Towards Automatic Verification of C Programs with Heap

Zafer Esen and Philipp Rümmer

Dept. of IT, Uppsala University, Sweden; {zafar.esen,philipp.ruemmer}@it.uu.se

Abstract

Programs containing dynamic data structures pose a challenge for static verification, because the shape of data-structures has to be reconstructed, and invariants involving unboundedly many heap locations have to be found. In previous work in the context of the Java model checker JayHorn, a simple Horn encoding through object invariants has been proposed to model programs with heap. This abstract presents ongoing work on TriCera, a model checker for C programs with a similar representation of heap interactions.

1 Introduction

Effective handling of heap-allocated data-structures is one of the main challenges in automatic verification approaches such as software model checking. In order to verify programs operating on such data-structures, a verification tool has to analyse the *shape* of the data-structures (which is usually not explicitly expressed in a program), and also has to find *data invariants* that cover an unbounded number of heap locations. A plethora of approaches to address this challenge has been developed over the years; for instance, separation logic [5] provides a general verification methodology, but has mostly been successful in interactive verification systems. In software model checkers, the most common solution is to use a low-level representation of heap as an array; this necessitates quantified invariants to verify programs with unbounded data-structures, which in itself is a challenging research problem.

A different methodology, inspired by *refinement type systems* [2] and based on instance invariants associated with the various classes in a program, is used in the Java model checker JayHorn [4]. The model checker *TriCera*[1], presented in this abstract, applies a similar heap representation to verify C programs, but extends the method to handle also language features not present in Java: among others, pointers to stack-allocated data, and primitive heap-allocated data like integers.

The architecture of TriCera is given in Figure 1. TriCera encodes all programs as sets of Horn clauses, and then uses Eldarica [3] as its backend to try and solve these. This means that state invariants, function contracts, and object invariants are all computed automatically by the Horn solver; in the end, making the whole process fully automatic.

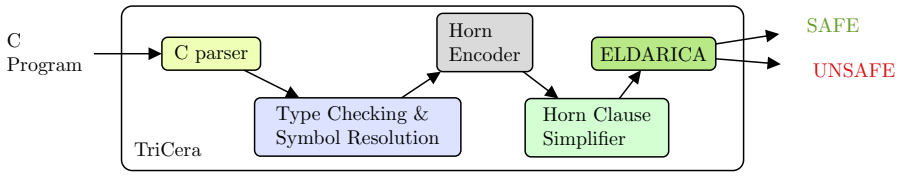

Figure 1: TriCera Architecture

[1] https://github.com/uuverifiers/tricera

```
1  typedef struct S1 {int f;} S1;
2  typedef struct S2 {int f1, f2;} S2;
3
4  S1 *s1_1 = calloc(sizeof(S1));
5  S1 *s1_2 = calloc(sizeof(S1));
6  S2 *s2   = calloc(sizeof(S2));
7  s1_1->f  = 42;       //push operation
8  int t    = s1_1->f; //pull operation
9
10 assert(t == 0 || t == 42);
```

Figure 2: A simple C program.

```
1  typedef struct S1 {int f;} S1;
2  typedef struct S2 {int f1, f2;} S2;
3
4  S1 *s1_1 = H++;     //allocation S1
5  push(s1_1, S1(0)); //push from calloc
6  S1 *s1_2 = H++;     //allocation S1
7  push(s1_2, S1(0)); //push from calloc/
8  S2 *s2 = H++;       //allocation S2
9  push(s2, S2(0,0)); //push from calloc
10 push(s1_1, S1(42); //push from assignment
11 S1 pulled = pull(s1_1);
12 int t = pulled.f
13
14 assert(t == 0 || t == 42);
```

Figure 3: A simple C program where heap interactions are replaced with `push` and `pull` operations.

2 Heap Encoding Using Invariants

Instead of modeling each data item precisely, a *heap invariant* (ϕ_{Type}) is used to represent each data type on the heap. These invariants capture the properties of the data type they correspond to. They are symbolic place-holders, which are later computed automatically by the Horn solver. Interactions with the heap are done via *push* and *pull* operations which use these invariants.

Figure 2 shows a very simplistic C program, and Figure 3 shows its reduced version where heap related operations are automatically replaced with `push` and `pull` operations. These operations are then reduced into `assert` and `assume` statements using the replacement rules given below. The final translation into Horn clauses then follows, which is straightforward.

$$y = \mathbf{pull}(x) \rightsquigarrow \{\mathbf{assume}(\phi_{Type}(ptr, x_{fresh})); \quad y = x_{fresh;}\}$$
$$\mathbf{push}(ptr, val) \rightsquigarrow \mathbf{assert}(\phi_{Type}(ptr, val))$$

C `structs` are encoded using the theory of algebraic data types (ADTs), meaning that they are represented by a single value on the heap similarly to primitive data types.

Figure 4 shows how the heap would look like for the simple program given in Figure 2, and Figure 5 shows how the heap is encoded using invariants. The properties of the objects of same type are captured by the same invariant, as in the case of `S1`.

Memory allocation, in the example using `calloc`[2], is done by assigning the value of the heap counter `H` to the pointer variable, and then incrementing the value of the counter. A zero initialized value is also *pushed* to that location in the case of `calloc`.

Assigning to a variable on the heap can be seen as updating its property, thus the heap invariant must now satisfy the new property as well. This means that the push operation *asserts* the heap invariant using the newly assigned value. On the other hand, reading from the heap can be done by creating a fresh variable which satisfies the properties associated with that type, by *assuming* that the heap invariant holds with this new variable as its argument.

3 Experiments and Results

A total of 114 benchmarks were used to evaluate the initial performance of TriCera. The benchmarks were chosen from files located under the *ReachSafety-Heap* and *MemSafety-Heap* categories of SVCOMP'19[3], and which did not contain unsupported constructs such as arrays.

[2]This differs from the standard C `calloc` function by having no argument for the number of items, as arrays are currently not supported in TriCera.

[3]https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp19

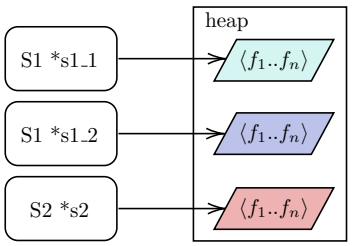

Figure 4: Heap representation as a partial function which maps locations to values

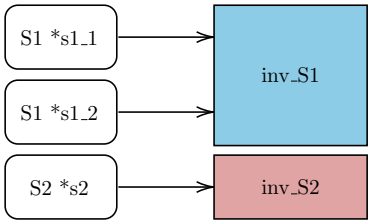

Figure 5: Heap invariants which are created for each type

The benchmark programs are provided as inputs to both TriCera (v0.1) and CPAchecker (v1.8), using the default settings of the tools.

With a timeout of 5 minutes, TriCera could verify 25 (8 safe and 17 unsafe) out of 114 programs in 11 minutes. The rest of the programs were flagged as unsafe due to the imprecision of the current heap encoding. For comparison, CPAchecker [1] could verify 53 (40 safe and 13 unsafe) out of 114 programs in 183 minutes. Both tools produced no unsound results.

While TriCera could verify correctly almost half of what CPAchecker could in total, it took one tenth of the time to do so. However, this was mostly due to CPAchecker timing out trying to verify tasks, on which TriCera gave up much earlier and produced false alarms. It is expected that the refinements discussed in Section 4 should reduce the number of these false alarms, while keeping a similar level of performance with respect to execution time.

4 Conclusions and Future Work

This paper presented the ongoing work with TriCera. The initial results are promising; however, there are several planned improvements to increase the precision, such as using *allocation sites* as done in JayHorn [4] and adding flow sensitivity. There are also plans to to support a wider subset of the C language, such as arrays and pointer arithmetic.

Since the whole encoding is over-approximate, one cannot directly trust the generated counterexamples. To get a genuine counterexample, the encoding can also be complemented with an under-approximate encoding, as done in JayHorn.

References

[1] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proc. of 23rd Int. Conf. on Computer Aided Verification, CAV 2011*, volume 6806 of *Lect. Notes in Comput. Sci.*, pages 184–190. Springer, 2011.

[2] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. of ACM SIGPLAN 1991 Conf. on Program. Language Design and Implementation, PLDI '91*, pages 268–277. ACM, 1991.

[3] Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *18th Int. Conf. on Formal Methods in Computer Aided Design, FMCAD 2018*, 7 pp. IEEE, 2018.

[4] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. Jayhorn: A framework for verifying java programs. In Swarat Chauduri and Azadeh Farzan, editors, *Proc. of 28th Int. Conf. on Computer Aided Verification, CAV 2016, Part I*, volume 9779 of *Lect. Notes in Comput. Sci.*, pages 352–358. Springer, 2016.

[5] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of 17th IEEE Symp. on Logic in Computer Science, LICS 2002*, pages 55–74. IEEE, 2002.