floyd and C. A. R. Hoare, **based on program specification using first-orden logic assertions.**



Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

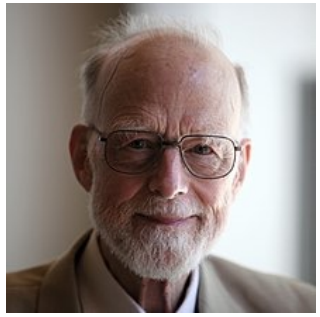
Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

[1] This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

19

Assigning Meanings to Programs, 1967



An Axiomatic Basis for Computer Programming, 1969

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$x = x + y \times 0$

$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$

The proof of the second of these is:

A5	$(r - y) + y \times (1 + q)$
	$= (r - y) + (y \times 1 + y \times q)$
A9	$= (r - y) + (y + y \times q)$
A3	$= ((r - y) + y) + y \times q$
A6	$= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to nonnegative numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

A10$_i$ $\neg\exists x \forall y \quad (y \leqslant x)$,

where all finite arithmetics satisfy:

A10$_f$ $\forall x \quad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

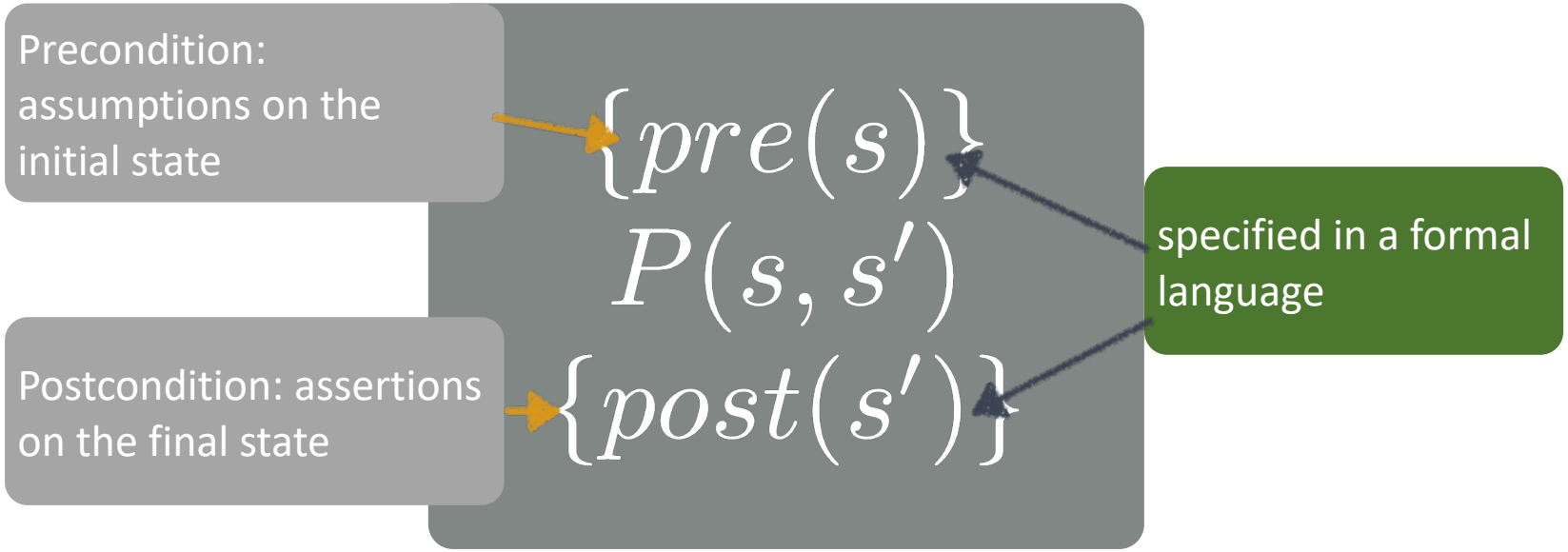
A11$_s$	$\neg\exists x \quad (x = \max + 1)$	(strict interpretation)
A11$_b$	$\max + 1 = \max$	(firm boundary)
A11$_M$	$\max + 1 = 0$	(modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

Program Verification

Gives rise to the use of **correctness assertions** (Hoare triples) to specify program behavior:



Assertion holds iff every execution of **P** that starts in a state **s** that satisfies **pre**, terminates and does so in a state **s'** that satisfies **post**.

Examples of Hoare Triples

{**false**} S {**Q**}

- Valid for all S and Q.

{**P**} S {**true**}

- Valid for all P and S.

{**P**} while (**true**) do skip {**Q**}

- Valid for all P and Q.

{**P**} S {**false**}

- False when P holds and S terminates, otherwise Valid

{**true**} S {**Q**}

- If S terminates, then Q must hold.

Example

{x = X & y = Y}

tmp := x;

x:=y;

y:=tmp;

{x = Y & y = X}

Example - Sort

$\{true\}$

```
i := 1;
while (i<=length(A)) do
   j := i;
   while (j>1) do
      if (A[j-1]>A[j]) then
         swap(j-1,j)
      j := j-1
   end
   i := i+1
end
```

$\{\forall x \in [1..\, length(A) - 1] : A[x] \leq A[x+1]\}$

Hoare Rules

Skip

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

Assignment

$$\frac{}{\{Q[x \rightarrow E]\}\ \texttt{x := E}\ \{Q\}}$$

Sequential composition

$$\frac{\{A\}\ \texttt{s1}\ \{C\}\quad \{C\}\ \texttt{s2}\ \{B\}}{\{A\}\ \texttt{s1;s2}\ \{B\}}$$

Conditional

$$\frac{\{A \mathbin{\&\&} cond\}\ \texttt{s1}\ \{B\}\quad \{A \mathbin{\&\&} !cond\}\ \texttt{s2}\ \{B\}}{\{A\}\ \texttt{if(cond) \{s1\} else \{s2\}}\ \{B\}}$$

Consequence

$$\frac{A' \rightarrow A\quad \{A\}\ \texttt{s}\ \{B\}\quad B \rightarrow B'}{\{A'\}\ \texttt{s}\ \{B'\}}$$

Iteration

$$\frac{\{A \mathbin{\&\&} cond\}\ \texttt{body}\ \{A\}\qquad (A \mathbin{\&\&} !cond) \Rightarrow B}{\{A\}\ \texttt{while(cond)\{body\}}\ \{B\}}$$

Program Verification

{true}
x := 5
{x = 5}

$$\frac{}{\{Q[x \rightarrow E]\}\ \texttt{x := E}\ \{Q\}}$$

Proof:

$$\{5 = 5\}\ \texttt{x := 5}\ \{\texttt{x} = 5\}\quad (\text{assignment})$$
$$\{true\}\ \texttt{x := 5}\ \{\texttt{x} = 5\}\quad (\text{arithmetic})$$

Program Verification

{x = 0}

x := x +1

{x > 0}

$$\frac{}{\{Q[x \rightarrow E]\}\ \texttt{x := E}\ \{Q\}}$$

$$\frac{A' \rightarrow A \quad \{A\}\ s\ \{B\} \quad B \rightarrow B'}{\{A'\}\ s\ \{B'\}}$$

Proof:

$$\{\mathtt{x}+1>0\}\ \mathtt{x := x+1}\ \{\mathtt{x}>0\} \quad \text{(assignment)}$$

$$\{\mathtt{x}=0\}\ \mathtt{x := x+1}\ \{\mathtt{x}>0\} \quad \text{(consequence}$$

$$(\mathtt{x}=0) \Rightarrow (\mathtt{x}+1>0))$$

Program Verification

{true}

x :=-x

{x < 0}

$$\frac{}{\{Q[x \rightarrow E]\}\ \mathtt{x := E}\ \{Q\}}$$

$$\frac{A' \rightarrow A \quad \{A\}\ s\ \{B\} \quad B \rightarrow B'}{\{A'\}\ s\ \{B'\}}$$

Proof:

$$\{-\mathtt{x} < 0\}\ \mathtt{x := -x}\ \{\mathtt{x} < 0\} \quad (\text{assignment})$$

$$\{\mathtt{x} > 0\}\ \mathtt{x := -x}\ \{\mathtt{x} < 0\} \quad (\text{arithmetic})$$

$true \Rightarrow \mathtt{x} > 0?$

Cannot apply consequence rule

Program Verification

Proof:

$$\{1 \le n\}$$
```
f = 1;
i = 1;
```
$$\{1 < i \wedge i < n \wedge f = i!\}$$

This step reduces the problem of proving the correctness of the original program to the three smaller problems:

(1) proving the initialization part;

(2) proving (inductively) that the premise R of rule 4 is valid for all iterations of the loop; and

(3) proving the finalization part.

These subproblems may be proved in any convenient order.

The third subproblem is easiest, since it involves only the *Skip* statement. Since that statement does nothing, its precondition must directly imply its post-condition. This can be shown by repeated applications of rule 5 and using our algebraic skills:

$$\begin{aligned} i \ge n \wedge 1 \le i \wedge i \le n \wedge f = i! &\Rightarrow \\ (i = n) \wedge f = i! &\Rightarrow \\ f = n! & \end{aligned}$$

That is, since $i \ge n$ and $i \le n$, it follows that $i = n$.

A strategy for solving the first subproblem uses rule 2 to break a *Block* into its individual components and then find the linking assertion $\{R'\}$:

$\{1 \le n\}$
```
f = 1;
```
$\{R'\}$
```
i = 1;
```
$\{1 \le i \wedge i \le n \wedge f = i!\}$

The linking assertion R' can be found by using rule 1 with the second assignment, so that $R' = \{1 \le 1 \wedge 1 \le n \wedge f = 1!\}$. So now we can insert this expression for R' and apply rule 1 to the first assignment:

$\{1 \le n\}$
```
f = 1;
```
$\{1 \le 1 \wedge 1 \le n \wedge f = i!\}$

obtaining $\{1 \le 1 \wedge 1 \le n \wedge 1 = 1!\}$, which simplifies to $1 \le n$. Thus, we have proved the validity of the *Block* by showing the validity of:

$$\begin{array}{c} \{1 \le n\} \quad \texttt{f = 1;} \quad \{1 \le 1 \wedge 1 \le n \wedge f = 1!\}, \\ \{1 \le 1 \wedge 1 \le n \wedge f = 1!\} \quad \texttt{i = 1;} \quad \{1 \le i \wedge i \le n \wedge f = i!\} \quad \vdash \\ \{1 \le n\}\ \texttt{f = 1; i = 1;}\ \{1 \le i \wedge i \le n \wedge f = i!\} \end{array}$$

Solving the second subproblem requires that we validate rule 4 for our invariant R and every iteration of the loop. So we must validate:

$\{s.\text{test} \wedge R\}s.\text{body}\{R\} \vdash \{R\}s\{\neg s.\text{test} \wedge R\}$, where **s** is a loop statement.

To do this for our particular loop test $i < n$, invariant R, and loop body, we need to show the validity of the following Hoare triple:

$\{i < n \wedge 1 \le i \wedge i \le n \wedge f = i!\}$
```
i = i + 1;
f = f * i;
```
$\{1 \le i \wedge i \le n \wedge f = i!\}$

Let us use rule 2 again to derive a linking assertion R' between the two statements in this *Block*:

$\{i < n \wedge 1 \le i \wedge i \le n \wedge f = i!\}$
```
i = i + 1;
```
$\{R'\}$
```
f = f * i;
```
$\{1 \le i \wedge i \le n \wedge f = i!\}$

Applying rule 1 to the second assignment gives:

$$R' = 1 \le i \wedge i \le n \wedge f \times i = i!$$

Applying rule 1 to the first assignment and R' gives:

$$1 \le i \wedge i \le n \wedge f \times (i+1) = (i+1)!$$

Now, we need to show that:

$$i < n \wedge 1 \le i \wedge i \le n \wedge f = i! \Rightarrow 1 \le i+1 \wedge i+1 \le n \wedge f \times (i+1) = (i+1)!,$$

in which case 5 can be used to complete our proof. To do this, we use the following principle from logic:

$$p \Rightarrow q, p \Rightarrow r, p \Rightarrow s \vdash p \Rightarrow q \wedge r \wedge s$$

and proceed by proving each term of this consequent separately.

First, we must show:

$$i < n \wedge 1 \le i \wedge i \le n \wedge f = i! \Rightarrow 1 \le i+1$$

This is valid, since $1 \le i$ is a term in the antecedent and $i \le i+1$ is always valid, algebraically.

Second, we must show:

$$i < n \wedge 1 \le i \wedge i \le n \wedge f = i! \Rightarrow i+1 \le n,$$

This is also valid, since $i < n$ is in the antecedent and it follows algebraically that $i+1 \le n$.

Third, we must show:

$$i < n \wedge 1 \le i \wedge i \le n \wedge f = i! \Rightarrow f \times (i+1) = (i+1)!$$

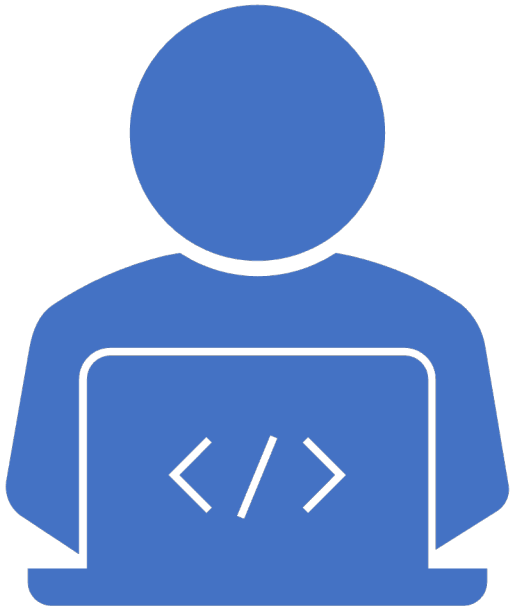
We can safely divide both sides of $f \times (i+1) = (i+1)!$ by $i+1$, since $i \ge 1$, resulting in:

$$i < n \wedge 1 \le i \wedge i \le n \wedge f = i! \Rightarrow f = i!$$

This is valid, since its consequent appears as a term in its antecedent.

This last step amounts to an induction proof, in which we show both: (1) the basis step in which $R(1)$ is established, and (2) the induction step in which $R(i) \Rightarrow R(i+1)$ is established for invariant $R(i)$ over all $i = \{i, \ldots n\}$. Since loops have indeterminate length, the invariant R is expressed as a function $R(i)$ on the number of iterations i that have taken place. The basis step, in which $R(1)$ is valid, corresponds to the validity of R before the first iteration.

This concludes our proof of the (partial) correctness of the *Factorial* function in Figure 12.3. Note that our proof does not address correctness when the calculation of $n!$ cannot be completed because too large a value for n was passed. We return to this important issues in a later section.



How can a **computer** tell if a program is correct?

Can we do it automatically?

Weakest precondition

{true}	{x = 3}	{x >= 3}	{x > 0}
x :=-x	x :=-x	x :=-x	x :=-x
{x < 0}	{x < 0}	{x < 0}	{x < 0}

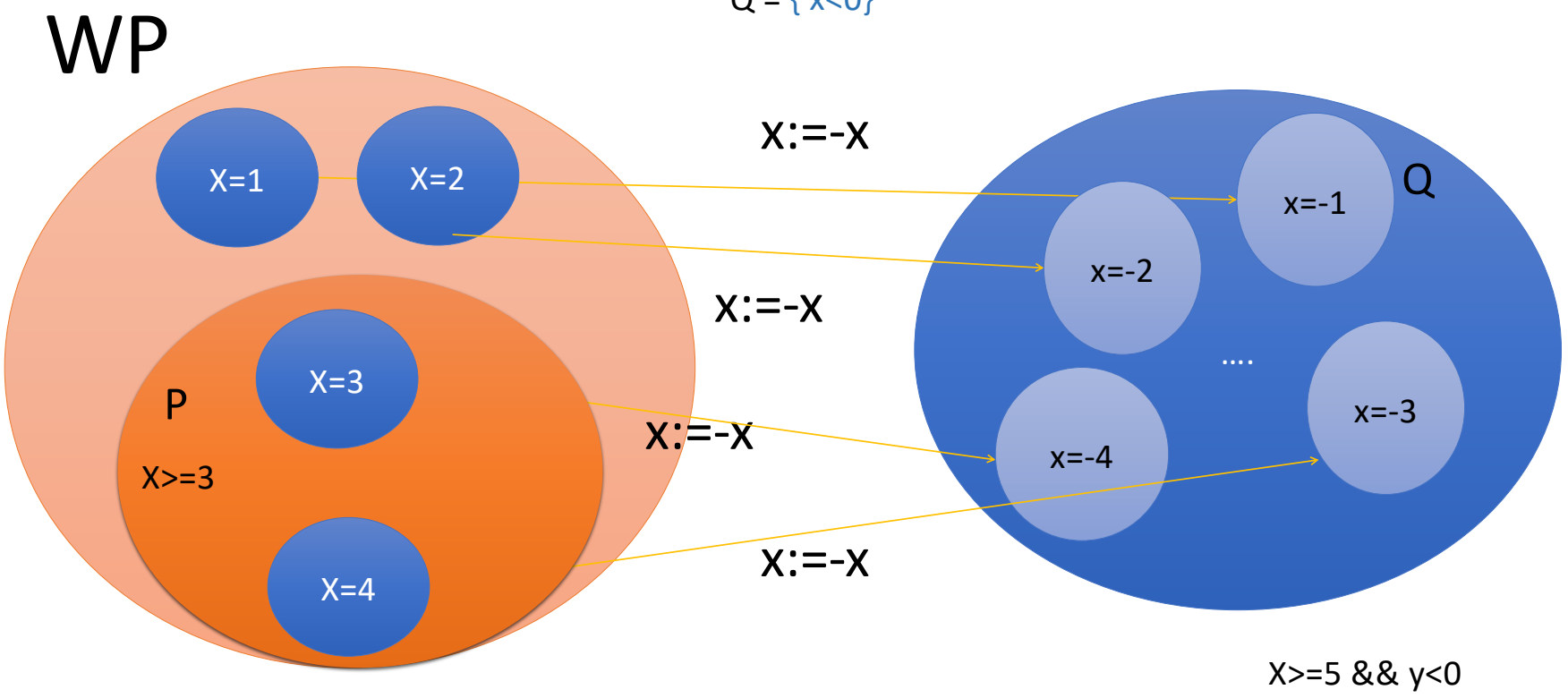
Weakest precondition

Is P the weakest precondition?

P = { x>=3}

x := -=x

Q = { x<0}



Weakest precondition

- WP(`skip`, Q) $=_{\text{def}}$ Q

$\{x < 0\}$

skip

$\{x < 0\}$

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]

{?} **x:= x+2** {x>=5}

{x>=5[x→x+2]} **x:=x+2**{x>=5}

{x+2>=5} **x:=x+2** {x>=5}

{x>=3} **x:=x+2** {x>=5}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))

{?}

x:=x+1

x:= x+2

{x>=5}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))

{?}

x:=x+1

{x>=3}

x:= x+2

{x>=5}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))

{x>=2}

x:=x+1

{x>=3}

x:= x+2

{x>=5}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))
- WP(`if(B){s1}else{s2}`, Q) $=_{def}$

  B=> WP(s1,Q) &&

  !B => WP(s2,Q)

{?}

If(x>0) then x:=x+1 else x:=2

{x=2}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))
- WP(`if(B){s1}else{s2}`, Q) $=_{def}$

  B=> WP(s1,Q) &&

  !B => WP(s2,Q)

Case X>0:

{?}

x:=x+1

{x=2}

Case X<=0:

{?}

x:=2

{x=2}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))
- WP(`if(B){s1}else{s2}`, Q) $=_{def}$

  B=> WP(s1,Q) &&

  !B => WP(s2,Q)

Case X>0:

{x=1}

x:=x+1

{x=2}

Case X<=0:

{True}

x:=2

{x=2}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))
- WP(`if(B){s1}else{s2}`, Q) $=_{def}$
  B=> WP(s1,Q) and !B => WP(s2,Q)

:

{(X>0 => x=1) and (X<=0 => True)}

If(x>0) then x:=x+1 else x:=2

{x=2}

Weakest precondition

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))
- WP(`if(B){s1}else{s2}`, Q) $=_{def}$
  B=> WP(s1,Q) and !B => WP(s2,Q)

{(X>0 => x=1) and (X<=0 => True)}
could be simplified into
{ x=1 or X<=0}

:

{ x=1 or X<=0}

If(x>0) then x:=x+1 else x:=2

{x=2}

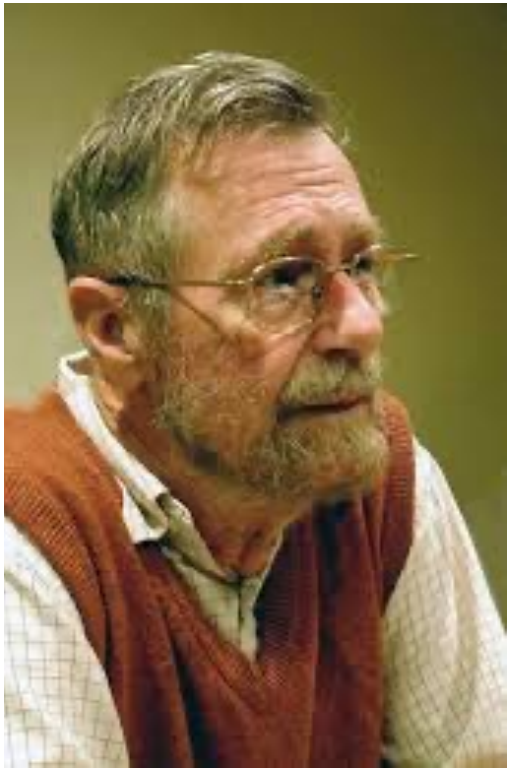
Why the weakest precondition is important?

$\{P\}\ S\ \{Q\}$

if and only if

$P \Rightarrow WP(S, Q)$

So, if we have an effective (algorithmic) way to compute the WP we are done ☺



Verification Condition

Given the following Hoare triple:

{Pre}

Program

{Post}

A **Verification Condition** (VC) for that triple:

Pre => WP(Program, Post)

Example

- WP(`skip`, Q) $=_{def}$ Q
- WP(`x:=E`, Q) $=_{def}$ Q[x→E]
- WP(`s1;s2` , Q) $=_{def}$ WP(s1, WP(s2, Q))
- WP(`if(B){s1}else{s2}`, Q) $=_{def}$ B=> WP(s1,Q) && !B => WP(s2,Q)

```
int abs(int a)
requires true
ensures result = |a|
{
  result = a;
  if (a<0)
    result = -result;
  else
    skip;
}
```

WP(`result=a; if(a<0)…`, result=|a|) =

P'

WP(`result=a; WP(if(a<0)…`, result=|a|))

P': a<0 => WP(`result=-result`, result=|a|) &&

a>=0 => WP(`skip`, result=|a|)

= (a<0 =>-result=|a|) && (a>=0 => result=|a|)

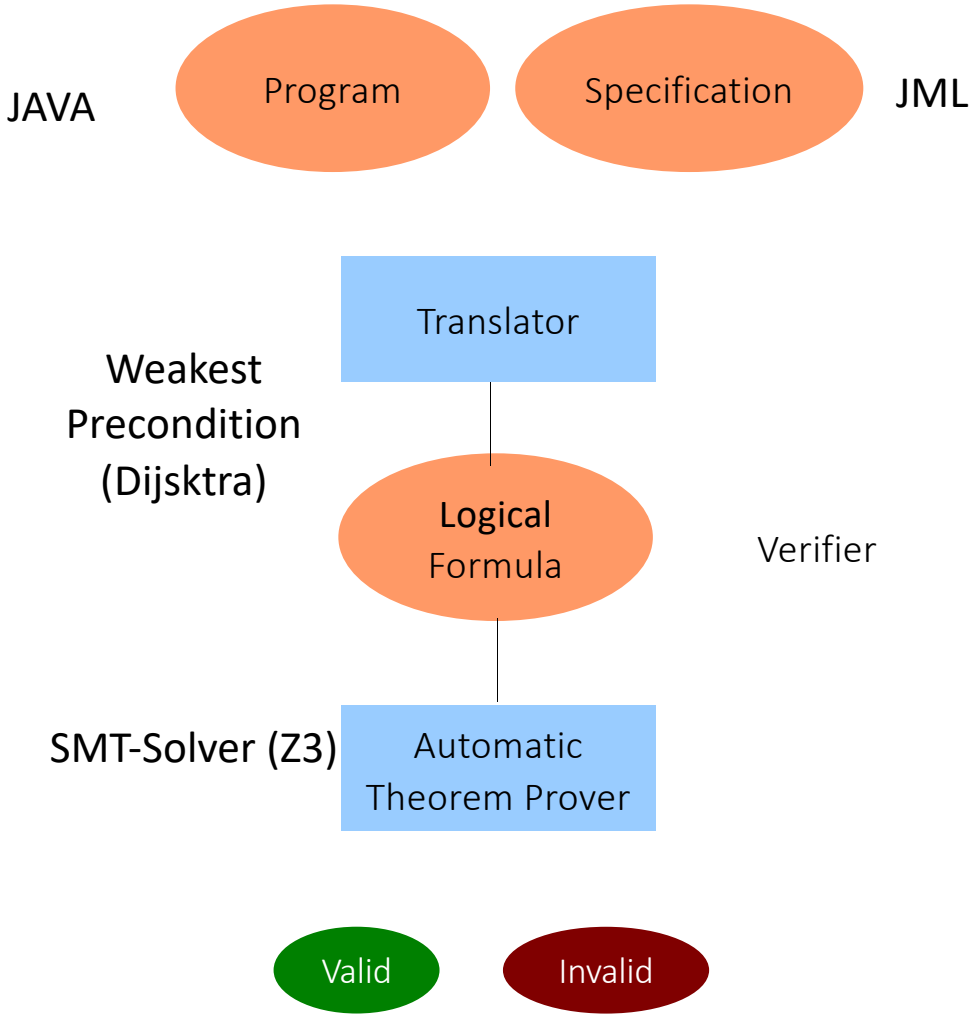
WP(`result=a`, P') = (a<0 =>-a=|a|) && (a>=0 => a=|a|)

What we need to prove: preM => WP(P, result=|a|)

true => (a<0 =>-a=|a|) && (a>=0 => a=|a|) ✓

Verifier Archtecture

- Programming Language
- Specification Language
- Logical representation of the program
- Automatic Decision Procedure



So, what is the problem with using WP?

1. Limitation in the **Logic** used in specifications: is **decidable**?

2. **Loops**!

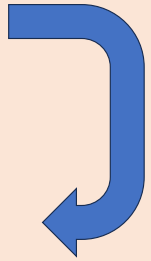
What is the problem with WP?

Let call H_k: precondition for executing exactly **k** times

```
while (B) {
      S;
}
{Q}
```

Execute 0 times

```
¬B
while (B) {
      S;
}
{Q}
```



What is the problem with WP?

H_k: precondition for executing exactly k times

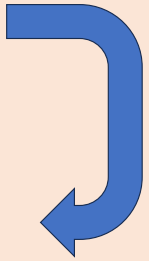
- $H_0(B,S,Q) = def(B) \wedge \neg B \wedge Q$

```
while (B) {
      S;
}
{Q}
```

Execute 0 times

$\{\neg B \wedge Q\}$

```
while (B) {
      S;
}
{Q}
```



What is the problem with WP?

H_k: precondition for executing exactly k times

- $H_0(B,S,Q) = def(B) \wedge \neg B \wedge Q$
- $H_1(B,S,Q) =_{def} def(B) \wedge B \wedge wp(S, \neg B \wedge Q))$
  $= def(B) \wedge B \wedge wp(S, H_0(Q))$.

Execute once

{B && ?}

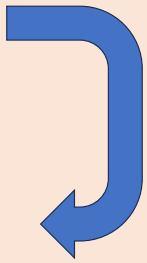
S;

while (B) {

S;

}

{Q}



{B ∧ wp(S, ¬B ∧ Q)}

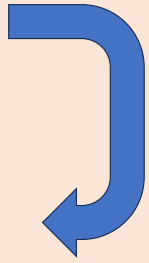
S;

while (B) {

S;

}

{Q}



What is the problem with WP?

H_k: precondition for executing exactly k times

- $H_0(B,S,Q)$ = def(B) ∧ ¬B ∧ Q,
- $H_1(B,S,Q) =_{def}$ def(B) ∧ B ∧ wp(S, ¬B ∧ Q))
  = def(B) ∧ B ∧ wp(S, $H_0(Q)$).
- ….
- $H_{k+1}(B,S,Q)$ ≡ def(B) ∧ B ∧ wp(S, $H_k(Q)$) for k ≥ 0.

$B \wedge wp(S, H_k(Q))$

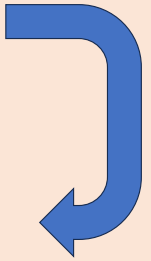
S; S;.. S;

while (B) {

S;

}

{Q}



What is the problem with WP?

H_k: precondition for executing exactly k times

- $H_0(B,S,Q)$ = def(B) ∧ ¬B ∧ Q,
- $H_1(B,S,Q) =_{def}$ def(B) ∧ B ∧ wp(S, ¬B ∧ Q))
  = def(B) ∧ B ∧ wp(S, $H_0(Q)$).
- ….
- $H_{k+1}(B,S,Q)$ ≡ def(B) ∧ B ∧ wp(S, $H_k(Q)$) for k ≥ 0.

$B \wedge wp(S, H_k(Q))$

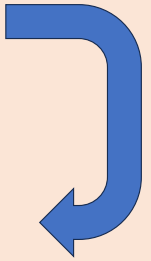
S; S;.. S;

while (B) {

S;

}

{Q}



WP(`while(B){S}`,Q) = $V_{i=0} H_i(B,S,Q)$

We get an infinitary formula…

Loop invariants

A **loop invariant** is a property of a program loop that is true **before** (and **after**) each **iteration**

- It's part of the (perhaps unconscious) reasoning we do when writing a loop
  - Formal Description:
    - What we **assume** at the beginning of each iteration
    - What **progress** we expect at the end of the iteration
- Fundamental for the verification of a method if it has loop/cycles (**inductive reasoning**)

```
{Pre}
Init;
{Pc}=>{ I }
while (Cond) {
      { I & Cond}
      S;
      { I }
}
{ I & !Cond}=> {Qc}
…
{Pos}
```

Loop invariant Theorem

```
… ;  // some code
{ Pc }
{ I }
 while (B) {
        { I & Cond}
        S;
        { I}
 }
 { I & !Cond}
 {Qc}
… ;  // some code
```

If you can show that:

1) Pc ==> I
2) {B && Inv} S {Inv}
3) (Inv && !B) => Qc

Then:

{Pc} while (B) S {Qc}

Loop invariants

```
int sumx(x: Int) {
//@   requires x >= 0;
//@   ensures result == \sum(i: int; 0<=i<=x; i)
```

```
int sumx(int x) {
   int s = 0, i = 0;
   while (i < x) {
      // state 1
      i = i + 1;
      s = s + i;
     // state 2
   }
   return s;
}
```

i@e1	s@e1	i@e2	s@e2
0	0	1	1
1	1	2	3
2	3	3	6
3	6	4	10

Loop invariant

```
s == \sum(j: int; 0<=j<=i; j)
&& 0 <= i <= x
```

Handling Loops

If you can show that:
1) Pc ==> I
2) {B && Inv} S {Inv}
3) (Inv && !B) => Qc

Then:
{Pc} while (B) S {Qc}

How we codify the rule for the loop?

- We require an **invariant** satisfying the invariant theorem
- We need to add this statements
  - **assume** E // believe E is true
  - **assert** E // check E is true (fail otherwise)
  - **havoc** x (a non-deterministic value to x)

WP (**assume** E, Q) == E=>Q
WP (**assert** E, Q) == E && Q
WP (**havoc** x, Q) == \forall x. Q

Handling Loops

If you can prove that:
1) Pc ==> I
2) **{B && Inv}** S **{Inv}**
3) (Inv && !B) => Qc
Then
{Pc} while (B) S {Qc}

While_(I,T) B do S **end** ==

assert I

havoc T

assume I

if (B) **then**

S

assert I

assume false

endif

1) Check that loop invariant holds at the beginning

2a)Forget info of variables affected T, use only the information provided by the invariant

2b) Check invariant is preserved after executing loop body

3) Here I holds and B is False

```
int sumx (int x) {
    int s = 0, i = 0;
    while (i < x) {
        i = i + 1;
        s = s + i;
    }
    return s;
}
```

While_(I,T) B do S **end** ==

assert I
havoc T
assume I
if (B) **then**
S
assert I
assume false
endif

```
int sumx (int x) {
  int s = 0, i = 0;
  assert 0 <= i <= x && s == \sum(j: int; 0<=j<=i; j)
  havoc i, sum
  assume 0 <= i <= x && s == \sum(j: int; 0<=j<=i; j)
  if (i<x) then
    i = i + 1;
    s = s + i;
    assert 0 <= i <= x && s == \sum(j: int; 0<=j<=i; j)
    assume false
  endif
  return s;
}
```

x >= 0 **==>** **WP(**sumx, result == \sum(i: int; 0<=i<=x; i)**)**

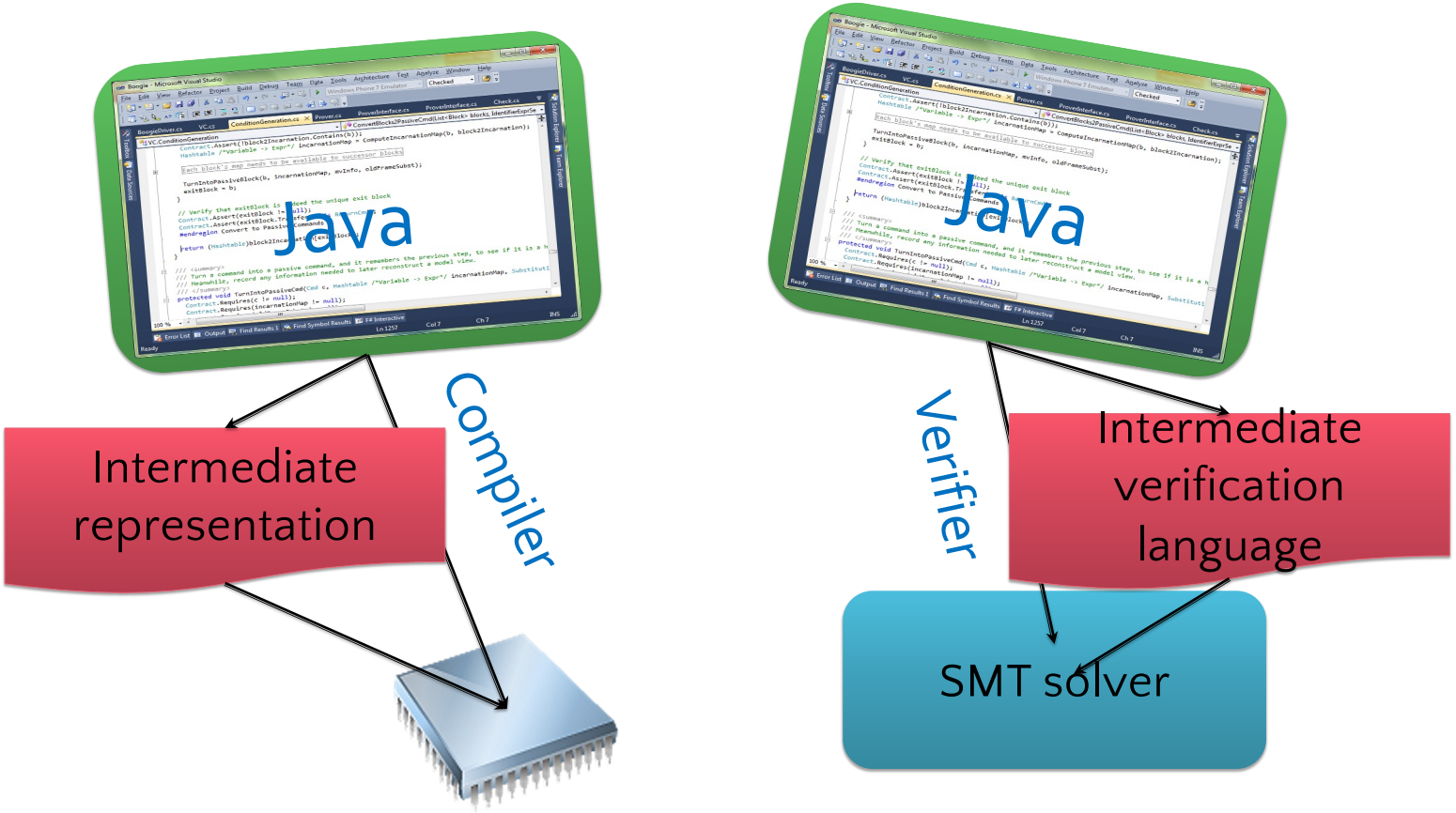
```
int sumx (int x) {
    int s = 0, i = 0;
    while (i < x) {
        i = i + 1;
        s = s + i;
    }
    return s;
}
```

Are we still computing the weakest precondition on the original program?

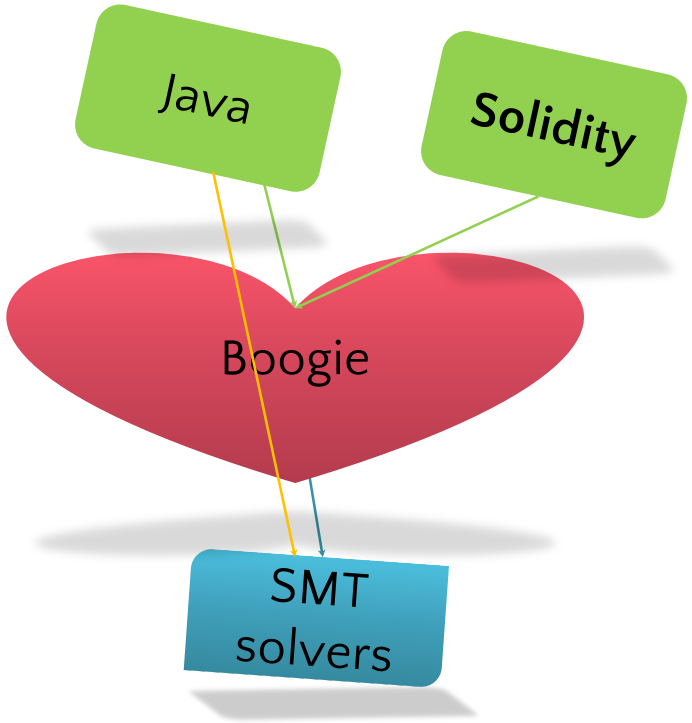
```
int sumx (int x) {
  int s = 0, i = 0;
  assert 0 <= i <= x && s == \sum(j: int; 0<=j<=i; j)
  havoc i, sum
  assume 0 <= i <= x && s == \sum(j: int; 0<=j<=i; j)
  if (i<x) then
    i = i + 1;
    s = s + i;
    assert 0 <= i <= x && s == \sum(j: int; 0<=j<=i; j)
    assume false
  endif
  return s;
}
```

x >= 0 **==>** **WP(**`sumx`, result == \sum(i: int; 0<=i<=x; i)**)**

Separation of concerns

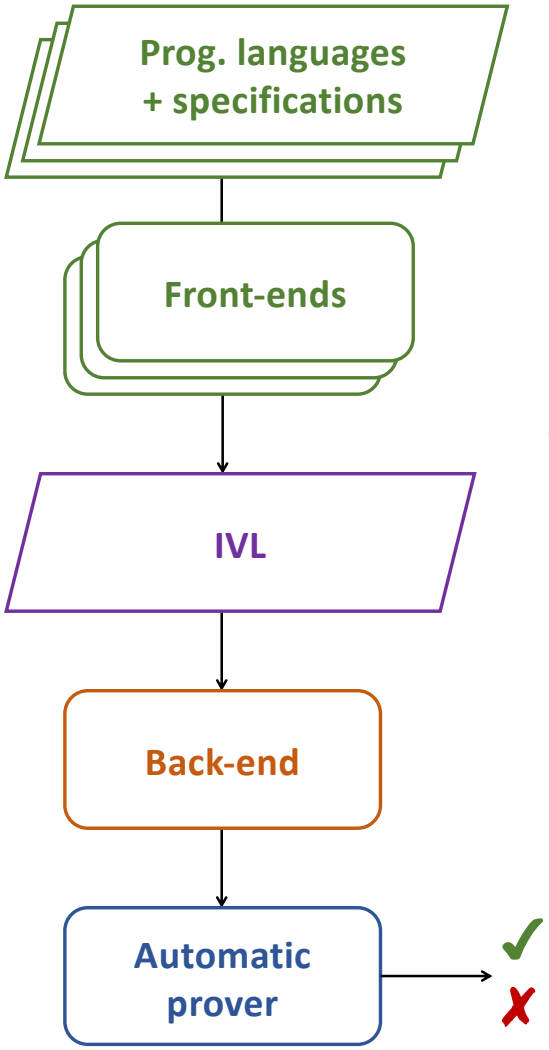


Verification architecture



Intermediate
Verification
Language

Modern software verification tools



+ Automatic first-order logic tools – major progress in the last decade (SAT, SMT)

+ Intermediate verification languages - Boogie, Why, …

+ Back-end: verifier (verification condition generator)

= **Common infrastructure** for building front-end verifiers

PROGRAM VERIFICATION

$\forall s, s' \cdot pre(s) \wedge P(s, s') \Rightarrow post(s')$
$\forall s \cdot pre(s) \Rightarrow P(s)$ terminates

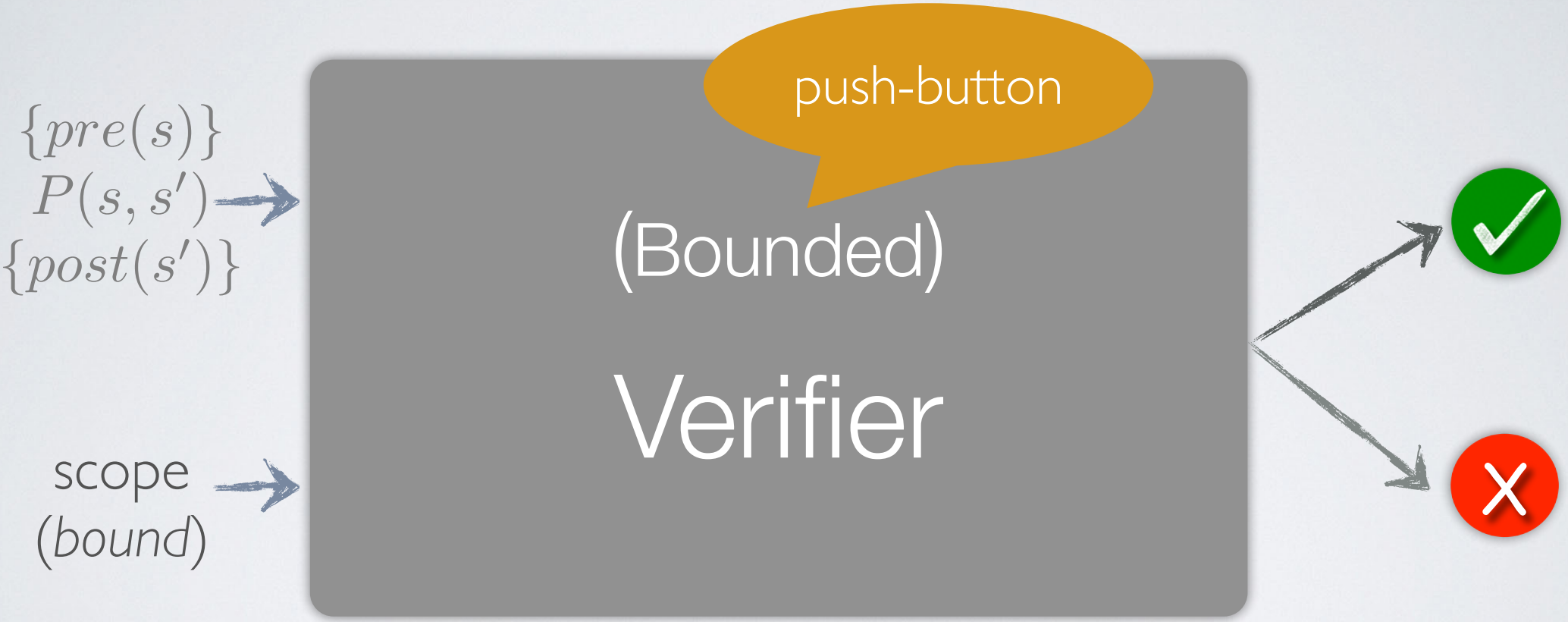
BOUNDED PROGRAM VERIFICAT

limit on the number of objects in program states, the range for numeric types, …

limit on the number of loop iterations and depth of recursive calls

$$\forall s_B, s'_B \cdot pre(s_B) \wedge P_B(s_B, s'_B) \Rightarrow post(s'_B)$$
$$\forall s_B \cdot P \text{ terminates within } B \text{ iterations}$$

BOUNDED VERIFICATION



THE BOOLEAN SATISFIABILITY PROBLEM

Given a **propositional** formula *A*, is it *satisfiable*?

i.e., find a valuation:

$$v : Vars(A) \rightarrow \{\mathtt{T}, \mathtt{F}\}$$

that makes the formula true

- **Decidable** problem
- No known polynomial time algorithm to solve it (problem is **NP-complete**)
- Several **efficient implementations for SAT** based on Davis-Putnam-Logemann-Loveland (DPLL)

BASIC ALGORITHM FOR SAT

Given a **propositional** formula *F*, try out all possible valuations for F in the search a satisfying one.

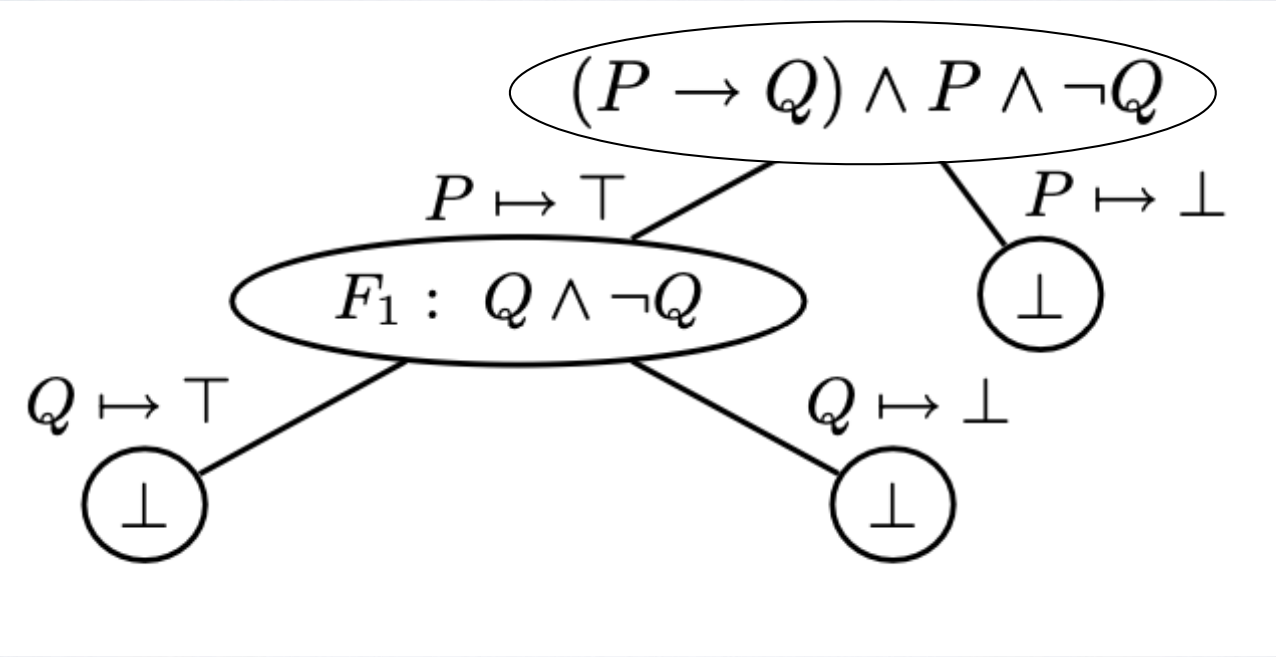
- Valuations are finite (but exponential on the number of propositional variables)
- This process is the so-called **truth-table construction**

$$F: \; P \wedge Q \; \rightarrow \; P \vee \neg Q$$

P	Q	$P \wedge Q$	$\neg Q$	$P \vee \neg Q$	F
0	0	0	1	1	1
0	1	0	0	0	1
1	0	0	1	1	1
1	1	1	0	1	1

BASIC ALGORITHM FOR SAT

The same process can be implemented without the explicit construction of a table:



SAT ALGORITHMIC IMPROVEMENTS

$$(P \vee Q) \wedge (\neg P \vee R) \wedge (\neg R \vee S) \wedge P$$

P appears positively, clause can be removed

P does not appear in the clause, no change in it

Identify literal clause

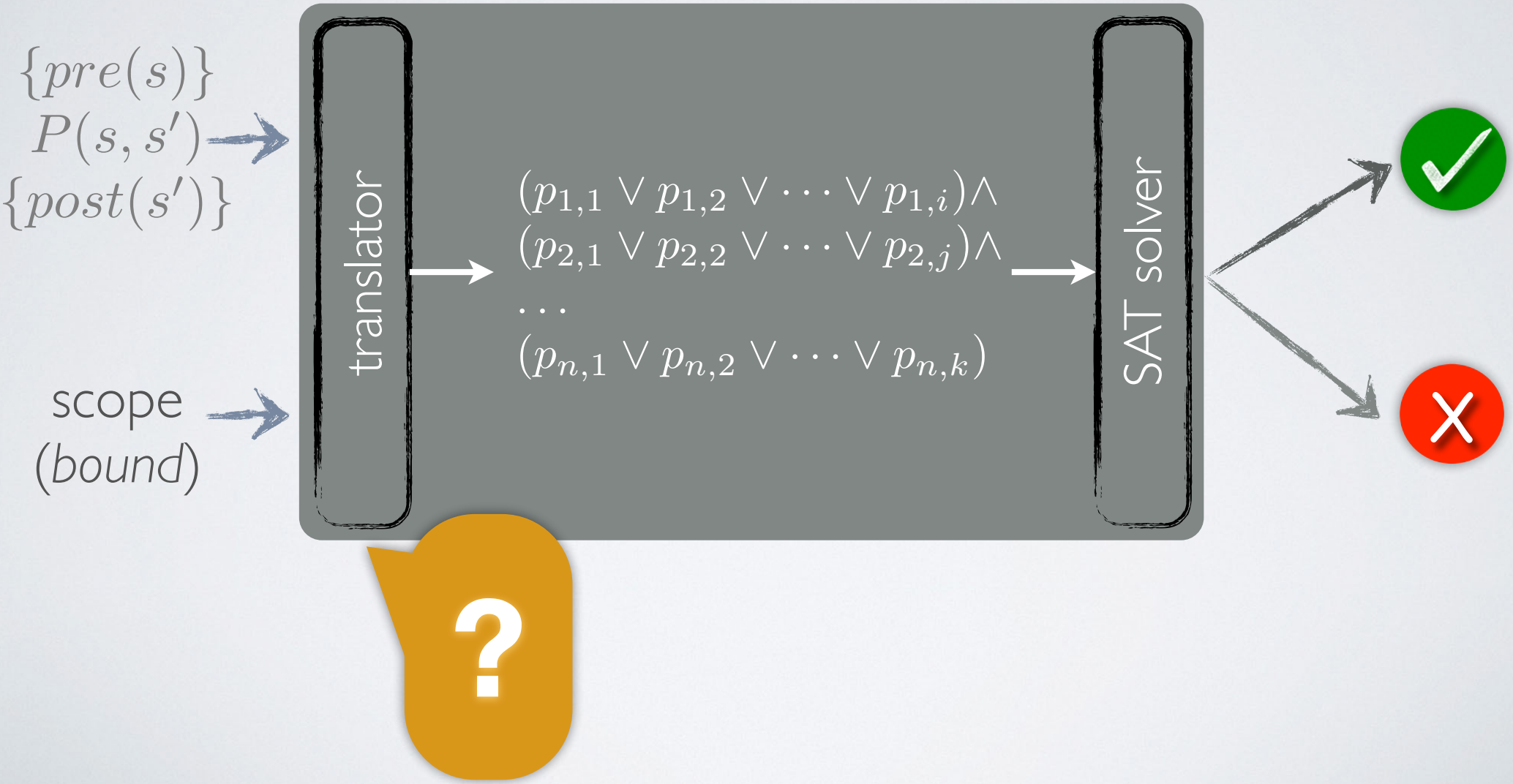
"removing" a clause is turning it into $\top$,
"removing" a literal is turning it into $\bot$

$$\top \wedge (\bot \vee R) \wedge (\neg R \vee S) \wedge \top$$

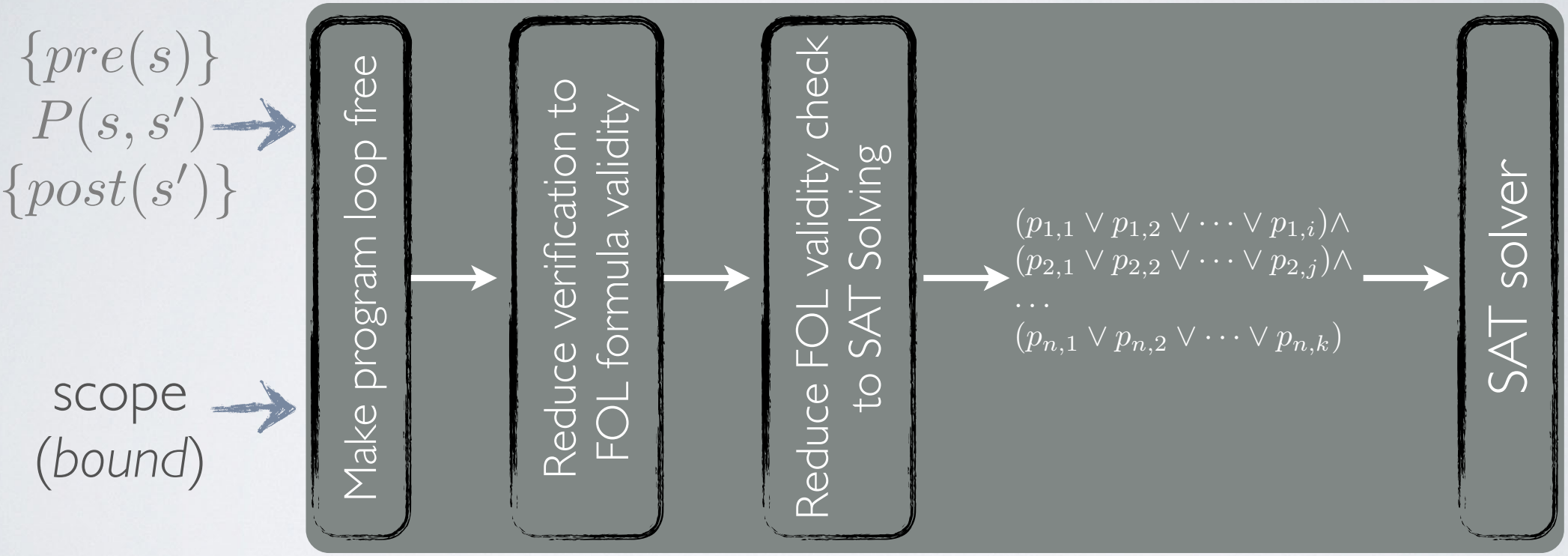
That is:

$$R \wedge (\neg R \vee S)$$

SAT-BASED BOUNDED VERIFICATION



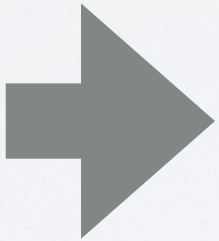
BOUNDED VERIFICATION SAT ENCODING



PROGRAM VERIFICATION ENCODING

A program **Prog with iteration bounded by k can be transformed into an equivalent loop-free program Prog'** by unrolling loops k times:

```
while (cond) {
      loopBody();
}
```



```
if (cond) {
      loopBody();
      if (cond) {
            loopBody();
            if (cond) {
                  loopBody();
                  if (cond) {
                        loopBody();
                        If (cond) {
                              ignoreRun();
                        }
                  }
            }
      }
}
```

PROGRAM VERIFICATION ENCODING

If a program P is **loop-free**, the **validity of a partial correctness assertion involving P can be mechanically reduced to checking the validity of a logical formula**:

$$\{pre(s)\}$$
$$P(s, s')$$
$$\{post(s')\}$$

$$\forall s \cdot pre(s) \Rightarrow \mathsf{WP}(P, post(s))$$

Formula is first-order

Dijkstra, Edsger W. (August 1975). "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *Comm. ACM*. **18** (8).

PROPOSITIONAL ENCODING OF FIRST-ORDER LOGIC OVER BOUNDED DOMAINS

A first-order logic formula involving quantification over a domain of finite size k can be turned into an equivalent propositional formula:

$$\forall s \in D \cdot p(s) \equiv p(d_0) \wedge p(d_1) \wedge \cdots \wedge p(d_{k-1})$$

LIMITATIONS

{ arr != null }

```
public static int linearSearch(int[] arr, int target) {
        // Iterate through each element of the array
        for (int i = 0; i < arr.length - 1; i++) {
            // If the current element matches the target, return its index
            if (arr[i] == target) {
                return i;
            }
        }
        // If the loop completes without finding the target, return -1
        return -1;
}
```

$$\{\text{result} = -1 \Leftrightarrow (\forall i \cdot 0 \leq i \leq arr.length \Rightarrow arr[i] \neq target)\}$$

Bug won't be found if loops are unrolled less then array length

LIMITATIONS

{ a != null }

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

$$\{\text{result} = -1 \Leftrightarrow (\forall i \cdot 0 \leq i \leq a.length \Rightarrow a[i] \neq key)\}$$

Bug won't be found if only small sized arrays are considered

CHALLENGES OF PROGRAM VERIFICATION

Efficiency and scalability

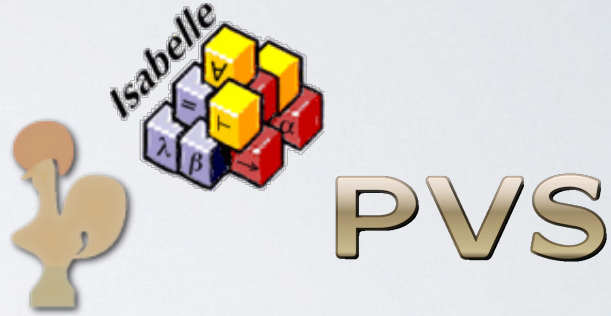
- SAT formula grows exponentially with program complexity and scope/bound

Generalizability

- Custom heap-allocated data is non-trivial to deal with
- Concurrent programs lead to huge number of program interleavings
  - Extremely difficult to deal with for bounded verifiers (scalability)

SOME RESEARCH DIRECTIONS

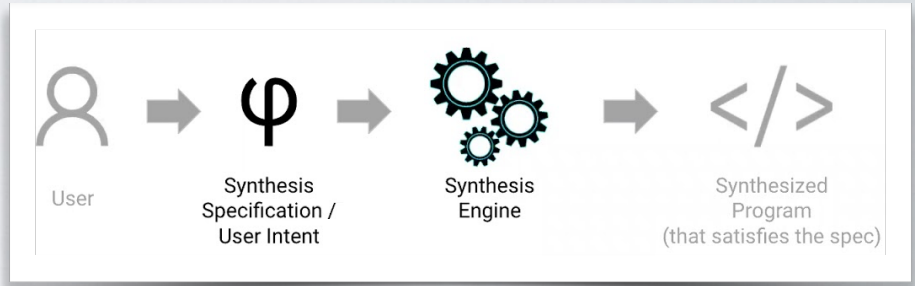
- Development of proof assistants for deductive verification
  - Very valuable for mathematicians!



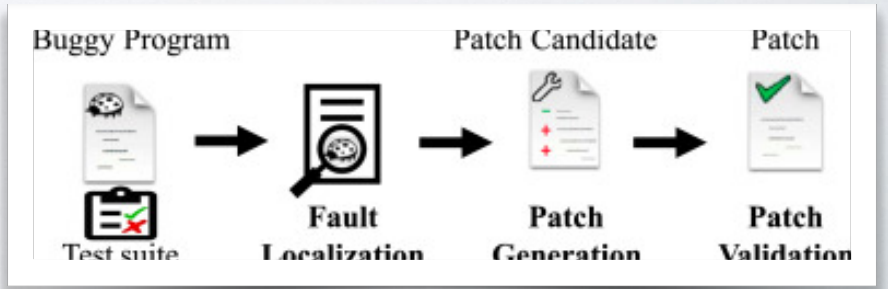
- Design of specification languages (logic engineering)
  - Tradeoff between expressiveness and tractability
  - Target specific program properties

SOME RESEARCH DIRECTIONS

- Program derivation/synthesis
  - Given a specification, systematically generate a program that satisfies the specification
- Fault localization
  - Given a faulty program, where's the error?
- Program repair
  - Given a faulty program, automatically fix it so that it satisfies its specification




```
525    if(! Model.fixed_id) {
526        fixed_id = ((u32)(radio_ch[0] ^ cyrfmfg_id[0] ^ cyrfmfg_id[3]) << 16)
527                 | ((u32)(radio_ch[1] ^ cyrfmfg_id[1] ^ cyrfmfg_id[4]) << 8)
528                 | ((u32)(radio_ch[2] ^ cyrfmfg_id[2] ^ cyrfmfg_id[5]) << 0);
529        fixed_id = fixed_id % 1000000;
530        bind_counter = BIND_COUNT;
531        state = DEVO_BIND;
532        PROTOCOL_SetBindState(0x1388 * 2400 / 1000); //msecs
533    } else {
534        fixed_id = Model.fixed_id;
535        use_fixed_id = 0;
536        state = DEVO_BOUND_1;
537        bind_counter = 0;
538        cyrf_set_bound_sop_code();
539    }
```



Is software verification also relevant for modern software systems?

THANK YOU!