Construction and Verification of Software

2022 - 2023

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 3 - Specification and Verification

Bernardo Toninho (btoninho@fct.unl.pt)
based on previous editions by **João Seco** and **Luís Caires**

NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Administrivia

- First handout will be out next week! (~2 week to turn in).

Outline

- Hoare Logic revisited (Axiomatic approach)
- Iteration (Loop Invariants)
- Algorithmic approach to verification
- Examples

Part I
Hoare Logic (Recap)

Procedures and method calls

RECAP

$$
\begin{array}{lcll}
E & ::= & \text{Expressions} & \\
& \mid & \ldots & \\
\\
S & ::= & \text{Statements} & \\
& \mid & \ldots & \\
& \mid & x := m(E_1, \ldots, E_n) & \text{Call + Assignment} \\
\\
D & ::= & \text{Declarations} & \\
& \mid & \texttt{method}\ m(x_1, \ldots, x_n)\ \texttt{returns}\ (r) & \\
& & \texttt{requires}\ Pre(x_1, \ldots, x_n) & \\
& & \texttt{ensures}\ Post(x_1, \ldots, x_n, r) & \\
& & \{S\} & \\
\\
P & ::= & \text{Program} & \\
& \mid & \overline{D} &
\end{array}
$$

RECAP

Procedures and method calls

- Declarations annotated with pre- and post-conditions.
- Method calls built into a form of assignment.
- A program P is a set of method declarations.
- Each method decl. is validated, assuming its pre-condition and establishing its post-condition:

$$\frac{\{Pre(x_1, \ldots, x_n)\}\, S\, \{Post(x_1, \ldots, x_n, r)\}}{\begin{array}{l}\texttt{method}\ m(x_1, \ldots, x_n)\ \texttt{returns}\ (r) \\ \texttt{requires}\ Pre(x_1, \ldots, x_n) \\ \texttt{ensures}\ Post(x_1, \ldots, x_n, r)\ \{S\}\end{array}}$$

RECAP

Procedures and method calls

- Method calls built into a form of assignment:

$$\begin{array}{l}\texttt{method}\ m(x_1,\ldots,x_n)\ \texttt{returns}\ (r)\\ \texttt{requires}\ Pre(x_1,\ldots,x_n)\\ \texttt{ensures}\ Post(x_1,\ldots,x_n,r)\ \{S\}\end{array} \quad \in P$$

$$\frac{A \Rightarrow Pre(E_1,\ldots,E_n) \qquad Post(E_1,\ldots,E_n,r) \Rightarrow B[{}^{r}/{}_{x}]}{\{A\}\, x := m(E_1,\ldots,E_n)\, \{B\}}$$

Procedures and method calls

RECAP

$$\begin{array}{ll}\texttt{method}\ m(x_1,\ldots,x_n)\,\texttt{returns}\,(r) & \\ \texttt{requires}\ Pre(x_1,\ldots,x_n) & \in P \\ \texttt{ensures}\ Post(x_1,\ldots,x_n,r)\ \{S\} & \end{array}$$

$$\frac{A \Rightarrow Pre(E_1,\ldots,E_n) \qquad Post(E_1,\ldots,E_n,r) \Rightarrow B[{}^{r}/_{x}]}{\{A\}\,x := m(E_1,\ldots,E_n)\,\{B\}}$$

- Instantiated method pre-condition must follow from A
- Instantiated method post-condition must imply B
- Calls are **opaque**! We only know what's in the post-condition.
- Verification with method calls is **modular**.

Procedures and method calls

RECAP

```
method maxImp(x:int,y:int) returns (r:int)
  ensures r >= x && r >= y
{
  if x > y { r := x; } else { r := y; }
  return r;
}

method Main() {
  var a := -10;
  var b := 23;
  var c := maxImp(a,b);
  assert (c == b);          ⊗ assertion violation Verifier


}
```

Procedures and method calls

RECAP

```
method maxImp(x:int,y:int) returns (r:int)
  ensures r >= x && r >= y
{
  if x > y { r := x; } else { r := y; }
  return r;
}


method Main() {
  var a := -10;
  var b := 23;
  var c := maxImp(a,b);
  assert (c >= b);

}
```

RECAP

Procedures and method calls

```
method maxImp(x:int,y:int) returns (r:int)
  ensures (x>y ==> r == x) && (x <= y ==> r == y)
{
  if x > y { r := x; } else { r := y; }
  return r;
}


method Main() {
  var a := -10;
  var b := 23;
  var c := maxImp(a,b);
  assert (c == b);

}
```

Hoare Logic - Rules

RECAP

$$\{A\}\ \texttt{skip}\ \{A\}$$

$$\{A[{}^{E}/_{x}]\}\ x := E\ \{A\}$$

$$\frac{\{A\}\ P\ \{B\} \quad \{B\}\ Q\ \{C\}}{\{A\}\ P;Q\ \{C\}}$$

$$\frac{A' \implies A \quad \{A\}\ P\ \{B\} \quad B \implies B'}{\{A'\}\ P\ \{B'\}}$$

Rule for Assignment

RECAP

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

- A[E/x] means:
  - the result of replacing all free occurrences of variable x in assertion A by the expression E
- For this rule to be sound, we require E to be an expression without side effects (a pure expression)

Rule for Assignment

RECAP

$$\{A[^{E}/_{x}]\}\ x := E\ \{A\}$$

- We can think of A as a condition where “x” appears in some places. A is a condition dependent on “x”.
- The assignment x := E changes the value of x to E, but leaves everything else unchanged
- So everything that could be said of E in the precondition, can be said of x in the postcondition, since the value of x after the assignment is E
- Example: { x + 1 > 0 } x := x + 1 { x > 0 }

Exercises

Prove using the assignment rule that:

```
assert y > 0;
x := y;
assert x > 0 && y == x;
```

```
assert y == x;
x := 2 * x;
assert y == x / 2;
```

Exercises

Prove using the assignment rule that:

```
function P(x:int):bool {
  …
}
function Q(x:int):bool {
  …
}

  var x := …;
  var y := …;
  var z;

  assert P(x) && Q(y);
  z := x;
  x := y;
  y := z;
  assert P(y) && Q(x);
```

Example

- Consider the program

  P ≜ if (x>y) then z := x else z := y

- We (mechanically) check the triple

  { true } P { z == max(x,y) }

Example

- Consider the program

  P ≜ if (x>y) then z := x else z := y

- We (mechanically) check the triple

  { true } P { z == max(x,y) }

  { x == max(x,y) } z := x { z == max(x,y) }
  { x > y } z := x { z == max(x,y) }

  { y == max(x,y) } z := y { z == max(x,y) }
  { y >= x } z := y { z == max(x,y) }

Part II
Loops and Loop Invariants

Rule for Iteration

$$\frac{\{? \wedge E\}\ P\ \{?\}}{\{A\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{\neg E \wedge ?\}}$$

Any precise post condition depends on how many times P is executed … P can be executed 0, 1, 2 … n times, n is generally not known at compile/verification time.

Rule for Iteration

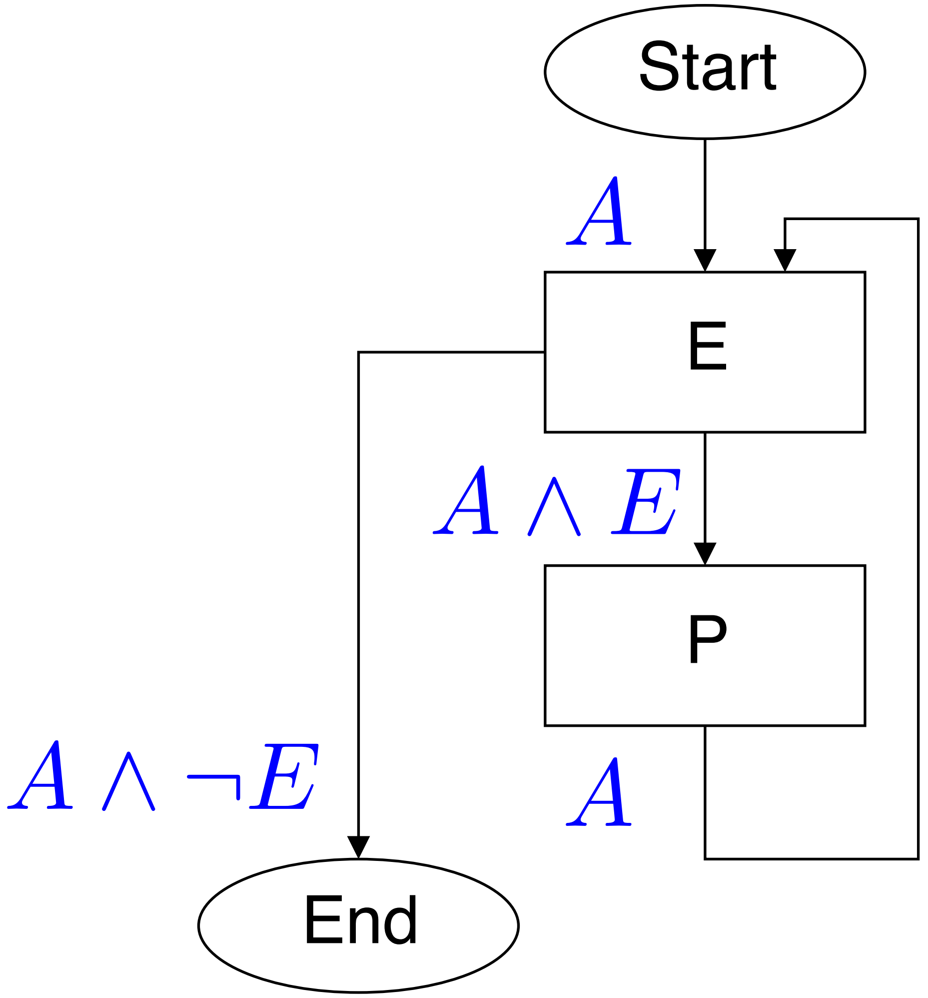
$$
\begin{array}{l}
\texttt{while } E \texttt{ do } P \triangleq \\
\quad \texttt{if } E \texttt{ then } P; \\
\qquad \texttt{if } E \texttt{ then } P; \\
\qquad\quad \texttt{if } E \texttt{ then } P, \ldots \\
\qquad\quad \texttt{else skip} \\
\qquad \texttt{else skip} \\
\quad \texttt{else skip}
\end{array}
$$

Rule for Iteration

$$
\begin{array}{l}
\{A\}\,\texttt{while}\ E\ \texttt{do}\ P\,\{B\} \triangleq \\
\quad \{A\}\,\texttt{if}\ E\ \texttt{then}\ \{A \land E\}\,P\,\{B_1\}; \\
\qquad \{B_1\}\texttt{if}\ E\ \texttt{then}\ \{B_1 \land E\}\,P\,\{B_2\}; \\
\qquad\quad \{B_2\}\texttt{if}\ E\ \texttt{then}\ \{B_2 \land E\}\,P\,\{B_3\}; \\
\qquad\quad \ldots \\
\qquad\quad \texttt{else skip}\,\{\neg E \land B_2\} \\
\qquad \texttt{else skip}\,\{\neg E \land B_1\} \\
\quad \texttt{else skip}\,\{\neg E \land A\}
\end{array}
$$

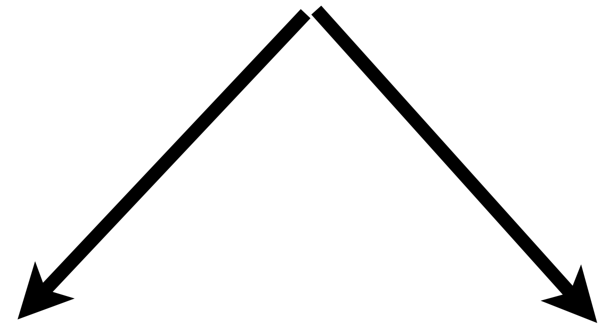
Rule for Iteration

$$\frac{\{A \wedge E\}\ P\ \{A\}}{\{A\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{A \wedge \neg E\}}$$



Rule for Iteration

INV = Invariant Condition



$$\frac{\{Inv \wedge E\}\ P\ \{Inv\}}{\{Inv\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{Inv \wedge \neg E\}}$$

- We cannot predict in general for how many iterations the while loop will run (undecidability of the halting problem).
- We approximate all iterations by an invariant condition
- A loop invariant is a condition that holds at loop entry and at loop exit.

Rule for Iteration

INV = Invariant Condition

$$\frac{\{Inv \wedge E\}\ P\ \{Inv\}}{\{Inv\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{Inv \wedge \neg E\}}$$

- If the invariant holds initially and is preserved by the loop body, it will hold when the loop terminates!
- It does not matter how many iterations will run.
- Unlike the other rules of Hoare logic, finding the invariant requires human intelligence and creativity.

Rule for Iteration

INV = Invariant Condition

$$\frac{\{Inv \wedge E\}\ P\ \{Inv\}}{\{Inv\}\ \texttt{while}\ E\ \texttt{do}\ P\ \{Inv \wedge \neg E\}}$$

- The invariant depicts the state in all iterations of a loop.
- The invariant works like the induction hypothesis in a proof. The base case is the loop executed 0 times, the loop body is the induction step that iterates from step n to n+1.There must exist a valid induction measure.

Loop Invariants

$$\{0 \leq n\}$$

$$i := 0;$$

$$\texttt{while}\ i < n\ \texttt{do}\ \{$$

$$\quad i := i + 1$$

$$\}$$

$$\{i == n\}$$

Loop Invariants

$$\{0 \leq n\}$$
$$i := 0;$$
$$\{i == 0 \wedge 0 \leq n\}$$
$$\{0 \leq i \leq n\}$$
$$\texttt{while } i < n \texttt{ do } \{$$
$$\{0 \leq i \leq n \wedge i < n\}$$
$$\{0 \leq i < n\}$$
$$\{0 \leq i + 1 \leq n\}$$
$$i := i + 1$$
$$\{0 \leq i \leq n\}$$
$$\}$$
$$\{0 \leq i \leq n \wedge i >= n\}$$
$$\{i == n\}$$

Part III
Breaking and Fixing Loop Invariants

Loop Invariants

- Consider program P defined by

$$P \triangleq s := 0;\ i := 0;\ \texttt{while}\ i < n\ \texttt{do}\ \{i := i + 1;\ s := s + i\}$$

- What is the specification of P? What does P do?

$$\{A\}\ P\ \{B\}$$

Loop Invariants

- Consider program P defined by

$$P \triangleq s := 0;\ i := 0;\ \texttt{while}\ i < n\ \texttt{do}\ \{i := i + 1;\ s := s + i\}$$

- What is the specification of P? What does P do?

$$\{n \geq 0\}\ P\ \{s = \sum_{j=0}^{n} j\}$$

Is this a good specification for program P?

Can we mechanically check the Hoare triple?

Loop Invariants

$$\{0 \leq n\}$$

$$s := 0;$$

$$i := 0;$$

$$\texttt{while } i < n \texttt{ do } \{$$

$$\quad i := i + 1;$$

$$\quad s := s + i$$

$$\}$$

$$\{s == \sum_{j=0}^{n} j\}$$

Loop Invariants

$$\{0 \leq n\}$$

$$s := 0;$$

$$\{s = 0 \land 0 \leq n\}$$

$$i := 0;$$

$$\{s = 0 \land 0 \leq i \leq n\}$$

$$\texttt{while}\ i < n\ \texttt{do}\ \{$$

$$\quad i := i + 1;$$

$$\quad s := s + i$$

$$\}$$

$$\{s = \sum_{j=0}^{n} j\}$$

Loop Invariants

$$\{0 \leq n\}$$
$$s := 0;$$
$$\{s = 0 \wedge 0 \leq n\}$$
$$i := 0;$$
$$\{s = 0 \wedge 0 \leq i \leq n\}$$
$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$$
$$\texttt{while } i < n \texttt{ do } \{$$
$$\quad i := i + 1;$$
$$\quad s := s + i;$$
$$\}$$
$$\{i = n \wedge s = \sum_{j=0}^{i} j\}$$
$$\{s = \sum_{j=0}^{i} j\}$$

Loop Invariants

$\{0 \leq n\}$
$s := 0;$
$\{s = 0 \wedge 0 \leq n\}$
$i := 0;$
$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$
$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$
$\texttt{while } i < n \texttt{ do } \{$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$
$\quad i := i + 1;$
$\quad s := s + i$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$
$\}$
$\{i = n \wedge s = \sum_{j=0}^{i} j\}$
$\{s = \sum_{j=0}^{n} j\}$

Invariant holds

Loop Invariants

- The loop invariant may be broken inside the body of the loop, but must be re-established at the end.
- Notice the assignment rule

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

- that breaks the invariant…

$$\{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^{i} j\}$$

$$\{0 \leq i < n \wedge s = \sum_{j=0}^{i} j\}$$

$$i := i + 1$$

$$\{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} j\}$$

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} j\}$$

Loop Invariants

- The loop invariant may be broken inside the body of the loop, but must be re-established at the end.
- Notice the assignment rule

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

- and then re-establishes it

$$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} j\}$$

$$s := s + i$$

$$\{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} j) + i\}$$

$$\{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i} j)\}$$

Loop Invariants

$\{0 \leq n\}$

$s := 0;$

$\{s = 0 \wedge 0 \leq n\}$

$i := 0;$

$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$

$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$

$\texttt{while } i < n \texttt{ do } \{$

$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$

$\quad i := i + 1;$

$\quad s := s + i$

$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$

$\}$

$\{i = n \wedge s = \sum_{j=0}^{i} j\}$

$\{s = \sum_{j=0}^{n} j\}$

Invariant holds

Loop Invariants

$$\{0 \leq n\}$$
$$s := 0;$$
$$\{s = 0 \land 0 \leq n\}$$
$$i := 0;$$
$$\{s = 0 \land i = 0 \land 0 \leq i \leq n\}$$
$$\{0 \leq i \leq n \land s = \sum_{j=0}^{i} j\}$$
$$\texttt{while } i < n \texttt{ do } \{$$
$$\{0 \leq i \leq n \land i < n \land s = \sum_{j=0}^{i} j\}$$
$$\{0 \leq i < n \land s = \sum_{j=0}^{i} j\}$$
$$i := i + 1;$$
$$\{0 \leq i - 1 < n \land s = \sum_{j=0}^{i-1} j\}$$

Invariant broken

$$\{0 \leq i \leq n \land s = \sum_{j=0}^{i-1} j\}$$
$$s := s + i$$
$$\{0 \leq i \leq n \land s = (\sum_{j=0}^{i-1} j) + i\}$$

Invariant restored

$$\{0 \leq i \leq n \land s = \sum_{j=0}^{i} j\}$$
$$\}$$
$$\{i = n \land s = \sum_{j=0}^{i} j\}$$
$$\{s = \sum_{j}^{n} j\}$$

Loop Invariants

$\{0 \leq n\}$
$s := 0;$
$\{s = 0 \wedge 0 \leq n\}$
$i := 0;$
$\{s = 0 \wedge i = 0 \wedge 0 \leq i \leq n\}$
$\{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$
$\texttt{while } i < n \texttt{ do } \{$
$\quad \{0 \leq i \leq n \wedge i < n \wedge s = \sum_{j=0}^{i} j\}$
$\quad \{0 \leq i < n \wedge s = \sum_{j=0}^{i} j\}$
$\quad i := i + 1;$
$\quad \{0 \leq i - 1 < n \wedge s = \sum_{j=0}^{i-1} j\}$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} j\}$
$\quad s := s + i$
$\quad \{0 \leq i \leq n \wedge s = (\sum_{j=0}^{i-1} j) + i\}$
$\quad \{0 \leq i \leq n \wedge s = \sum_{j=0}^{i} j\}$
$\}$
$\{i = n \wedge s = \sum_{j=0}^{i} j\}$
$\{s = \sum_{j}^{n} j\}$

Invariant holds

Hints for finding loop invariants

- **First:** carefully think about the post condition of the loop
  - Typically the post-condition talks about a property “accumulated” across a “range”
    (this is why you are using a loop, right?)
  - e.g., maximum of all elements of an array
  - e.g., sort visited elements in a data structure

Hints for finding loop invariants

- **Second:** design a "generalized" version of the post-condition, in which the already visited part of the data is made explicit as a function of the "loop control variable" (generalizing the i.h, remember?)

- The loop body may temporarily break the invariant, but must restore it at the end of the body

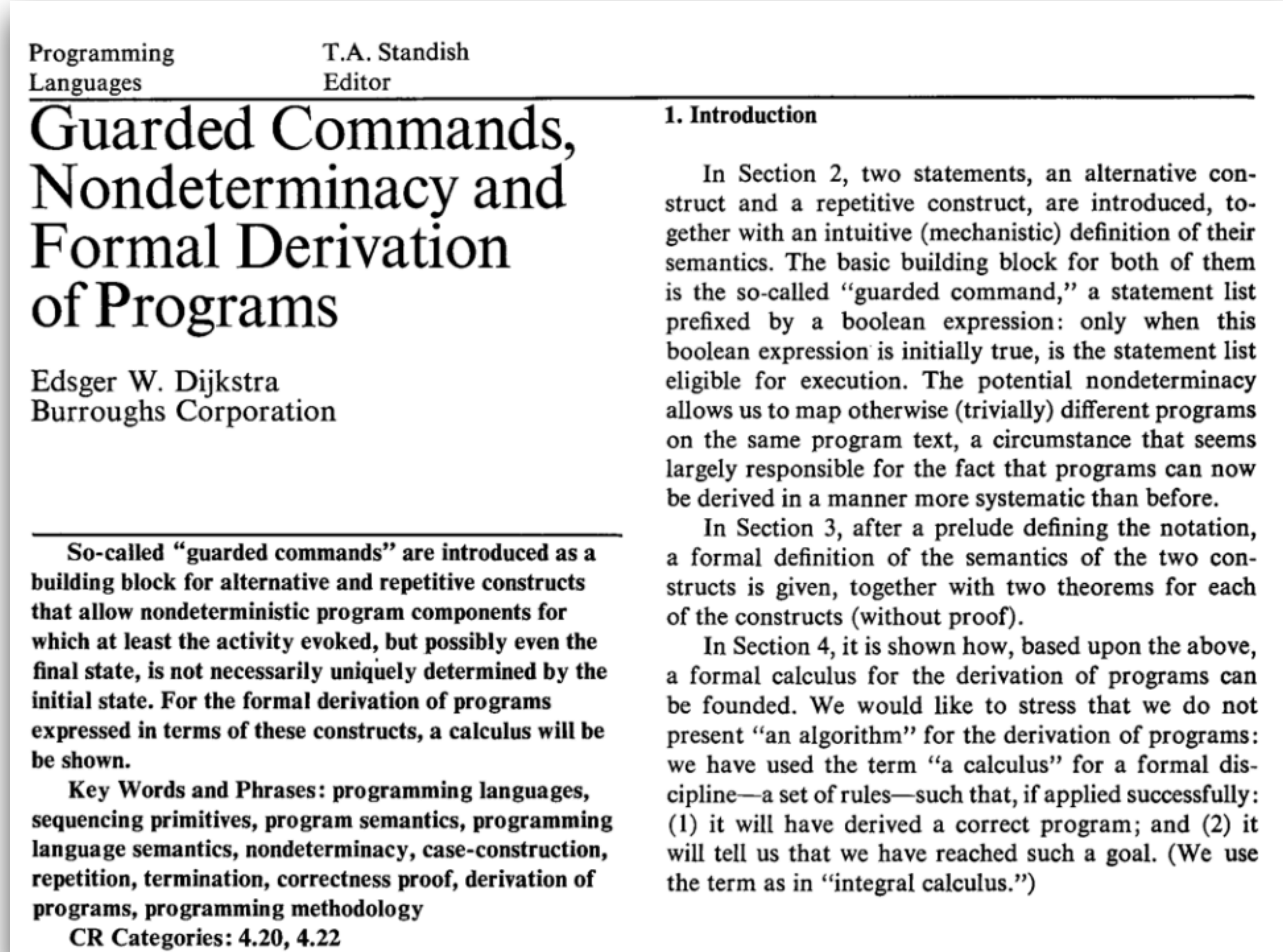
- **Important:** make sure that the invariant together (&&) with the termination condition really implies your post-condition

Examples, what kind of invariant we need for…

- **Max of an array**
  - All elements to the left are smaller than the max so far
- **Array Searching (unsorted)**
  - All elements left of the index are different from the value being searched
- **Array Searching (sorted)**
  - The element is between the lower and the higher limits
- **Sorting (bubblesort, insertion sort, etc.)**
  - Everything to the left of the cursor is sorted
- **List Reversing**
  - All elements to the left of the cursor are placed on the right of the result

Part IV
Weakest Pre-condition Algorithm

The Weakest Precondition



Programming Languages

T.A. Standish
Editor

Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
Burroughs Corporation

So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be be shown.

Key Words and Phrases: programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology

CR Categories: 4.20, 4.22

1. Introduction

In Section 2, two statements, an alternative construct and a repetitive construct, are introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called "guarded command," a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential nondeterminacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that programs can now be derived in a manner more systematic than before.

In Section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs is given, together with two theorems for each of the constructs (without proof).

In Section 4, it is shown how, based upon the above, a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present "an algorithm" for the derivation of programs: we have used the term "a calculus" for a formal discipline—a set of rules—such that, if applied successfully: (1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal. (We use the term as in "integral calculus.")

Predicate transformer semantics

- Algorithmic approach to verify a while program
- Defines a predicate transformer that produces the weakest precondition for a pair of program/assertion

$$wp(P, B)$$

- Any Hoare triple $\{A\}\ P\ \{B\}$ is provable if and only if the predicate $A \Rightarrow wp(P, B)$ holds.
- The predicate is (recursively) defined on the cases of the program syntax.

Predicate transformer semantics

$$\{A\}\ \texttt{skip}\ \{A\}$$

$$wp(\texttt{skip}, A) = A$$

$$\frac{\{A\}\ P\ \{B\} \quad \{B\}\ Q\ \{C\}}{\{A\}\ P; Q\ \{C\}}$$

$$wp(P; Q, C) = wp(P, wp(Q, C))$$

$$\{A[^E/_x]\}\ x := E\ \{A\}$$

$$wp(x := E, A) \triangleq A[^E/_x]$$

$$\frac{\{A \wedge E\}\ P\ \{B\} \quad \{A \wedge \neg E\}\ Q\ \{B\}}{\{A\}\ \texttt{if}\ E\ \texttt{then}\ P\ \texttt{else}\ Q\ \{B\}}$$

$$wp(\texttt{if}\ E\ \texttt{then}\ P_1\ \texttt{else}\ P_2, B) \triangleq E \Rightarrow wp(P_1, B) \wedge \neg E \Rightarrow wp(P_2, B)$$

Example (again)

- Consider the program

  $P \triangleq$ if (x>y) then z := x else z := y

- We (mechanically) check the triple

  { true } P { z == max(x,y) }

Example (again)

- Consider the program

$$P \triangleq x := y \,;\, y := z;\;\; z := x$$

We (mechanically) check the triple

{ P(y) && Q(z) } P { P(z) && Q(y) }

(here P and Q are any properties)

Algorithmic approach for Iteration

$$
\begin{aligned}
&wp(\texttt{while } E \texttt{ do } P, B) \triangleq \\
&\qquad I \wedge (E \wedge I \Rightarrow wp(P, I)) \wedge (\neg E \wedge I \Rightarrow B)
\end{aligned}
$$