Using Horn Solvers to Generate Memory Access Permissions for Deductive Verification – A Preliminary Report *

Lukas Armborst[0000−0001−7565−0954] Marieke Huisman[0000−0003−4467−072X]
University of Twente, The Netherlands
{l.armborst, m.huisman}@utwente.nl

Deductive verifiers for concurrent software often require the user to specify access permissions to ensure memory safety whenever shared memory is used. There is existing work on automatically generating specifications from source code, such as the TriCera model checker, which is based on constrained Horn clauses. However, the existing approaches usually do not consider memory access permissions. We present our ongoing work to extend the heap memory model in the Princess SMT solver underneath TriCera, where we add permissions and use the existing generation framework to also obtain access permissions.

1 Introduction

In deductive software verification, the developer annotates the source code with the intended behaviour, and the verifier automatically checks that the code complies with that. This approach requires significant effort from the developer to provide the annotations. This burden increases even more if the program is using concurrency: To exclude for example data races, verifiers like VerCors [BDHO17] require that the developer specifies memory access permissions whenever the program tries to access shared memory.

For deductive software verification to be accepted in practise, this effort needs to be reduced, i.e. the tool should be able to derive most annotations from as few lines of user-provided specifications as possible. Several approaches exist to help in this, but they usually do not consider the complexity coming with concurrent memory accesses.

We are therefore extending the approach by Alshnakat et al. [AGLR20] to also infer memory access permissions. They encode the verification problem with constrained Horn clauses, and use the solution from the Horn solver to infer annotations such as method preconditions. We extended the underlying memory model to use access permissions, and leverage the existing Horn inference mechanism to produce the respective source code annotations. In this extended abstract, after providing a brief overview of the background (Section 2), we present our first preliminary results (Section 3), and discuss how to further improve the encoding (Section 4).

Related Work Apart from Alshnakat et al. [AGLR20], there are other attempts to generate functional specifications, but without using Horn solvers, such as Beyer et al. [BSU22], who leverage the automatic verifier CPAchecker, or Daikon [EPG+07], which uses templates. Others do use Horn solvers, like Ezudheen et al. [END+18] who use them as part of a teacher-learner framework for annotations. However, none of them consider permissions. Dohrau [Doh22] does infer permissions, using various strategies including a learning framework, but again not using Horn solvers. Our main work is in extending the heap encoding. Iosif et al. [ISRS18] similarly extend the SMT encoding to support separation logic, but target "pure" rather than permission-based separation logic, as input format for the SL-COMP competition.

*Work on this project is support by the NWO VICI 639.023.710 Mercedes project.

To appear in EPTCS.

2 Background

Permission-Based Separation Logic Permission-based separation logic (PBSL) [AHHH15] is an extension of first-order logic. It has dedicated permission predicates to control access to the heap memory, which is shared among threads and therefore susceptible to data races. Typically, one distinguishes between `read` permission and `write` permission: Either only one thread has `write` permission to a specific heap location, disallowing any simultaneous access to that location by other threads, or multiple threads can all have `read` permission simultaneously. Whenever a method wants to access a heap location, the respective thread needs to have sufficient permission, which means that the permission predicates frequently occur for instance in method contracts. The comments on Lines 8, 9 and 13 of Figure 1 show examples how method contracts with such permission predicates might look like, with `write` permission for `change` and `read` permission for `check`.

Generating ACSL via Horn Solvers Alshnakat et al. [AGLR20] use the Horn solver Eldarica [HR18] to generate specifications for C programs in the ACSL format [BCJF+18], which can be checked for instance with the verifier Frama-C. They take a C program that contains `assert` statements about properties of interest. First, the model checker TriCera [ER22b] transforms this into a set of constrained Horn clauses. In this encoding, method preconditions occur as uninterpreted functions. Eldarica solves the encoded problem using the satisfiability-modulo-theory (SMT) solver Princess [Rüm08]. The solution also contains an interpretation for those functions, which TriCera transforms back to specifications on the C level. While the generated conditions are always sound, their completeness highly depends on the asserted properties in the input, and they can be overly tailored to the specific use case.

Heap Encoding The heap memory is a partial function, mapping (allocated) memory addresses to the stored values. In the SMT solver Princess, this is modelled as a total function [ER22a], with a dedicated $\perp$ object stored at non-allocated addresses: $\mathit{Heap} : \mathit{Addr} \to \mathit{HeapObject}$, where addresses are internally represented by positive integers. Reading is a function $\mathit{read} : \mathit{Heap} \times \mathit{Addr} \to \mathit{HeapObject}$, and writing returns an updated heap: $\mathit{write} : \mathit{Heap} \times \mathit{Addr} \times \mathit{HeapObject} \to \mathit{Heap}$. The type *HeapObject* is an algebraic data type (ADT), with multiple constructors: One for $\perp$, and one wrapper for each type of data to be stored in the heap, for example *O_Int*.

3 Generating Permission Annotations

Extending Heap Encoding with Permissions We use the existing heap encoding as described in Section 2, but added permissions by replacing the *HeapObject* type with a new ADT *PermHeapObject*, which can be either $\perp$ or $\mathit{Field}(\mathit{target}, \mathit{perm})$. The parameter *target* is the data value stored at the location (whose type is the *HeapObject* as defined in Section 2). The parameter *perm* is the permission amount, currently modelled as integers from $[0, 100]$, with 0 representing no permission, 100 being `write` and the rest indicating `read` (see also Section 4).

We defined wrapped functions such as $\mathit{wrappedRead} : \mathit{Heap} \times \mathit{Addr} \to \mathit{HeapObject}$,

$$(h, addr) \mapsto \begin{cases} t & \text{if } \mathit{read}(h, addr) = \mathit{Field}(t, p) \text{ and } p > 0 \\ \perp & \mathit{otherwise} \end{cases}$$

This represents the expected behaviour of *read* with permission logic: The operation only returns a valid value (i.e. not $\perp$), if there is sufficient permission to access that heap location. Note that *wrappedRead*

```
1 void main() {
2   int *arr = (int*) malloc(sizeof(int));
3   if (arr == 0) return;
4   change(arr); change(arr);
5   check(arr);
6 }
7
8 // requires ptr!=NULL && Perm(ptr,write);
9 // ensures Perm(ptr,write) && *ptr==5;
10 /*@ contract @*/
11 void change(int *ptr) { *ptr = 5; }
12
13 // requires a!=NULL && Perm(a,read) && *a==5;
14 /*@ contract @*/
15 void check(int *a) { assert(*a == 5); }
```

```
16
17
18 /* contract for change
19   requires ptr == 1
20         && read(@h, ptr).getPerm == 100;
21   ensures @h == wrappedWrite(\old(@h), ptr, O_Int(5));
22             // == write(\old(@h), ptr, Field(O_Int(5), 100))
23 */
24
25 /* contract for check
26   requires a != nullAddr
27         && read(@h, a).getPerm != 0
28         && read(@h, a).getTarget.getInt == 5;
29   ensures \true;
30 */
```

Figure 1: A simple C program (left) and the generated contracts (right, simplified)

has the same signature as *read* in the original heap encoding in Section 2. All *read* operations in existing SMT files should therefore now use *wrappedRead* instead. The same holds for *write* and *alloc*, which have analogous wrapped versions. A program, which accesses the heap only through these three new functions, never encounters insufficient permissions: *wrappedAlloc* initialises the permission with write, and these functions never change the permission amount afterwards. We can therefore adapt Princess' parser to interpret e.g. `read(h,a)` in an SMT file as meaning $\mathit{wrappedRead}(h,a)$. That way, the permission logic is completely transparent; old SMT files behave as before without requiring any changes, the user reads and writes just as before, and they never have to interact with permissions, the ADT *PermHeapObject*, et cetera.

However, we can also define new Princess functions that change the permission amount, for instance when modelling thread forking (see Section 4). When using those functions, it is possible that for example *wrappedRead* afterwards encounters insufficient permissions and enters its second case. To prevent that, when TriCera translates a C file to SMT, it adds checks before each read and write operation to ensure that there is enough permission. For instance, the C pointer operation $*p = i$ also generates a check that the *perm* at address p allows write access. These additional SMT assertions influence the solver result of Princess, which in turn triggers the annotation generation on the TriCera level, such that for example a generated method precondition should also require that there is write permission for p.

Preliminary Results Figure 1 shows on the left an example C program, with comments indicating how manually written PBSL contracts might look like. On the right, there are the generated contracts for two methods. These contracts are simplified here, for example removing duplicate clauses, and omitting some type checks. They refer to the current heap as $@h$. Note that the write access in `change` caused the generated contract to require a permission of 100 for `ptr` (Line 20), representing write. For `check`, read is sufficient, represented by a non-zero permission amount (Line 27). The argument for `check` is required to be a valid address (no null pointer, Line 26), while for `change` it is overfitted to be exactly the address of this use case, i.e. 1 (Line 19). The postcondition of `check` is overapproximated to *true*, because there are no further assertions or memory accesses in the `main` program. Meanwhile, `change` assures that the only change to the current heap is the written value 5 at `ptr` (Line 21, with Line 22 showing its meaning with the definition of *wrappedWrite* applied). While the generated contracts are mostly similar to the manually written ones on the left, this postcondition is the most different: Manually, we would specify the write permission and the new value. The generated contract combines them in the *Field* object. The

generated contract makes explicit that this is the only change, by performing a single write operation on the old heap, i.e. the one before the method execution. In contrast, the manual contract leaves this implicit: `change` never has permission to any other location, so the logic of PBSL guarantees that no other location has been changed.

4 Next Steps

The work on this project is ongoing, and there are several improvements we plan to address in the short term. First, we want to add support for *resource predicates*. They bundle permission predicates together, including other resource predicates. This allows reasoning about unbounded data structures, such as linked lists, by recursively containing the respective resource predicate for the tail of the list. Second, permission-based separation logic targets concurrent programs, so we want to add support for example for forking threads, and distributing permissions across them. Ideally, the solver can determine which thread requires which permissions, similar to determining pre- and postconditions. However, some manual guidance about how to split the heap may be required. Another important kind of concurrency constructs are locks. They are expected to work similar to resource predicates, so the implementation details will become clear once the predicate implementation is finalised. Third, we are investigating how to best represent various permission amounts. One frequently used model are fractional permissions [Boy03], where `write` is represented by 1, and `read` by a fraction between 0 and 1. However, fractions are difficult to reason about, so counting permissions [BCOP05] or symbolic permissions [HM15] may be more suitable. Lastly, Figure 1 shows that the generated annotations are still using low-level functions of the heap encoding. For better usability, they should more closely resemble actual verifier syntax, similar to the manually provided contracts in the figure.

5 Conclusion

We presented our ongoing work about using Horn solvers to generate memory access permissions for permission-based separation logic, to support deductive verification efforts. In particular, we described how we integrated permissions into the existing heap model of Princess, and used TriCera to generate method contracts. We also mention some improvements that we are currently implementing.

References

[AGLR20] Anoud Alshnakat, Dilian Gurov, Christian Lidström & Philipp Rümmer (2020): *Constraint-Based Contract Inference for Deductive Verification*, pp. 149–176. *Lecture Notes in Computer Science* 12345, Springer International Publishing, `https://doi.org/10.1007/978-3-030-64354-6_6`.

[AHHH15] A. Amighi, C. Haack, M. Huisman & C. Hurlin (2015): *Permission-based separation logic for multi-threaded Java programs*. *LMCS* 11(1), `https://doi.org/10.2168/LMCS-11(1:2)2015`.

[BCJF+18] P. Baudin, P. Cuoq, J.C.-Filliâttre, C. Marché, B. Monate, Y. Moy & V. Prevosto (2018): *ACSL: ANSI/ISO C Specification language, version 1.14*. Available at `https://frama-c.com/download/acsl-1.14.pdf`.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn & Matthew Parkinson (2005): *Permission accounting in separation logic*. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pp. 259–270, `https://doi.org/10.1145/1040305.1040327`.

[BDHO17] S. Blom, S. Darabi, M. Huisman & W. Oortwijn (2017): *The VerCors Tool Set: Verification of Parallel and Concurrent Software*. In Nadia Polikarpova & Steve Schneider, editors: *integrated Formal Methods 2017*, *LNCS* 10510, Springer, pp. 102 – 110, https://doi.org/10.1007/978-3-319-66845-1_7.

[Boy03] J. Boyland (2003): *Checking Interference with Fractional Permissions*. In Radhia Cousot, editor: *SAS*, *LNCS* 2694, Springer, pp. 55–72, https://doi.org/10.1007/3-540-44898-5_4.

[BSU22] Dirk Beyer, Martin Spiessl & Sven Umbricht (2022): *Cooperation Between Automatic and Interactive Software Verifiers*. In Bernd-Holger Schlingloff & Ming Chai, editors: *Software Engineering and Formal Methods (SEFM)*, *Lecture Notes in Computer Science* 13550, Springer International Publishing, pp. 111–128, https://doi.org/10.1007/978-3-031-17108-6_7.

[Doh22] Jérôme Dohrau (2022): *Automatic Inference of Permission Specifications*. Ph.D. thesis, ETH Zurich.

[END+18] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg & P. Madhusudan (2018): *Horn-ICE learning for synthesizing invariants and contracts*. *Proc. ACM Program. Lang.* 2(OOPSLA), https://doi.org/10.1145/3276501.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz & Chen Xiao (2007): *The Daikon system for dynamic detection of likely invariants*. *Science of Computer Programming* 69(1–3), pp. 35–45, https://doi.org/https://doi.org/10.1016/j.scico.2007.01.015. Tool website: https://plse.cs.washington.edu/daikon/.

[ER22a] Zafer Esen & Philipp Rümmer (2022): *An SMT-LIB Theory of Heaps*. In David Déharbe & Antti E. J. Hyvärinen, editors: *Proceedings of the 20th International Workshop on Satisfiability Modulo Theories*, *CEUR Workshop Proceedings* 3185, CEUR-WS.org, pp. 38–53. Available at https://ceur-ws.org/Vol-3185/paper1180.pdf.

[ER22b] Zafer Esen & Philipp Rümmer (2022): *Tricera: Verifying C Programs Using the Theory of Heaps*. In Alberto Griggio & Neha Rungta, editors: *2022 Formal Methods in Computer-Aided Design (FMCAD)*, IEEE, pp. 380–391, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_45.

[HM15] Marieke Huisman & Wojciech Mostowski (2015): *A Symbolic Approach to Permission Accounting for Concurrent Reasoning*. In Daniel Grosu, Hai Jin & George Papadopoulos, editors: *14th International Symposium on Parallel and Distributed Computing*, IEEE, pp. 165–174, https://doi.org/10.1109/ISPDC.2015.26.

[HR18] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In Nikolaj Bjørner & Arie Gurfinkel, editors: *2018 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, pp. 1–7, https://doi.org/10.23919/FMCAD.2018.8603013.

[ISRS18] Radu Iosif, Cristina Serban, Andrew Reynolds & Mihaela Sighireanu (2018): *Encoding separation logic in SMT-LIB v2.5*. Available at https://sl-comp.github.io/docs/smtlib-sl.pdf.

[Rüm08] Philipp Rümmer (2008): *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* 5330, Springer, pp. 274–289, https://doi.org/978-3-540-89439-1_20.