A Theory of C-Style Memory Allocation*

Stephan Falke, Florian Merz, and Carsten Sinz

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{stephan.falke, florian.merz, carsten.sinz}@kit.edu
http://verialg.iti.kit.edu

Abstract. This paper introduces the theory $\mathcal{T}_{\mathcal{H}}$ for reasoning about the correctness of memory access operations in the context of a C-style heap memory. The proposed approach makes a clear distinction between reasoning about the values stored in memory and checking whether access to a specific memory location is allowed. The theory provides support for `malloc` and `free` and is presented in the form of axioms that can be converted into conditional rewrite rules. It is also shown how $\mathcal{T}_{\mathcal{H}}$ can be used in a bounded model checker for C programs.

1 Introduction

Reasoning about memory access operations is an important part of many program verification tasks. Memory access checks can, e.g., be used to detect heap or stack buffer overflows which may be exploited by malware in attacks. In general, accessing unallocated memory can result in unpredictable program behavior, loss of data, or program crashes.

Whereas several approaches for formalizing computer memory have been presented in the past (see, e.g., [1, 2, 6, 7, 11–14, 17, 18]), models of heap (or stack) memory access control are not as widespread. This is in particular true for weakly-typed programming languages such as C.

This paper develops the theory $\mathcal{T}_{\mathcal{H}}$ for reasoning about validity of heap memory access operations. $\mathcal{T}_{\mathcal{H}}$ is suitable for a C-like memory management system using function calls to `malloc` and `free` for allocating and deallocating memory on the heap. The formalization of $\mathcal{T}_{\mathcal{H}}$ has similarities to the theory of arrays $\mathcal{T}_{\mathcal{A}}$ which is governed by McCarthy's axioms for array read and write operations (sometimes also called read-*over*-write *axioms*)

$$\begin{aligned} p = q &\Rightarrow \mathsf{read}(\mathsf{write}(a, p, x), q) = x \\ p \neq q &\Rightarrow \mathsf{read}(\mathsf{write}(a, p, x), q) = \mathsf{read}(a, q) \end{aligned}$$

These axioms state that writing the value x into an array a at index p and subsequently reading a's value at index q results in the value x if indices p and q are identical. Otherwise, the read operation is not influenced by the preceding write operation. Arrays are often used to model the content of computer memory. In the programming language C, memory can be regarded as a large array of byte values.

In [16], we have extended $\mathcal{T}_{\mathcal{A}}$ with capabilities for reasoning about the correctness of memory access operations by adding suitable global constraints formalizing heap properties and memory access correctness predicates. There are two major drawbacks to this approach. First, the approach does not perform local reasoning but requires knowledge about all past heap-modifying operations. This global view does not lend itself very well to modular reasoning. Second, the approach does not provide a "separation of concerns", i.e., memory access control is intermixed with read and write operations. `malloc` and `free` modify the state

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

of the memory allocation system, but do not modify the memory content in any of the allocated memory blocks. Memory write operations, on the other hand, modify the content of allocated memory blocks, but do not change the state of the memory allocation system. Memory accesses and their correctness are thus separate concepts. Because of this, memory content and memory allocation state should be represented by different objects since this makes it possible to reason about them separately.

2 Background

We first recall McCarthy's theory of arrays $\mathcal{T}_{\mathcal{A}}$.

Sorts	E : elements I : indices A : arrays
Functions	$\mathsf{read} : A \times I \to E$ $\mathsf{write} : A \times I \times E \to A$
Axioms	$p = q \;\Rightarrow\; \mathsf{read}(\mathsf{write}(a, p, x), q) = x$ $p \neq q \;\Rightarrow\; \mathsf{read}(\mathsf{write}(a, p, x), q) = \mathsf{read}(a, q)$

Objects of sort A denote arrays, i.e., maps from indices of type I to elements of type E. The write function is used to store an element in an array. Its counter-part, the read function, is used to retrieve an element from an array.

In SMT-solvers for $\mathcal{T}_{\mathcal{A}}$, the read-over-write axioms are typically applied from left to right using the *if-then-else* operator ITE, i.e., a term $\mathsf{read}(\mathsf{write}(a, p, x), q)$ is replaced by $\mathsf{ITE}(p = q, x, \mathsf{read}(a, q))$. After this transformation has been applied exhaustively, only read operations remain, which can then be treated as uninterpreted functions. The resulting formula can—if needed—be further transformed into pure equality logic using Ackermann's construction: for all array variables a, let Q_a be the set of all index arguments that occur in a read operation for a. Then, each occurrence of $\mathsf{read}(a, q)$ is replaced by a fresh variable A_q, and further (consistency) constraints of the form $q_1 = q_2 \;\Rightarrow\; A_{q_1} = A_{q_2}$ for all $q_1, q_2 \in Q_a$ are added as constraints to the formula. An alternative way to deal with McCarthy's axioms was presented in [4], adding instances of this axiom lazily (on demand) in a refinement loop.

3 The Theory $\mathcal{T}_{\mathcal{H}}$

This section gives the signature and axioms of the theory $\mathcal{T}_{\mathcal{H}}$.

Sorts	I : indices (pointers) S : sizes H : allocation system states
Functions	$\varepsilon : \to H$ $\mathsf{malloc} : H \times I \times S \to H$ $\mathsf{free} : H \times I \to H$ $\mathsf{mallocsize} : H \times I \to S$
Predicates	$\mathsf{accessible} : H \times I \times S$ $\mathsf{freeable} : H \times I$ $\mathsf{mallocable} : H \times I \times S$

ε denotes an "empty" heap object, i.e., a heap to which no memory allocation or deallocation operations have been applied. $\mathsf{malloc}(h, p, s)$ denotes the heap obtained from h by allocating a memory block of size s starting at address p (if the allocation is possible, i.e., if the block does not overlap with previously allocated blocks; otherwise the heap state is not modified). Accesses to this memory region are valid in the new heap. $\mathsf{free}(h, p)$ denotes the heap obtained from h by freeing the memory block starting at address p (if it is currently allocated; otherwise the heap state is not modified). Accesses to this memory region are invalid in the new heap. $\mathsf{mallocsize}(h, p)$ returns the size s if p is the first address of a memory region $[p, p + s)$ that is currently allocated in h.

The theory $\mathcal{T}_{\mathcal{H}}$ does not allow equality tests between objects of sort H. Thus, extensionality axioms for the equality of heap states are not needed.

The predicate $\mathsf{accessible}(h, p, s)$, the main predicate of $\mathcal{T}_{\mathcal{H}}$, is used for checking validity of memory read and write operations. It determines whether access to the memory region $[p, p + s)$ is valid in the heap h, i.e., whether it falls completely within a currently allocated memory region. $\mathsf{freeable}(h, p)$ determines if p is the first address of a currently allocated memory region. In this case, the memory region pointed to by p can safely be deallocated. Finally, $\mathsf{mallocable}(h, p, s)$ determines whether the memory region $[p, p + s)$ can be allocated in h, i.e., does not interfere with any other currently allocated memory region.

In order to simplify presentation, we restrict ourselves to $I = S = \mathbb{N}$ in the following. Alternatively, fixed-width bitvectors could be used (and we do so in our implementation).[1]

Auxiliary Predicates. For memory regions $[p, p + s)$ and $[q, q + t)$, the predicate $\mathsf{disjoint}(p, s, q, t)$ determines whether the regions are disjoint:

$$\mathsf{disjoint}(p, s, q, t) := p + s \leq q \vee q + t \leq p$$

The predicate $\mathsf{contained}(p, s, q, t)$ determines whether the memory region $[q, q + t)$ is completely contained in the region $[p, p + s)$:

$$\mathsf{contained}(p, s, q, t) := p \leq q \wedge q + t \leq p + s$$

Axioms for mallocable. Recall that the intended semantics of $\mathsf{mallocable}(h, p, s)$ is that the region $[p, p+s)$ can be allocated in h. The exact meaning of this is explained in the following. First, $\mathcal{T}_{\mathcal{H}}$ assumes that mallocs of size zero are not allowed.[2] Thus,

$$\mathsf{mallocable}_{\mathsf{size}}(h, p, s) \Leftrightarrow s \neq 0$$

Additionally, it needs to be ensured that distinct mallocs do not allocate overlapping regions of the heap. There are several ways to formalize this requirement. In a first step, we use a simplified formalization which achieves the non-overlapping property by enforcing that a malloc always returns an address that is larger than all addresses used in previous mallocs (a more general formalization will be presented in Section 4). This is stated by

$$\mathsf{mallocable}_{\mathsf{top}}(h, p, s) \Leftrightarrow p \geq \mathsf{heaptop}(h)$$

[1] Fixed-width bitvectors complicate the presentation due to overflow effects in bitvector arithmetic.

[2] The C standard states that in this case the result is implementation-defined. Specific ways of how this is handled in concrete implementations can easily be modeled in theories that extend $\mathcal{T}_{\mathcal{H}}$.

This axiom makes use of the additional function symbol $\mathsf{heaptop} : H \to I$ which is formalized by

$$\begin{aligned}&\mathsf{heaptop}(\varepsilon) = 0\\&\mathsf{heaptop}(\mathsf{free}(h,p)) = \mathsf{heaptop}(h)\\&\mathsf{heaptop}(\mathsf{malloc}(h,p,s)) = p+s\end{aligned}$$

In the definition of $\mathsf{heaptop}(\varepsilon)$, a different constant that more accurately reflects the lowest address used for heap memory on a system can be used instead of 0. The predicate mallocable is now defined as

$$\mathsf{mallocable}(h,p,s) := \mathsf{mallocable}_{\mathsf{size}}(h,p,s) \wedge \mathsf{mallocable}_{\mathsf{top}}(h,p,s)$$

Axioms for freeable. The intended semantics of $\mathsf{freeable}(h,p)$ is that p is the first address of a memory region that is currently allocated in h. This semantics is captured by the following axioms:[3]

$$\begin{aligned}&&\mathsf{freeable}(\varepsilon,q) &\Leftrightarrow \bot\\\mathsf{mallocable}(h,p,s) \wedge p = q &\Rightarrow& \mathsf{freeable}(\mathsf{malloc}(h,p,s),q) &\Leftrightarrow \top\\\neg(\mathsf{mallocable}(h,p,s) \wedge p = q) &\Rightarrow& \mathsf{freeable}(\mathsf{malloc}(h,p,s),q) &\Leftrightarrow \mathsf{freeable}(h,q)\\p = q &\Rightarrow& \mathsf{freeable}(\mathsf{free}(h,p),q) &\Leftrightarrow \bot\\p \neq q &\Rightarrow& \mathsf{freeable}(\mathsf{free}(h,p),q) &\Leftrightarrow \mathsf{freeable}(h,q)\end{aligned}$$

Notice that, for each possible first argument of freeable (i.e., ε, malloc, or free), the conditions on the left side of the implications cover all possible cases. This means that exactly one equivalence (on the right hand side of the implication) is usable under any circumstances. This observation also holds for the axioms in the following paragraphs.

Axioms for mallocsize. $\mathsf{mallocsize}(h,p)$ denotes the size of the currently allocated memory region which starts at p (if such a region exists; otherwise $\mathsf{mallocsize}(h,p)$ is zero):

$$\begin{aligned}&&\mathsf{mallocsize}(\varepsilon,q) &= 0\\\mathsf{freeable}(h,p) \wedge p = q &\Rightarrow& \mathsf{mallocsize}(\mathsf{free}(h,p),q) &= 0\\\neg(\mathsf{freeable}(h,p) \wedge p = q) &\Rightarrow& \mathsf{mallocsize}(\mathsf{free}(h,p),q) &= \mathsf{mallocsize}(h,q)\\\mathsf{mallocable}(h,p,s) \wedge p = q &\Rightarrow& \mathsf{mallocsize}(\mathsf{malloc}(h,p,s),q) &= s\\\neg(\mathsf{mallocable}(h,p,s) \wedge p = q) &\Rightarrow& \mathsf{mallocsize}(\mathsf{malloc}(h,p,s),q) &= \mathsf{mallocsize}(h,q)\end{aligned}$$

This function will be used in the axioms for accessible below.

[3] For all axiom groups stated below, it would also be possible to add further, more complex "axioms" that can be derived from the stated axioms (e.g., $p \neq q \wedge \mathsf{contained}(p,s,q,1) \Rightarrow \mathsf{freeable}(\mathsf{malloc}(h,p,s),q) \Leftrightarrow \bot$, which states that a free operation with an address "in the middle" of an allocated block is not valid). How these derived "axioms" affect the runtime of the implementation will be investigated in future work.

Axioms for accessible. Finally, we present the axioms for accessible. Recall that $\mathsf{accessible}(h, p, s)$ determines whether the region $[p, p + s)$ is completely contained within an allocated memory region.

$$
\begin{aligned}
& & \mathsf{accessible}(\varepsilon, p, s) &\Leftrightarrow \bot \\
\mathsf{mallocable}(h, p, s) \wedge \mathsf{contained}(p, s, q, t) &\Rightarrow & \mathsf{accessible}(\mathsf{malloc}(h, p, s), q, t) &\Leftrightarrow \top \\
\neg(\mathsf{mallocable}(h, p, s) \wedge \mathsf{contained}(p, s, q, t)) &\Rightarrow & \mathsf{accessible}(\mathsf{malloc}(h, p, s), q, t) & \\
& & &\Leftrightarrow \mathsf{accessible}(h, q, t) \\
\neg\mathsf{freeable}(h, p) &\Rightarrow & \mathsf{accessible}(\mathsf{free}(h, p), q, t) & \\
& & &\Leftrightarrow \mathsf{accessible}(h, q, t) \\
\mathsf{freeable}(h, p) \wedge \mathsf{disjoint}(p, \mathsf{mallocsize}(h, p), q, t) &\Rightarrow & \mathsf{accessible}(\mathsf{free}(h, p), q, t) & \\
& & &\Leftrightarrow \mathsf{accessible}(h, q, t) \\
\mathsf{freeable}(h, p) \wedge \neg\mathsf{disjoint}(p, \mathsf{mallocsize}(h, p), q, t) &\Rightarrow & \mathsf{accessible}(\mathsf{free}(h, p), q, t) &\Leftrightarrow \bot
\end{aligned}
$$

4 A Generalized Version of mallocable

Figure 1 contains refined axioms for mallocable. Here, $\mathsf{mallocable}_{\mathsf{fit}}(h, p, s)$ is true iff $[p, p + s)$ is disjoint from all currently allocated memory regions. The most complex axioms are the mallocable-over-free axioms (1)–(3) that are concerned with partial overlaps of a freed memory region with a memory region whose mallocability is to be determined. The different possible overlap situations are depicted in Fig. 1. Then, mallocable itself is defined by

$$\mathsf{mallocable}(h, p, s) := \mathsf{mallocable}_{\mathsf{size}}(h, p, s) \wedge \mathsf{mallocable}_{\mathsf{fit}}(h, p, s)$$

5 Extensions of the Theory

In order to be able to use the memory model for the verification of C programs, some peculiarities of the C programming language have to be taken into account. This is mostly related to the special role of `NULL`-pointers in C.

$\mathsf{malloc}(h, 0, s)$: The `malloc`-function may return `NULL` to indicate that the memory allocation could not be performed, for example because of an out-of-memory situation. Notice that the heap allocation state is not altered in this situation. To take this into account, mallocable can be replaced by $\mathsf{mallocable}'$, which is defined by

$$\mathsf{mallocable}'(h, p, s) := (p \neq 0) \wedge \mathsf{mallocable}(h, p, s)$$

Furthermore, $\mathsf{heaptop}(\varepsilon)$ needs to be changed to a non-zero constant.

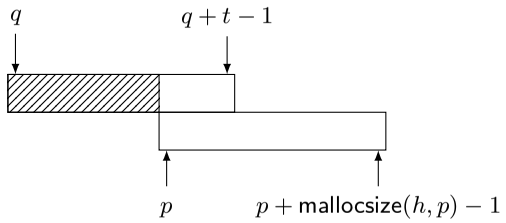
$\mathsf{free}(h, 0)$: Passing `NULL` to `free` is explicitly allowed in the C standard, but is specified to have no effect. To take this into account, $\mathsf{freeable}'$, defined by

$$\mathsf{freeable}'(h, p) := (p = 0) \vee \mathsf{freeable}(h, p)$$

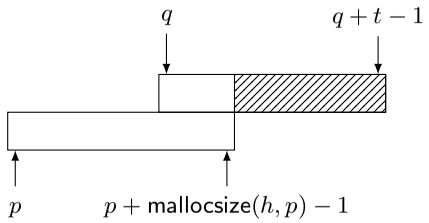
can be used instead of freeable for checking the correctness of free operations.

$$\begin{array}{rcl}
 & & \mathsf{mallocable}_{\mathsf{fit}}(\varepsilon, p, s) \Leftrightarrow \top \\
\mathsf{mallocable}_{\mathsf{fit}}(h,p,s) \wedge \neg\mathsf{disjoint}(p,s,q,t) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{malloc}(h,p,s),q,t) \Leftrightarrow \bot \\
\neg(\mathsf{mallocable}_{\mathsf{fit}}(h,p,s) \wedge \neg\mathsf{disjoint}(p,s,q,t)) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{malloc}(h,p,s),q,t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h,q,t) \\
\neg\mathsf{freeable}(h,p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h,p),q,t) \Leftrightarrow \top \\
\mathsf{freeable}(h,p) \wedge \mathsf{contained}(p,\mathsf{mallocsize}(h,p),q,t) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h,p),q,t) \Leftrightarrow \top \\
\mathsf{freeable}(h,p) \wedge \mathsf{disjoint}(p,\mathsf{mallocsize}(h,p),q,t) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h,p),q,t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h,q,t) \\
\mathsf{freeable}(h,p) \wedge q < p \wedge q+t > p \wedge q+t \leq p+\mathsf{mallocsize}(h,p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h,p),q,t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h,q,p-q) \qquad (1) \\
\mathsf{freeable}(h,p) \wedge q \geq p \wedge q < p+\mathsf{mallocsize}(h,p) \wedge q+t > p+\mathsf{mallocsize}(h,p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h,p),q,t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h,p+\mathsf{mallocsize}(h,p), \\
 & & \qquad q+t-(p+\mathsf{mallocsize}(h,p))) \qquad (2) \\
\mathsf{freeable}(h,p) \wedge q < p \wedge q+t > p+\mathsf{mallocsize}(h,p) & \Rightarrow & \mathsf{mallocable}_{\mathsf{fit}}(\mathsf{free}(h,p),q,t) \Leftrightarrow \mathsf{mallocable}_{\mathsf{fit}}(h,q,p-q) \wedge \\
 & & \qquad \mathsf{mallocable}_{\mathsf{fit}}(h,p+\mathsf{mallocsize}(h,p), \\
 & & \qquad q+t-(p+\mathsf{mallocsize}(h,p))) \qquad (3)
\end{array}$$

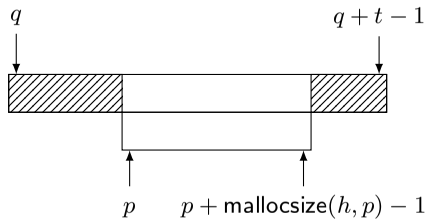
(a) Axioms



(b) Illustration for axiom (1)



(c) Illustration for axiom (2)



(d) Illustration for axiom (3)

Fig. 1. Axioms and illustrations for $\mathsf{mallocable}_{\mathsf{fit}}$.

6 Implementation

We have implemented $\mathcal{T}_{\mathcal{H}}$ in our software bounded model checking tool LLBMC as an alternative to the combined theory approach presented in [16]. Since current SMT solvers do not support $\mathcal{T}_{\mathcal{H}}$ (yet?), we apply the axioms in a pre-processing step before passing the resulting formula to an SMT solver. This pre-processing is done similarly to the case of $\mathcal{T}_{\mathcal{A}}$ as discussed in Section 2:

1. The equalities or logical equivalences in the axioms are oriented from left to right, turning them into conditional rewrite rules.
2. ITE-terms (possibly nested) are used in order to replace instances of left-hand sides by instances of right-hand sides. In order to prevent a blow-up of the formula, newly created ITE-terms are immediately simplified.

In LLBMC, mallocable and the corresponding axioms are not needed since suitable non-overlapping assumptions (see [16]) ensure that mallocable is always true.

Example 1. In this example we show that the formula

$$\mathsf{accessible}(\mathsf{free}(\mathsf{malloc}(\varepsilon, x, 1), x), x, 1) \tag{4}$$

is unsatisfiable using the above pre-processing and additional formula simplifications. Using the accessible-over-free axioms and simplifications of the introduced disjoint-subformula, (4) is equivalent to

$$\mathsf{ITE}(\mathsf{freeable}(\mathsf{malloc}(\varepsilon, x, 1), x), \bot, \mathsf{accessible}(\mathsf{malloc}(\varepsilon, x, 1), x, 1)) \tag{5}$$

The freeable-over-malloc axioms imply that the predicate $\mathsf{freeable}(\mathsf{malloc}(\varepsilon, x, 1), x)$ is equivalent to the predicate $\mathsf{ITE}(\mathsf{mallocable}(\varepsilon, x, 1), \top, \mathsf{freeable}(\varepsilon, x))$. Next, the subformula $\mathsf{mallocable}(\varepsilon, x, 1)$ is simplified to $\top$. Thus, (5) is equivalent to

$$\mathsf{ITE}(\mathsf{ITE}(\top, \top, \mathsf{freeable}(\varepsilon, x)), \bot, \mathsf{accessible}(\mathsf{malloc}(\varepsilon, x, 1), x, 1)) \tag{6}$$

Using ITE-simplifications, (6) is simplified to $\bot$, thus showing unsatisfiability of the original formula. $\Diamond$

7 Evaluation

We have evaluated LLBMC using the implementation of $\mathcal{T}_{\mathcal{H}}$ as described in Section 6 and the implementation of the approach from [16]. In [16], the $\mathcal{T}_{\mathcal{H}}$ predicates accessible, freeable, and mallocable are used as well. In contrast to $\mathcal{T}_{\mathcal{H}}$, however, the encoding of $\mathsf{accessible}(h, p, s)$ iterates over all mallocs that took place when obtaining the heap state h and have not been deallocated since then. $\mathsf{accessible}(h, p, s)$ is then encoded as a disjunction over these mallocs, where each disjunct checks whether the access operation falls within the memory block that is allocated by the malloc.

The evaluation has been performed on a collection of 97 small to medium-sized C programs from various sources. The largest part of the evaluated benchmarks was selected from the NEC Laboratories America benchmark suite[4], the Run Time Error Detection Test Suites[5], and the WCET benchmark selection[6]. Of these, only those benchmarks using dynamic heap memory allocation were included.

After unrolling of loops and inlinng of function calls, an average of 95.32 memory allocations per benchmark remained. The benchmark with the largest number of memory allocations was an algorithm for the flattening of a tree datastructure. This benchmark contained a total of 6930 memory allocations.

[4] Available at http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
[5] Available at http://rted.public.iastate.edu/
[6] Available at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Example 2. The following (artificial) example illustrates the difference between $\mathcal{T}_\mathcal{H}$ and [16].

```
1  int main()
2  {
3      int i;
4      int *p = NULL;
5
6      for (i = 0; i < N; ++i) {
7          p = malloc(4 * sizeof(int));
8      }
9
10     return *(p + 3);
11 }
```

Using $\mathcal{T}_\mathcal{H}$, the validity of the memory access operation in line 10 can easily be established by considering the last malloc in the heap state history. Using the approach from [16], on the other hand, builds a disjunction of N accessible predicates, one for each malloc in the loop. ◊

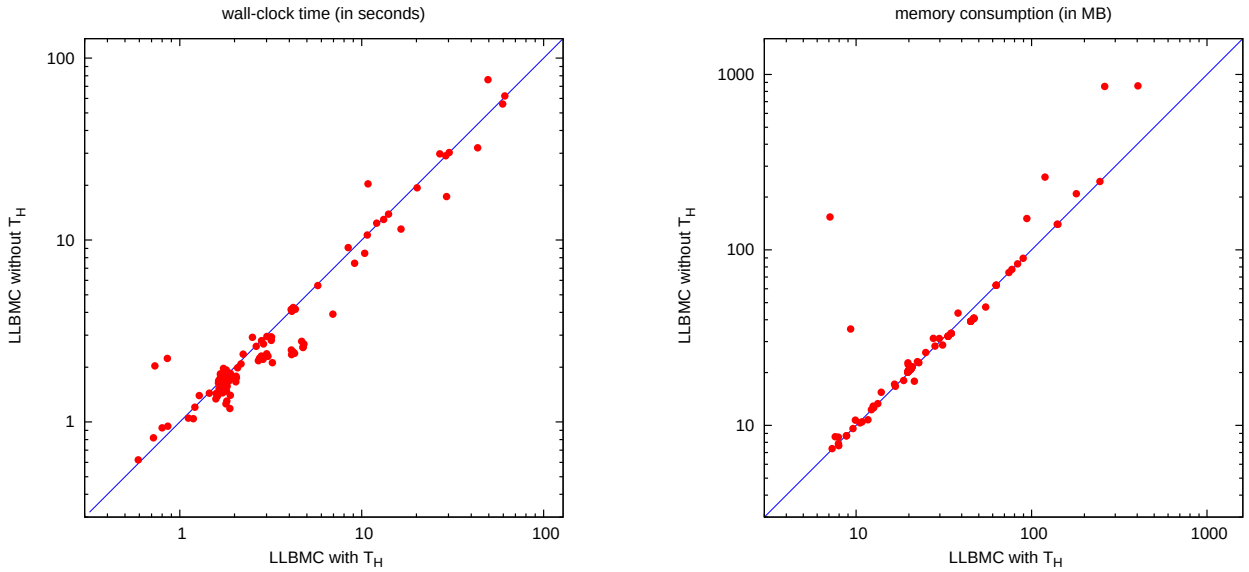


Fig. 2. Scatter plots of the run-time and memory consumption of LLBMC comparing the implementation of $\mathcal{T}_\mathcal{H}$ (x-axis) to the implementation following [16] (y-axis).

Scatter plots of the results are given in Fig. 2. The run-times of $\mathcal{T}_\mathcal{H}$ and [16] are roughly comparable. The same is true for memory consumption, but $\mathcal{T}_\mathcal{H}$ has a significantly lower ionsumption than [16] in certain cases. Table 1 contains a detailed comparison for selected benchmarks.

8 Related Work

Several low-level memory models[7] for C-like languages have been proposed in the past ([1, 2, 6, 7, 11–14, 17, 18]). However, they do not emphasize memory protection or ignore it completely.

[7] In a low-level memory model the memory is not much more than an array of bytes and suitable disjointness or consistency conditions are stated explicitly.

benchmark name	#mallocs/ #frees	#accessible	time SSV	time $\mathcal{T}_\mathcal{H}$	memory SSV	memory $\mathcal{T}_\mathcal{H}$
sparsemem	129/51	8374	76.2	49.5	861	404
plenty-of-mallocs	333/0	2	2.0	0.7	154	7
binary-tree	127/127	3048	7.5	9.1	150	94
flatten-trees	6930/0	42420	29.8	26.9	854	261
inplace-reverse	100/100	1800	20.4	10.8	260	119
wcet-bsort100	3/0	120204	12.4	12.1	246	246
wcet-statemate	106/0	2816	2.2	0.9	35	9

Table 1. Comparison of local ($\mathcal{T}_\mathcal{H}$) and global (SSV, see[16]) memory access formalizations on selected benchmarks. Reported times are wall-clock times in seconds; memory consumption is given in MBs.

Tuch *at al.* [18, 17] discuss a typed memory model in the context of interactive theorem proving with the proof assistant `Isabelle/HOL`. It is shown that this typed memory model is sound with respect to the untyped memory model assumed by C.

The memory model presented by Leroy and Blazy [13] is similar to our model and considers read, write, malloc, and free operations. While the disjointness of memory blocks allocated by separate mallocs is guaranteed, no such separation for accesses performed within the same memory block is ensured (e.g., accesses to different members of a structure). Leroy and Blazy prove properties of their memory model using the proof assistant `Coq` (such as semantic preservation of compiler passes). Cohen *et al.* [7] introduce a typed memory model similar to [18] for a C-like toy programming language and show that this typed memory model is sound with respect to the untyped memory model assumed by C. They support pointer arithmetic and memory access (read and write operations) at arbitrary locations in the memory, but do not consider memory protection (malloc and free operations). Mehta and Nipkow [14] present a mechanism to reason about pointer-based programs. Gast [11] gives a formalism for reasoning about memory layouts of C programs. In both cases, proof obligations are formulated in Hoare logic and verified using `Isabelle/HOL`. The memory model used in `Havoc` is presented in [6]. `Havoc` uses a reachability predicate based on the memory model in order to reason about heap-based data structures, but does not support memory protection.

Böhme and Moskal describe several typed heap encodings used by `VCC2` and `VCC3`. A memory model that is suitable for verification using separation logic is presented in [2], while a separation-based approach for deductive verification in `Caduceus` is given in [12]. Neither paper considers memory protection.

`KLEE` [5], a symbolic execution engine developed by Cadar *et al.*, shares the use of `LLVM`'s intermediate representation with our tool. `KLEE` uses an untyped, segmented memory model, where each object is represented by a separate array in the SMT solver `STP` [10]. How memory access correctness is modeled is not explicitly mentioned, though. `CUTE` [15] is a tool that combines symbolic and concrete execution in an approach called *concolic testing*. Their memory model uses fixed addresses for memory objects plus a global variable to store the next free address available for allocation. From what is published, it is not clear how they handle memory allocation. In general, symbolic execution requires techniques similar to the ones presented here. E.g., accessing an array of pointers at a "symbolic index" requires some kind of case distinction.

9 Conclusions and Future Work

We have presented $\mathcal{T}_\mathcal{H}$, a theory of heap memory allocation, that closely matches the semantics of `malloc` and `free` in C. Furthermore, we have shown how the theory's axioms can be applied as conditional

rewrite rules to reduce a problem from $\mathcal{T}_\mathcal{H}$ to a problem that can be solved by current SMT solvers such as `Boolector` [3] or `Z3` [8].

An Evaluation in the software bounded model checking tool `LLBMC` shows that even though application of $\mathcal{T}_\mathcal{H}$ eliminates the disadvantages of the approach presented in [16] by applying local simplifications to the formula, it does not impose a performance penalty in comparison to that approach.

As future work based on the results presented in this paper, we intend to combine spatially related accessible statements and hope to be able to reduce size and complexity of the generated SMT formula this way. Furthermore, we are planning to develop an approach based on lemmas-on-demand [9, 4] for solving $\mathcal{T}_\mathcal{H}$ formulas. A further possibility is to use SMT solvers that support quantified axioms (such as `Z3` [8]).

In the long term, we hope to be able to use the work presented in this paper as groundwork for a more modular software bounded model checking approach. For this, we intend to translate each function separately into our intermediate logic representation and apply syntactic and semantic rewriting on these functions. Only after this simplification has been performed, the final formula is created and passed on to the SMT solver. $\mathcal{T}_\mathcal{H}$ represents an important step towards this goal.

References

1. Sascha Böhme and Michał Moskal. Heaps and data structures: A challenge for automated provers. In *Proc. CADE 2011*, 2011. To appear.
2. Matko Botincan, Matthew Parkinson, and Wolfram Schulte. Separation logic verification of C programs with an SMT solver. *ENTCS*, 254:5–23, 2009.
3. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. TACAS 2009*, volume 5505 of *LNCS*, pages 174–177, 2009.
4. Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6:165–201, 2009.
5. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI 2008*, pages 209–224, 2008.
6. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A low-level memory model and an accompanying reachability predicate. *STTT*, 11(2):105–116, 2009.
7. Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *ENTCS*, 254:85–103, 2009.
8. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS 2008*, volume 4963 of *LNCS*, pages 337–340, 2008.
9. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Proc. SAT 2002*, 2002.
10. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV 2007*, volume 4590 of *LNCS*, pages 519–531, 2007.
11. Holger Gast. Reasoning about memory layouts. In *Proc. FM 2009*, volume 5850 of *LNCS*, pages 628–643, 2009.
12. Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Proc. HAV 2007*, pages 81–93, 2007.
13. Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.
14. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *IC*, 199(1–2):200–227, 2005.
15. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proc. FSE 2005*, pages 263–272, 2005.
16. Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *Proc. SSV 2010*, 2010.
17. Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *JAR*, 42(2–4):125–187, 2009.
18. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Proc. POPL 2007*, pages 97–108, 2007.