TRICERA: Verifying C Programs Using the Theory of Heaps

Zafer Esen
Uppsala University, Uppsala, Sweden
zafer.esen@it.uu.se

Philipp Rümmer
University of Regensburg, Regensburg, Germany
Uppsala University, Uppsala, Sweden
philipp.ruemmer@it.uu.se

***Abstract*—TRICERA is an automated, open-source verification tool for C programs based on the concept of Constrained Horn Clauses (CHCs). In order to handle programs operating on heap, TRICERA applies a novel theory of heaps, which enables the tool to hand off most of the required heap reasoning directly to the underlying CHC solver. This leads to a cleaner interface between the language-specific verification front-end and the language-independent CHC back-end, and enables verification tools for different programming languages to share a common heap back-end. The paper introduces TRICERA, gives an overview of the theory of heaps, and presents preliminary experimental results using SV-COMP benchmarks.**

I. INTRODUCTION

This paper presents TRICERA, an automated open-source verification tool for C programs. TRICERA accepts programs in a subset of the C11 standard [1] with the purpose of checking whether explicit and implicit safety assertions in a program are valid. The tool has been developed mainly with applications in the embedded systems area in mind: restrictions in the supported language features are aligned with the recommendations made in the MISRA C coding guidelines [2]. TRICERA works by translating C programs to sets of Constrained Horn Clauses (CHCs), which are then processed and solved by the CHC solvers ELDARICA [33] or SPACER [37], thus either proving that assertions can never fail, or computing counterexample traces leading to an assertion violation.

TRICERA is a model checker for C programs, but includes a plethora of additional features that go beyond C11, such as processing specifications in the ACSL language [6], modelling concurrent and parameterised systems, and augmenting programs with timing constraints. A distinguishing feature of TRICERA is the handling of heap data-structures, which are among the most challenging aspects in the verification of imperative programs. Existing verification tools based on CHCs tend to handle heap either using the theory of arrays (e.g., as done by SEAHORN [30]), or apply bespoke encodings of heap data using refinement types [28], invariants (JAYHORN [35]) or prophecies (RUSTHORN [43]). As the heap encoder is often one of the most complex components of a CHC-based verification tool, this implies repeated implementation effort when designing verification tools for different programming languages, and migrating a tool to a different style of heap encoding is an extremely complex task.

We propose a departure from this conventional architecture of CHC-based verification tools, instead using a language-independent *theory of heaps* [24] augmenting the interface between verification tools and CHC solvers. The theory of heaps is designed to cover the features of many existing programming languages; it is deliberately kept simple, so that it can be integrated easily in verifiers; and it is kept high-level, so that CHC solvers are able to implement a wide range of methods for solving problems involving heap, including the aforementioned encodings through arrays and invariants. The resulting architecture is shown in Figure 1.

TRICERA is the first verification tool that produces CHCs modulo the theory of heaps. At the point of writing this paper, in addition a project is underway to convert the Java verification tool JAYHORN [35] to use the theory. The development of effective solvers for CHCs modulo heaps is an ongoing effort as well; currently the CHC solver ELDARICA provides direct support for CHCs modulo heaps by integrating a native decision and interpolation procedure for the theory of heaps [23]. In addition a tool is available for translating CHCs with heaps to CHCs with algebraic data-types (ADTs) and arrays. A more detailed description of the theory of heaps is available as a technical report [24].

TRICERA is developed at Uppsala University and the University of Regensburg. It is open source[1] and distributed under a 3-Clause BSD license. A web-interface to try it online is available[2].

The contributions of this paper are (i) a presentation of the verification tool TRICERA, including an overview of its features, the verification approach, and architecture; (ii) a definition of a high-level encoding of heap data using the theory of heaps; (iii) an experimental evaluation of TRICERA, on C benchmarks taken from SV-COMP, with and without heap.

II. TRICERA FEATURES

A. Input Language

We start with an overview of the features and languages supported by TRICERA. As its main input language TRICERA can handle a large subset of C11 [1], extended with additional features that are useful for verification purposes. An overview

[1]https://github.com/uuverifiers/tricera
[2]http://logicrunch.it.uu.se:4096/~zafer/tricera/

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_45

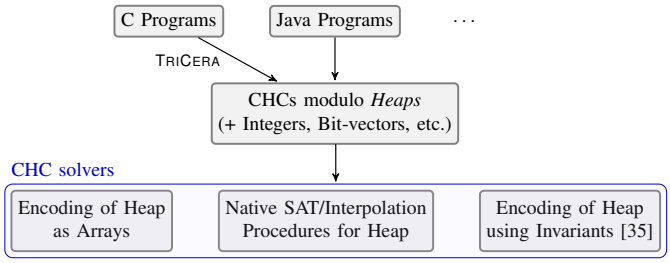

Fig. 1: Program verification using the theory of heaps.

of the currently supported and unsupported types, operations and constructs is given in Table I.

The initially supported subset of C11 is selected as to provide a strong foundation that can be easily extended. Our choice of language features to support is mainly influenced by safety-critical programs from the embedded systems area, and largely aligned with the recommendations made by the MISRA C [2].

TriCera supports most of the standard C types with the exception of floating-point numbers, function pointers and strings. Integer types can be treated either as mathematical integers or as bit-vectors, in the latter case modelling the standard wrap-around semantics. TriCera has full support for the C operators and statements, including several extensions discussed later in this section. TriCera can also handle a (small) part of the C library, in particular functions for memory allocation. Heap data, pointers, and arrays are encoded through the theory of heaps, which we discuss in more detail in Section V.

The partial support in Table I for arrays refers to the following restrictions: (i) the type of array cells must be specified during allocation using `malloc`; (ii) pointers used to index arrays (either through brackets or pointer arithmetic) must be declared as arrays when they are first declared. For instance, **int** `*a` cannot be used to index an array, but **int** `a[]` can. Pointers to array cells are allowed: for instance **int** `*b = &a[i]` is allowed where `a` is an **int** array, but `b` cannot later be indexed as an array. Since arrays are a recent addition to TriCera, these are restrictions of the current TriCera C front-end and not a theoretical limitation of the theory of heaps.

TriCera has limited support for pointers to stack variables. Such pointers are statically associated with the variables they point to on the stack. This imposes some restrictions on such pointers: they cannot be mixed and matched with pointers to the heap, and they cannot be reassigned. The restrictions result in easier to solve encodings.

The following paragraphs survey some of the additional features beyond C11.

TABLE I: Supported subset of the C language (not exhaustive). ✓ represents fully or almost-fully supported, ✢ represents partially supported, and ✗ represents unsupported 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)

B. Supported Code Annotations

In line with other model checkers, TriCera uses **assert** and **assume** statements for explicitly specifying properties, which have their usual semantics as given by Flanagan and Saxe [27]. TriCera in addition automatically adds several implicit properties:

- all pointer de-references are checked for type safety,
- array accesses are checked to be within array bounds,
- (optionally) memory leaks are detected by ensuring all allocated memory on the heap is freed at program exit.

Checking pointer de-references for type safety also implies memory safety, because TriCera encodes unallocated locations using a special type. More information is provided in Section V.

Given a program with an entry function (default `main`), TriCera will attempt to prove that none of the explicit and implicit properties can be violated. When TriCera reports that an assertion is reachable, a counterexample trace is provided for debugging purposes.

TriCera supports the declaration of non-deterministically initialised (local or global) variables (with program type `T`) using the notation `T x = _`.

Function calls in a program are handled, by default, through inlining; by annotating a function with the comment `/*@ contract @*/`, TriCera can be instructed to instead compute a contract consisting of a pre- and post-condition for the function (also see Section II-C). Functions that do not have a body are assumed to produce some non-deterministic result, but not change global variables or heap data.

Function contracts can optionally be specified using the ACSL specification language [6]. At the moment, TriCera can parse and encode `requires`, `ensures` and `assigns` clauses. Listing 1 shows an example program that TriCera can check. Programs annotated with contracts are verified modularly: for each function `f` with a contract, TriCera will try to prove that `f` will never violate its contract that will then be used for encoding `f` at its call sites. More details about the supported ACSL features are given in [21].

C. Annotation Inference

TriCera can be used to automatically infer function contracts and loop invariants for *safe* programs (with respect to implicit and explicit assertions) [4]. An example program is given in Listing 2, encoding the *tak* function [44]. Based on the properties assumed and asserted at lines 12 and 14,

Listing 1: Example of ACSL function contracts in TRICERA. The program is *unsafe* because `q` is accessed but only `p` is specified in the `assigns` clause.

```
1  /*@
2    requires \valid(p, q);
3    assigns *p;
4  */
5  void foo(int* p, int* q) {
6    *q = 42;
7  }
```

Listing 2: An example contract inference problem in TRICERA

```
1  /*@ contract @*/
2  int tak(int x, int y, int z) {
3    if (y < x)
4      return tak(tak(x-1, y, z),
5                 tak(y-1, z, x),
6                 tak(z-1, x, y));
7    else return y;
8  }
9
10 void main() {
11   int x, y, z;
12   assume(x > y && y <= z);
13   int r = tak(x, y, z);
14   assert(r == z);
15 }
```

respectively, TRICERA is able to compute a contract for `tak` that is sufficient to show the safety of the program:

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

where f_{pre} and f_{post} are the pre- and post-conditions of `tak`.

The inferred contracts and invariants can be printed in the ACSL language [6], as well as in SMT-LIB2 and in Prolog. As of writing this paper, ACSL printing is limited to programs without heap.

D. Uninterpreted Predicates

TRICERA allows declaration of *uninterpreted predicates* as annotations, which can then be used in **assert** and **assume** statements. Uninterpreted predicates provide a way to directly affect the generated set of CHCs, as assumptions about the shape of invariants can be manually specified. A program annotated with uninterpreted predicates is considered safe if and only if an interpretation of the predicates (in the sense of first-order logic) exists such that all assertions hold.

An example application is given in Figure 2. In the left column, an array `a` is updated in a loop, and the loop at line 8 encodes the property $\forall j : 0 \leq j < n \rightarrow a[j] = 2j$. Although the program is simple, it turns out to be challenging for software model checkers, since a universally quantified property about the array elements is needed.

The right column shows a version of the program rewritten for verification purposes; the uninterpreted predicate `p_a` is now used to specify a data invariant for the array `a`. The two arguments are selected to correspond to the index, and the value residing at that index, respectively. Writes to `a` are replaced with assertions to `p_a` as in line 6, which asserts that the array `a` contains the value `2*i` at index `i`. Reads from `a` are replaced with assumptions with an additional fresh variable in lines 9–10. The program in the right column can be verified by TRICERA almost instantaneously.

The encoding in Figure 2 closely corresponds to the encoding of universally quantified properties in [13], and is also similar to the invariant encoding of [35], where heap data and operations are encoded through data invariants. Uninterpreted predicates in TRICERA make it possible to easily experiment with encoding tricks of this kind.

E. Concurrency

TRICERA has basic support for handling concurrency in programs. Static threads, executing concurrently with the main program, can be declared using the keyword **thread**. TRICERA currently applies a relatively simple, sequentially consistent thread model that is defined in [34]. This support for concurrency is mainly intended for modelling purposes, but is also useful, e.g., for defining monitors that check temporal properties during execution. For instance, the following thread asserts that the global variable `x` will never decrease during program execution.

```
1 thread Monitor {
2   int t = x;
3   assert(x >= t);
4 }
```

Thread interleaving can be controlled using **atomic** blocks, which mandate that all statements in the block are executed in one atomic step. Threads can moreover be controlled using synchronous rendezvous, which are introduced through UPPAAL-style binary communication channels [7]. In the following program, the two statements **chan_send** and **chan_receive** can only be executed together, thus ensuring that the assertion will be checked after the assignment:

```
1 chan s; int x;
2 thread A {x = 42; chan_send(s);}
3 thread B {chan_receive(s); assert(x>0);}
```

Finally, TRICERA also supports the declaration of infinitely replicated threads, which are useful to model dynamic thread creation and parameterised systems. An example of a parameterised model is given in the next section.

```
1
2  void main () {
3    int i, n = _;
4    int a[n];
5    for (i = 0; i < n; ++i) {
6      a[i] = 2*i;
7    }
8    for (i = 0; i < n; ++i) {
9
10
11     assert(a[i] == 2*i);
12   }
13 }
```

```
1  /*$  p_a(int, int)  $*/
2  void main () {
3    int i, n = _;
4
5    for (i = 0; i < n; ++i) {
6      assert(p_a(i, 2*i));
7    }
8    for (i = 0; i < n; ++i) {
9      int v = _;
10     assume(p_a(i, v));
11     assert(2*i == v);
12   }
13 }
```

Fig. 2: Encoding an array program (left column) using uninterpreted predicates (right column).

Listing 3: The parameterised Fischer protocol [3]

```
1  int lock = 0;
2  thread[tid] Proc {
3    clock C;
4    assume(tid > 0);
5
6    while (1) {
7      atomic { assume(lock == 0); C = 0; }
8      within (C <= 1) { lock = tid; }
9      C = 0; assume(C > 1);
10
11     if (lock == tid) { // critical sect.
12       assert(lock == tid);
13       lock = 0;
14     }
15   }
16 }
```

F. Timing Constraints

For modelling purposes, TRICERA supports timing constraints in C programs. C programs with time have semantics similar to UPPAAL timed automata [7], which means that computations (program instructions) consume zero time, but are interleaved with explicit time-elapse transitions. The passing of time can be observed using clocks, which are declared as variables of type **clock**, can be reset to 0, and can be compared with constants in **assert** and **assume** statements.

As an example, Listing 3 shows a parameterised version of the well-known Fischer mutual exclusion protocol [3]. An arbitrary number of processes can participate in the protocol by communicating through a shared variable `lock`. In line 2, for this purpose an infinitely replicated thread `Proc` is declared. Each instance of `Proc` has a unique thread id `tid` of type **int** and a clock `C`. Each process executes a simple loop: it waits until it observes that `lock == 0`, and then writes its thread id to `lock`. The **within** block in line 8 has similar semantics as an UPPAAL time invariant: it enforces execution of the assignment before the condition `C <= 1` has become false, i.e., at most one time unit after executing the block in line 7. The process then waits for more than one time unit in line 9, and then checks that no other process has meanwhile overwritten the value in `lock`. Line 12 asserts that at most one process is able to enter the critical section at a time.

TRICERA is able to verify the safety of this model for an unbounded number of participating threads, using an encoding of the program as CHCs over k-indexed invariants [34].

III. THE TRICERA VERIFICATION APPROACH

A. Constrained Horn Clauses

TRICERA analyses programs by translating them to sets of Constrained Horn Clauses (*CHCs*, or just *clauses* in this paper), in such a way that the CHCs are satisfiable iff the program is safe. A Constrained Horn Clause is a sentence $\forall \bar{x}.\ \left(C \wedge B_1 \wedge \ldots \wedge B_n \rightarrow H\right)$ where H is either an *atom* (application of a predicate to first-order terms) or *false*, B_i (for $1 \leq i \leq n$) is an atom, and C is a constraint over some background theories (including heaps). A CHC with at least one positive *literal* (an atom or its negation) is called a *definite clause*, and a CHC with no positive literals is called a *goal clause* (or an *assertion clause*). In the rest of the paper we leave the universal quantification of variables implicit, and write the clauses from right to left in the spirit of logic programming.

B. The Architecture of TRICERA

An overview of the TRICERA architecture is given in Figure 3. The preprocessor and the CHC solver are external tools; we call the whole toolchain "TRICERA".

a) Preprocessor: Input programs are preprocessed in order to simplify parsing and encoding. To simplify parsing, all **typedef**s are removed and some language constructs are normalised into a standard form. Unused type and function declarations are removed; removing unused data-types makes

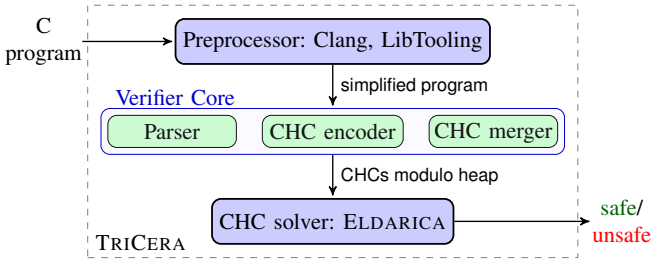

Fig. 3: An overview of TRICERA.

modelling heap simpler as the number of possible types for a heap object is reduced. The preprocessor is written as a stand-alone tool using the LibTooling[3] library in C++.

b) Verifier Core: The TRICERA core component is a translator from C programs into CHCs, written in Scala. The verifier core works by first creating a parse tree of the input program, and then translating this tree into a set of CHCs. The translator supports the language features given in Table I.

The CHC encoder also includes a CHC simplifier that post-processes the generated CHCs before being sent to a CHC solver. This simplifier attempts to merge the CHCs in order to produce a smaller but equisatisfiable set of CHCs.

c) CHC solver: The resulting set of CHCs are finally sent to a CHC solver to check if their conjunction is satisfiable. TRICERA primarily uses ELDARICA [33] for this purpose as it has native support for the theory of heaps, and can easily be integrated as a Scala library; however, the final set of CHCs can also be post-processed by eliminating heap operations, instead encoding using the theory of arrays, and then be checked by other solvers such as Z3/SPACER [37].

C. Programs as Constrained Horn Clauses (CHCs)

The overall translation from sequential programs to CHCs applied by TRICERA follows the strategy defined, e.g., in [12], [29]. In this setting, linear CHCs are used to model the control-flow graph of a program: each node of the graph is interpreted to represent the set of possible states at a program location and each transition corresponds to a program control instruction. Asserted properties add additional sink nodes to the graph, whose edges are the negations of those properties. The goal of the process is to discover program *invariant*s that are sufficient to show that none of the sink nodes is reachable.

A program is thus encoded in CHCs as follows:

- An uninterpreted predicate is declared for each program location to represent program invariants: the interpretations of these predicates (provided by the CHC solver when the set of CHCs is satisfiable) correspond to sets of program states that hold at each location. The arguments to a predicate are all program variables currently in scope, as well as additional terms required in the encoding, for instance terms representing the heap.
- A definite clause consisting of only a single positive literal is added as program entry, e.g., $P(\ldots) \leftarrow \mathit{true}$. The CHC (1) in Figure 4 is an example.
- A definite clause is introduced for each program control instruction. These CHCs encode Hoare triples between locations [32]. The set of CHCs can be cyclic (e.g., $\{P_1(\ldots) \leftarrow P_0(\ldots), P_0(\ldots) \leftarrow P_1(\ldots)\}$), representing program loops. Guarded control instructions are encoded by adding the guards as constraints. The CHCs (2) – (5) in Figure 4 provide an example.
- Two clauses are added for each asserted property: a goal clause whose constraint is the negation of the asserted property, and a definite clause whose constraint is the asserted property. The CHCs (6) and (7) in Figure 4 provide an example.
- Functions are encoded either through predicates representing their pre-/post-conditions, or by inlining them.

An example encoding is provided in Figure 4.

The translation of concurrent and timed programs follows the calculus defined in [34]. To handle concurrency, TRICERA uses a variant of the Owicki-Gries proof rules [47], [34], to which explicit variables to represent time and clocks are added. The representation of replicated threads uses the k-indexed invariants approach [52], [34].

IV. THE THEORY OF HEAPS

One of the most challenging aspects of encoding computer programs as CHCs is the encoding of heap-allocated data-structures and heap-related operations. One approach to represent such data-structures is using the theory of arrays (e.g., [36], [17]). This is a natural encoding since a heap can be seen as an array of memory locations; however, as the encoding is byte-precise, in the context of CHCs it tends to be low-level and often yields clauses that are hard to solve.

An alternative approach is to transform away such data-structures with the help of invariants or refinement types (e.g., [49], [13], [45], [35], and the example in Section II-D). The resulting CHCs tend to be over-approximate (i.e., can lead to false positives), even with smart refinement strategies that aim at increasing precision. This is because every operation that reads, writes, or allocates a heap object is replaced with assertions and assumptions about local object invariants, so that global program invariants might not be expressible. In cases where local invariants are sufficient, however, they can enable efficient and modular verification even of challenging programs.

Both approaches leave little design choice with respect to handling of heaps to CHC solvers. Dealing with heaps at the encoding level also implies repeated effort when designing verifiers for different programming languages.

The vision of the presented line of research is to extend CHCs to a standardised interchange format for programs with heaps. We apply a high-level theory of heaps [24] that does not restrict the way in which CHC solvers approach heap reasoning, while covering the main functionality needed for program verification: (i) representation of the type system

[3]https://clang.llvm.org/docs/LibTooling.html

```
1  /*P0*/
2  if(x > 0)
3    x+=1; /*P1*/
4  else
5    x-=1; /*P2*/
6  /*P3*/
7  assert(x > 0);
8  /*P4*/
```

$$P_0(x) \leftarrow \mathit{true} \quad (1)$$
$$P_1(x) \leftarrow P_0(x) \wedge x > 0 \quad (2)$$
$$P_2(x) \leftarrow P_0(x) \wedge x \leq 0 \quad (3)$$
$$P_3(x') \leftarrow P_1(x) \wedge x' = x + 1 \quad (4)$$
$$P_3(x') \leftarrow P_2(x) \wedge x' = x - 1 \quad (5)$$
$$P_4(x') \leftarrow P_3(x) \wedge x > 0 \quad (6)$$
$$\mathit{false} \leftarrow P_3(x) \wedge x \leq 0 \quad (7)$$

Fig. 4: The CHC encoding of a branching statement

Listing 4: SMT-LIB-style declaration of a heap. In lines 4–8 the constructors and the selectors of the data-types are declared. The constructors and selectors in lines 5–6 serve as the wrappers and the getters for the program types `Node` and `int`. `Node` is encoded as an ADT (line 4) and the C type `int` is encoded using mathematical integers (`Int`).

```
1 (declare-heap
2  Heap Addr Object O_Empty
3  ((Node 0) (Object 0))
4  (((Node (data Int) (next Addr)))
5   ((O_Node (getNode Node))
6    (O_Int  (getInt  Int ))
7    (O_Uninit_Node) (O_Uninit_Int)
8    (O_Empty))))
```

associated with heap data; (ii) reading and updating of data on the heap; (iii) object allocation.

The theory of heaps employs algebraic data-types (ADTs), as already standardised by SMT-LIB [5], as a flexible way to handle (i). The theory offers operations akin to the theory of arrays to handle (ii) and (iii). Arithmetic operations on pointers are excluded in the theory, as are low-level tricks like extracting individual bytes from bigger pieces of data through pointer manipulation. Being language-agnostic, the theory of heaps allows for common encodings across different applications.

a) Sorts: To encode a program using the theory of heaps, first a heap data-type has to be declared that covers the required program types; a declaration in SMT-LIB notation is shown in Listing 4. Each declared heap introduces the three sorts, *Heap*, *Addr* and *AddrRange*, and in addition can declare any number of ADTs later used to represent the data stored on the heap (lines 5–8, see Section V). A *Heap* address has the sort *Addr*. Although an address itself does not carry type information, the type of a heap *Object* can be checked using ADT discriminator functions. A range of addresses can be defined with the *AddrRange* sort, which is needed when encoding contiguous data-structures such as arrays.

The objects on the heap are represented with a single *Object* sort, which can either be selected from one of the pre-declared sorts, or declared as an ADT in a heap theory declaration. The latter makes referring to heap theory sorts possible, such as *Addr*, as done in Line 4 of Listing 4. In the sequel we call a constructor function that produces an *Object* a *wrapper*, and a selector that returns the underlying term from an *Object* a *getter*.

b) Operations: The operations of the theory of heaps are given in Table II. The function allocate is used for allocating new objects on the heap, and each allocation returns a new $\langle \mathit{Heap}, \mathit{Addr} \rangle$ pair that is valid and contains the passed object. The allocatedness of an *Addr* in a *Heap* can be tested using the predicate valid. The function emptyHeap returns a heap that is invalid at all addresses, and nullAddr returns an address that is invalid in all heaps.

The functions read and write are used for reading from and writing to heap addresses. If a read address is invalid, the *default object* is returned (`O_Empty` in line 2 in Listing 4). An invalid write returns the heap that was passed to the function without any modifications. The default *Object* to be returned on invalid reads is specified in the heap declaration, and this is needed to make the read function total.

Operations (14)–(17) are used for *batch* heap operations, which are needed when encoding array-like data on the heap. These operations operate over address ranges rather than single addresses (*Addr*). The functions batchAllocate and batchWrite allow batch allocation and batch update of address ranges. Given an address range, nthInAddrRange allows the extraction of an individual address, and the predicate withinAddrRange allows testing if an address is within a range.

c) Implementation: The theory of heaps is currently implemented in the SMT solver PRINCESS [50] and in the CHC solver ELDARICA [33]. The decision procedure for solving

TABLE II: Operations defined by the theory of heaps

$$\mathsf{emptyHeap} : () \rightarrow \mathit{Heap} \quad (8)$$
$$\mathsf{nullAddr} : () \rightarrow \mathit{Addr} \quad (9)$$
$$\mathsf{allocate} : \mathit{Heap} \times \mathit{Object} \rightarrow \mathit{Heap} \times \mathit{Addr} \quad (10)$$
$$\mathsf{valid} : \mathit{Heap} \times \mathit{Addr} \rightarrow \mathit{Bool} \quad (11)$$
$$\mathsf{read} : \mathit{Heap} \times \mathit{Addr} \rightarrow \mathit{Object} \quad (12)$$
$$\mathsf{write} : \mathit{Heap} \times \mathit{Addr} \times \mathit{Object} \rightarrow \mathit{Heap} \quad (13)$$
$$\mathsf{batchAllocate} : \mathit{Heap} \times \mathit{Object} \times \mathbb{N} \rightarrow \mathit{Heap} \times \mathit{AddrRange} \quad (14)$$
$$\mathsf{batchWrite} : \mathit{Heap} \times \mathit{AddrRange} \times \mathit{Object} \rightarrow \mathit{Heap} \quad (15)$$
$$\mathsf{nthInAddrRange} : \mathit{AddrRange} \times \mathbb{N} \rightarrow \mathit{Addr} \quad (16)$$
$$\mathsf{withinAddrRange} : \mathit{AddrRange} \times \mathit{Addr} \rightarrow \mathit{Bool} \quad (17)$$

TABLE III: O_T is the object wrapper for sort T, which is the encoding of the program type `T`. O_$T(T_0)$ constructs a zero-valued term of sort T. h represents the heap. Non-primed and primed terms encode the same program variable (and the heap) before and after the execution of a statement. `x` is a variable, `p` is a (non-array) pointer. `a` and `b` are pointers to arrays of type `T`. `i`, `j` and `n` are integers.

C statement	Mathematical encoding with heaps
`*p = x;`	$h' = \mathsf{write}(h, p, \mathsf{O\_}T(x)) \wedge$ $(\mathsf{is\text{-}O\_Uninit\_}T(\mathsf{read}(h, p)) \vee \mathsf{is\text{-}O\_}T(\mathsf{read}(h, p)))$
`x = *p;`	$x = \mathsf{get}T(\mathsf{read}(h, p)) \wedge \mathsf{is\text{-}O\_}T(\mathsf{read}(h, p))$
`x = a[i];`	$x = \mathsf{get}T(\mathsf{read}(h, \mathsf{nthInAddrRange}(a, i))) \wedge$ $\mathsf{is\text{-}O\_}T(\mathsf{read}(h, \mathsf{nthInAddrRange}(a, i))) \wedge$ $\mathsf{withinAddrRange}(a, i)$
`a[i] = x;`	$h' = \mathsf{write}(h, \mathsf{nthInAddrRange}(a, i), \mathsf{O\_}T(x)) \wedge$ $(\mathsf{is\text{-}O\_Uninit\_}T(\mathsf{read}(h, \mathsf{nthInAddrRange}(a, i))) \vee$ $\mathsf{is\text{-}O\_}T(\mathsf{read}(h, \mathsf{nthInAddrRange}(a, i)))) \wedge$ $\mathsf{withinAddrRange}(a, i)$
`p = malloc(sizeof(T));`	$\langle h', p\rangle = \mathsf{allocate}(h, \mathsf{O\_Uninit\_}T)$
`p = calloc(sizeof(T));`	$\langle h', p\rangle = \mathsf{allocate}(h, \mathsf{O\_}T(T_0))$
`a = malloc(sizeof(T) * n);`	$\langle h', p\rangle = \mathsf{batchAllocate}($ $h, \mathsf{O\_Uninit\_}T, n)$
`free(p);`	$h' = \mathsf{write}(h, p, \mathsf{O\_Empty}) \wedge$ $\neg\mathsf{is\text{-}O\_Empty}(\mathsf{read}(h, p)))$
`free(a);`	$h' = \mathsf{batchWrite}(h, a, \mathsf{O\_Empty}) \wedge$ $\forall q : \mathit{Addr}.(\mathsf{withinAddrRange}(a, q) \rightarrow$ $\neg\mathsf{is\text{-}O\_Empty}(\mathsf{read}(h, q)))$

formulas over the theory in PRINCESS is introduced in [23]. ELDARICA mostly defers the solving of heap theory formulas to PRINCESS; there is ongoing work to implement additional static analysis of heap properties directly in ELDARICA.

V. ENCODING OF C PROGRAMS WITH HEAP

When translating programs with heaps, TRICERA augments all introduced relation symbols (state invariants and pre-conditions) with explicit *Heap* arguments; post-conditions receive both the pre- and the post-heap.

A heap $Addr$ can be seen as a direct counterpart of an (untyped) C pointer. Any program type that makes use of an $Addr$, such as a list node, needs to be declared as part of the heap theory declaration. Lastly, *Object wrappers* and *getters* need to be declared for all program types that can be on the heap. For instance, Listing 4 shows a heap declaration for a program over (mathematical) integers and a node struct:

```
struct Node {
  int data;
  struct Node* next;
};
```

Since `Node` has a pointer field, it is declared as an ADT as part of the heap declaration as shown in line 4 of Listing 4. The object wrappers and getters for all program types are declared in lines 5–6. Additional *empty* object wrappers are defined in lines 7–8 to serve as the *uninitialised* and *default* objects respectively. TRICERA uses the default object to mark de-allocated locations as shown in Table III. The uninitialised objects are used as initial values for allocated memory locations with uninitialised values, as is the case with `malloc`. An uninitialised object constructor is declared for each programming type on the heap.

After the heap is declared, every statement that accesses the heap is encoded as shown in Table III. A new heap term h' is produced for statements modifying the starting heap term h.

Each de-reference of a pointer is also coupled with a type-safety assertion. In the table, those assertions are conjoined with the actual transition relations of the CHCs; as a result, the stated formulas describe all correct executions of a statement. TRICERA in addition introduces assertions that will detect cases in which these conditions are violated. For instance, for the expression `*p`, assuming that `p` is encoded using the sort T, TRICERA asserts the predicate $\mathsf{is\text{-}O\_}T(\mathsf{read}(h, p))$. is-O_$T$ is the discriminator predicate for the ADT sort T. Since invalid reads would return the default object, this type-safety assertion doubles as a memory safety assertion. C also allows the allocation of uninitialised memory; TRICERA models this by placing the object $\mathsf{O\_Uninit\_}T$ in these addresses, which represents an uninitialised value for the sort T.

Functions that require byte-level access to data-structures such as `memset` are currently not supported by TRICERA; however, these can be handled without introducing a full byte-level memory representation. It is sufficient to infer which values a heap object can assume when setting all its bytes to a certain value, taking into account the compiler and architecture when necessary. To prevent aliasing when using such functions, safety assertions that ensure the accessed memory region belongs to a single object can be automatically added to each access.

a) Arrays: TRICERA uses the theory of heaps also to model C arrays. Arrays are allocated and freed using the *batch* operations of the theory. The address of an array cell is obtained with the nthInAddrRange function, which can then be used as any other address. Whenever an array cell is accessed (`a[i]`), TRICERA automatically asserts that the accessed index is within bounds ($\mathsf{withinAddrRange}(a, i)$).

Arithmetic operations on array pointers can be supported by augmenting $AddrRange$ terms with offsets (not shown in Table III). This yields a model in which arithmetic on array pointers is possible, but modified pointers have to remain in the same array, which is again in line with the MISRA C coding guidelines [2].

The theory of heaps does not provide a direct operation for de-allocation, i.e., an allocated address always remains valid. TRICERA overcomes this limitation by writing the default object (O_Empty) to de-allocated addresses, and provides an option to add a memory safety assertion such that all addresses must contain the default object at program exit. Double-freeing of memory is caught by an additional assertion that the freed addresses do not contain the default object.

Stack-allocated arrays are also modeled using the theory of heaps, and the functions to free their memory are automatically added by TRICERA when they go out of scope. Non-array

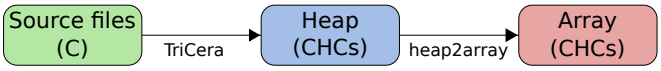

Fig. 5: The three sets of heap benchmarks: (i) the source C benchmarks from SV-COMP are encoded into (ii) CHCs modulo the theory of heaps using TRICERA, then heap2array is applied to these benchmarks to produce (iii) CHCs modulo the theory of arrays.

stack pointers do not make use of the theory, and are supported with some limitations as described in [22].

Table III does not show the encoding of field updates for record types, for instance `p->f = x` where `p` is a pointer to a record type with `f` as one of its fields. This is encoded by first reading the record from `p`, creating a new ADT term with only the field `f` updated, and then writing back the new ADT to the address pointed to by `p`.

VI. EXPERIMENTAL RESULTS

A. Benchmarks

As TRICERA does not have a model of pthreads yet, we focus in our evaluation on sequential C programs. We collected C benchmarks from SV-COMP 2022's *ReachSafety* and *MemSafety* categories [10], and generated CHCs in the SMT-LIB [5] format for all benchmarks that the current version of TRICERA could parse and encode (see Section II-A). This resulted in 396 heap (i.e., where the heap is modelled using the theory of heaps) benchmarks (349 in the *ReachSafety* and 128 in the *MemSafety* categories, with some benchmarks occurring in both categories), and 1453 non-heap benchmarks in the *ReachSafety* category. Many of the benchmarks that TRICERA could not parse were under the Juliet test and the Linux device driver suites, which failed mainly due to currently unsupported operations and constructs such as `memcpy` and function pointers. Mathematical integer semantics was used in the benchmarks encoded by TRICERA.

For the heap benchmarks, an additional set of benchmarks was created through a translation of the theory of heaps into the theory of arrays, using an extended version of the encoding given in [24] implemented in the tool *heap2array*[4]. This serves the purpose of making additional back-ends available to solve the generated CHCs. Similarly generated benchmarks were also submitted to CHC-Comp 2022 and were part of the *LIA-nonlin-Arrays-nonrecADT* track[5]. The benchmark creation process is depicted in Figure 5.

We then applied two of the top CHC solvers currently available [26] to the CHCs: ELDARICA, which is the default solver in TRICERA and natively supports the theory of heaps, and Z3/SPACER [37]. We have used the default settings in both ELDARICA and Z3/SPACER. ELDARICA was used in two different configurations for the heap benchmarks: TRICERA (ELDARICA-heap), using the native solver for the theory of heaps on the CHCs with heaps, and TRICERA (ELDARICA-array), applying ELDARICA's array solver to the array version of the CHCs. Z3/SPACER was only applied to the array benchmarks (TRICERA (Z3/SPACER)). The *portfolio* rows in the result tables show the results achieved by running both back-ends of TRICERA in parallel and taking the first result (TRICERA (portfolio)).

B. Experimental Setup

The experiments were ran on an AMD Opteron 2220 SE (2.8 GHZ with 4 CPUs) machine running 64-bit Linux with 6 GB of RAM and a wall-clock timeout of 900 seconds. To compare TRICERA[6] against the state of the art, we gathered the results published by SV-COMP 2022 [9] for the *ReachSafety* and *MemSafety* tracks.

C. Results

The results are given in Table IV for the non-heap benchmarks in the *ReachSafety* category, in Table V for the heap benchmarks in the *ReachSafety category* and in Table VI for the heap benchmarks in the *MemSafety* category. All benchmarks can be found in [25].

For non-heap, TRICERA showed performance competitive with the best tools evaluated at SV-COMP, in particular on safe problems. The TRICERA results are not completely comparable to the results of SV-COMP tools due to the use of mathematical integer semantics in TRICERA, however. For 19 benchmarks, the statuses reported by TRICERA were inconsistent with the expected SV-COMP statuses for this reason. The two TRICERA back-ends, ELDARICA and Z3/SPACER, always produced the same answer.

For heap problems, TRICERA performed worse than some of the tools based on bounded model checking or symbolic execution, but was comparable with CEGAR-based tools like CPACHECKER. Comparing the TRICERA back-ends, ELDARICA applied to the array encoding performs best by some margin (TRICERA (ELDARICA-array)).

TRICERA currently cannot check for reachability and memory-safety properties separately, it always adds the implicit memory-safety assertions. This, coupled with the use of mathematical integers, led to results that did not match their expected SV-COMP statuses in 25 heap benchmarks (13 reported incorrectly unsafe, 12 reported incorrectly safe) using the portfolio method; again there were no inconsistencies between the different TRICERA back-ends.

VII. RELATED WORK

There are several other verification tools that make use of CHCs, and many others for verifying C programs. As discussed in Section I, these tools either transform away the heap, or use the theory of arrays for encoding heap.

JAYHORN, a model checker for Java programs, encodes heap by using invariants that summarise the possible states of a reference at a program location [35], which is inspired by methods like liquid types [49]. Although this method is

[4]https://github.com/zafer-esen/heap2array

[5]https://github.com/zafer-esen/tricera-adt-arr

[6]https://github.com/uuverifiers/tricera/commit/5ffd2b6

TABLE IV: Results for the non-heap benchmarks in the *ReachSafety* category. The column "solved" gives the total number of "safe" or "unsafe" results.

	safe	unsafe	unknown	solved
GOBLINT [51]	180	0	1273	180
THETA [53]	250	140	1063	390
UKOJAK [46]	278	221	954	499
VERIFUZZ [15]	0	515	938	515
2LS [42]	428	265	760	693
CBMC [38]	313	394	746	707
TRICERA (Z3/SPACER)	442	271	740	713
CRUX [19]	293	427	733	720
LART [40]	346	392	715	738
ESBMC-KIND [41]	484	380	589	864
SYMBIOTIC [14]	423	458	572	881
UTAIPAN [18]	598	298	557	896
UAUTOMIZER [31]	612	302	539	914
PESCO [48]	584	458	411	1042
TRICERA (ELDARICA)	698	360	395	1058
GRAVES-CPA [41]	636	442	375	1078
TRICERA (portfolio)	730	379	344	1109
CPACHECKER [11]	666	470	317	1136
VERIABS [16]	739	507	207	1246

TABLE V: Results for the heap benchmarks in the *ReachSafety* category.

	safe	unsafe	unknown	solved
THETA [53]	10	7	332	17
GOBLINT [51]	27	0	322	27
GRAVES-CPA [41]	22	26	301	48
TRICERA (ELDARICA-heap)	12	36	301	48
2LS [42]	35	21	293	56
TRICERA (Z3/SPACER)	20	40	289	60
UTAIPAN [18]	32	33	284	65
UAUTOMIZER [31]	32	35	282	67
UKOJAK [46]	25	42	282	67
VERIFUZZ [15]	0	71	278	71
TRICERA (ELDARICA-array)	36	49	264	85
TRICERA (portfolio)	39	58	252	97
CRUX [19]	55	48	246	103
CPACHECKER [11]	58	46	245	104
PESCO [48]	65	47	237	112
CBMC [38]	65	51	233	116
LART [40]	90	30	229	120
SYMBIOTIC [14]	102	62	185	164
ESBMC-KIND [41]	122	49	178	171
VERIABS [16]	223	81	45	304

TABLE VI: Results for the heap benchmarks in the *MemSafety* category.

	safe	unsafe	unknown	solved
VERIFUZZ [15]	0	5	123	5
SESL	0	11	117	11
UAUTOMIZER [31]	6	11	111	17
UTAIPAN [18]	7	10	111	17
UKOJAK [46]	9	10	109	19
2LS [42]	14	11	103	25
TRICERA (ELDARICA-heap)	12	19	97	31
TRICERA (Z3/SPACER)	23	16	89	39
CPACHECKER [11]	53	10	65	63
TRICERA (ELDARICA-array)	39	24	65	63
TRICERA (portfolio)	40	26	62	66
ESBMC-KIND [41]	54	18	56	72
CPA-BAM-SMG	53	30	45	83
CBMC [38]	54	36	38	90
SYMBIOTIC [14]	68	36	24	104

incomplete (i.e., can lead to false positives), with various optimisations the authors have managed to significantly improve its effectiveness. Using the theory of heaps, much of the work in JayHorn could be shifted to a CHC solver. TRICERA and JAYHORN both use ELDARICA for solving the generated CHCs, but otherwise do not share any infrastructure.

RUSTHORN is a verifier for Rust programs, and also transforms away the heap [43] by exploiting the ownership system of Rust. Since the method is not directly applicable in case of unsafe code blocks, a theory of heaps could be used to extend the tool in this direction.

SEAHORN is a verifier for LLVM-based languages [30]. SEAHORN employs Z3/SPACER as one of its back-ends for CHC-based model-checking. It also employs various static analyses that can be used on their own as a verification engine, or to provide invariants to its CHC back-ends. SEAHORN encodes the heap as a set of non-overlapping arrays that are created by a *data structure analysis* (DSA) [39]. Since SEAHORN works with the LLVM intermediate representation, it can be used to target other LLVM-based languages than C. In contrast, TRICERA comes with its own parser that currently cannot handle all the peculiarities of C; however, its custom parser can handle several non-standard C constructs as shown in Table I and can easily be extended.

KORN is a verifier for C programs that uses CHCs; however its main focus is showing the feasibility of using *loop contracts* as opposed to loop invariants and currently supports a small fragment of C [20]. KORN uses ELDARICA as one of its back-ends.

Information about the other verification tools evaluated in Section VI can be found in the SV-COMP report [8].

VIII. CONCLUSIONS AND OUTLOOK

This paper has introduced the verification tool TRICERA, given an overview of the encoding of C programs using the theory of heaps, and provided first experimental results using SV-COMP benchmarks. Both TRICERA and the theory of heaps are still under development, and planned future work includes support for further features of C (see Table I), improved decision and interpolation procedures for the theory of heaps, and the development of additional heap back-ends (in particular along the lines of [35]). Once multiple CHC solvers with support for the theory of heaps are available, we will also propose a heap track at the Horn solver competition CHC-COMP.

ACKNOWLEDGEMENTS

This work was supported by the Swedish Research Council (VR) under grant 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and by the Knut and Alice Wallenberg Foundation under the project UPDATE.

References

[1] Information technology — programming languages — C. ISO/IEC 9899:2011, International Organization for Standardization, Geneva, Switzerland (2011)

[2] Guidelines for the use of the C language in critical systems. MISRA C:2012, The MISRA Consortium Limited, Norfolk, England (2012)

[3] Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Trans. Program. Lang. Syst. **16**(5), 1543–1571 (sep 1994). https://doi.org/10.1145/186025.186058, https://doi.org/10.1145/186025.186058

[4] Amilon, J., Esen, Z., Gurov, D., Lidström, C., Rümmer, P.: An exercise in mind reading: Automatic contract inference for Frama-C. In: Guide to Software Verification with Frama-C. Core Components, Usages, and Applications (2022), (To appear)

[5] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`

[6] Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language Version 1.17. (2021)

[7] Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA. Lecture Notes in Computer Science, vol. 1066, pp. 232–243. Springer (1995). https://doi.org/10.1007/BFb0020949, https://doi.org/10.1007/BFb0020949

[8] Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20, https://doi.org/10.1007/978-3-030-99527-0_20

[9] Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022) (Jan 2022). https://doi.org/10.5281/zenodo.5831008, https://doi.org/10.5281/zenodo.5831008

[10] Beyer, D.: SV-Benchmarks: Benchmark Set for Software Verification and Testing (SV-COMP 2022 and Test- Comp 2022) (Jan 2022). https://doi.org/10.5281/zenodo.5831003, https://doi.org/10.5281/zenodo.5831003

[11] Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16, https://doi.org/10.1007/978-3-642-22110-1_16

[12] Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015). https://doi.org/10.1007/978-3-319-23534-9_2, https://doi.org/10.1007/978-3-319-23534-9_2

[13] Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 105–125. Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_8, https://doi.org/10.1007/978-3-642-38856-9_8

[14] Chalupa, M., Mihalkovic, V., Rechtácková, A., Zaoral, L., Strejcek, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding - (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 462–467. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32, https://doi.org/10.1007/978-3-030-99527-0_32

[15] Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 244–249. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22, https://doi.org/10.1007/978-3-030-17502-3_22

[16] Darke, P., Agrawal, S., Venkatesh, R.: Veriabs: A tool for scalable verification by abstraction (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 458–462. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32, https://doi.org/10.1007/978-3-030-72013-1_32

[17] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Program verification using constraint handling rules and array constraint generalizations. Fundam. Inform. **150**(1), 73–117 (2017). https://doi.org/10.3233/FI-2017-1461, https://doi.org/10.3233/FI-2017-1461

[18] Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate taipan with symbolic interpretation and fluid abstractions - (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 418–422. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32, https://doi.org/10.1007/978-3-030-45237-7_32

[19] Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Blazy, S., Chechik, M. (eds.) Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9971, pp. 56–72 (2016). https://doi.org/10.1007/978-3-319-48869-1_5, https://doi.org/10.1007/978-3-319-48869-1_5

[20] Ernst, G.: A complete approach to loop verification with invariants and summaries. CoRR **abs/2010.05812** (2020), https://arxiv.org/abs/2010.05812

[21] Ernstedt, P.: Contract-Based Verification in TriCera. Master's thesis, Uppsala University, Department of Information Technology (2022)

[22] Esen, Z.: Extension of the ELDARICA C model checker with heap memory. Master's thesis, Uppsala University, Department of Information Technology (2019)

[23] Esen, Z., Rümmer, P.: Reasoning in the theory of heap: Satisfiability and interpolation. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12561, pp. 173–191. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_9, https://doi.org/10.1007/978-3-030-68446-4_9

[24] Esen, Z., Rümmer, P.: A theory of heap for constrained horn clauses (extended technical report). CoRR **abs/2104.04224** (2021), https://arxiv.org/abs/2104.04224

[25] Esen, Z., Rümmer, P.: TriCera Benchmarks: SMT-LIB Encodings of SV-COMP 2022 Benchmarks by TriCera (Aug 2022). https://doi.org/10.5281/zenodo.6950363, https://doi.org/10.5281/zenodo.6950363

[26] Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021). https://doi.org/10.4204/EPTCS.344.7, https://doi.org/10.4204/EPTCS.344.7

[27] Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Hankin, C., Schmidt, D. (eds.) Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001. pp. 193–205. ACM (2001). https://doi.org/10.1145/360204.360220, https://doi.org/10.1145/360204.360220

[28] Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI. pp. 268–277. ACM, New York, NY, USA (1991). https://doi.org/10.1145/113445.113468, http://doi.acm.org/10.1145/113445.113468

[29] Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012). https://doi.org/10.1145/2254064.2254112, https://doi.org/10.1145/2254064.2254112

[30] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20, https://doi.org/10.1007/978-3-319-21690-4_20

[31] Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants - (competition contribution). In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30, https://doi.org/10.1007/978-3-319-89963-3_30

[32] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259, https://doi.org/10.1145/363235.363259

[33] Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: FMCAD 2018. pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013

[34] Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: Bjørner, N., Fioravanti, F., Rybalchenko, A., Senni, V. (eds.) Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014. EPTCS, vol. 169, pp. 39–52 (2014). https://doi.org/10.4204/EPTCS.169.6, https://doi.org/10.4204/EPTCS.169.6

[35] Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017), https://easychair.org/publications/paper/Pmh

[36] Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 89–96. IEEE (2015)

[37] Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 846–862. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59, https://doi.org/10.1007/978-3-642-39799-8_59

[38] Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26, https://doi.org/10.1007/978-3-642-54862-8_26

[39] Lattner, C., Adve, V.S.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 129–142. ACM (2005). https://doi.org/10.1145/1065010.1065027, https://doi.org/10.1145/1065010.1065027

[40] Lauko, H., Rockai, P.: LART: compiled abstract execution - (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 457–461. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_31, https://doi.org/10.1007/978-3-030-99527-0_31

[41] Leeson, W., Dwyer, M.B.: Graves-cpa: A graph-attention verifier selector (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 440–445. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28, https://doi.org/10.1007/978-3-030-99527-0_28

[42] Malík, V., Schrammel, P., Vojnar, T.: 2ls: Heap analysis and memory safety - (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 368–372. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_22, https://doi.org/10.1007/978-3-030-45237-7_22

[43] Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based Verification for Rust Programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021). https://doi.org/10.1145/3462205, https://doi.org/10.1145/3462205

[44] McCarthy, J.: An interesting lisp function. ACM Lisp Bulletin (3), 6–8 (1979)

[45] Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free Horn clauses. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 361–382. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_18, https://doi.org/10.1007/978-3-662-53413-7_18

[46] Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks - (competition contribution). In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 458–460. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44, https://doi.org/10.1007/978-3-662-46681-0_44

[47] Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Inf. **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134

[48] Richter, C., Wehrheim, H.: Pesco: Predicting sequential combinations of verifiers - (competition contribution). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 229–233. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19, https://doi.org/10.1007/978-3-030-17502-3_19

[49] Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 159–169. ACM (2008). https://doi.org/10.1145/1375581.1375602, https://doi.org/10.1145/1375581.1375602

[50] Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNCS, vol. 5330, pp. 274–289. Springer (2008)

[51] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints - (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of

Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 438–442. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28, https://doi.org/10.1007/978-3-030-72013-1_28

[52] Sánchez, A., Sankaranarayanan, S., Sánchez, C., Chang, B.Y.E.: Invariant generation for parametrized systems using self-reflection - (extended version). In: Miné, A., Schmidt, D. (eds.) SAS. Lecture Notes in Computer Science, vol. 7460, pp. 146–163. Springer (2012). https://doi.org/10.1007/978-3-642-33125-1_12, https://doi.org/10.1007/978-3-642-33125-1

[53] Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., Majzik, I.: Theta: A framework for abstraction refinement-based model checking. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 176–179. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102257, https://doi.org/10.23919/FMCAD.2017.8102257