Hypersafety Verification and Programming Assignment Evaluation

Kumar Madhukar

TCS Research

Indian Institute of Technology Goa

January 21, 2021

Assignment Evaluation and Hypersafety

- problem of evaluating an assignment submission, given a reference implementation
- property: for the same input, the outputs always match
- can be asserted in a composed program, but not easy to verify
- such proofs often require that the functionality of every component program be captured fully
- background: a *k*-safety (hypersafety) property is a program safety property whose violation is witnessed by at least *k* finite runs of a program (e.g. determinism is a 2-safety property)

[1] Jude Anil, Sumanth Prabhu, **M**, and R Venkatesh. 2020. Using hypersafety verification for proving correctness of programming assignments. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20).

[2] ongoing work with Akshatha Shenoy, Sumanth Prabhu, Ron Shemer, and Mandayam Srivas

Assignment Evaluation and Hypersafety

- such proofs often require that the functionality of every component program be captured fully

```
sum-v1 (int n)

  s1 = 0; i = 1;

  // 2*s1 == i(i-1)
  while (i <= n)
    s1 = s1 + i;
    i = i + 1;

  return s1;
```

```
sum-v2 (int m)

  s2 = 0; j = 1;

  // 2*s2 == j(j-1)
  while (j <= m)
    s2 = s2 + j;
    j = j + 1;

  return s2;
```

```
pre: (n == m)

// (i == j) & (s1 == s2)
while ((i <= n) || (j <= m))
  if (i <= n)
    s1 = s1 + i; i = i + 1
  if (j <= m)
    s2 = s2 + j; j = j + 1

post: (s1 == s2)
```

- hypersafety verification techniques also face this challenge

[1] Jude Anil, Sumanth Prabhu, **M**, and R Venkatesh. 2020. Using hypersafety verification for proving correctness of programming assignments. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20).

[2] ongoing work with Akshatha Shenoy, Sumanth Prabhu, Ron Shemer, and Mandayam Srivas

Assignment Evaluation and Hypersafety

- such proofs often require that the functionality of every component program be captured fully

```
sum-v1 (int n)

  s1 = 0; i = 1;

  // 2*s1 == i(i-1)
  while (i <= n)
    s1 = s1 + i;
    i = i + 1;

  return s1;
```

```
sum-v2 (int m)

  s2 = 0; j = 1;

  // 2*s2 == j(j-1)
  while (j <= m)
    s2 = s2 + j;
    j = j + 1;

  return s2;
```

```
pre: (n == m)

// (i == j) & (s1 == s2)
while ((i <= n) || (j <= m))
  if (i <= n)
    s1 = s1 + i; i = i + 1
  if (j <= m)
    s2 = s2 + j; j = j + 1

post: (s1 == s2)
```

- hypersafety verification techniques also face and partially address this challenge

[1] Jude Anil, Sumanth Prabhu, **M**, and R Venkatesh. 2020. Using hypersafety verification for proving correctness of programming assignments. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20).

[2] ongoing work with Akshatha Shenoy, Sumanth Prabhu, Ron Shemer, and Mandayam Srivas

Property Directed Self Composition

Ron et al., CAV 2019

- if some interleaving violates the postcondition, then all of them will
- any self-composition is sufficient to reduce *k*-safety to safety (e.g. lockstep, sequential)
- different self-composed programs would require different (safe) inductive invariants
- find the "right" composition, and the inductive invariant for that
- work in a restricted language $\mathcal{L}$
- explore all compositions, discarding "bad" ones (that cannot have inductive invariants in $\mathcal{L}$)

Property Directed Self Composition

- if some interleaving violates the postcondition, then all of them will
- any self-composition is sufficient to reduce k-safety to safety (e.g. lockstep, sequential)
- different self-composed programs would require different (safe) inductive invariants
- find the "right" composition, and the inductive invariant for that
- work in a restricted language $\mathcal{L}$ fixed, and user-supplied
- explore all compositions, discarding "bad" ones (that cannot have inductive invariants in $\mathcal{L}$)

```
doubleSquare-v1(x)
  int z, y=0;
  z = 2*x;

  while (z>0)
    z -= 1;
    y = y+x;

  return y;
```

```
doubleSquare-v2(x)
  int z, y=0;
  z = x;

  while (z>0)
    z -= 1;
    y = y+x;

  y = 2*y
  return y;
```

pre:

```
 (x1 > 0) & (x2 > 0)
 (y1 == 0) & (y2 == 0)
 (z1 == 2*x1) & (z2 == x2)
 (x1 == x2)
```

post: (y1 == y2)

user-supplied predicates:

```
 (z1 == 2*z2),(z1 == 2*z2-1)
 (y1 == 2*y2),(y1 == 2*y2+x2)
```

- not all compositions are easy to prove
- the lockstep composition these does not even have a safe inductive invariant in LIA

DoubleSquare

```
doubleSquare-v1(x)
  int z, y=0;
  z = 2*x;

  while (z>0)
    z -= 1;
    y = y+x;

  return y;
```

```
doubleSquare-v2(x)
  int z, y=0;
  z = x;

  while (z>0)
    z -= 1;
    y = y+x;

  y = 2*y
  return y;
```

```
pre:
 (x1 > 0) & (x2 > 0)
 (y1 == 0) & (y2 == 0)
 (z1 == 2*x1) & (z2 == x2)
 (x1 == x2)

post: (y1 == y2)

user-supplied predicates:
 (z1 == 2*z2),(z1 == 2*z2-1)
 (y1 == 2*y2),(y1 == 2*y2+x2)
```

- an “easy” proof if we compose two loop iterations of `v1` with one of `v2`

DoubleSquare

```
doubleSquare-v1(x)
  int z, y=0;
  z = 2*x;

  while (z>0)
    z -= 1;
    y = y+x;

  return y;
```

```
doubleSquare-v2(x)
  int z, y=0;
  z = x;

  while (z>0)
    z -= 1;
    y = y+x;

  y = 2*y
  return y;
```

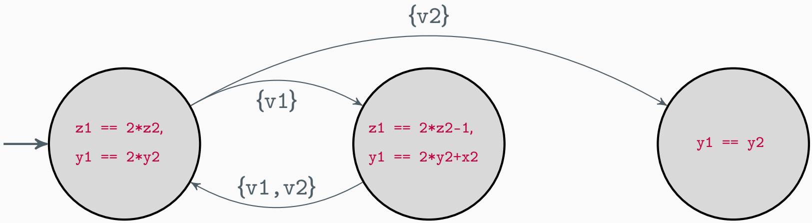
pre:

```
(x1 > 0) & (x2 > 0)
(y1 == 0) & (y2 == 0)
(z1 == 2*x1) & (z2 == x2)
(x1 == x2)
```

post: `(y1 == y2)`

user-supplied predicates:

```
(z1 == 2*z2),(z1 == 2*z2-1)
(y1 == 2*y2),(y1 == 2*y2+x2)
```



Semantic Self Composition Function

- program semantics as transition systems $T = (S, R, F)$
- every terminal state (in F) has only one outgoing transition to itself
- $f : S^k \to \mathbb{P}(\{1..k\})$ maps each state to a set of copies that run next
- represented as a set of logical conditions, C_M for every non-empty subset $M \subseteq \{1..k\}$
- $f(s_1, ..., s_k) = M \iff (s_1, ..., s_k) \models C_M$
- f must also be *well-defined* and *fair*

Finding Composition-Invariant Pair

- $T^f = (S^k, R^f, F^k)$
- R^f includes a transition from $(s_1, ..., s_k)$ to $(s'_1, ..., s'_k)$ *iff*
  - $f(s_1, ..., s_k) = M$ and
  - $(\forall i \in M.\, (s_i, s'_i) \in R) \wedge (\forall i \notin M.\, s_i = s'_i)$
- finding a composition-invariant pair (f, Inv)
  - undecidable in general; fix a language
  - a set of predicates $\mathcal{P}$ and their boolean combinations ($\mathcal{L}_\mathcal{P}$)
- a transition system has an inductive invariant in $\mathcal{L}_\mathcal{P}$ *if and only if* its abstraction using $\mathcal{P}$ is safe

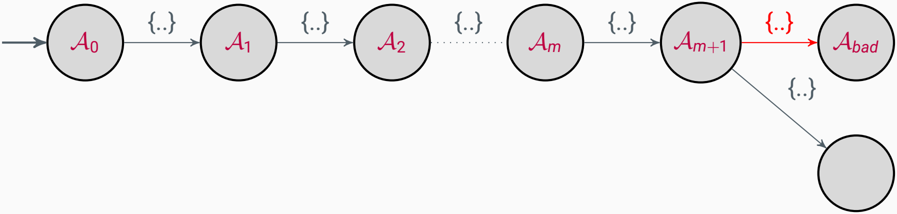
The PDSC Algorithm

- initialize the composition function to lockstep (default)
- abstract T^f with the predicates $\mathcal{P}$
- check if it is possible to start from *pre* and violate the *post*
- if not, then proved
- else, take the trace, modify composition, and try again
- if no more compositions left to try
    - return (language is insufficient)

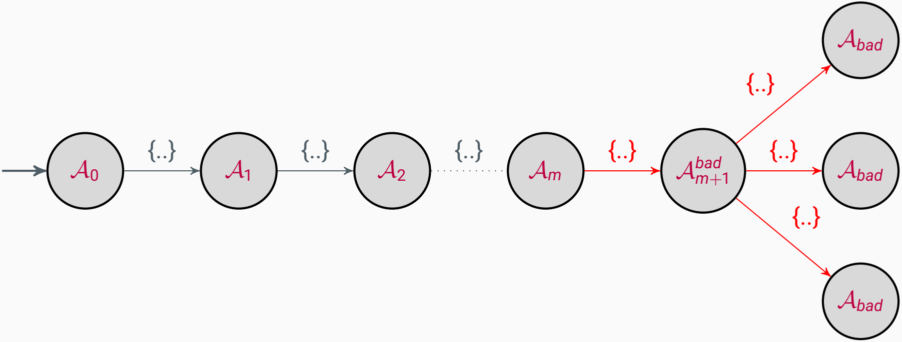
Finding the right composition

disallow the (state, composition) pair reaching bad

Finding the right composition

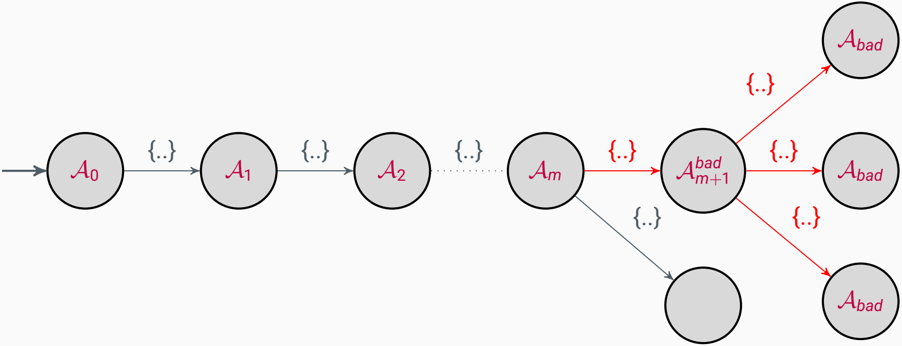


Finding the right composition



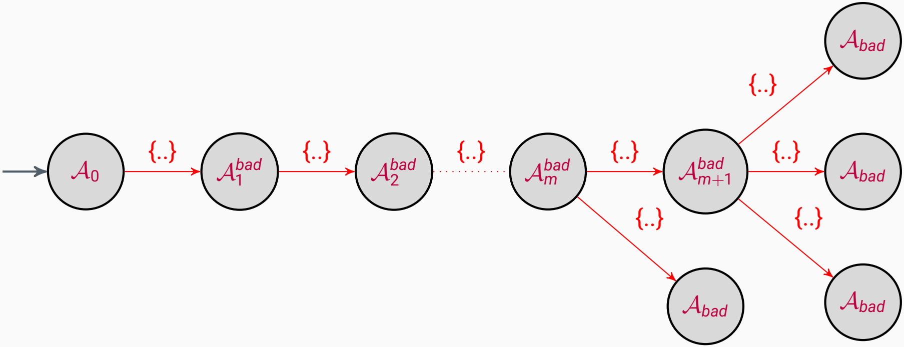
if all compositions disallowed from a state, mark it as bad

Finding the right composition



and so on ...

Finding the right composition



until the initial state must be marked bad

return, i.e. no composition-invariant pair exists (in the given language)

Extending PDSC with Refinement

- initialize the composition function to lockstep (default)
- abstract T^f with the predicates $\mathcal{P}$
- check if it is possible to start from *pre* and violate the *post*
- if not, then proved
- else, take the trace, modify composition, and try again
- if no more compositions left to try
    - return (language is insufficient)

Extending PDSC with Refinement

- initialize the composition function to lockstep (default)
- abstract T^f with the predicates $\mathcal{P}$
- check if it is possible to start from *pre* and violate the *post*
- if not, then proved
- else, take the trace, modify composition, and try again
- if no more compositions left to try
  - ~~return (language is insufficient)~~
  - check if the abstract trace is spurious; if not, return unsafe (and counterexample)
  - if yes, add a predicate to remove the spurious transition, and restart the search

- spurious transition $\langle a_{src}, tr, a_{tgt} \rangle$

$$a_{src}(X) \wedge tr(X, X') \not\Rightarrow \neg a_{tgt}(X')$$

$$p(Y \subseteq X) \wedge a_{src}(X) \wedge tr(X, X') \Rightarrow \neg a_{tgt}(X')$$

$$p(Y \subseteq X) \wedge a_{src}(X) \wedge tr(X, X') \not\Rightarrow \bot$$

- the problem of abductive inference

$$\forall\, ((X \cup X') \setminus Y).\ a_{src}(X) \wedge tr(X, X') \Rightarrow \neg a_{tgt}(X')$$

$$\exists\, ((X \cup X') \setminus Y).\ a_{src}(X) \wedge tr(X, X') \wedge a_{tgt}(X')$$

- solve for $p(Y)$ using SyGuS and SMT solvers (CVC4 and Z3)

Claim

The refinement ensures progress, i.e. the synthesized predicate eliminates the spurious transition.

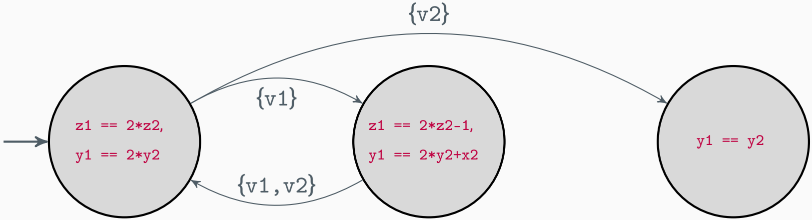
```
doubleSquare-v1(x)
  int z, y=0;
  z = 2*x;

  while (z>0)
    z -= 1;
    y = y+x;

  return y;
```

```
doubleSquare-v2(x)
  int z, y=0;
  z = x;

  while (z>0)
    z -= 1;
    y = y+x;

  y = 2*y
  return y;
```

pre:
(x1 > 0) & (x2 > 0)
(y1 == 0) & (y2 == 0)
(z1 == 2*x1) & (z2 == x2)
(x1 == x2)

post: (y1 == y2)

refinement predicates:
(z1 == 2*z2),(z1 == 2*z2-1)
(y1 == 2*y2),(y1 == 2*y2+x2)



Implementation

- implemented our ideas in the pdsc tool
- SyGuS (CVC4-1.8) gets nice-looking predicates, but is slower
- QE (Z3) works quicker, but the predicates can be big formulas
  - eliminate more variables to get shorter expressions

Experiments

S. No.	Benchmark	Source	Safe/Unsafe	SyGuS (#pred, time)	QE (#preds, time)
1.	sum_to_n	crafted	safe	timeout	8, 1m30s
2.	sum_to_n_err	crafted	unsafe	0, 1.1s	0, 0.8s
3.	inc-dec	crafted	safe	5, 39 secs	8, 35.8 secs
4.	squareSum	cav19	safe	0, 2.2 secs	0, 1.1 secs
5.	sum-pc	cav19	safe	5, 4m5.3s	1, 11.9 secs
6.	fig4_1	icse16	unsafe	timeout	2, 7.63 secs
7.	fig4_2	icse16	unsafe	timeout	2, 7.65 secs
8.	fig4_ref_ref	icse16	safe	0, 0.8 secs	0, 0.6 secs
9.	subsume_1	icse16	unsafe	timeout	3, 13 secs
10.	subsume_2	icse16	unsafe	timeout	2, 8.8 secs
11.	subsume_ref_ref	icse16	safe	timeout	1, 3.9 secs
12.	puzzle_1	derived from icse16	unsafe	timeout	4, 26.8 secs
13.	puzzle_2	derived from icse16	unsafe	timeout	8, 2m25s
14.	puzzle_3	derived from icse16	safe	timeout	2, 11.9s
15.	halfSquare	cav19	safe	timeout	4, 1m10s
16.	doubleSquare_1	derived from cav19	safe	timeout	6, 1m55s
17.	doubleSquare_2	derived from cav19	safe	timeout	3, 43.8s
18.	doubleSquare_3	derived from cav19	safe	timeout	5, 1m29s

Related work in 2019

- Automated Hypersafety Verification (CAV 2019)
- Semantic Program Alignment for Equivalence Checking (PLDI 2019)
- Property Directed Self Composition (CAV 2019)

Summary

- hypersafety verification/program equivalence/assignment evaluation
- finding correctness proof in an easy-to-prove composition
- need to generalize the discovered predicates
- interpolants from infeasibility proofs

Thanks for your attention.

Questions?

kumar.madhukar@tcs.com