TRICERA: Verifying C Programs Using the Theory of Heaps

Zafer Esen[1] and Philipp Rümmer[1,2]

[1]Uppsala University
[2]University of Regensburg

FMCAD 2022
21 October 2022
Trento, Italy

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

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

- 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

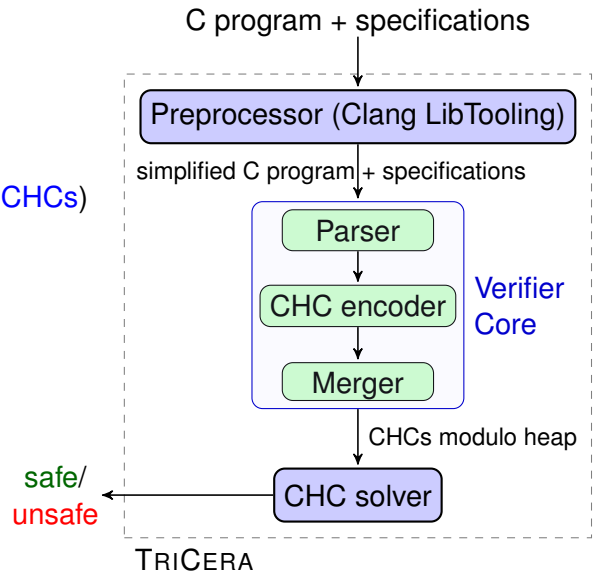
- Supports a large subset of C11
- Assertion-based (some support for ACSL)
- Concurrency - declare threads & monitors
- C programs with timing constraints
- Automatic inference of function contracts and loop invariants
- Uninterpreted predicates

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

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

TRICERA's architecture

- Based on Constrained Horn Clauses (CHCs)
- Open source, implemented in Scala
- Custom, extensible parser

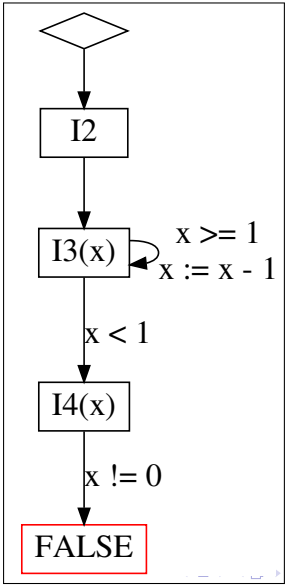


A program and its Constrained Horn Clauses

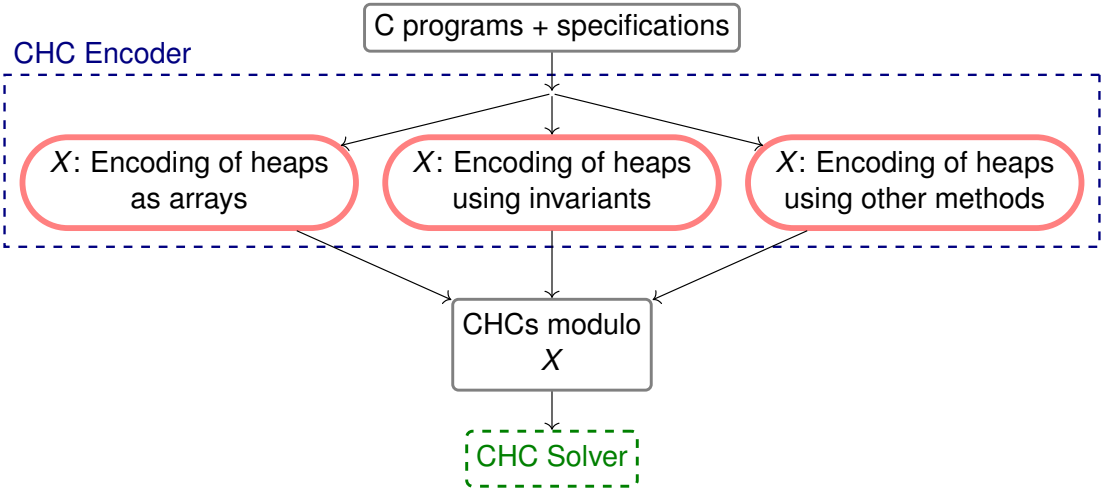
```
int x = _;
while(x > 0){
  x--;
}
assert(x == 0);
```

A program and its Constrained Horn Clauses

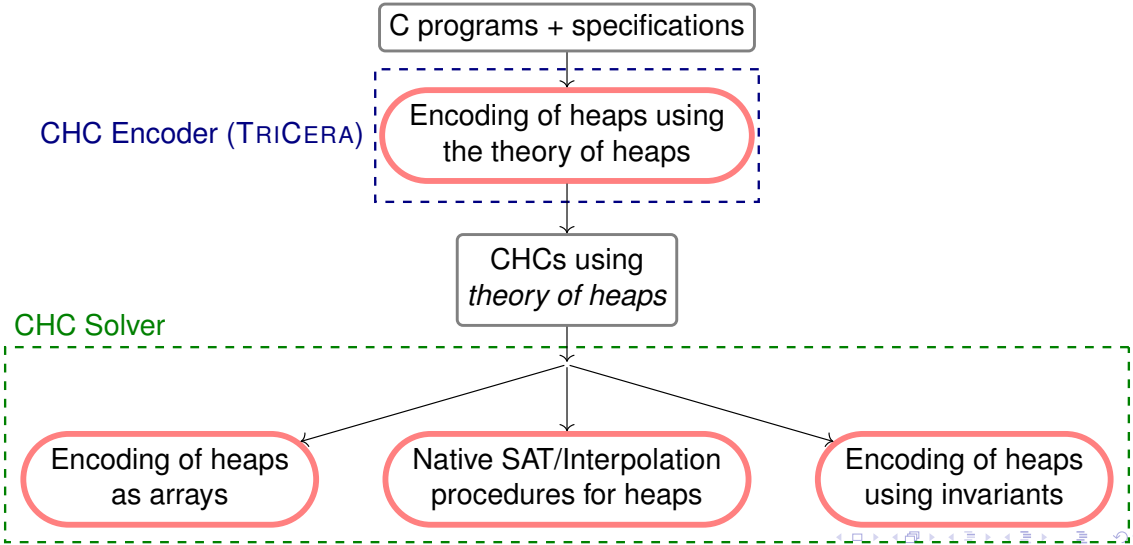
```
int x = _;
while(x > 0){
  x--;
}
assert(x == 0);
```



Encoding programs with heap



TRICERA: move heap reasoning to the back-end



Example encoding

```
1 Node* list = calloc(sizeof(Node));
2 Node n = list->data;
```

```
1 I1(emptyHeap).
```

Example encoding

```
1 Node* list = calloc(sizeof(Node));
2 Node n = list->data;
```

```
1 I1(emptyHeap).
2 I2(ar._1, ar._2) :- I1(h),
3                     ar = alloc(h, Node(0, nullAddress)).
```

Example encoding

```
1 Node* list = calloc(sizeof(Node));
2 Node n = list->data;
```

```
1 I1(emptyHeap).
2 I2(ar._1, ar._2) :- I1(h),
3                     ar = alloc(h, Node(0, nullAddress)).
4
5 I3(h, list, n)   :- I2(h, list), n = read(h, list).
```

Example encoding

```
1 Node* list = calloc(sizeof(Node));
2 Node n = list->data;
```

```
1 I1(emptyHeap).
2 I2(ar._1, ar._2) :- I1(h),
3                     ar = alloc(h, Node(0, nullAddress)).
4
5 I3(h, list, n)   :- I2(h, list), n = read(h, list).
6 false            :- I2(h, list), !valid(h, list).
7 ...
```

Experiments / SV-COMP 2022 *ReachSafety* (Heap)

	safe	unsafe	unknown	solved
THETA	10	7	332	17
GOBLINT	27	0	322	27
GRAVES-CPA	22	26	301	48
TRICERA (ELDARICA-heap)	12	36	301	48
2LS	35	21	293	56
TRICERA (Z3/SPACER)	20	40	289	60
UTAIPAN	32	33	284	65
UAUTOMIZER	32	35	282	67
UKOJAK	25	42	282	67
VERIFUZZ	0	71	278	71
TRICERA (ELDARICA-array)	36	49	264	85
TRICERA (portfolio)	39	58	252	97
CRUX	55	48	246	103
CPACHECKER	58	46	245	104
PESCO	65	47	237	112
CBMC	65	51	233	116
LART	90	30	229	120
SYMBIOTIC	102	62	185	164
ESBMC-KIND	122	49	178	171
VERIABS	223	81	45	304

Experiments / SV-COMP 2022 *ReachSafety* (Non-heap)

	safe	unsafe	unknown	solved
GOBLINT	180	0	1273	180
THETA	250	140	1063	390
UKOJAK	278	221	954	499
VERIFUZZ	0	515	938	515
2LS	428	265	760	693
CBMC	313	394	746	707
TRICERA (Z3/SPACER)	442	271	740	713
CRUX	293	427	733	720
LART	346	392	715	738
ESBMC-KIND	484	380	589	864
SYMBIOTIC	423	458	572	881
UTAIPAN	598	298	557	896
UAUTOMIZER	612	302	539	914
PESCO	584	458	411	1042
TRICERA (ELDARICA)	698	360	395	1058
GRAVES-CPA	636	442	375	1078
TRICERA (portfolio)	730	379	344	1109
CPACHECKER	666	470	317	1136
VERIABS	739	507	207	1246

Outlook

Next steps:

- further C features (floats, function pointers, etc.),
- improve heap reasoning (at solver level),
- improve user-friendliness

Outlook

Next steps:

- further C features (floats, function pointers, etc.),
- improve heap reasoning (at solver level),
- improve user-friendliness

Try TRICERA:
`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>
```

Experiments – Benchmark selection

Benchmark sets:

(A) 361 source benchmarks taken from *reach-safety* and *memsafety* tracks of SV-COMP 2022.

(B) benchmarks in (A) are encoded into CHCs with heaps by TRICERA.

(C) benchmarks in (B) are encoded into CHCs with Arrays by heap2array.

Runs:

- ELDARICA (heap): ELDARICA on (B),
- ELDARICA (array): ELDARICA on (C),
- Z3/Spacer (array): Z3/Spacer on (C),
- *Portfolio*: combined results of previous three steps,

Axioms about read, write & validity

read-over-write

$\texttt{valid}(h, p) \implies \texttt{read}(\texttt{write}(h, p, o), p) = o$

$p_1 \neq p_2 \implies \texttt{read}(\texttt{write}(h, p1, o), p2) = \texttt{read}(h, p2)$

read-over-allocate

$\texttt{alloc}(h, o) = ar \implies \texttt{read}(ar.\_1, ar.\_2) = o$

$\texttt{alloc}(h, o) = ar \ \wedge \ p \neq ar.\_2 \implies \texttt{read}(ar.\_1, p) = \texttt{read}(h, p)$

invalid write/read

$\neg\texttt{valid}(h, p) \implies \texttt{write}(h, p, o) = h$

$\neg\texttt{valid}(h, p) \implies \texttt{read}(h, p) = \mathit{defObj}$

validity of the empty heap and the null address

$\neg\texttt{valid}(\texttt{emptyHeap}, p)$ $\qquad$ $\neg\texttt{valid}(h, \texttt{nullAddr})$

Axioms about allocation

allocation

$$\texttt{alloc}(h, o) = ar \implies \neg\texttt{valid}(h, ar.\_2) \wedge \texttt{valid}(ar.\_1, ar.\_2) \wedge (\forall p.\ ar.\_2 \neq p \implies (\texttt{valid}(h, p) \Leftrightarrow \texttt{valid}(ar.\_1, p)))$$

deterministic allocation

$$(\forall p.\ \texttt{valid}(h_1, p) \Leftrightarrow \texttt{valid}(h_2, p)) \implies \texttt{alloc}(h1, o1).\_2 = \texttt{alloc}(h2, o2).\_2$$

Axioms about extensionality and constructability

extensionality

$$(\forall p.\ (\texttt{valid}(h1, p) \Leftrightarrow \texttt{valid}(h2, p)) \ \wedge\ \texttt{read}(h1, p) = \texttt{read}(h2, p)) \\ \implies h1 = h2$$

constructability / no junk

$$\exists f : Nat \to Heap, g : Nat \to Address. \\ f(0) = \texttt{emptyHeap} \wedge g(0) = \texttt{nullAddr} \wedge \\ \forall i : Nat.\ \langle f(i+1), g(i+1)\rangle = \texttt{alloc}(f(i), defObj) \wedge \\ \forall p : Addr.\ \exists i : Nat.\ g(i) = p$$

Example: Encoding of an IntList

```
1  (declare-heap ...
2   ((IntList 0) (Cons 0) (Nil 0) ...)
3   (((IntList   (size          Int)))
4    ((Cons      (parentCons IntList) (head Int) (tail Addr)))
5    ((Nil       (parentNil  IntList)))
6    ...))
```

Example: Encoding of an IntList – full heap declaration

```
1  (declare-heap
2   Heap                                    ; declared Heap sort
3   Addr                                    ; declared Address sort
4   Object                                  ; chosen Object sort
5   O_Empty                                 ; the default Object
6   ((IntList 0) (Cons 0) (Nil 0) (Object 0)) ; ADTs
7   (((IntList   (sz         Int)))          ; Class ctors
8    ((Cons      (parentCons IntList) (hd Int) (tl Addr)))
9    ((Nil       (parentNil  IntList)))
10   ((O_Cons    (getCons    Cons))          ; Object sort ctors
11    (O_Nil     (getNil     Nil))
12    (O_Empty   ))))
```

Constrained Horn Clauses

A constrained Horn clause (CHC) in predicate logic is the formula:

$$\overbrace{H}^{\text{Head}} \leftarrow \overbrace{C \wedge B_1 \wedge ... \wedge B_n}^{\text{Body}}$$

Constrained Horn Clauses

A constrained Horn clause (CHC) in predicate logic is the formula:

$$\overbrace{H}^{\text{Head}} \leftarrow \overbrace{C \wedge B_1 \wedge ... \wedge B_n}^{\text{Body}}$$

$$I_2(x) \leftarrow I_1(x) \wedge x > 0$$

```
(assert (forall ((x Int))
          (=> (and (> x 0) (I1 x)) (I2 x))))
```

Constrained Horn Clauses

A constrained Horn clause (CHC) in predicate logic is the formula:

$$\overbrace{H}^{\text{Head}} \leftarrow \overbrace{C \wedge B_1 \wedge ... \wedge B_n}^{\text{Body}}$$

$$I_2(x) \leftarrow I_1(x) \wedge x > 0$$

```
(assert (forall ((x Int))
          (=> (and (> x 0) (I1 x)) (I2 x))))
```

Often written in Prolog:

```
I2(x) :- I1(x), x > 0.
```

Algebraic Data-Types (ADTs)

An enumeration

```
(declare-datatype Colour (Red Green Blue))
```

A list

```
(declare-datatype IntList (Nil (Cons (head Int) (tail IntList))))
```

Algebraic Data-Types (ADTs)

An enumeration

```
(declare-datatype Colour (Red Green Blue))
```

A list

```
(declare-datatype IntList (Nil (Cons (head Int) (tail IntList))))
```

ADTs can be used to encode heap *Object*s.

Encoding of programs using CHCs

```
1  int x = _;
2  while(x > 0){
3    x--;
4  }
5  assert(x == 0);
```

Encoding of programs using CHCs

```
1  int x = _; l1
2  while(x > 0){ l2
3    x--;
4  } l3
5  assert(x == 0);
```

Encoding of programs using CHCs

```
1  int x = _; I1
2  while(x > 0){ I2
3    x--;
4  } I3
5  assert(x == 0);
```

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

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

Encoding of programs using CHCs

```
1  int x = 
2  while(x > 0
3    x--;
4
5  assert(x == 0);
```

$l_1 : true$

$l_2 : x \geq 0$

$l_3 : x = 0$

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

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...

Encoding of programs using CHCs

```
1  int x = I1 : true
2  while(x > 0) I2 : x ≥ 0
3    x--;
4  I3 : x = 0
5  assert(x == 0);
```

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

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: $\mathit{true} \rightarrow I_1(-1) \xrightarrow{x \not> 0} I_3(-1) \xrightarrow{x \neq 0} \mathit{false}$

Functions & predicates of the theory

$$
\begin{array}{rcll}
\texttt{nullAddr} & : & () & \rightarrow \ \mathit{Address} \\
\texttt{emptyHeap} & : & () & \rightarrow \ \mathit{Heap} \\
\texttt{alloc} & : & \mathit{Heap} \times \mathit{Object} & \rightarrow \ \mathit{Heap} \times \mathit{Address} \\
\texttt{read} & : & \mathit{Heap} \times \mathit{Address} & \rightarrow \ \mathit{Object} \\
\texttt{write} & : & \mathit{Heap} \times \mathit{Address} \times \mathit{Object} & \rightarrow \ \mathit{Heap} \\
\texttt{valid} & : & \mathit{Heap} \times \mathit{Address} & \rightarrow \ \mathit{Bool}
\end{array}
$$