VerifyThis Benchmarks for SV-COMP

Gidon Ernst, (Gregor Alexandru)

LMU Munich, Germany
gidon.ernst@lmu.de

CPA 2019

VerifyThis

- On-site program verification competition
  - FoVeOOS 2011, FM 2012, ETAPS 2015–2019
  - permanent SC; changing organizers each year
  - $\approx$20 participants per event, in teams
- Goals: describe state-of-the-art; community exchange
- 1st day: Tutorial, then 3 Challenges, 90 min each
- 2nd day: Discussions: judging $\|$ plenum

VerifyThis

- On-site program verification competition
  - FoVeOOS 2011, FM 2012, ETAPS 2015–2019
  - permanent SC; changing organizers each year
  - ≈20 participants per event, in teams
- Goals: describe state-of-the-art; <u>community exchange</u>
- 1st day: Tutorial, then 3 Challenges, 90 min each
- 2nd day: Discussions: judging ‖ plenum
- Plenty of interesting verification challenges!
  But: hard for SV-COMP verifiers
  (need quantifiers, \old, abstractions, inductive lemmas)

VerifyThis

- On-site program verification competition
  - FoVeOOS 2011, FM 2012, ETAPS 2015–2019
  - permanent SC; changing organizers each year
  - $\approx$20 participants per event, in teams
- Goals: describe state-of-the-art; community exchange
- 1st day: Tutorial, then 3 Challenges, 90 min each
- 2nd day: Discussions: judging $\parallel$ plenum
- Plenty of interesting verification challenges!
  But: hard for SV-COMP verifiers
  (need quantifiers, \old, abstractions, inductive lemmas)
- You might want to be able to solve them:
  6 challenges from 2011/2012 now in sv-benchmarks

Challenge Descriptions

- Given: pseudocode + natural language spec
- Task: write down algorithm, formalize spec, prove correctness
- Characteristics: arrays, heap, concurrency
- Some tools: Dafny, Why3, VeriFast, KeY, KIV, VerCors, Isabelle, Frama-C, Viper, mCRL2, CIVL, ...
- `www.pm.inf.ethz.ch/research/verifythis.html`

Example Challenge: Linked Tree

```
struct node *tree_del(struct node *t, int *min) {
    struct node *r;
    if (!t->left) {
        *min = t->data; r = t->right; free(t); return r;
    } else {
        t->left = tree_del(t->left, min);      return t;
    }
}

void task(struct node *t) {
    int n = size(t);
    int a = min(t), b;  int x[n], y[n-1];
    tree_inorder(t, x);
    assert(a == x[0]);

    struct node *r = tree_del(t, &b);
    assert(a == b);
    tree_inorder(t, y);

    int i;
    assume(0 <= i < n-1);
    assert(x[i+1] == y[i]);
}
```

Example Challenge: Elimination Max

```
int max(int a[n], int n) {
    assume(0 < n);

    int i = 0, j = n-1;
    while(i < j) {
        if(a[i] < a[j]) i++; else j--;
    }

    int k;
    assume(0 <= k < n);
    assert(a[k] <= a[i]);
    return i;
}
```

Example Challenge: Elimination Max

```
int max(int a[n], int n) {
    assume(0 < n);

    int i = 0, j = n-1;
    while(i < j) {
        if(a[i] < a[j]) i++; else j--;
    }

    int k;
    assume(0 <= k < n);
    assert(a[k] <= a[i]);
    return i;
}
```

- Invariant (Why3)
  $\forall k.\ (0 \leq k < i \vee j < k < n) \Longrightarrow (a[k] \leq a[i] \wedge a[k] \leq a[j])$

Example Challenge: Elimination Max

```
int max(int a[n], int n) {
    assume(0 < n);

    int i = 0, j = n-1;
    while(i < j) {
        if(a[i] < a[j]) i++; else j--;
    }

    int k;
    assume(0 <= k < n);
    assert(a[k] <= a[i]);
    return i;
}
```

- Invariant (Why3)
  $\forall k.\ (0 \leq k < i \lor j < k < n) \implies (a[k] \leq a[i] \land a[k] \leq a[j])$
- Invariant (KIV)
  $\exists k.\ (0 \leq i \leq k \leq j < n) \land a[k] = max\{a[0], .., a[n-1]\}$

Example Challenge: Elimination Max

```
int max(int a[n], int n) {
    assume(0 < n);

    int i = 0, j = n-1;
    while(i < j) {
        if(a[i] < a[j]) i++; else j--;
    }

    int k;
    assume(\old(i) <= k < \old(j)+1);
    assert(a[k] <= a[i]);
    return i;
}
```

- Invariant (Why3)
  $\forall k.\ (0 \leq k < i \vee j < k < n) \Longrightarrow (a[k] \leq a[i] \wedge a[k] \leq a[j])$
- Invariant (KIV)
  $\exists k.\ (0 \leq i \leq k \leq j < n) \wedge a[k] = max\{a[0], .., a[n-1]\}$
- Sufficient: invariant $0 \leq i \leq j < n$ & induction & `\old`

Demo

Recursive Variant

```
int max(int a[n], int i, int j, int n, int k) {
    int m;
    if(i >= j) {
        m = i;
    } else {
        if(a[i] < a[j]) i++; else j--;
        m = max(a, i, j, n, k);
    }

    if(i <= k <= j);
        assert(a[k] <= a[m]);
    return m;
}
```

Note trick: fix the k upfront which we want to compare!

Recursive Variant

```
int max(int a[n], int i, int j, int n, int k) {
    int m;
    if(i >= j) {
        m = i;
    } else {
        if(a[i] < a[j]) i++; else j--;
        m = max(a, i, j, n, k);
    }

    if(i <= k <= j);
        assert(a[k] <= a[m]);
    return m;
}
```

Note trick: fix the k upfront which we want to compare!

Should be in reach of SV-COMP verifiers:
no quantifier in procedure summary

Reasoning about Loops

- Hoare's invariant rule

$$\frac{\{I \land c\}\ \ \texttt{p}\ \ \{I\}}{\{I\}\ \ \texttt{while(c)\{p\}}\ \ \{I \land \neg c\}}$$

Reasoning about Loops

- Hoare's invariant rule

$$\frac{\{I \land c\}\ \ \texttt{p}\ \ \{I\}}{\{I\}\ \ \texttt{while(c)\{p\}}\ \ \{I \land \neg c\}}$$

- Loop specifications (e.g. Morgan, Hehner, Tuerk)
- Intuition: treat loop as recursive procedure, use induction
  - base case = loop termination
  - step case = unwind once, assume that rest of program will be safe after remaining iterations

Reasoning about Loops

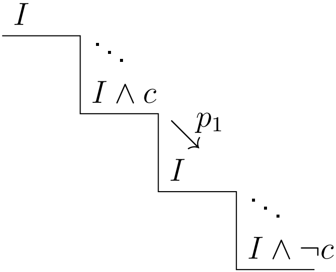
- Hoare's invariant rule

$$\frac{\{I \land c\}\ \ \texttt{p}\ \ \{I\}}{\{I\}\ \ \texttt{while(c)\{p\}}\ \ \{I \land \neg c\}}$$

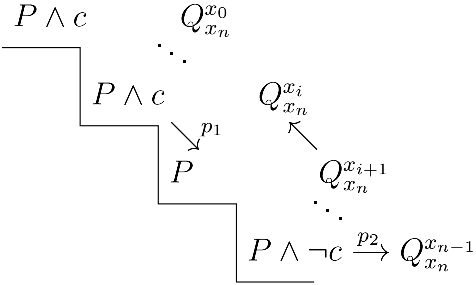
- Loop specifications (e.g. Morgan, Hehner, Tuerk)
- Intuition: treat loop as recursive procedure, use induction
  - base case = loop termination
  - step case = unwind once, assume that rest of program will be safe after remaining iterations

$$\frac{\begin{array}{c}\{P_x \land \neg c \land x_0 = x\}\ \ \texttt{p}_2\ \ \{Q_x^{x_0}\} \\ \{P_x \land c \land x_i = x\}\ \ \texttt{p}_1\ \ \{P_x \land Q_{x_n}^{x_i} \Longrightarrow Q_{x_n}^{x}\}\end{array}}{\{P_x \land x_0 = x\}\ \ \texttt{while(c)\{p}_1\texttt{\}; p}_2\ \ \{Q_x^{x_0}\}}$$

Comparison of Proof Structure



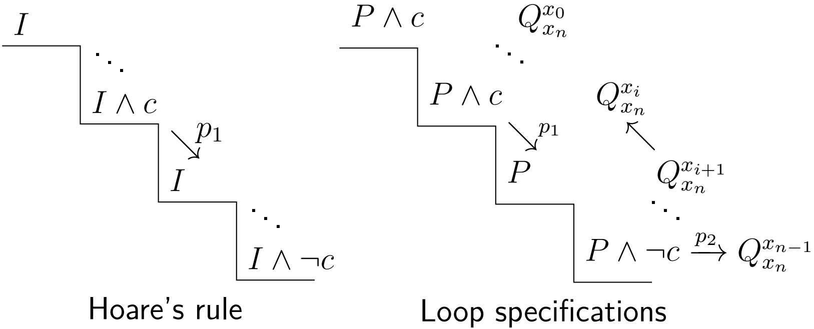
Hoare's rule



Loop specifications

(Illustration by Gregor Alexandru)

Comparison of Proof Structure



(Illustration by Gregor Alexandru)

not like k-Induction

(which does not use hypothesis about remaining execution)

Proof Sketch

- Base case:

```
assume(0 < i₀ <= k <= j₀ < n);
assume(i₀ >= j₀);
assert(a[k] <= a[i₀]);
```

Proof Sketch

- Base case:
```
assume(0 < i₀ <= k <= j₀ < n);
assume(i₀ >= j₀);
assert(a[k] <= a[i₀]);
```
- Step case:
```
assume(0 < i₀ <= k <= j₀ < n);
assume(i₀ < j₀);
if(a[i₀] < a[j₀]) i₁ = i₀+1; else j₁ = j₀-1;
assume(IndHyp);
assert(a[k] <= a[iₙ]); // iₙ denotes final result
```
where IndHyp is that for <u>all</u> k this is safe:
```
assume(0 < i₁ <= k <= j₁ < n); // different bounds!
assert(a[k] <= a[iₙ]);
```

Onwards to new and difficult tasks!

- Some benchmarks are available, more to come
  - Properties $\rightsquigarrow$ (recursive) procedures
  - Quantifiers $\rightsquigarrow$ nondeterministic choice

Onwards to new and difficult tasks!

- Some benchmarks are available, more to come
  - Properties $\rightsquigarrow$ (recursive) procedures
  - Quantifiers $\rightsquigarrow$ nondeterministic choice
- Loop specifications:
  - Can be taken from assertions/postconditions, mild generalizations necessary (btw: old not required in assertion language)
  - Actual loop invariants typically simple
  - Alternative to existing summarization techniques?

Onwards to new and difficult tasks!

- Some benchmarks are available, more to come
  - Properties $\rightsquigarrow$ (recursive) procedures
  - Quantifiers $\rightsquigarrow$ nondeterministic choice
- Loop specifications:
  - Can be taken from assertions/postconditions, mild generalizations necessary (btw: old not required in assertion language)
  - Actual loop invariants typically simple
  - Alternative to existing summarization techniques?
- (watch out for bugs in the slides/tasks)

Big thanks to: Vladimir Klebanov, Marieke Huisman, Rosemary Monahan, Peter Müller, Mattias Ulbrich, and all VerifyThis organizers.