The Calculus of Computation

Decision Procedures with Applications to Verification

Aaron R. Bradley and Zohar Manna

Stanford University

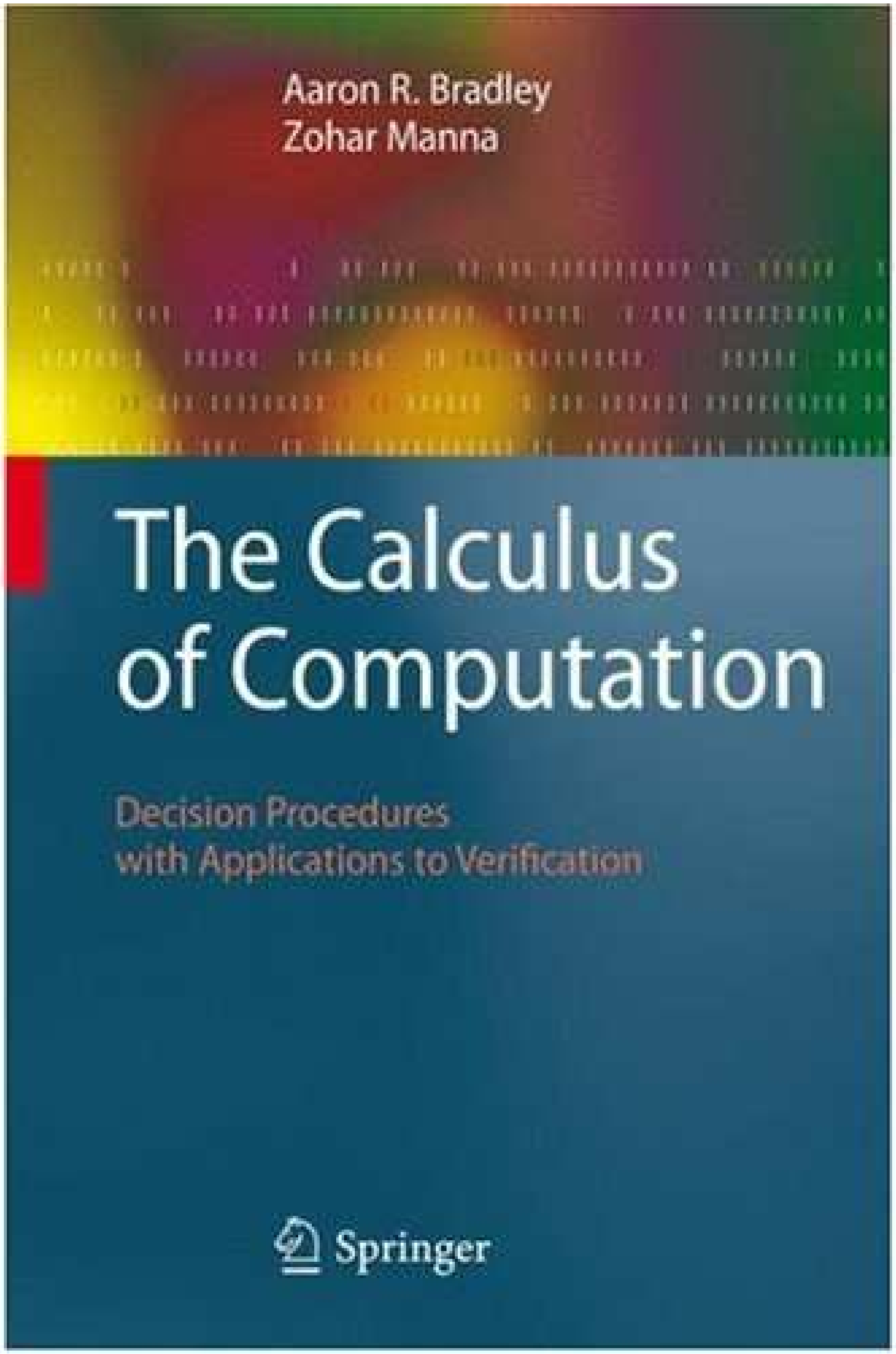
(Aaron is visiting EPFL and will soon be at CU Boulder)

The Calculus of Computation?

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and mathematical elegance.

— John McCarthy

A Basis for a Mathematical Theory of Computation, 1963


Aaron R. Bradley
Zohar Manna
The Calculus of Computation
Decision Procedures with Applications to Verification
Springer

Goals

Teach logic as a fundamental tool in engineering.

- Present computational view of logic.
- Apply logic to specification and verification.
  - Promote a practical understanding of logic.
  - Teach the fundamental concepts in verification.
- Connect to other topics.

Audience

- Advanced undergraduate students
- Beginning graduate students
- Computer scientists and engineers who want to apply decision procedures

But assumes very little.

Topics: Overview

- First-order logic
- Specification & verification
- Satisfiability decision procedures
- Static analysis

Part I: Foundations

1. Propositional Logic
2. First-Order Logic
3. First-Order Theories
4. Induction
5. Program Correctness: Mechanics
   Inductive assertion method, Ranking function method
6. Program Correctness: Strategies

Pi: *Prove it*

```
@pre ⊤
@post ∀m, n. 0 ≤ m ≤ n < |rv| → rv[m] ≤ rv[n]
int[] BubbleSort(int[] a₀) {
  int[] a := a₀;
  for
    @L₁ : [ −1 ≤ i < |a|
            ∧ ∀m, n. i ≤ m ≤ n < |a| → a[m] ≤ a[n]
            ∧ ∀m, n. 0 ≤ m ≤ i ∧ i + 1 ≤ n < |a| → a[m] ≤ a[n] ]
    (int i := |a| − 1; i > 0; i := i − 1)
    for
      @L₂ : [ 1 ≤ i < |a| ∧ 0 ≤ j ≤ i
              ∧ ∀m, n. i ≤ m ≤ n < |a| → a[m] ≤ a[n]
              ∧ ∀m, n. 0 ≤ m ≤ i ∧ i + 1 ≤ n < |a| → a[m] ≤ a[n]
              ∧ ∀m. 0 ≤ m < j → a[m] ≤ a[j] ]
      (int j := 0; j < i; j := j + 1)
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
  return a;
}
```

Part II: Algorithmic Reasoning

7. Quantified Linear Arithmetic
   Quantifier elimination for integers and rationals
8. Quantifier-Free Linear Arithmetic
   Linear programming for rationals
9. Quantifier-Free Equality and Data Structures
10. Combining Decision Procedures
    Nelson-Oppen combination method
11. Arrays
    More than quantifier-free fragment
12. Invariant Generation
    Abstract interpretation without the Greek

Courses

Full course

- Semester: time for theorems
- Quarter: fast pace or skip some theorems

Partial course

- Combination procedure track: 5-10 lectures
  Incorporate into course on theorem proving
- Verification track: 5-10 lectures
  Prepare students for depth in static analysis

Track: Combination Procedures

1. Propositional Logic
2. First-Order Logic

   Theorems: Compactness, Craig Interpolation
3. First-Order Theories

8. Quantifier-Free Linear Arithmetic
9. Quantifier-Free Equality and Data Structures
10. Combining Decision Procedures

    Theorem: Correctness of Nelson-Oppen

Track: Verification

Partial & total correctness of sequential programs

1. Propositional Logic
2. First-Order Logic
3. First-Order Theories
4. Induction
5. Program Correctness: Mechanics
6. Program Correctness: Strategies

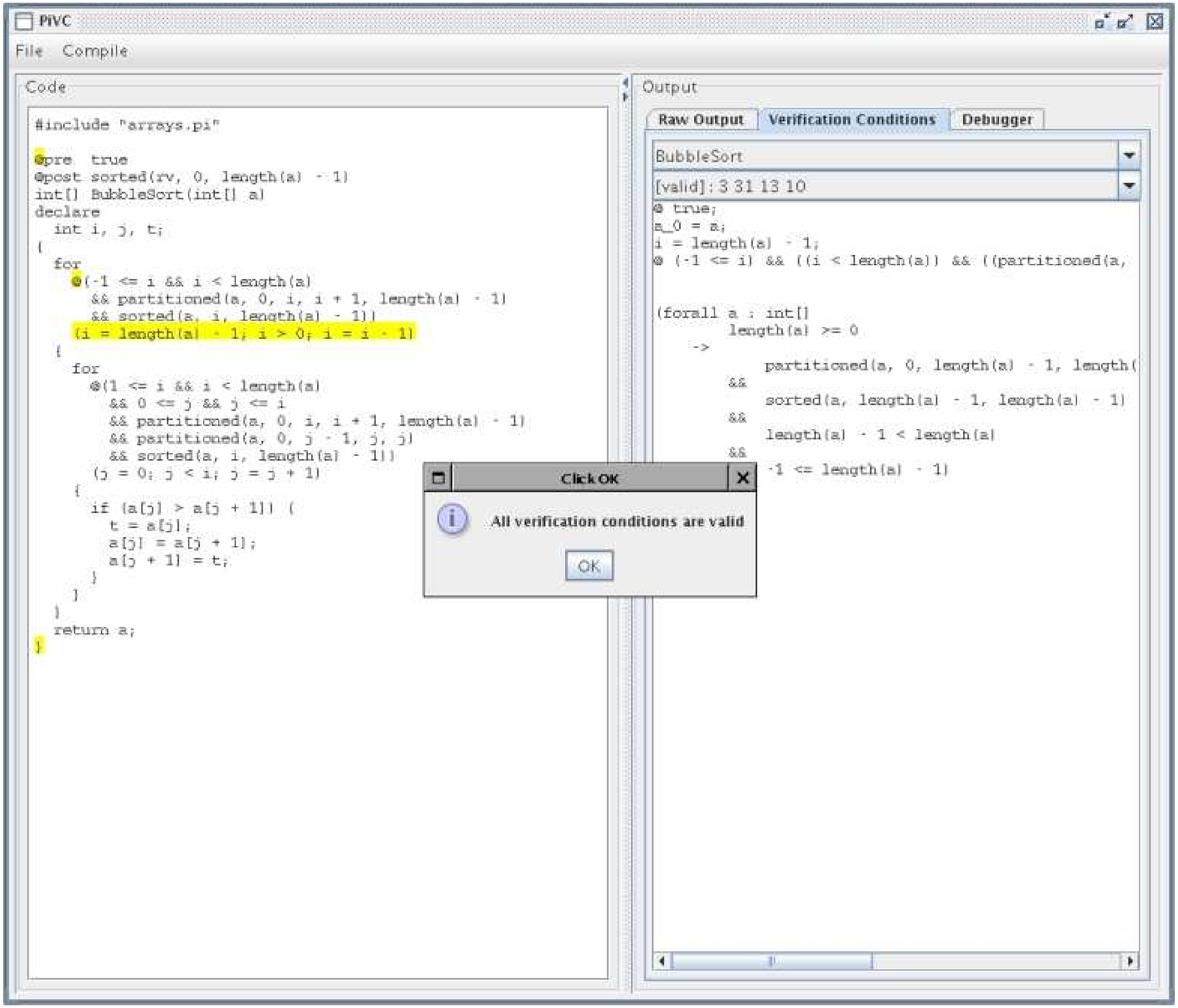
12. Invariant Generation

Courses

Exercises

- Each chapter includes exercises.
  Range from applied to theoretical
- πVC: Assign exercises throughout course.
  - Students need time to learn skills.
  - Students learn to use logic.

πVC


PiVC
File Compile
Code
#include "arrays.pi"
@pre true
@post sorted(rv, 0, length(a) - 1)
int[] BubbleSort(int[] a)
declare
int i, j, t;
{
for
@(-1 <= i && i < length(a)
&& partitioned(a, 0, i, i + 1, length(a) - 1)
&& sorted(a, i, length(a) - 1))
(i = length(a) - 1; i > 0; i = i - 1)
{
for
@(1 <= i && i < length(a)
&& 0 <= j && j <= i
&& partitioned(a, 0, i, i + 1, length(a) - 1)
&& partitioned(a, 0, j - 1, j, j)
&& sorted(a, i, length(a) - 1))
(j = 0; j < i; j = j + 1)
{
if (a[j] > a[j + 1]) {
t = a[j];
a[j] = a[j + 1];
a[j + 1] = t;
}
}
}
return a;
}
Output
Raw Output
Verification Conditions
Debugger
BubbleSort
[valid]: 3 31 13 10
@ true;
a_0 = a;
i = length(a) - 1;
@ (-1 <= i) && ((i < length(a)) && ((partitioned(a,
(forall a : int[]
length(a) >= 0
->
partitioned(a, 0, length(a) - 1, length(
&&
sorted(a, length(a) - 1, length(a) - 1)
&&
length(a) - 1 < length(a)
&&
-1 <= length(a) - 1)
Click OK
All verification conditions are valid
OK

πVC

- Download:
  `http://theory.stanford.edu/~arbrad/pivc`
- Runs on Linux & Mac OS X
- Minimal technical overhead
- All exercises from Chapters 5 & 6

Verification Exercises

Focus on arrays. Why?

- Data structure invariants are common.
- Most expressive decidable fragment in book.
- Personal bias (previous research).

Exercises:

- Sorting: from BubbleSort to QuickSort
- Searching: linear and binary search
- Set operations

More Information

- `http://theory.stanford.edu/~arbrad`
- I have a copy of the book with me.