Dance of the Dragons: Induction, Difference Computation, SMT Solving

Divyesh Unadkat

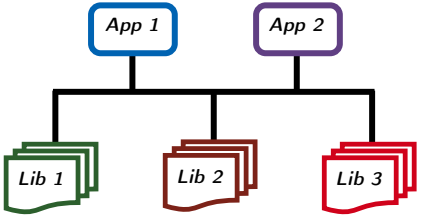
Joint work with Prof. Supratik Chakraborty, Prof. Ashutosh Gupta

FM Update - 5^{th} July'22

tcs Research

Motivation

Photo by Sahin Sezer Dincer from Pexels



- Software in safety critical applications often use arrays
- Implemented with arrays of parametric size
  - Model-specific parameter instantiation at deployment
- Libraries with parametric sized arrays frequently used
- Important to verify parametric properties of such software

Prove Correctness of Parametric Array Programs

tcs Research

```
assume(∀x∈[0,N) A[x] = *)                    φ(N)

void foo(int A[], int N)
{
  for (int i=0; i<N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
}                                            } P_N

assert(∀x∈[0,N) A[x] ≥ 5)                    ψ(N)
```

Arrays of parametric size N

Branch conditions dependent on N; Nested loops

Pre- and post-condition formulas

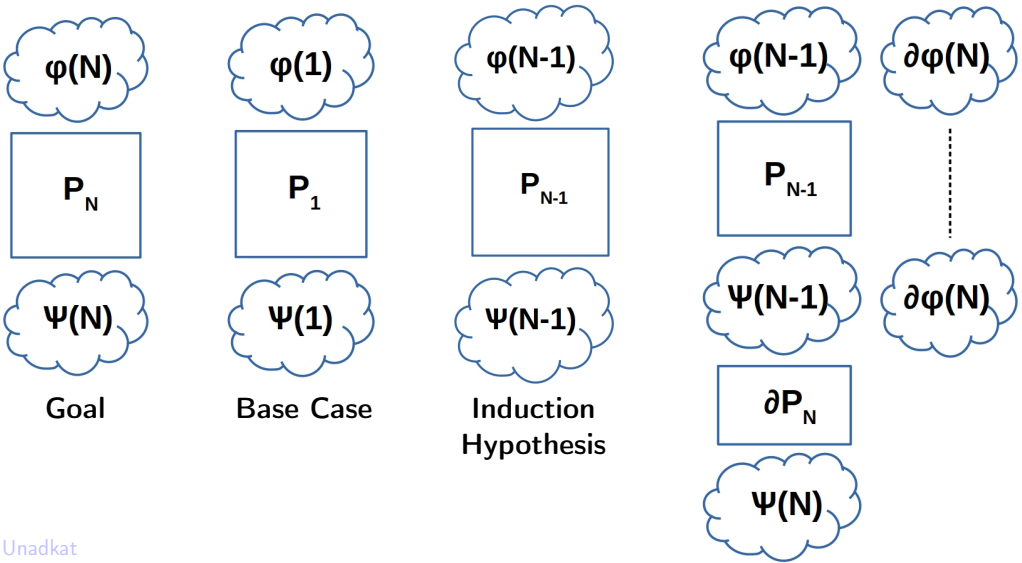
- Quantified or quantifier-free
- Non-linear terms

Prove the parametric Hoare triple
$\{\varphi(N)\}\ \mathrm{P}_N\ \{\psi(N)\}$ for all $N > 0$

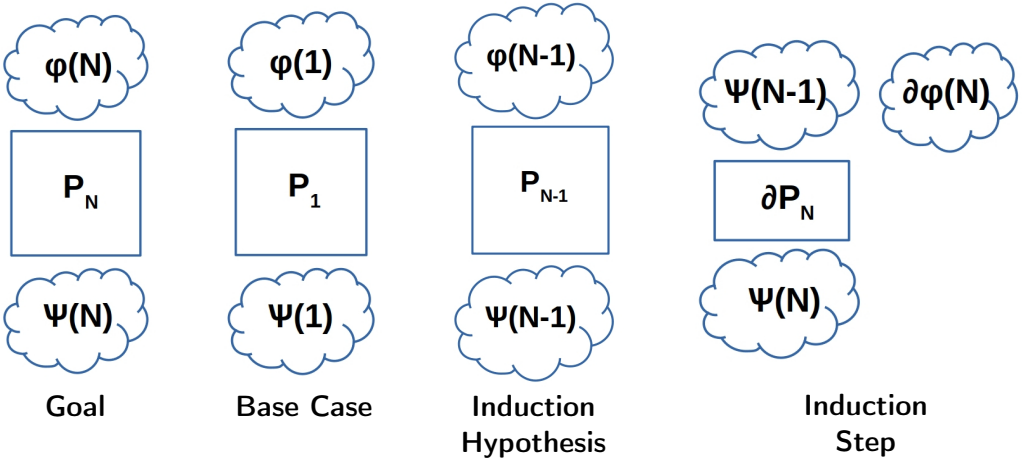
Classical Hoare logic-based methods

- Need quantified loop invariants
- Difficult to compute

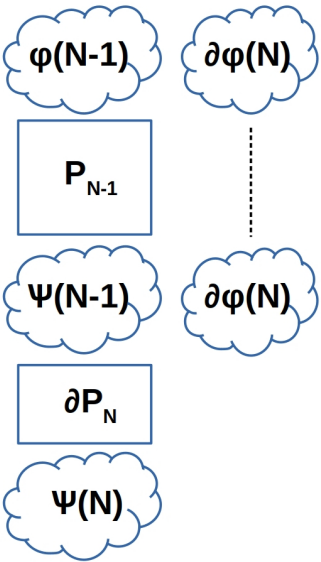
Full-Program Induction [TACAS'20, STTT'22*]



Full-Program Induction [TACAS'20, STTT'22*]



Difference Computation Example




```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.    A[i] = A[i] + S;
7.  }

8.  for(i=0; i<N; i++) {
9.    S = S + A[i];
10. }

assert(S = N × (N+2))
```

P_N

```
assume(∀i∈[0,N-1) A_Nm1[i] = 1)

1.  S_Nm1 = 0;
2.  for(i=0; i<N-1; i++) {
3.    S_Nm1 = S_Nm1 + A_Nm1[i];
4.  }

5.  for(i=0; i<N-1; i++) {
6.    A_Nm1[i] = A_Nm1[i] + S_Nm1;
7.  }

9.  for(i=0; i<N-1; i++) {
10.   S_Nm1 = S_Nm1 + A_Nm1[i];
11. }

assert(S_Nm1 = (N-1) × (N+1))
```

P_{N-1}

If Only Difference Computation Were Easy!!!

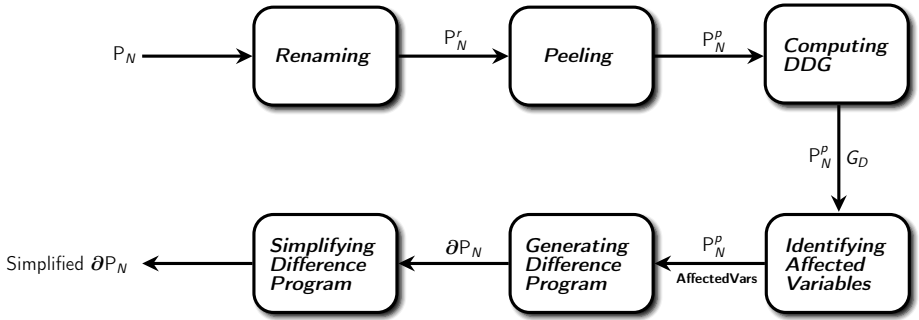
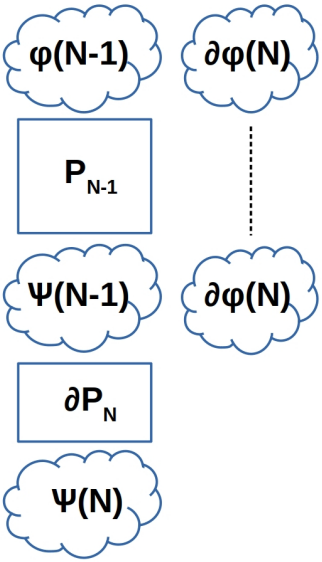


Figure: Sequence of steps for the computation of the difference program ∂P_N

Lemma (Sound Decomposition)

$$\{\varphi(N)\}\ P_N\ \{\psi(N)\} \iff \{\varphi(N)\}\ P_{N-1}; \partial P_N\ \{\psi(N)\}$$

Renaming

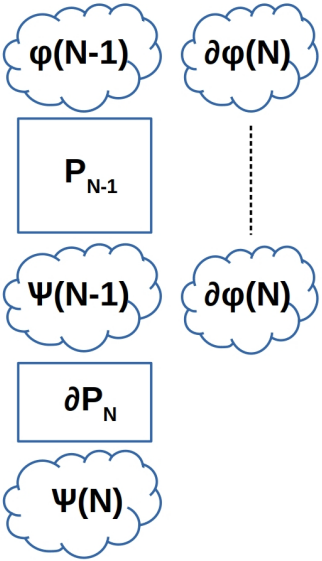



```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.    A[i] = A[i] + S;
7.  }


8.  for(i=0; i<N; i++) {
9.    S = S + A[i];
10. }

assert(S = N × (N+2))
```

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.    A1[i] = A[i] + S;
7.  }

8.  S1 = S;
9.  for(i=0; i<N; i++) {
10.   S1 = S1 + A1[i];
11. }

assert(S1 = N × (N+2))
```

Divyesh Unadkat

Peeling



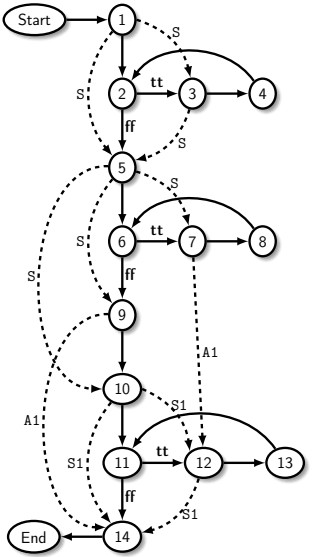

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.    A1[i] = A[i] + S;
7.  }

8.  S1 = S;
9.  for(i=0; i<N; i++) {
10.   S1 = S1 + A1[i];
11. }

assert(S1 = N × (N+2))
```

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N-1; i++) {
3.    S = S + A[i];
4.  }
5.  S = S + A[N-1];

6.  for(i=0; i<N-1; i++) {
7.    A1[i] = A[i] + S;
8.  }
9.  A1[N-1] = A[N-1] + S;

10. S1 = S;
11. for(i=0; i<N-1; i++) {
12.   S1 = S1 + A1[i];
13. }
14. S1 = S1 + A1[N-1];

assert(S1 = N × (N+2))
```


Divyesh Unadkat

Affected Variable Analysis using DDG

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N-1; i++) {
3.    S = S + A[i];
4.  }
5.  S = S + A[N-1];

6.  for(i=0; i<N-1; i++) {
7.    A1[i] = A[i] + S;
8.  }
9.  A1[N-1] = A[N-1] + S;

10. S1 = S;
11. for(i=0; i<N-1; i++) {
12.   S1 = S1 + A1[i];
13. }
14. S1 = S1 + A1[N-1];

assert(S1 = N × (N+2))
```



A is **not** affected; not modified

S is **not** affected

- No dependenece on peel or N

A1 is affected

- Depends on a value in the peel

S1 is affected

- Depends on a value in the peel, and
- Depends on an affected array

Generating the Difference Program ∂P_N

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N-1; i++) {
3.    S = S + A[i];
4.  }
5.  S = S + A[N-1];

6.  for(i=0; i<N-1; i++) {
7.    A1[i] = A[i] + S;
8.  }
9.  A1[N-1] = A[N-1] + S;

10. S1 = S;
11. for(i=0; i<N-1; i++) {
12.   S1 = S1 + A1[i];
13. }
14. S1 = S1 + A1[N-1];

assert(S1 = N × (N+2))
```

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + (S - S_Nm1);
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + (S - S_Nm1);
7.  for(i=0; i<N-1; i++) {
8.    S1 = S1 + (A1[i] - A1_Nm1[i]);
9.  }
10. S1 = S1 + A1[N-1];

assert(S1 = N × (N+2))
```

∂P_N

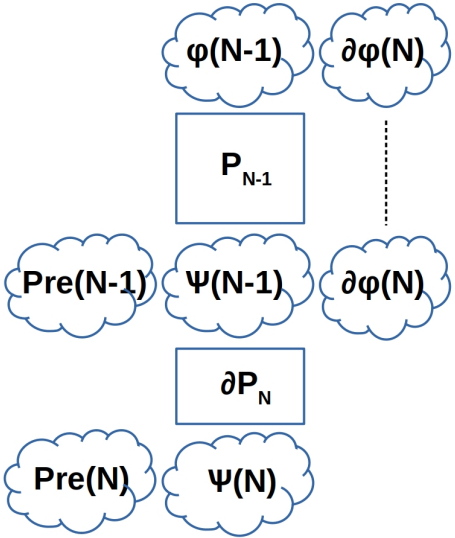
Simplifying the Difference Program ∂P_N

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N-1; i++) {
3.    S = S + A[i];
4.  }
5.  S = S + A[N-1];

6.  for(i=0; i<N-1; i++) {
7.    A1[i] = A[i] + S;
8.  }
9.  A1[N-1] = A[N-1] + S;

10. S1 = S;
11. for(i=0; i<N-1; i++) {
12.   S1 = S1 + A1[i];
13. }
14. S1 = S1 + A1[N-1];

assert(S1 = N × (N+2))
```

```
assume(∀i∈[0,N) A[i] = 1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + 1;
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + A[N-1];
7.  S1 = S1 + (N-1);
8.  S1 = S1 + A1[N-1];

assert(S1 = N × (N+2))
```

Simplified ∂P_N

Strengthening Pre and Post




```
assume(A[N-1] = 1)                    //∂φ(N)
assume(S1_Nm1 = (N-1)×(N+1))          //ψ(N-1)
assume(∀i∈[0,N-1) A1_Nm1[i] = N)
assume(S_Nm1 = N-1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + 1;
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + A[N-1];
7.  S1 = S1 + (N-1);
8.  S1 = S1 + A1[N-1];

assert(S1 = N×(N+2))                  //ψ(N)
assert(∀i∈[0,N) A1[i] = N+1)
assert(S = N)
```

Divyesh Unadkat

Vajra - Making the Dragons Dance

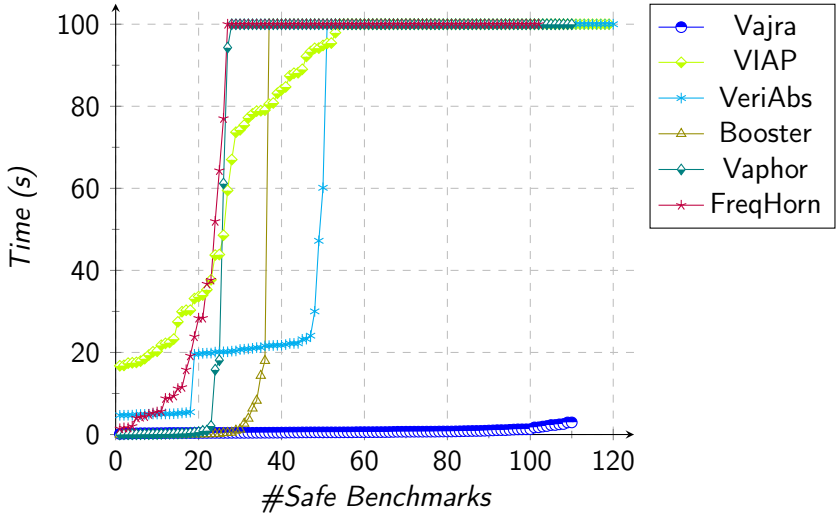
Vajra Artifact

https://doi.org/10.6084/
m9.figshare.11875428.v1

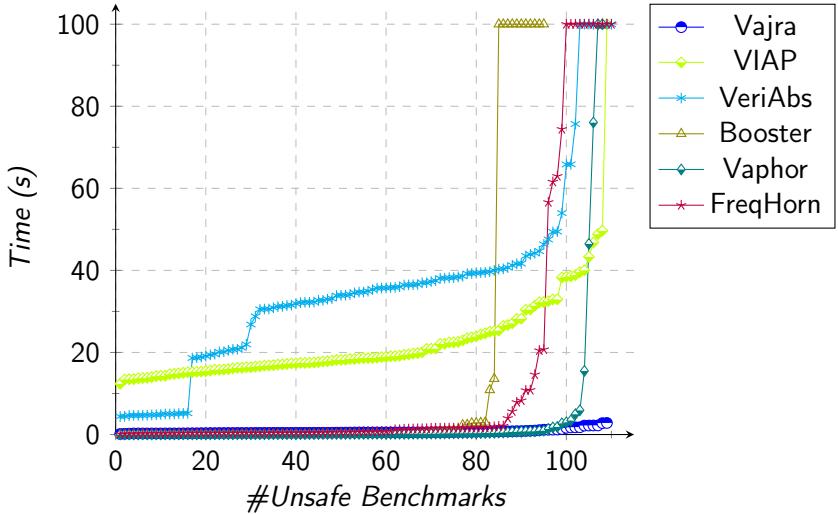
- Evaluation on 231 challenging benchmarks; performance comparison with:
  - ⋆ VIAP v1.1, ⋆ VeriAbs v1.3.10, ⋆ Booster v0.2, ⋆ Vaphor v1.2, ⋆ FreqHorn v3
- Verified 110/121 safe, 109/110 unsafe, inconclusive on 12 programs
- Intel i7-6500U CPU, 2.5 GHz, 16GB RAM, Ubuntu 18.04.5 LTS; Time limit - 60s

Divyesh Unadkat

Trends in Safe Benchmarks



Trends in Unsafe Benchmarks



Difficulties and Challanges

```
x = 0;
for(i=0; i<N; i++)
    x = x + N*N;
    a[i] = a[i] + N;

for(j=0; j<N; j++)
    b[j] = x + j;
```

P_N

Non-trivial construction of ∂P_N in some cases

Computed ∂P_N may have loops

- Two loops in this example

Analyzing ∂P_N as hard as P_N

Recursive application of FPI on ∂P_N may be required even for non-nested loops

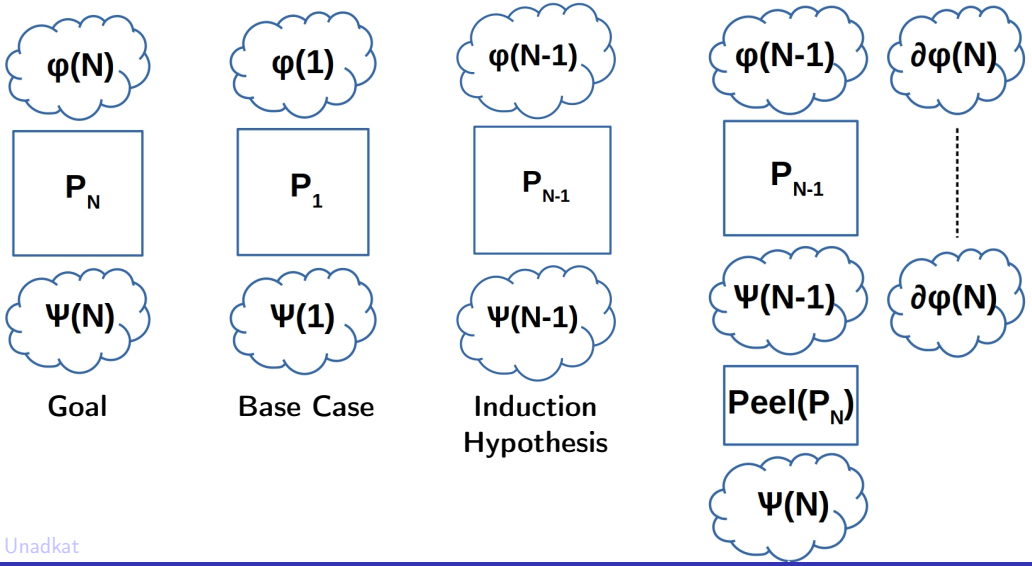
```
x = 0;
for(i=0; i<N-1; i++)
    x = x+(N-1)*(N-1);
    a[i] = a[i] + N-1;
for(j=0; j<N-1; j++)
    b[j] = x + j;
```

P_{N-1}

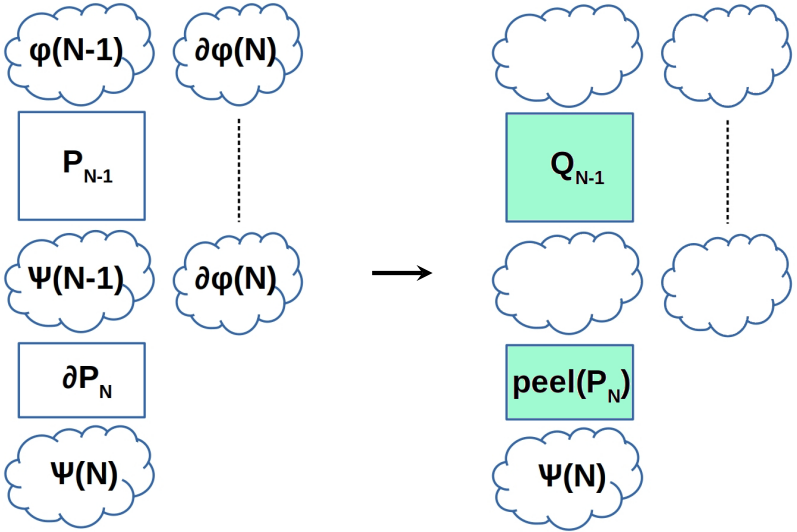
```
for(i=0; i<N-1; i++)
    x = x + 2*N-1;
    a[i] = a[i] + 1;
x = x + N*N;
a[N-1] = a[N-1]+N;
for(k=0; k<N-1; k++)
    b[k] = b[k] +
    (N-1)*(2*N-1)+N*N;
b[N-1] = x + N-1;
```

∂P_N

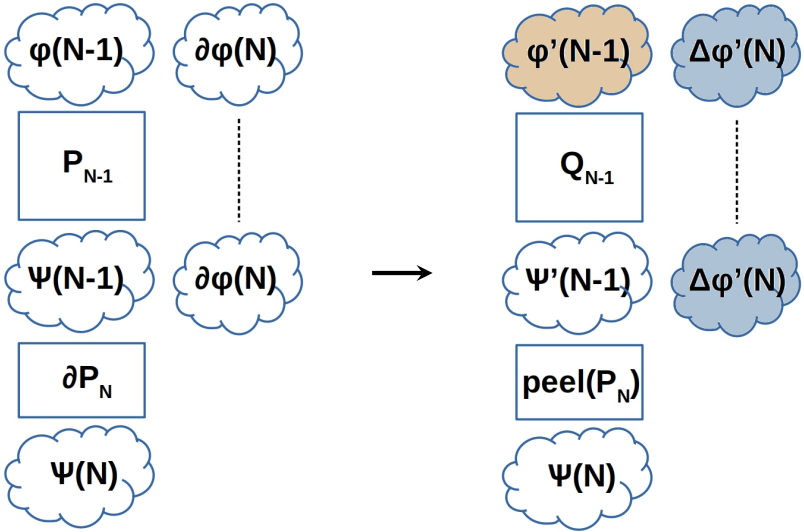
Full-Program Induction in Simple Cases



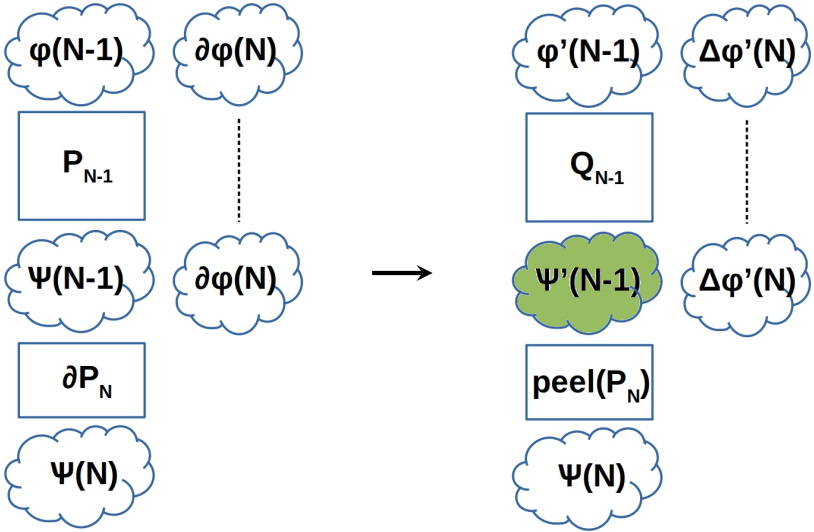
Evolution of 'Difference Computation'



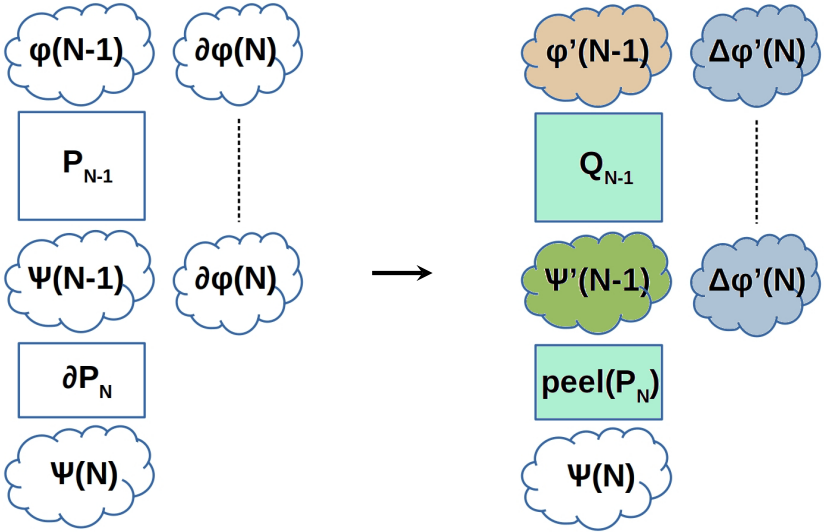
Evolution of 'Difference Computation'



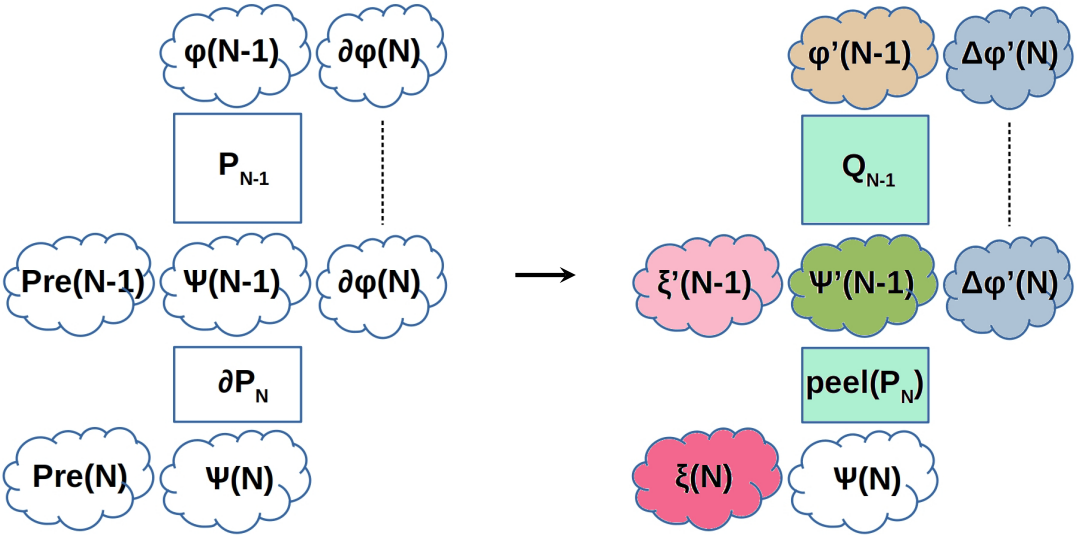
Evolution of 'Full-Program Induction'



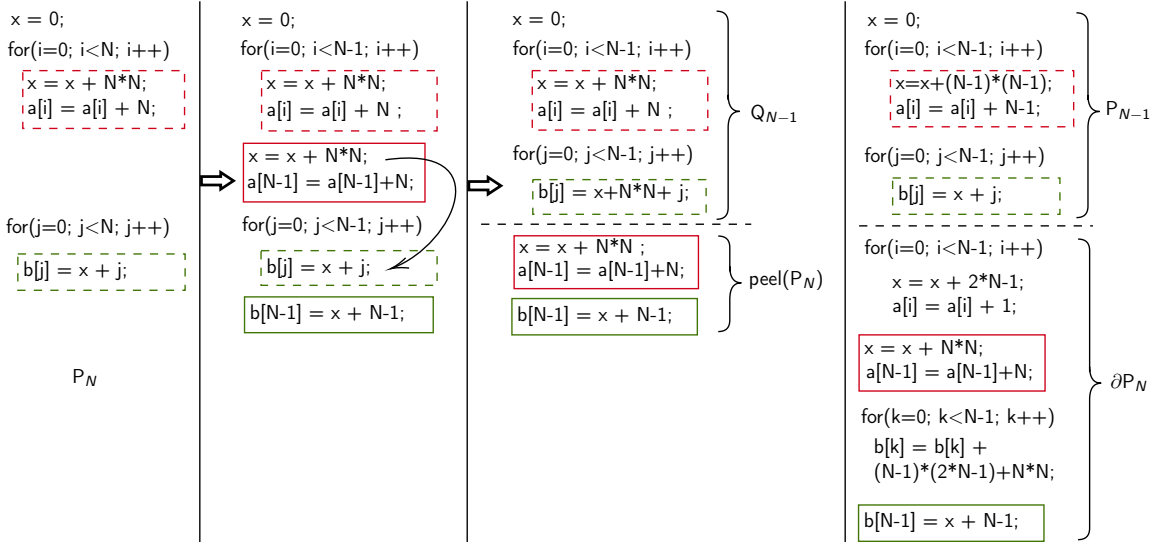
Evolution of 'Full-Program Induction'



Evolution of 'Full-Program Induction'



Decomposing P_N into Q_{N-1}; peel(P_N)



Divyesh Unadkat

Correctness and Progress

Theorem (Correctness of the Decomposition)

$$Q_{N-1}; \textbf{peel}(P_N) \equiv P_N$$

Lemma (Reduction in Verification Complexity)

For a class of programs where upper bound expressions of loops are linear in N & loop counters

Max loop nesting depth in $\text{peel}(P_N)$ $<$ **Max loop nesting depth in** P_N

Relating Q_{N-1} and P_{N-1}

Difference invariants $D(V_Q, V_P)$:

ϕ(N-1) $\equiv \forall$i$\in$[0,N-1) a[i] = N-1

```
void foo(int a[], int N)
{
  int b[N-1], x = 0;

  for(int i=0; i<N-1; i++) {
    x = x + a[i];
  }

  for(int j=0; j<N-1; j++) {
    b[j] = x + j;
  }
}
```

P_{N-1}

ψ(N-1)$\equiv\forall$i$\in$[0,N-1) b[i]=i+(N-1)^2

$\forall i \in [0, N-1),\ a'[i] - a[i] = 1$

$x' - x = N-1$

$\forall j \in [0, N-1),\ b'[j] - b[j] = 2 \times N - 1$

$\psi'(N-1) \equiv \exists V_P.\ (\psi(N-1) \wedge D(V_Q, V_P))$

ϕ'(N-1) $\equiv \forall$i$\in$[0,N-1) a'[i] = N

```
void bar(int a'[], int N)
{
  int b'[N-1], x' = 0;

  for(int i=0; i<N-1; i++) {
    x' = x' + a'[i];
  }

  for(int j=0; j<N-1; j++) {
    b'[j] = x' + N + j;
  }
}
```

Q_{N-1}

ψ'(N-1)$\equiv\forall$i$\in$[0,N-1) b'[i]=i+N^2

Verification using Evolved Full-Program Induction

tcs Research

Base Case	Inductive Step	Strengthening
$\phi(1) \equiv \forall i \in [0,1)$ a[i] = 1	$\partial\phi'(N) \equiv$ a[N-1] = N $\psi'(N-1) \equiv \forall i \in [0,N-1)$ b[i] = i+N^2	$\partial\phi'(N) \equiv$ a[N-1] = N $\psi'(N-1) \equiv \forall i \in [0,N-1)$ b[i] = i+N^2 $\xi'(N-1) \equiv$ x = N^2 - N

void foo(int a[], int N)
{
int b[N], x = 0;
 	
for(int i=0; i<1; i++) {
x = x + a[i];
}
 	
for(int j=0; j<1; j++) {
b[j] = x + j;
}
}
	x = x + a[N-1];
	
	b[N-1] = x + N-1;
		x = x + a[N-1];
		
		b[N-1] = x + N-1;

$\psi(1) \equiv \forall i \in [0,1)$ b[i] = i+1	$\psi(N) \equiv \forall i \in [0,N)$ b[i] = i+N^2	$\xi(N) \equiv$ x = N^2 $\psi(N) \equiv \forall i \in [0,N)$ b[i] = i+N^2

Soundness

Theorem (Soundness of Evolved Full-Program Induction)

1) $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds for* $1 \leq N \leq M$, *for some* $M > 0$
2) $\xi(N) \wedge D(V_\mathsf{Q}, V_\mathsf{P}) \Rightarrow \xi'(N)$ *holds for all* $N > 0$
3) $\{\xi'(N-1) \wedge \Delta\varphi'(N)\ \wedge\ \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\xi(N) \wedge \psi(N)\}$ *holds for all* $N \geq M$

$$\boldsymbol{\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}\ \text{holds for all}\ N > 0}$$

Proving Post-conditions with Arbitrary Quantifiers

- No data-dependence on N simplifies verification significantly
- Corresponding vars in Q_{N-1} and P_{N-1} are equal; $D(V_{\mathrm{Q}}, V_{\mathrm{P}})$ is restricted to $V' = V$
- $\psi'(N-1) \equiv \psi(N-1)$ (same post-condition for Q_{N-1} and P_{N-1})
- $\xi'(N-1) \equiv \xi(N-1)$ (simplifies the strengthening step)

Theorem (Relative Completeness)

FPI is `sound and relatively complete` *for post-conditions with arbitrary quantifiers when*

- *programs have only non-nested loops*
- *variables and arrays computed in the program do not depend on N*

Completeness is relative to the capabilites of the underlying SMT solver

Diffy - Dance of the Evolved Dragons

Diffy Artifact

https://doi.org/10.6084/m9.figshare.14509467

- Experiments on 300+ benchmarks; performance comparison with:
  - ⋆ Vajra v1.0 - Chakraborty et al.'20
  - ⋆ VeriAbs v1.4.1-12 - Afzal et al.'20
  - ⋆ VIAP v1.1 - Rajkhowa & Lin'19
- Intel i7-6500U CPU, 2.5 GHz, 16GB RAM, Ubuntu 18.04.5 LTS; Time limit - 60s

Divyesh Unadkat

Diffy - Experimental Evaluation

Program Category		Diffy			Vajra		VeriAbs		VIAP		
		S	U	TO	S	U	S	TO	S	U	TO
Safe C1	110	110	0	0	110	0	96	14	16	1	93
Safe C2	24	21	0	3	0	24	5	19	4	0	20
Safe C3	23	20	3	0	0	23	9	14	0	23	0
Total	157	**151**	3	3	110	47	110	47	20	24	113
Unsafe C1	99	98	1	0	98	1	84	15	98	0	1
Unsafe C2	24	24	0	0	17	7	19	5	22	0	2
Unsafe C3	23	20	3	0	0	23	22	1	0	23	0
Total	146	**142**	4	0	115	31	125	21	120	23	3

Table: Summary. S is successful result. U is inconclusive result. TO is timeout of 60s.

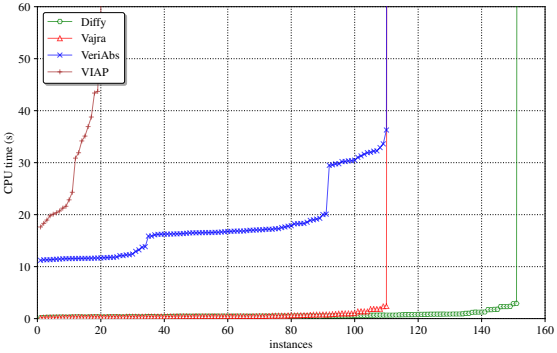
C1 - programs with standard array operations such as min, max, copy

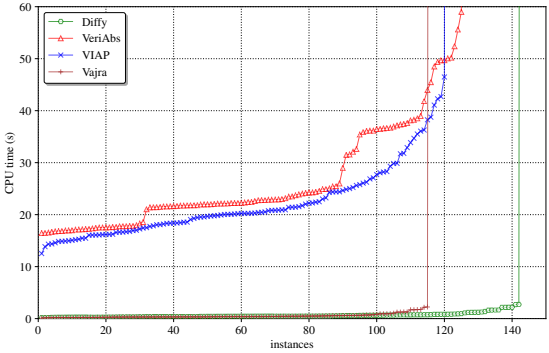
C2 - branch conditions affected by N, modulo operators in programs

C3 - nested loop programs

Diffy out-performs *winners* of SV-COMP from Arrays sub-category

Diffy - Performance Evaluation





★ Diffy is 10× faster than VIAP, VeriAbs; proves more benchmarks than Vajra

★ Violations are reported by simple bmc when base case fails

Conclusion

- ★ Full-Program Induction a novel and efficient verification technique
- ★ Adapts induction in ways different from classical methods
- ★ Prototyped in the verification tools Vajra & Diffy
- ★ Outperforms state-of-the-art tools and techniques
- ★ Vajra & Diffy are incorporated in VeriAbs, a portfolio verifier from TCS Research
  - ⋆ Recurring winner of the International Software Verification Competition (SV-COMP)

- ◇ Future Prospects
  - ▸ Support for larger classes of programs and properties
  - ▸ Expand scope of inductive reasoning beyond arrays

Thank you