Annotation synthesis for C programs using TriCera

Zafer Esen[1] and Philipp Rümmer[1,2]

[1]Uppsala University
[2]University of Regensburg

EuroProofNet WG3 Meeting
8 February 2023
Timişoara, Romania

Encoding of programs using CHCs

```
int x = _;
while(x > 0){
  x--;
}
assert(x == 0);
```

Encoding of programs using CHCs

```
int x = _; l1
while(x > 0){ l2
  x--;
} l3
assert(x == 0);
```

Encoding of programs using CHCs

```
int x = _; I1
while(x > 0){ I2
  x--;
} I3
assert(x == 0);
```

$$
\begin{aligned}
I_1(x) &\leftarrow \mathit{true}\\
I_2(x) &\leftarrow I_1(x) \wedge x > 0\\
I_1(x-1) &\leftarrow I_2(x)\\
I_3(x) &\leftarrow I_1(x) \wedge x \not> 0\\
\mathit{false} &\leftarrow I_3(x) \wedge x \neq 0.
\end{aligned}
$$

I_1, I_2, I_3 are *uninterpreted* predicates (i.e., program *invariants*).

Encoding of programs using CHCs

```
int x =
while(x > 0
  x--;

assert(x == 0);
```

$I_1 : true$

$I_2 : x \geq 0$

$I_3 : x = 0$

$$
\begin{aligned}
I_1(x) &\leftarrow true \\
I_2(x) &\leftarrow I_1(x) \wedge x > 0 \\
I_1(x-1) &\leftarrow I_2(x) \\
I_3(x) &\leftarrow I_1(x) \wedge x \not> 0 \\
\textcolor{red}{false} &\leftarrow I_3(x) \wedge x \neq 0.
\end{aligned}
$$

I_1, I_2, I_3 are *uninterpreted* predicates (i.e., program *invariants*).

A CHC solver (e.g., ELDARICA, Z3/Spacer) tries to compute a solution...

Encoding of programs using CHCs

```
int x =
while(x > 0
  x--;

assert(x == 0);
```

$l_1 : true$

$l_2 : x \geq 0$

$l_3 : x = 0$

$$
\begin{aligned}
l_1(x) &\leftarrow true \\
l_2(x) &\leftarrow l_1(x) \wedge x > 0 \\
l_1(x-1) &\leftarrow l_2(x) \\
l_3(x) &\leftarrow l_1(x) \wedge x \not> 0 \\
\textit{false} &\leftarrow l_3(x) \wedge x \neq 0.
\end{aligned}
$$

l_1, l_2, l_3 are *uninterpreted* predicates (i.e., program *invariants*).

A CHC solver (e.g., ELDARICA, Z3/Spacer) tries to compute a solution...

... or fails and provides a *counterexample trace* to *false*: e.g., any trace starting with $x < 0$ at l_1.

Counterexample: $true \rightarrow l_1(-1) \xrightarrow{x \not> 0} l_3(-1) \xrightarrow{x \neq 0} \textit{false}$

Encoding of programs using CHCs

```
1  int x =        // I_1 : true
2  while (x > 0)  // I_2 : x ≥ 0
3    x--;
4                 // I_3 : x = 0
5  assert(x == 0);
```

$$
\begin{aligned}
I_1(x) &\leftarrow true \\
I_2(x) &\leftarrow I_1(x) \wedge x > 0 \\
I_1(x-1) &\leftarrow I_2(x) \\
I_3(x) &\leftarrow I_1(x) \wedge x \not> 0 \\
false &\leftarrow I_3(x) \wedge x \neq 0.
\end{aligned}
$$

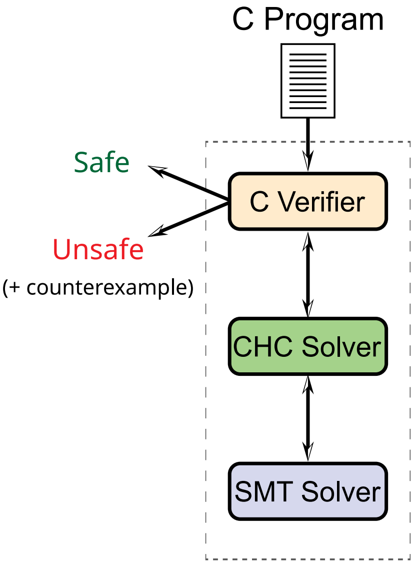
I_1, I_2, I_3 are *uninterpreted* predicates (i.e., program *invariants*).

A CHC solver (e.g., ELDARICA, Z3/Spacer) tries to compute a solution...

... or fails and provides a *counterexample trace* to *false*: e.g., any trace starting with $x < 0$ at I_1.

Counterexample: $true \rightarrow I_1(-1) \xrightarrow{x \not> 0} I_3(-1) \xrightarrow{x \neq 0} false$
`http://logicrunch.it.uu.se:4096/~zafer/tricera/?ex=perma%2F1660054620_1229897514`

Verification of C programs using TriCera and Eldarica



- the TriCera C model checker
  `https://github.com/uuverifiers/tricera`
- the Eldarica CHC Solver
  `https://github.com/uuverifiers/eldarica`
- the Princess SMT Solver
  `https://github.com/uuverifiers/princess`

TRICERA: An open-source verification tool

- Supports a large subset of C11

TriCera: An open-source verification tool

- Supports a large subset of C11
- Assertion-based

```
int foo (int x) {
  int res = x*2;
  assert(res >= x);
  return x;
}
```

TriCera: An open-source verification tool

- Supports a large subset of C11
- Assertion-based (some support for ACSL)

```
/*@
  requires \valid(p, q);
  assigns *p;
*/
void foo(int* p, int* q) {
  *q = 42;
}
```

TriCera: An open-source verification tool

- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors

```
thread Monitor {
  int t = x;
  assert(x >= t);
}
```

TriCera: An open-source verification tool

- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors
- C programs with timing constraints

```
int lock = 0;
thread[tid] Proc {
  clock C;
  assume(tid > 0);

  while (1) {
    atomic {
      assume(lock == 0); C = 0;
    }
    within (C <= 1) { lock = tid; }
    C = 0; assume(C > 1);

    if (lock == tid) {
      assert(lock == tid);
      lock = 0;
    }
  }
}
```

TriCera: An open-source verification tool

- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors
- C programs with timing constraints
- Automatic inference of function contracts and loop invariants

```
/*@ contract @*/
int tak(int x, int y, int z) {
  if (y < x)
    return tak(tak(x-1, y, z),
               tak(y-1, z, x),
               tak(z-1, x, y));
  else return y;
}
```

TriCera: An open-source verification tool

$$f_{pre}: true$$
$$f_{post}: (r \neq z \vee y \geq z \vee x > y) \wedge (r \neq y \vee y \geq z \vee y \geq x) \wedge$$
$$(r = z \vee r = y \vee y > z) \wedge (r = y \vee z \geq y \vee x > y)$$

- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors
- C programs with timing constraints
- Automatic inference of function contracts and loop invariants

TriCera: An open-source verification tool

- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors
- C programs with timing constraints
- Automatic inference of function contracts and loop invariants
- Uninterpreted predicates

```
/*$  p_a(int, int)  $*/
void main () {
  int i, n = _;

  for (i = 0; i < n; ++i) {
    assert(p_a(i, 2*i));
  }
  for (i = 0; i < n; ++i) {
    int v = _;
    assume(p_a(i, v));
    assert(2*i == v);
  }
}
```

TriCera: An open-source verification tool

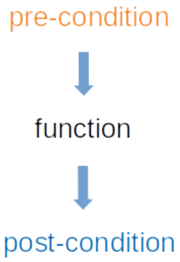
- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors
- C programs with timing constraints
- Automatic inference of function contracts and loop invariants
- Uninterpreted predicates

`https://github.com/uuverifiers/tricera`

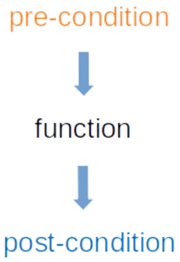
Zafer Esen and Philipp Rümmer
further contributions by Pontus Ernstedt and Hossein Hojjat

Function Contracts

A function's post-condition is guaranteed to hold after the function returns, as long as the function's pre-condition is satisfied.



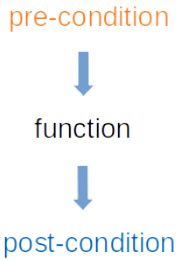
Function Contracts



A function's post-condition is guaranteed to hold after the function returns, as long as the function's pre-condition is satisfied.

Pre-condition: obligation of the caller - assert it at call-sites to function - assume at function entry

Function Contracts

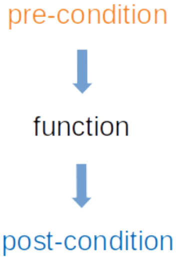


A function's post-condition is guaranteed to hold after the function returns, as long as the function's pre-condition is satisfied.

Pre-condition: obligation of the caller - assert it at call-sites to function - assume at function entry

Post-condition: obligation of the callee (the function) - assert it at function exit - assume it at return point

Function Contracts



A function's post-condition is guaranteed to hold after the function returns, as long as the function's pre-condition is satisfied.

Pre-condition: obligation of the caller - assert it at call-sites to function - assume at function entry

Post-condition: obligation of the callee (the function) - assert it at function exit - assume it at return point

Semantically equivalent to Hoare triples: $\{Q\}$ function call $\{R\}$

Contract inference in TRICERA

```
int f(int n) {
  int s;
  if(n <= 0)
    s = 0;
  else
    s = n + f(n-1);
  return s;
 }

void main() {
  int x, y;
  x = f(y);
  assert(x >= y);
}
```

Contract inference in TriCera

```c
int f(int n) {
  int s;
  if(n <= 0)
    s = 0;
  else
    s = n + f(n-1);
  return s;
 }

void main() {
  int x, y;
  x = f(y);
  assert(x >= y);
}
```

$$f_{post}(n, 0) \leftarrow f_{pre}(n) \wedge n \leq 0$$
$$f_{pre}(n-1) \leftarrow f_{pre}(n) \wedge n > 0$$
$$f_{post}(n, n+s) \leftarrow f_{pre}(n) \wedge f_{post}(n-1, s)$$

$$main_1(x, y) \leftarrow true$$
$$f_{pre}(y) \leftarrow main_1(x, y)$$
$$main_2(s, y) \leftarrow main_1(x, y) \wedge f_{post}(y, s)$$
$$false \leftarrow main_2(x, y) \wedge x < y$$

Contract inference in TriCera

```c
int f(int n) {
  int s;
  if(n <= 0)
    s = 0;
  else
    s = n + f(n-1);
  return s;
 }

void main() {
  int x, y;
  x = f(y);
  assert(x >= y);
}
```

$$f_{post}(n, 0) \leftarrow f_{pre}(n) \wedge n \leq 0$$
$$f_{pre}(n-1) \leftarrow f_{pre}(n) \wedge n > 0$$
$$f_{post}(n, n+s) \leftarrow f_{pre}(n) \wedge f_{post}(n-1, s)$$

$$main_1(x, y) \leftarrow true$$
$$f_{pre}(y) \leftarrow main_1(x, y)$$
$$main_2(s, y) \leftarrow main_1(x, y) \wedge f_{post}(y, s)$$
$$false \leftarrow main_2(x, y) \wedge x < y$$

```
/*@
  requires \true;
  ensures \result >= \old(n) &&
          \result >= 0;
*/
```

(Option “-sol” and “-acsl” to see inferred contracts.)

Function Contracts - Outlook

Limitations

- Generated contracts are w.r.t. asserted (safe) properties at call sites.
- Generated contracts are usually not human-readable.

Function Contracts - Outlook

Limitations

- Generated contracts are w.r.t. asserted (safe) properties at call sites.
- Generated contracts are usually not human-readable.

Challenges and ongoing work

- Not always straightforward to go from solutions in first-order logic to ACSL annotations.
- Smart solutions are needed for aggregation functions (e.g., max, min, sum etc.).
- Inferring contracts over collection types (sets etc.).

Outlook

- produce proofs that can be checked by other tools (e.g., Frama-C)

Outlook

- produce proofs that can be checked by other tools (e.g., Frama-C)
- symbolic execution (at solver level)

Outlook

- produce proofs that can be checked by other tools (e.g., Frama-C)
- symbolic execution (at solver level)
- improve heap reasoning
  (at solver level: *the theory of heaps*).

Outlook

- produce proofs that can be checked by other tools (e.g., Frama-C)
- symbolic execution (at solver level)
- improve heap reasoning
  (at solver level: *the theory of heaps*).
- further C features (floats, function pointers, etc.)
- SV-COMP

Outlook

- produce proofs that can be checked by other tools (e.g., Frama-C)
- symbolic execution (at solver level)
- improve heap reasoning
  (at solver level: *the theory of heaps*).
- further C features (floats, function pointers, etc.)
- SV-COMP

Try TriCera online:

`http://logicrunch.it.uu.se:4096/~zafer/tricera/`

or

```
$ git clone https://github.com/uuverifiers/tricera.git
$ cd tricera && sbt assembly
$ ./tri <your_program.c>
```

TriCera – User-specified uninterpreted predicates

```c
void main () {
  int i, n = _;
  int a[n];
  for (i = 0; i < n; ++i) {
    a[i] = 2*i;
  }
  for (i = 0; i < n; ++i) {
    assert(a[i] == 2*i);
  }
}
```

```c
/*$ p(int, int, int) $*/
void main () {
  int i, n = _;
  int a;                        // a[n];
  for (i = 0; i < n; ++i){
    assert(p(a, i, 2*i)); // a[i] = 2*i;
  }
  for (i = 0; i < n; ++i) {
    int v;
    assume(p(a, i, v));   // v = a[i];
    assert(2*i == v);
  }
}
```

On the right, `p(a, i, v)` is used for specifying a data invariant for the array.

TriCera – User-specified uninterpreted predicates

```
void main () {
  int i, n = _;
  int a[n];
  for (i = 0; i < n; ++i) {
    a[i] = 2*i;
  }
  for (i = 0; i < n; ++i) {


    assert(a[i] == 2*i);
  }
}
```

```
/*$  p(int, int, int)  $*/
void main () {
  int i, n = _;
  int a;                        // a[n];
  for (i = 0; i < n; ++i){
    assert(p(a, i, 2*i)); // a[i] = 2*i;
  }
  for (i = 0; i < n; ++i) {
    int v;
    assume(p(a, i, v));    // v = a[i];
    assert(2*i == v);
  }
}
```

On the right, `p(a, i, v)` is used for specifying a data invariant for the array.
http://logicrunch.it.uu.se:4096/~zafer/tricera/?ex=perma%2F1653061937_379790425

ELDARICA – Theories

	SAT Checking	Proof Generation	Craig Interpolation	Quantifier Elimination
Linear Integers (LIA)	✓	✓	✓	✓
Non-linear Integers (NIA)	✓	✓	(✓)[3]	(✓)[4]
Linear Reals (LRA)	✓			
Bit-vectors (BV)	✓	✓	✓	✓
Algebraic Datatypes (ADT)	✓	✓	✓	
Strings	✓[1]	(✓)[2]		
Equality, Functions (EUF)	✓	✓	✓	
Arrays	✓	✓	(✓)[3]	
Heap	✓	✓	(✓)[3]	

(1) Separate solver OSTRICH
(2) Ongoing research
(3) Quantifier-free interpolation not guaranteed
(4) Best-effort, not possible in general

TriCera – Supported features

Types	✓integers (mathematical, machine arithmetic), ✓structs, ✓enums, ✓heap pointers, ✢arrays, ✢stack pointers, ✗floating point, ✗strings, ✗function pointers,
Expressions	✓(postfix, unary, logical, bitwise, arithmetic, cast operators)
Statements and Blocks	✓(compound, expression, selection, iteration statements), ✓(atomic, within and thread blocks (non-standard C))
Other	✓(assert and assume statements), ✓(malloc, calloc, and free) ✓threads, ✓communicating timed systems, ✓function contract and loop invariant inference, ✢ACSL parser (only for function contracts)

More info: tool paper to appear at FMCAD 2022.