CS 350 Algorithms and Complexity

Winter 2019

Lecture 16: Proving Programs Correct

Andrew P. Black
Department of Computer Science
Portland State University

How to Prove a Program Correct

1. Decide what programming language statements mean.
   - Some sort of formal semantics
2. Decide what "correct" means.
   - This means writing a specification of the problem
3. Write the program with the proof in mind
4. Check the proof mechanically.

Correctness

Correctness

- Original characterization:
  - Given a specification (in a formal logic) and
  - an implementation (in a programming language),
  - *prove* that the implementation satisfies the specification

Correctness

- Original characterization:
  - Given a specification (in a formal logic) and
  - an implementation (in a programming language),
  - *prove* that the implementation satisfies the specification
- Maybe possible
  - … but very, very hard

Correctness

Correctness

- Modern characterization
  - Given a specification
  - *construct* a program that satisfies the specification

Correctness

- Modern characterization
  - Given a specification
  - *construct* a program that satisfies the specification
- Program and proof are written together

Correctness

- Modern characterization
  - Given a specification
  - *construct* a program that satisfies the specification
- Program and proof are written together
  - not necessarily easy,

Correctness

- Modern characterization
  - Given a specification
  - *construct* a program that satisfies the specification
- Program and proof are written together
  - not necessarily easy,
  - but much easier!

Example

```
{ true }
var x: int;
read(x);
{ x: Integer }
if (x mod 2 = 1) then
    { odd(x) }
    x := x + 1;
fi
{ even(x) }
```

Example

```
{ true }
var x: int;
read(x);
{ x: Integer }
if (x mod 2 = 1) then
    { odd(x) }
    x := x + 1;
fi
{ even(x) }
```

Note: Modern notation is different from that of Hoare's *Axiomatic Basis*:
the predicates are in braces, not program statements.

"An Axiomatic Basis ..."

- Classic paper shows how to define a language, specify a problem, & design a program with the proof in mind; foreshadowed mechanical proof.
- He won the Turing Award for this work
  - basic ideas have not changed.

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

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:

... the result of an overflowing

Notation

- The two most common notations are Hoare triples and Dijkstra weakest preconditions (or predicate transformers)
- I'll focus primarily on Hoare triples
- A Hoare triple is a logical predicate:

  {P} S {Q}

  – P and Q are predicates, S is a program

  - {P} S {Q} is true when
    – *if* P is true initially, *then* after S terminates, Q will be true

Notation

- The two most common notations are Hoare triples and Dijkstra weakest preconditions (or predicate transformers)
- I'll focus primarily on Hoare triples
- A Hoare triple is a logical predicate:

  {P} S {Q}

  - P and Q are predicates, S is a program
  - {P} S {Q} is true when
    - *if* P is true initially, *then* after S terminates, Q will be true

Note: the notation in "An Axiomatic Basis..." is different

True or False?

True or False?

Assume that you know what assignment does!

1. $\{true\}\ y{:=}x{*}x\ \{y \geq 0\}$

True or False?

Assume that you know what assignment does!

1. {true} y:=x*x {y≥0}
2. {true} y:=x*x {y<0}

True or False?

Assume that you know what assignment does!

1. $\{true\}\ y:=x*x\ \{y \geq 0\}$
2. $\{true\}\ y:=x*x\ \{y < 0\}$
3. $\{x>0\}\ x:=x+1\ \{x>1\}$

True or False?

Assume that you know what assignment does!

1. {true} y:=x*x {y≥0}
2. {true} y:=x*x {y<0}
3. {x>0} x:=x+1 {x>1}
4. {x>1} x:=x+1 {x>0}

True or False?

Assume that you know what assignment does!

1. {true} y:=x*x {y≥0}
2. {true} y:=x*x {y<0}
3. {x>0} x:=x+1 {x>1}
4. {x>1} x:=x+1 {x>0}
5. {x=X ∧ y=Y} t := x; x := y; y := t {x=Y∧y=X}

True or False?

Assume that you know what assignment does!

1. {true} y:=x*x {y≥0}
2. {true} y:=x*x {y<0}
3. {x>0} x:=x+1 {x>1}
4. {x>1} x:=x+1 {x>0}
5. {x=X ∧ y=Y} t := x; x := y; y := t {x=Y∧y=X}
6. {x=X ∧ y=Y} t := x; x := y; y := t {x=Y∧t=Y}

True or False?

Assume that you know what assignment does!

1. {true} y:=x*x {y≥0}
2. {true} y:=x*x {y<0}
3. {x>0} x:=x+1 {x>1}
4. {x>1} x:=x+1 {x>0}
5. {x=X ∧ y=Y} t := x; x := y; y := t {x=Y∧y=X}
6. {x=X ∧ y=Y} t := x; x := y; y := t {x=Y∧t=Y}
7. {x≠0} **if** x > 0 **then** skip **else** x:= -x {x>0}

But what *does* assignment do?

- We can also use axioms to *define* what programming language statements do
  - the meaning of a programming language can be given by axioms!
  - big advantage: axioms let us leave some details up to the implementation

Landmark Paper:

Acta Informatica 2, 335—355 (1973)
© by Springer-Verlag 1973

An Axiomatic Definition of the Programming Language PASCAL

C. A. R. Hoare and N. Wirth

Received December 11, 1972

Summary. The axiomatic definition method proposed in reference [5] is extended and applied to define the meaning of the programming language PASCAL [1]. The whole language is covered with the exception of real arithmetic and go to statements.

Introduction

The programming language PASCAL was designed as a general purpose language efficiently implementable on many computers and sufficiently flexible to be able to serve in many areas of application. Its defining report [1] was given in the style of the ALGOL 60 report [2]. A formalism was used to define

Data Types: Enumerated types

type $T = (c_1, c_2, \ldots, c_n)$

1. $c_1, c_2, \ldots c_n$ are distinct elements of T
2. These are the only elements of T
3. $c_{i+1} = succ(c_i)$ for $i = 1 \ldots n-1$
4. $pred(succ(u)) = u$, $succ(pred(v)) = v$
5. $\neg (v < v)$
6. $(u < v) \wedge (v \leq w) \supset u < w$
7. $v = succ(u) \vee u = pred(v) \supset u < v$
8. $(u > v) \equiv (v < u)$
9. $(u \leq v) \equiv \neg (u > v)$
10. $(u \geq v) \equiv \neg (u < v)$
11. $(u \neq v) \equiv \neg (u = v)$

We define $min_T = c_1$ and $max_T = c_n$ (not available to the Pascal Programmer)

Declarations

- D is a sequence of declarations
- S is a compound statement
- then D; S is a block and obeys the following rule of inference

$$\frac{H \vdash \{P\}\ S\ \{Q\}}{\{P\}\ D;\ S\ \{Q\}}$$

where H is the set of assertions describing the properties established by the declarations in D

Statements

- Assignment

$$\{P[exp/x]\}\ \ x := exp\ \ \{P\}$$

- Compound Statements

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\},\quad \text{for } i = 1\ldots n}{\{P_0\}\ \textbf{begin}\ S_1;\ S_2;\ \ldots\ S_n\ \textbf{end}\ \{P_n\}}$$

- If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P\wedge B\supset Q_1,\ P\wedge\neg B\supset Q_2}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{R\}}$$

- While Statements

$$\frac{\{Q\wedge B\}S\{Q\}}{\{Q\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q\wedge\neg B\}}$$

Statements

Also written P^x_{exp}

- Assignment

$$\{P[exp/x]\}\ \ x := exp\ \ \{P\}$$

- Compound Statements

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\},\quad \text{for } i = 1\ldots n}{\{P_0\}\ \textbf{begin}\ S_1;\ S_2;\ \ldots\ S_n\ \textbf{end}\ \{P_n\}}$$

- If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P\wedge B\supset Q_1,\ P\wedge\neg B\supset Q_2}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{R\}}$$

- While Statements

$$\frac{\{Q\wedge B\}S\{Q\}}{\{Q\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q\wedge\neg B\}}$$

Rule of Assignment

$$\{P^{x}_{exp}\}\ x := exp\ \{P\}$$

$$\{P[exp/x]\}\ x := exp\ \{P\}$$

- Suppose that you want P to hold *after* the execution of some assignment to x:
  - What has to be true before? Consider two cases:
    - if P doesn't mention x (e.g., P ≡ w = y), then P better also hold *before* the assignment to x
    - if P mentions x (e.g., P ≡ even(x)), then P better hold with x replaced by exp (P[exp/x] ≡ even(exp))

Assignment Examples

rule: $\{P[exp/x]\}\ x := exp\ \{P\}$

Fill in the blank:

A. true
B. false
C. $y = 1$
D. $1 = 1$
E. None of the above

$$\{ \qquad \}\ x := 1\ \{ x = 1 \}$$

Assignment Examples

rule: {P[exp/x]} x := exp {P}

Fill in the blank:

A. true
B. false
C. $y = 7$
D. $x + 1 = 7$
E. $x = 6$
F. None of the above

{   } x := x + 1 { x = 7 }

Assignment Examples

rule: {P[exp/x]} x := exp {P}

Fill in the blank:

A. true
B. false
C. y = 1
D. x = 7
E. None of the above

{   } y := 4 { x = 7 }

Assignment Examples

rule: $\{P[exp/x]\}$ x := exp $\{P\}$

Fill in the blank:

A. x := true
B. x := 7
C. y := 1
D. 1 = 1
E. None of the above

{ true }   {x = 7}

Rule for Compound Statements

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\}, \quad \text{for } i = 1 \ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots\ S_n\ \{P_n\}}$$

- Basic idea:
  - The postcondition of each statement in the sequence has to achieve the precondition for the next
- Recall Hoare's rules of Consequences:
  - if $\{P\}\ Q\ \{R\}$ and $S \supset P$ then $\{S\}\ Q\ \{R\}$
  - if $\{P\}\ Q\ \{R\}$ and $R \supset S$ then $\{P\}\ Q\ \{S\}$

Example

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\}, \quad \text{for } i = 1 \ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots\ S_n\ \{P_n\}}$$

for n = 2:

$$\frac{\{P_0\}\ S_1\ \{P_1\}, \quad \{P_1\}\ S_2\ \{P_2\}}{\{P_0\}\ S_1;\ S_2\ \{P_2\}}$$

Suppose that $P_2 \equiv x \geq m \wedge y > n$. $P_0 \equiv \text{true}$.

(m and n are constants.) What are S_1 and S_2 ?

{ true } $\qquad$ { $x \geq m \wedge y > n$ }

Example

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\}, \quad \text{for } i = 1 \ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots\ S_n\ \{P_n\}}$$

for n = 2:

$$\frac{\{P_0\}\ S_1\ \{P_1\}, \quad \{P_1\}\ S_2\ \{P_2\}}{\{P_0\}\ S_1;\ S_2\ \{P_2\}}$$

Suppose that $P_2 \equiv x \geq m \wedge y > n$. $P_0 \equiv \text{true}$.

(m and n are constants.) What are S_1 and S_2 ?

{ true } *y := n+1* $\{ x \geq m \wedge y > n \}$

Example

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\}, \quad \text{for } i = 1\ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots\ S_n\ \{P_n\}}$$

for $n = 2$:

$$\frac{\{P_0\}\ S_1\ \{P_1\}, \quad \{P_1\}\ S_2\ \{P_2\}}{\{P_0\}\ S_1;\ S_2\ \{P_2\}}$$

Suppose that $P_2 \equiv x \geq m \land y > n$. $P_0 \equiv$ true.

(m and n are constants.) What are S_1 and S_2 ?

{ true } $\{ x \geq m \land n+1 > n\}$ *$y := n+1$* $\{ x \geq m \land y > n\}$

Example

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\}, \quad \text{for } i = 1 \ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots\ S_n\ \{P_n\}}$$

for n = 2:

$$\frac{\{P_0\}\ S_1\ \{P_1\}, \quad \{P_1\}\ S_2\ \{P_2\}}{\{P_0\}\ S_1;\ S_2\ \{P_2\}}$$

Suppose that $P_2 \equiv x \geq m \wedge y > n$. $P_0 \equiv$ true.

(m and n are constants.) What are S_1 and S_2 ?

{ true $\qquad$ $\{ x \geq m \wedge \text{true} \}$ *y := n+1* $\{ x \geq m \wedge y > n \}$

Example

$$\frac{\{P_{i-1}\}\ S_i\ \{P_i\},\quad \text{for } i = 1\ldots n}{\{P_0\}\ S_1;\ S_2;\ \ldots\ S_n\ \{P_n\}}$$

for n = 2:

$$\frac{\{P_0\}\ S_1\ \{P_1\},\quad \{P_1\}\ S_2\ \{P_2\}}{\{P_0\}\ S_1;\ S_2\ \{P_2\}}$$

Suppose that $P_2 \equiv x \geq m \wedge y > n$. $P_0 \equiv$ true.

(m and n are constants.) What are S_1 and S_2 ?

{ true } *x := m;* { $x \geq m \wedge$ true } *y := n+1* { $x \geq m \wedge y > n$}

If Statements

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P \wedge B \supset Q_1,\ P \wedge \neg B \supset Q_2}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{R\}}$$

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P \wedge B \supset Q_1,\ P \wedge \neg B \supset Q_2}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{R\}}$$

- Basic idea:

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P \wedge B \supset Q_1,\ P \wedge \neg B \supset Q_2}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{R\}}$$

- Basic idea:
  - What do you know when you execute S_1 ?

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P \wedge B \supset Q_1,\ P \wedge \neg B \supset Q_2}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{R\}}$$

- Basic idea:
  - What do you know when you execute S_1 ?
    - P

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P\wedge B\supset Q_1,\ P\wedge\neg B\supset Q_2}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{R\}}$$

- Basic idea:
  - What do you know when you execute S_1 ?
    - P
    - B

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P \wedge B \supset Q_1,\ P \wedge \neg B \supset Q_2}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{R\}}$$

- Basic idea:
  - What do you know when you execute S_1 ?
    - P
    - B
  - Life is good if, taken together, $P \wedge B$ implies the precondition of Q_1

If Statements

$$\frac{\{Q_1\}S_1\{R\},\ \{Q_2\}S_2\{R\},\ P\wedge B\supset Q_1,\ P\wedge\neg B\supset Q_2}{\{P\}\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\{R\}}$$

- Basic idea:
  - What do you know when you execute S_1 ?
    - P
    - B
  - Life is good if, taken together, $P\wedge B$ implies the precondition of Q_1
  - If you want R to be true after the **if**, then R better be true after *both* S_1 and S_2

Example

$\{ m > 0 \wedge n > 0 \wedge n \neq m \wedge n + m = s \}$

if $m > n$ **then**

$m := m - n$

else

$n := n - m$

$\{ m > 0 \wedge n > 0 \wedge n + m < s \}$

Example

$\{ m > 0 \land n > 0 \land n \neq m \land n + m = s \}$

if m > n **then**

    m := m – n

  else

    n := n – m

$\{ m > 0 \land n > 0 \land n + m < s \}$

Let's prove it!

Example

$\{ m > 0 \wedge n > 0 \wedge n \neq m \wedge n + m = s \}$

if $m > n$ **then**

$m := m - n$

else

$n := n - m$

$\{ m > 0 \wedge n > 0 \wedge n + m < s \}$

Example

$\{ m > 0 \wedge n > 0 \wedge n \neq m \wedge n + m = s \}$

if $m > n$ **then**

$m := m - n$

else $\{ (m > 0 \wedge n > 0 \wedge n + m < s)[(n - m)/n] \}$

$n := n - m$

$\{ m > 0 \wedge n > 0 \wedge n + m < s \}$

Example

$\{ m > 0 \land n > 0 \land n \neq m \land n + m = s \}$

if m > n **then**

$m := m - n$

else

$n := n - m$

$\{ m > 0 \land n > 0 \land n + m < s \}$

Example

$\{ m > 0 \land n > 0 \land n \neq m \land n + m = s \}$

if $m > n$ **then**

$m := m - n$

else $\{ m > 0 \land n - m > 0 \land (n - m) + m < s \}$

$n := n - m$

$\{ m > 0 \land n > 0 \land n + m < s \}$

Example

$\{ m > 0 \land n > 0 \land n \neq m \land n + m = s \}$

if m > n **then**

$m := m - n$

else

$n := n - m$

$\{ m > 0 \land n > 0 \land n + m < s \}$

Example

$\{ m > 0 \land n > 0 \land n \neq m \land n + m = s \}$

if m > n **then**

m := m – n

else

n := n – m

$\{ m > 0 \land n > 0 \land n + m < s \}$

Example

$\{ m > 0 \wedge n > 0 \wedge n \neq m \wedge n + m = s \}$

if $m > n$ **then**

$m := m - n$

else $\{ m > 0 \wedge n > m \wedge n < s \}$

$n := n - m$

$\{ m > 0 \wedge n > 0 \wedge n + m < s \}$

Example

$\{ m > 0 \wedge n > 0 \wedge n \neq m \wedge n + m = s \}$

if m > n **then**

$\{ (m > 0 \wedge n > 0 \wedge n + m < s)[(m - n)/m] \}$

m := m – n

else $\{ m > 0 \wedge n > m \wedge n < s \}$

n := n – m

$\{ m > 0 \wedge n > 0 \wedge n + m < s \}$

Example

$\{ m > 0 \ \wedge \ n > 0 \ \wedge \ n \neq m \ \wedge \ n + m = s \}$

if m > n **then**

$\{ (m - n) > 0 \wedge n > 0 \wedge n + (m - n) < s) \}$

m := m – n

else $\{ m > 0 \wedge n > m \wedge n < s \}$

n := n – m

$\{ m > 0 \ \wedge \ n > 0 \ \wedge \ n + m < s \}$

Example

$\{ m > 0 \ \wedge \ n > 0 \ \wedge \ n \neq m \ \wedge \ n + m = s \}$

if $m > n$ **then**

$\{ m > n \wedge n > 0 \wedge m < s) \}$

$m := m - n$

else $\{ m > 0 \wedge n > m \wedge n < s \}$

$n := n - m$

$\{ m > 0 \ \wedge \ n > 0 \ \wedge \ n + m < s \}$

While Statements

$$\frac{\{Q \wedge B\}\ S\ \{Q\}}{\{Q\}\ \textbf{while}\ B\ \textbf{do}\ S\{Q \wedge \neg B\}}$$

- This is the hard part!
- Q is called a "loop invariant":
  - it's true before the loop starts
  - it's true when the loop ends
  - we say: Q is "maintained" by the loop
    - typically, Q is invalidated by the initial code in S, and then it is restored by later code.

While Statements

$$\frac{\{Q \wedge B\}S\{Q\}}{\{Q\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q \wedge \neg B\}}$$

- Also often written:

$$\frac{P \supset J \quad \{J \wedge B\}S\{J\} \quad (J \wedge \neg B) \supset Q}{\{P\}\ \text{while}\ B\ \text{do}\ S\ \{Q\}}$$

Example

```
{ true }
s := 0;
n := 0;
while n ≠ a.Length do
    s := s + a[n];  n := n + 1;
od;
{ s = a[0] + a[1] + … + a[a.Length-1] }
```

- What's the invariant?

Example

```
{ true }
s := 0;
n := 0;
while n ≠ a.Length do
    s := s + a[n];  n := n + 1;
od;
{ s = a[0] + a[1] + … + a[a.Length-1] }
```

- What's the invariant?

  s =

Example

```
{ true }
s := 0;
n := 0;
while n ≠ a.Length do
    s := s + a[n];  n := n + 1;
od;
{ s = a[0] + a[1] + … + a[a.Length-1] }
```

- What's the invariant?

  s = a[0] +

Example

```
{ true }
s := 0;
n := 0;
while n ≠ a.Length do
    s := s + a[n];  n := n + 1;
od;
{ s = a[0] + a[1] + … + a[a.Length-1] }
```

- What's the invariant?

  $s = a[0] + a[1] + \ldots + a[n-1]$

Example

```
{ true }
s := 0;
n := 0;
while n ≠ a.Length do
   s := s + a[n];  n := n + 1;
od;
{ s = a[0] + a[1] + … + a[a.Length-1] }
```

- What's the invariant?

  s = a[0] + a[1] + … + a[n-1] $= \sum_{i=0}^{n-1} a[i]$

Loop invariants

- Every loop that you write has an invariant!
  - Figuring out what the invariant is is a major part of designing loop program.
  - Most specification tools can't guess your invariant—they make you write it down explicitly
  - Invariants are also a big help in informal correctness arguments.
- Every loop will also have a *variant*, **if** it can be guaranteed to terminate

Loop Invariants in Spec

- Video
  - http://channel9.msdn.com/Blogs/Peli/The-Verification-Corner-Loop-Invariants (22 mins)
  - more: http://channel9.msdn.com/search?term=verification+corner
- Spec# in your browser
  - http://rise4fun.com/specsharp/

Procedures

- Non-recursive procedures can be handled by substitution:
  - if **proc** p(a) = S and {P} S {Q}
  - then {P[e/a]} p(e) {Q[e/a]}
- What about recursive procedures?
  - programmer must specify the pre-and post-conditions explicitly
  - This is good practice anyway (*e.g.*, every algorithm in Levitin)

Binary Search Example

```
int BinarySearch(int[ ] a, int key)
  requires forall{int i in (0..a.Length-1),
                  int j in (i..a.Length-1); a[i] <= a[j] };
  ensures 0 <= result ==> a[result] == key;
  ensures result < 0 ==>
                 forall{int i in (0..a.Length-1); a[i] ≠ key};
{…}
```

Program Proofs in Real Life

- SLAM
  - Automated safety checks for drives in MS Windows
  - Drastically reduced the incidence of blue-screens of death
  - Part of the SDK

Program Proofs in Paris Metro

HOME PRODUCTS SERVICES COMPANY

- About
- Executive Team
- Employment
- Events
- **Press**
- Client List
- Downloads
- Partner Programs
- Contact

Press Releases

November 16, 2010

Paris Metro replaces testing of interlocking and CBTC systems with formal safety verification using Prover Certifier

Stockholm, Sweden and Toulouse, France – November 16, 2010. Prover Technology today announced that its software product Prover Certifier has replaced traditional safety testing of interlocking and Communication-Based Train Control (CBTC) systems that entered revenue service in 2010 at RATP (Regie Autonome des Transports Parisiens), the Paris Metro infrastructure manager.

The product was used for formal verification of the safety of the PIPC interlocking systems supplied by Thales for Paris Metro line 3b, and the wayside CBTC system supplied by Ansaldo STS for Paris Metro Line 3. The application of the product provided independent, formal proof that these state-of-the-art railway signaling software systems always respect the safety requirements stipulated by RATP.
Pierre Chartier, safety director at RATP, said "RATP and Prover have collaborated since 2004 on formal safety verification of interlocking and CBTC systems. We're very satisfied to have reached the point that formal verification with Prover Certifier can replace testing-based methods. We use formal verification to reduce costs while maintaining the highest possible level of safety

November 16, 2010

Paris Metro replaces testing of interlocking and CBTC systems with formal safety verification using Prover Certifier

Stockholm, Sweden and Toulouse, France - November 16, 2010. Prover Technology today announced that its software product Prover Certifier has replaced traditional safety testing of interlocking and Communication-Based Train Control (CBTC) systems that entered revenue service in 2010 at RATP (Regie Autonome des Transports Parisiens), the Paris Metro infrastructure manager.

The product was used for formal verification of the safety of the PIPC interlocking systems supplied by Thales for Paris Metro line 3b, and the wayside CBTC system supplied by Ansaldo STS for Paris Metro Line 3. The application of the product provided independent, formal proof that these state-of-the-art railway signaling software systems always respect the safety requirements stipulated by RATP.
Pierre Chartier, safety director at RATP, said "RATP and Prover have collaborated since 2004 on formal safety verification of interlocking and CBTC systems. We're very satisfied to have reached the point that formal verification with Prover Certifier can replace testing-based methods. We use formal verification to reduce costs while maintaining the highest possible level of safety verification."

Prover Certifier was developed to meet the highest Safety Integrity Level, SIL 4, of the CENELEC (European Committee for Electrotechnical Standardization) standard EN50128. This makes it possible to replace time-consuming and incomplete testing-based methods for safety verification with exhaustive, formal proof-based safety verification.
In addition to the signaling systems whose safety assessment was based solely on the application of Prover Certifier, a number of interlocking systems installed on line 11 and line 12 of Paris Metro have previously been formally verified using Prover Certifier.

Emerging Directions from Internal R&D

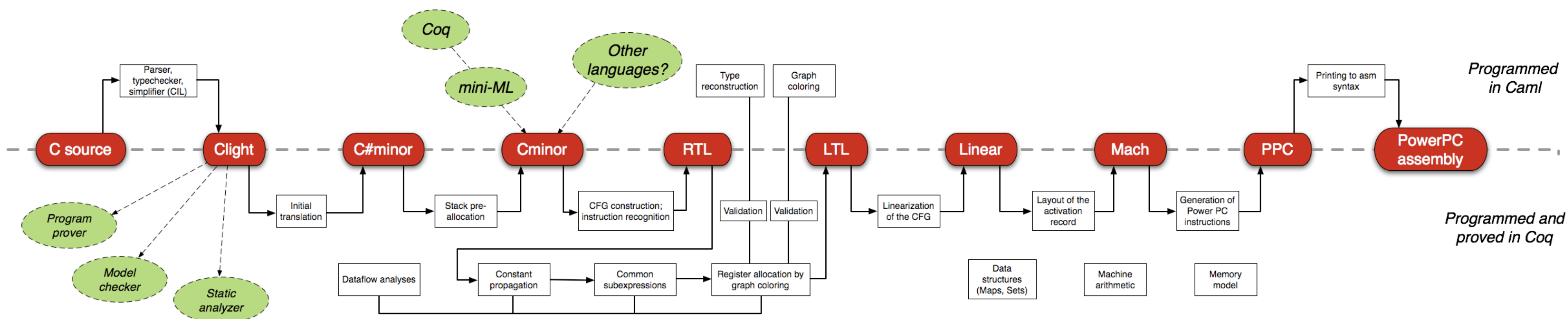
- Early analysis
  - Formal Methods / Model checking can detect errors earlier
    - Saves cost (from escaped defects)
    - Problems with scalability, use on production programs
  - Formal Language for requirements specification
- Automation
  - Generate test suite from design model or requirements
  - Automate verification to match automation of development

Program Proofs in Real Life

- seL4
  - "secure embedded" L4 operating system
  - 8,700 lines of C and ARM assembly
  - 200k lines of Isabelle proof script
    - 25 person-years of work!
  - Guarantees:
    - no null-pointer derefs, no indexing out of bounds,
      no space leaks.
  - http://ertos.nicta.com.au/research/sel4/

Program Proofs in Real Life

- CompCert (http://compcert.inria.fr/)
  - "Verified Compiler" from C to various assembly languages
  - comes with a "mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program"



Spec # Video

- Video:
  - http://channel9.msdn.com/Blogs/Peli/The-Verification-Corner-Specifications-in-Action-with-SpecSharp
- Binary Search:
  - http://rise4fun.com/SpecSharp/BinarySearch
- more examples:
  - http://www.rosemarymonahan.com/specsharp/