Giving Some Pointers for Abstraction-Based Model Checking

Levente Bajczi, Dániel Szekeres, Csanád Telbisz, Vince Molnár
Budapest University of Technology and Economics
Department of Artificial Intelligence and Systems Engineering
Budapest, Hungary
Email: {bajczi, szekeres}@mit.bme.hu, csanadtelbisz@edu.bme.hu, molnarv@mit.bme.hu

***Abstract*—Abstraction-based software model checkers often rely on external analyses or unbounded SMT arrays to reason about pointers, arrays, and dynamic heap manipulation. External analyses are precise but often require the modification of existing verification algorithms, while SMT arrays provide a native solution for solver-based verifiers but require strict type safety often forgone in real-world programs. We propose a novel way of integrating a precise pointer and array analysis as a plug-in for abstraction-based model checking, which does not require the modification of the underlying algorithms. Our solution keeps track of arbitrary predicates over potentially abstract memory locations, moving toward more efficient verification of software code by allowing a fine-grained and precise abstraction of memory accesses.**

I. Introduction

Verifying software remains a complex yet necessary task in modern systems engineering. One source of complexity comes from the use of dynamic data structures such as heap-allocated arrays. , and their formal analysis remains a vital part of practical software verification, especially as more and more applications rely on formal methods for safety guarantees. In this paper, we refer to Satisfiability Modulo Theories (SMT) solver-aided verification [1] as *software model checking*.

State-of-the-art software model checking tools generally fall into two categories in terms of heap analysis. They either encode the heap as unbounded SMT arrays [2] (which can be done either for each array individually, or monolithically per type [3]), or use an auxiliary analysis such as symbolic memory graphs (SMGs) [4] to keep track of pointer (and thus, array) information. However, for *abstraction-based* analyses [1], both solutions are problematic. Auxiliary analyses often result in both heightened complexity and a performance penalty, monolithic heap encoding as a single SMT array per type is also often costly as the solver needs to reason about the heap as a whole, while direct SMT-encoding of individual arrays often lacks support for the type conversions necessary for verifying real-world programs.

Consider for example a C function that takes a `void*` pointer, and depending on external factors, dereferences it as either an `int` or a `float`. Given their sizes are equivalent (e.g., in the `ILP32` architecture[1]), this is valid in C, and sometimes, such as when passing an argument to a new thread via `pthread_start`, even necessary. An SMT encoding, however, cannot implement this without somehow relying on a different array expression for each type. Therefore, every pointer used as an array has to have multiple encodings, which must also be kept in sync when updated.

We propose a solution that integrates directly into existing Counterexample-Guided Abstraction Refinement (CEGAR) [5] frameworks (sparing the complexity of standalone analyses), while also providing an easy and performant way of relying on heap information in the abstract state space. We also extend this approach to handle stack-based arrays, structs, and references to variables. We also discuss how our approach can be integrated with analyses relying on a dependency relation among transitions (e.g., partial order reduction).

II. Informal example

Consider the program in Fig. 1. The variant in Fig. 1a declares two variables a and b, both initialized to 0, then based on a nondeterministic choice (such as user input) sets the pointers c and d to each point to different ones. Afterwards, the program sets the memory location at address c to 1, and the memory location at address d to 2. It is required that at least a or b has the value 1, because we know that

1) c points to a or b, and
2) d cannot alias c, thus cannot overwrite the value 1.

Therefore, we expect a program analysis verdict of *safe*.

A. Model Checking without Abstraction

First, we construct a second variant of the same program Fig. 1a, seen in Fig. 1b – a and b are now pointers themselves, initialized to unique memory address via the call to `malloc`, and instead of references to a or b, c and d simply alias exactly one of them, each. The requirement is the same: at the memory location pointed to by either a or b, we must find the value 1. The advantage of this form is the lack of the `&` (reference) operator, thus we only need to handle the $\star$ (dereference) operator when we encode the data flow in an SMT formula.

This research was partially funded by the EKÖP-24-{2,3} New National Excellence Program under project numbers EKÖP-24-2-BME-118, EKÖP-24-3-BME-213 and EKÖP-24-3-BME-159, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023 and 400443/2023; funded by the NRDI Fund of Hungary.

[1] https://unix.org/whitepapers/64bit.html

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c =  cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

(a) Before preprocessing

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

(b) After preprocessing

Fig. 1: Program showcasing pointer aliasing

```
1 int arr[2];
2 arr[0] = 0;
3 arr[1] = 0;
4 int cond = nondet();
5 int c =  cond?0:1;
6 int d = !cond?0:1;
7 arr[c] = 1;
8 arr[d] = 2;
9 assert(
      arr[0]==1||arr[1]==1);
```

Fig. 2: Aliasing via offsets

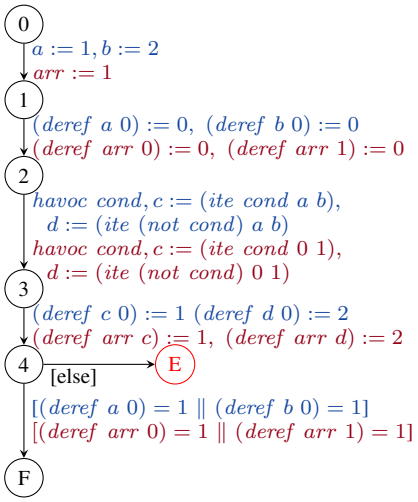

Fig. 4: CFA of Fig. 1b and Fig. 2

```
1 (declare-fun deref (Int Int Int) Int) ; (ptr, offset, idx) -> value
  (declare-fun arr () Int) (declare-fun c () Int) (declare-fun d () Int)
  (declare-fun cond () Bool)                          ; havoc cond
2 (assert (= arr 1))                                  ; unique arbitrary address
3 (assert (= (deref arr 0 1) 0)) (assert (= (deref arr 1 1) 0))
4 (assert (= c (ite cond 0 1))) (assert (= d (ite (not cond) 0 1)))
5 (assert
    (let ((idx1 (+ 1 (ite (= c 1) 1 (ite (= c 0) 1 0)))))
    (let ((idx2 (+ 1 (ite (= d c) idx1 (ite (= d 1) 1 (ite (= d 0) 1 0))))))
      (and (= (deref arr c idx1) 1)                   ; previous idx + 1
           (= (deref arr d idx2) 2)                   ; previous idx + 1
6          (let ((idx3 (ite (= 0 d) idx2 (ite (= 0 c) idx1 1))))
           (let ((idx4 (ite (= 1 d) idx2 (ite (= 1 c) idx1 1))))
             (or (= (deref arr 0 idx3) 1)             ; previous idx
                 (= (deref arr 1 idx4) 1))))))))      ; previous idx
```

Fig. 3: SMT-encoding of Fig. 2 over the trace from 0 to F in Fig. 4

ptr	*off*	*idx*	(*deref ptr off idx*)
1	0	1	0
1	1	1	0
1	0	2	**1**
1	1	2	**2**

Fig. 5: One of the models for *deref* in Fig. 3 (final results in **bold**)

Then, we construct a Control Flow Automaton (CFA) [6] seen in Fig. 4[2] where we also show the CFA of the program in Fig. 2 (shown in a different color on the edges), which is functionally the same as Fig. 1b but with a single stack-allocated array and two literal offsets instead of a and b. a, b, and arr are initialized to unique but unimportant values (here we used a small natural number for each), representing the memory address. For the dereferences in both cases (either via the $\star$ or the `[]` operator), we use an uninterpreted SMT function `deref`, taking two values: a *basis* pointer address and an *offset*. For $*a$ and $*b$ in Fig. 1b the bases are unique addresses returned by `malloc` and the offsets are 0, while for $arr[0]$ and $arr[1]$ in Fig. 2 the bases are the same (arr), and the offsets are 0 and 1.

Because the programs are *safe*, encoding the path from the *initial* location to the *error* location should be `UNSAT`, therefore, for demonstration purposes, let us encode the path to the *final* location instead in Fig. 3, which should be *SAT*. We use the array variant in Fig. 2.

After the function declarations in line 1, we assign a unique arbitrary address (here, 1) to arr in line 2. Then based on the nondeterministic value of $cond$, we set the values of c and d to either 0 or 1 in line 4. The writes to arr are encoded in line 5, and the reads for the `assert` are encoded in line 6.

Notice using a three-parameter `deref` in the SMT encoding instead of the two-parameter one in the CFA. The idea is similar to the static single assignment (SSA) [8] used for regular variable encoding for SMT queries: constants are created by associating indexes with variable usages, where each time a variable is updated, its corresponding index is incremented. This allows us to reason about changing values at a memory address, rather than a constant. We encode this index symbolically in the third (index) parameter of the `deref` function. In line 3, we know that the array updates are the first in the program, therefore we can use a literal 1 index. In line 5 however, we do not know if $arr[c]$ overwrites $arr[0]$ or $arr[1]$, so we must construct an expression that evaluates to the correct index instead of a literal. We use the parameter `idx1` for this, which must be 1 greater than the previous index, because we overwrite the value. The previous index is 1 if $c = 1$ or $c = 0$. Otherwise, the previous index is 0, referring to the uninitialized memory at the beginning of execution. We can also construct `idx2` the same way, but we must also consider the case where $d = c$, in which case the previous index is `idx1`. We use the same logic to construct `idx3` and `idx4` in line 6, but we do not increment the previous indices because these are *read* accesses.

By querying an SMT solver with the input in Fig. 3, we can retrieve a model for the `deref` function (see Fig. 5). Each entry in the table refers to a value in the memory throughout the execution of the program, and entries with the highest *idx* value for any given *(ptr, off)* pair represent the last write to the specific memory address, thus representing the state of the memory at the end of execution.

[2]We show the CFA in a partial Large Block Encoding (LBE) [7] for readability, but single block encoding or full LBE could also be used

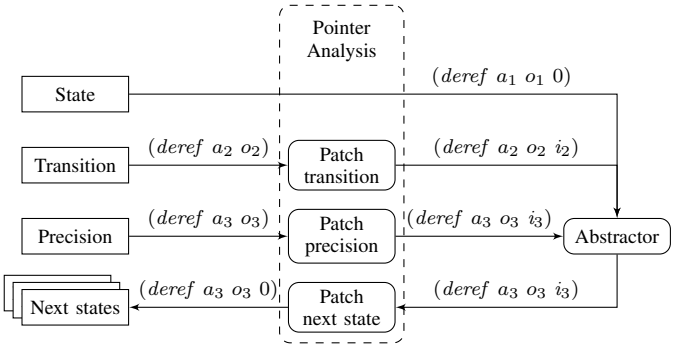

Fig. 6: Integration into abstraction frameworks. Labels denote example subexpressions in the in- and output of the abstractor.

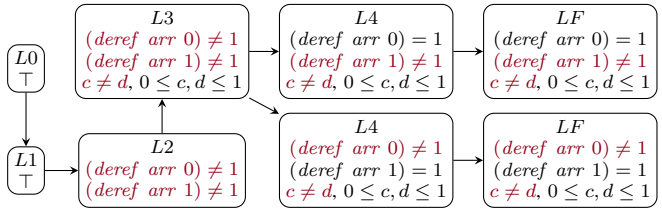

Fig. 7: ARG for Fig. 2 with $\Pi = \{(deref\ arr\ 0) = 1; (deref\ arr\ 1) = 1; c = d; 0 \leq c, d \leq 1\}$

B. Model Checking with Abstraction

Instead of using a monolithic SMT formula to represent a trace in the program, we can rely on abstraction to reason about the reachable state space of the program. Let us consider the same program in Fig. 2, using predicate abstraction [6] with the predicates $\{arr[0] = 1; arr[1] = 1; c = d; 0 \leq c, d \leq 1\}$ and explicit location tracking.

We assume we can rely on an existing predicate abstractor (that takes a state, a transition, and a precision and returns a list of successor states), and only create a *pointer analysis* adapter that patches the input and output of that abstractor, as seen in Fig. 6.

Because we want to represent the state of the memory in each abstract state, we use 0 as the index to the three-parameter `deref` in the state labels, and re-start indexing in every successor state calculation. Therefore we do not need to adapt the state's expression when invoking the abstractor. We do need to patch the transition, however, because on the labels of the CFA all *deref* calls had two parameters only. We calculate a suitable index for every dereference as demonstrated in Fig. 3, and pass it to the abstractor. We do not use dereferences preceding the input state in the index calculation, only those that come afterward – should a read not find a suitable previous write in the transition, it will read the 0-index value included in the state.

We also insert an additional read access at the end of the transition for every dereference present in the precision, thus querying the resulting values in the memory, including those (patched to use 0 as indices) in the next states when the truth value of a predicate is established.

Using this method, we can construct the abstract reachability graph (ARG) [6] in Fig. 7 containing no reachable abstract error state. We can thus conclude that the program is *safe*. Had we found an abstract counterexample, we would have checked its conretizability to avoid spurious results by using the approach in Sect. II-A. If a spurious counterexample is reached, we refine the precision using its refutation (e.g., via interpolation), remembering to remove the indices from all dereferences.

C. Advantages of `Deref`

While the presented encoding is similar to other pointer handling techniques, our approach combines and extends their advantages as follows:

Performance We only keep track of "interesting" abstract places in the memory (ensured by precision refinement)

Precision We can rely on precise aliasing information (similar to dedicated analyses [4]) but only when deemed necessary by the refiner, thus sparing overly detailed tracking

Array polymorphism Arrays can be polymorphic and require no explicit conversion when placing different types in an array (a trait of uninterpreted SMT functions)

Flat struct support Due to the aforementioned polymorphism, structs can be modeled as arrays, even when containing elements of different types

To expand on the struct support and polymorphism, consider a mapping from `struct A{char c; float f;}` to the `deref` function. While structs by themselves can be mapped to individual variables via a preprocessing step, when combined with pointers (especially when arrays-of-structs are used), this transformation can be hindered. Therefore, we deal with structs as if they were polymorphic arrays themselves. When accessing a value, the base pointer can be an abstract memory location a, and the offset can be the index of the field in the type (1 for `c` and 2 for `f`). Because these indices never change, $(deref\ a\ 1)$ will always refer to a 1-byte bitvector, and $(deref\ a\ 2)$ a single-precision float. Special care needs to be taken in cases when the semantics of the source language (such as C) passes the struct by value rather than by pointer or reference, as a deep copy of the values must be made in these cases (so that any modification will not be reflected in the original struct).

III. Assumptions and Consequences

We assume the following about the input program.

a) Program Traits: We assume the input is a control flow automaton (CFA) [6] with variables $V = \{v_1, v_2, \ldots, v_n\}$ over domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$, locations L, and edges $E \subseteq L \times Ops \times L$, where Ops can have $assume(expr)$, $assign(expr_{lhs}, expr_{rhs})$, and $havoc(v \in V)$ instructions for guards, variable updates, and nondeterministic value assignments, respectively. Sequential combination of operations are permitted in the complex operation $seq(op_1, ...)$, which is a shorthand for a sequence of edges and locations with individual operations. Domains of variables are subsets of domains

in SMT. Expressions can be variables, literals, and domain-specific operations over expressions. Besides conventional operators, we also use a distinguished binary operator `deref`, which maps a base pointer and offset pair to an in-memory value. Pointers and offsets are integers. Conventionally, the left-hand side of assignments could only be variable references, but we allow `deref` expressions as well, to facilitate address-based updates. While executing, the state of the CFA is identified by a valuation over V, the current location $l \in L$, and the state of the global memory, given by a collection of base-offset-value triples (we use a 2D memory model [2]). We assume that a memory location is identified uniquely by its base- and offset (an array cannot "index into" another array).

b) Pointer Semantics: We further assume that input programs follow the C standard [9] for pointer-, array- and heap-manipulating instructions. Furthermore, all memory accesses are *correct*, because our primary goal is answering error state reachability queries, for which we can assume no undefined behaviour may occur on trajectories leading up to the error state. Therefore we use no bounds-checking, nor do we check for invalid `free` usages.

c) Eliminating References: We rely on the CFA not containing variable references, provided by a pre-transformation step that transforms all referred variables to dynamically allocated 1-element arrays. Therefore, references will become variable accesses (see Fig. 1), and previous variable accesses will become dereferences. Because variables can only be referenced a finite amount of times, this pre-transformation step is always possible, and we can always rely on the CFA not containing any references.

d) Transforming Dereferences: During analysis (i.e., calculating successor states from a state and a CFA edge), we transform the binary `deref` instruction to ternary `deref` expressions for the SMT solver. The third argument is an index. The index 0 refers to the value in memory before any known writes (either uninitialized, or referring to the value in the previous state). A lower-index `deref` expression is considered overwritten by a higher-index one, if their base and offset values match. Indices are incremented when the corresponding `deref` expression is present on the left-hand side of an assignment. The highest index expression will appear in the successor state as the in-memory value for the corresponding base and offset.

e) Considerations for Dependency-Based Analyses: Algorithms used for the verification of concurrent software often require identifying dependency between interacting program operations from different threads [10]. Without pointers, this can be easily performed by a syntactic check on shared variable accesses. However, with pointers, this is insufficient: we also have to check if the operations access the same memory location. Since we may not know the exact basis and offset values of parallel memory accesses, we can use an SMT solver to decide whether they can point to the same memory location provided the information in the current (abstract) state. This is potentially a very expensive operation, but can provide accurate information in the abstract state space. A possible alternative is to use a safe overapproximation, like a static *points-to analysis* [11], which can detect dependencies of CFA edges as a pre-processing step.

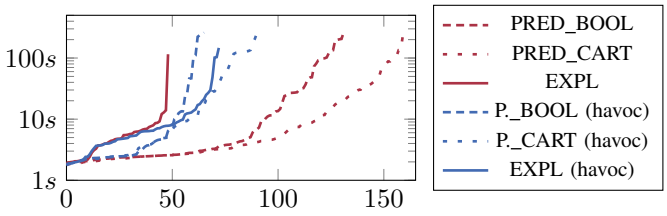

Fig. 8: Quantile plot of results

IV. Practical Evaluation

To evaluate the proposed approach, we developed a proof-of-concept implementation in the Theta model checking framework [6]. We introduced an adapter layer between the XCFA frontend [12] and the CEGAR backend [6], which implements the necessary transformations of the expressions exchanged between the two regarding pointers. We aimed to answer the following research questions:

RQ1 For verification tasks containing pointers, can the proposed approach solve more tasks than using the *havoc* memory model (where all reads are nondeterministic)?

RQ2 For verification tasks containing pointers, which abstract domain is better: explicit value, or predicate analysis?

Answering **RQ1** helps in establishing the rationale for implementing the pointer analysis extension for the abstractor. With the *havoc* memory model, all reads are nondeterminstic, thus they force coarser abstraction and may hinder safety proofs. Answering **RQ2** can result in suggestions for potential users on abstract domain choice when faced with verification problems containing pointers.

To facilitate the experiments, we selected 633 error label reachability tasks containing pointers from the sv-benchmarks repository[3] which can be parsed by Theta. We ran Theta with predicate and explicit value abstraction, using sequential interpolation in the refinement step [6]. We executed these tests on 2 dedicated AMD EPYC 7352 cores each, with 16GB of memory and 5 minutes of timeout. We used BenchExec [13] for precise resource measurements. The results can be seen in Fig. 8, with axis x displaying the number of solved tasks and axis y the time necessary to achieve that many solutions. There were no incorrect results.

RQ1 and **RQ2** can both be answered directly from the plot: for the explicit value analysis, it is *not worth* using the proposed new technique as it presumably introduces such an overhead that previously solved tasks now become unsolvable within the time limit. Predicate abstraction with the new approach, however, outperforms the *havoc* memory model with either abstract domain. Therefore, we can conclude that

[3]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

there is value in implementing the technique for predicate abstraction-based analyses. However, the performance differences between the two predicate abstraction strategies require further investigation.

For the explicit value analysis, we theorize that the lack of explicit *memory state* value information leads to the diminished performance. Therefore, we will attempt to realize this addition to the explicit abstract domain in the future, and re-evaluate the proposed approach.

REFERENCES

[1] D. Beyer, M. Dangl, and P. Wendler, "A Unifying View on SMT-Based Software Verification," *J. Autom. Reason.*, vol. 60, no. 3, pp. 299–335, 2018. [Online]. Available: https://doi.org/10.1007/s10817-017-9432-6

[2] H. Tuch, "Formal memory models for verifying C systems code," Ph.D. dissertation, University of New South Wales, Sydney, Australia, 2008. [Online]. Available: http://handle.unsw.edu.au/1959.4/41233

[3] R. Burstall, "Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7," *American Elsevier, New York*, 1972.

[4] D. Baier, "Implementation of Value Analysis over Symbolic Memory Graphs in CPAchecker," Master's thesis, Ludwig-Maximilians-Universität München, Institut für Informatik, 2022.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169. [Online]. Available: https://doi.org/10.1007/10722167_15

[6] Á. Hajdu and Z. Micskei, "Efficient Strategies for CEGAR-Based Model Checking," *J. Autom. Reason.*, vol. 64, no. 6, pp. 1051–1091, 2020. [Online]. Available: https://doi.org/10.1007/s10817-019-09535-x

[7] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009, pp. 25–32. [Online]. Available: https://doi.org/10.1109/FMCAD.2009.5351147

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[9] "Programming languages — C," International Organization for Standardization, International Electrotechnical Commission, International Standard, Dec. 2010.

[10] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 373–384, 2014.

[11] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 32–41.

[12] L. Bajczi, Z. Ádám, and V. Molnár, "C for yourself: comparison of front-end techniques for formal verification," in *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, ser. FormaliSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–11. [Online]. Available: https://doi.org/10.1145/3524482.3527646

[13] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 1, p. 1–29, feb 2019. [Online]. Available: https://doi.org/10.1007/s10009-017-0469-y