* can we infer these ??

Template-Based Verification of Heap-Manipulating Programs

Viktor Malík*‡ Martin Hruska‡ Peter Schrammel*† Tomáš Vojnar‡
*Diffblue Ltd, Oxford, UK †University of Sussex, Brighton, UK ‡FIT BUT, IT4Innovations Centre of Excellence, CZ

***Abstract*—We propose a shape analysis suitable for analysis engines that perform automatic invariant inference using an SMT solver. The proposed solution includes an abstract template domain that encodes the shape of the program heap based on logical formulae over bit-vectors. It is based on computing a points-to relation between pointers and symbolic addresses of abstract memory objects. Our abstract heap domain can be combined with value domains in a straightforward manner, which particularly allows us to reason about shapes and contents of heap structures at the same time. The information obtained from the analysis can be used to prove memory safety and reachability properties, expressed by user assertions, of programs manipulating dynamic data structures, mainly linked lists. The solution has been implemented in the 2LS framework and compared against state-of-the-art tools that perform the best in heap-related categories of the well-known Software Verification Competition (SV-COMP). Results show that 2LS outperforms these tools on benchmarks requiring combined reasoning about unbounded data structures and their numerical contents.**

I. Introduction

7.

Reasoning about dynamic data structures is one of the core problems in software verification. The techniques implemented in state-of-the-art verification tools for C programs such as those competing in the Software Verification Competition (SV-COMP) have shortcomings when it comes to combined reasoning about shape and content of data structures as our experiments revealed. We address this problem in this paper in the context of template-based program verification.

Template-based verification uses a logic-based synthesis approach to inferring the invariants required for proving program properties. It delegates semantic reasoning to SMT solvers and focusses on the design of appropriate template domains and efficient algorithms for finding the optimal template parameters (i.e. least fixed points in the abstract interpretation sense [14]). The use of such templates makes it straightforward to compute invariants describing both shape and value properties of data structures, which is more difficult when combining domains that are based on different principles.

Running example: To better illustrate the concepts and methods proposed in the paper, we use the program in Listing 1 as a running example. It creates a singly-linked list, each node containing a value between 10 and 20 (Lines 7–15). The list is afterwards traversed repeatedly and the value of each node is either incremented by 1 or halved (Lines 16–22). We add an assertion that, in every iteration, the value of each node stays between 10 and 20. The goal of the analysis is to prove that the assertion always holds. This requires an analysis capable of reasoning about unbounded linked data structures and numerical content of their nodes at the same time.

Listing 1: A running example

```
1  typedef struct node {
2    int val;
3    struct node *next;
4  } Node;
5
6  int main() {
7    Node *p, *list = malloc(sizeof(Node));
8    Node *tail = list;
9    *list = {.next = NULL, .val = 10};
10   while (__VERIFIER_nondet_int()) {
11     int x = __VERIFIER_nondet_int();
12     if (x < 10 || x > 20) continue;
13     p = malloc(sizeof(Node));
14     *p = {.next = NULL, .val = x};
15     tail→next = p; tail = p;
16   }
17   while (1) {   → terminate ?
18     for (p = list; p!= NULL; p = p→next) {
19       assert(p→val <= 20 && p→val >= 10);
20       if (p→val < 20) p→val++;
21       else p→val /= 2;
22 } } }
```

To prove this property we have to infer that the value of the `val` field of the dynamic objects allocated in Line 7 and 13 is always in the range $[10, 20]$.

With the help of our technique, we will infer an invariant for the loop on Line 10 that states the following:

- `tail` may point to the sets of `Node` objects created in Line 7 and 13. We denote these sets ao_7 and ao_{13}, resp.
- The `next` field of ao_7 may point to ao_{13} or `null`. Its `val` field has a value in the interval [10,10].
- The `next` field of ao_{13} may point to ao_{13} or `null`. However, its `val` field has a value in the interval [10,20]. This means that ao_{13} abstracts a set of `Node` objects whose `val` fields have values in the interval [10,20].

For the loop in Line 18, we infer the invariant that the `val` fields of ao_7 and ao_{13} must both be in the interval [10,20], which implies ~~that the property holds.~~ → how to check?

Contributions: The contributions of this paper, which form the contents of Sections III–VII, are as follows:

1) We propose a novel abstract template domain for reasoning over heap-allocated data structures such as singly and doubly linked lists using a template-based parameter synthesis engine.
2) We show how we can build product and power domain combinations of our heap domain with structural domains (e.g. trace partitioning) and value domains such as template polyhedra that capture the content of data structures.
3) We implement our abstract heap domain in the 2LS verification tool for C programs. We demonstrate the power

of the proposed domain on benchmarks, which require combined reasoning about the shape and content of data structures, showing that other tools, which performed well in SV-COMP, cannot handle these examples.

II. Template-Based Program Verification

This section describes the approach to program verification using template-based synthesis of inductive invariants which the 2LS tool [35] is based upon and that underlies our approach too. The source program is first translated into single static assignment (SSA) form. Using this program representation, the verification task can then be expressed as a second-order logical formula. However, since suitable solvers for such formulae are not available, the verification problem is reduced to synthesising loop invariants using parametrised templates and an SMT solver to find suitable values of the parameters.

A. Program Verification Using Inductive Invariants

A state of a program is a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—states in the set are defined by models of the formula. Given a vector of variables $\vec{x}$, the predicate $Init(\vec{x})$ describes the initial states. A transition relation is described as a formula $Trans(\vec{x}, \vec{x}')$.

From these, it is possible to determine the set of reachable states as the least fixed-point of the transition relation starting from the states described by $Init(\vec{x})$. This is, however, difficult to compute, so instead, we use an *inductive invariant*. A verification task requires showing that the set of reachable states does not intersect with the set of error states $Err(\vec{x})$. Using the concept of inductive invariants and existential second-order quantification ($\exists_2$), we can formalise it as:

$$\begin{aligned} \exists_2 Inv.\, \forall \vec{x}, \vec{x}'.\, &(Init(\vec{x}) \Longrightarrow Inv(\vec{x})) \wedge \\ &(Inv(\vec{x}) \wedge Trans(\vec{x}, \vec{x}') \Longrightarrow Inv(\vec{x}')) \wedge \\ &(Inv(\vec{x}) \Rightarrow \neg Err(\vec{x})) \end{aligned} \tag{1}$$

B. Invariant Inference via Templates

To directly handle Eq. (1) by a solver, it would require the capability to deal with second-order logic quantification. Since a suitably general and efficient second-order solver is not currently available, the problem is reduced to one that can be solved by an iterative application of a first-order solver. This reduction is done by restricting the form of the inductive invariant Inv to $\mathcal{T}(\vec{x}, \vec{\delta})$ where $\mathcal{T}$ is a fixed expression (a so-called *template*) over program variables $\vec{x}$ and template parameters $\vec{\delta}$. This restriction corresponds to the choice of an abstract domain in abstract interpretation—a template only captures the properties of the program state space that are relevant for the analysis. This reduces the second-order search for an invariant to a first-order search for the template parameters:

$$\begin{aligned} \exists \vec{\delta}.\, \forall \vec{x}, \vec{x}'.\, &(Init(\vec{x}) \Longrightarrow \mathcal{T}(\vec{x}, \vec{\delta})) \wedge \\ &(\mathcal{T}(\vec{x}, \vec{\delta}) \wedge Trans(\vec{x}, \vec{x}') \Longrightarrow \mathcal{T}(\vec{x}', \vec{\delta})) \end{aligned} \tag{2}$$

?

Although the problem is now expressible in first-order logic, the formula contains quantifier alternation, which poses a problem for current SMT solvers. This is solved by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants $\vec{d}$ as candidates for template parameters $\vec{\delta}$. For a value $\vec{d}$, the template formula $\mathcal{T}(\vec{x}, \vec{d})$ is an invariant if and only if Eq. (3) is unsatisfiable.

$$\begin{aligned} \exists \vec{x}, \vec{x}'.\, &\neg(Init(\vec{x}) \Longrightarrow \mathcal{T}(\vec{x}, \vec{d})) \vee \\ &\neg(\mathcal{T}(\vec{x}, \vec{d}) \wedge Trans(\vec{x}, \vec{x}') \Longrightarrow \mathcal{T}(\vec{x}', \vec{d})) \end{aligned} \tag{3}$$

From the abstract interpretation point of view, $\vec{d}$ is an abstract value, i.e. it represents (*concretises to*) the set of all program states $\vec{x}$ that satisfy the formula $\mathcal{T}(\vec{x}, \vec{d})$. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\vec{x}, \bot) \equiv false$ and $\mathcal{T}(\vec{x}, \top) \equiv true$ [8].

?

Formally, the concretisation function γ is: $\gamma(\vec{d}) = \{\vec{x} \mid \mathcal{T}(\vec{x}, \vec{d}) \equiv true\}$. In the abstraction function, to get the most precise abstract value representing the given concrete program state $\vec{x}$, we let $\alpha(\vec{x}) = \min(\vec{d})$ such that $\mathcal{T}(\vec{x}, \vec{d}) \equiv true$. Since the abstract domain forms a complete lattice, existence of such a minimal value $\vec{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\vec{d} = \bot$ and iteratively solves Eq. (3) using an SMT solver. If the formula is unsatisfiable, then an invariant has been found, otherwise a model of satisfiability is returned by the solver. The model represents a counterexample to the current instantiation of the template being an invariant. The value of the template parameter $\vec{d}$ is then updated by combining with the obtained model of satisfiability $\vec{d'}$ using a domain-specific join operator [8]. For example, assume we have a program with a loop that counts from 0 to 10 in variable x and we have a template $x \leq d$. Let's assume that the current value of the parameter d is 3 and we get a new model $d' = 4$. Then we update the parameter to 4 by computing $d \sqcup d' = \max(d, d')$, because max is the join operator for a domain that tracks numerical upper bounds.

C. Source Program Encoding

In this paper, we deal with non-recursive programs with all function calls inlined. As said above, we encode the program into a formula representing a specific *static single assignment form* (SSA). For acyclic programs, the SSA represents exactly the strongest postcondition of the program—as usual, with a fresh copy x_i of each variable x for each program location i where the value of x is modified. The effect of loops is over-approximated as described in [8]. In this encoding, special variables called *guards* are used to track the control flow of the program. In particular, for each program location i, a Boolean variable g_i is introduced, and its value encodes whether the program location is reachable.

To see how the over-approximation of program loops is achieved, note that, at the loop head, the program path coming from before the loop joins with the path coming from the end of the loop (assuming that all paths within the loop join before its end; and likewise for the paths coming from before the loop). To achieve acyclicity of the SSA, we cut the path coming from the end of the loop. We then represent the value of each variable x at the loop head using a *phi variable* x^{phi} whose value is defined by a non-deterministic choice between the value coming from before the loop, say x_0, and the value coming from the end of the loop. The latter value is represented by a newly introduced *loop-back* variable x^{lb}. In particular, we let $x^{phi} = g^{ls} \ ? \ x^{lb} : x_0$ where g^{ls} is a so-called *loop-select* Boolean guard that is unconstrained in order to model the non-deterministic choice. Moreover, to over-approximate the effect of the loop, the value of the loop-back variable x^{lb} is initially unconstrained too and later constrained by the derived candidate loop invariants.

Example. In Listing 1, the loop head at Line 10 joins two different values of variable $tail$ coming from program locations 8 and 15. The value of $tail$ coming from the end of the loop (denoted $tail_{15}$ in the SSA) is replaced by the loop-back variable $tail^{lb}_{16}$. The corresponding phi variable $list^{phi}_{10}$ then non-deterministically joins $tail^{lb}_{16}$ with the value of $tail$ from before the loop via the loop-select variable g^{ls}_{16}:

$$list^{phi}_{10} = g^{ls}_{16} \ ? \ list^{lb}_{16} \ : \ list_8 \tag{4}$$

III. Abstract Memory Operations in the SSA Form

We now propose a representation of heap memory and operations over it, designed to be used within the approach laid out in Section II. The proposal respects the fact that the considered SSA form is an acyclic program representation, over-approximating reachable values of variables used in loops.

A. Abstract Memory Representation

Under our assumption of fully inlined, non-recursive programs, *static memory objects* correspond simply to a finite set Var of *program variables*: we do not need to consider the ? stack. We let $PVar, SVar \subseteq Var$, $PVar \cap SVar = \emptyset$, be the sets of variables of *pointer* and *structure type*, respectively. A linked data structure in C is typically defined using a `struct` type, which groups together named *fields* for the payload data and the link pointers (see Lines 1–4 in Listing 1). We use Fld to denote the finite set of fields used in the given program. Let $PFld \subseteq Fld$ be the set of all pointer-typed fields.

1) Abstract Dynamic Objects: We use *abstract dynamic objects* to represent *dynamic memory objects*, i.e. those that are allocated using `malloc` (or some of its variants) on the heap. An abstract dynamic object represents a set of concrete dynamic objects allocated at the same *allocation site* i, e.g. by the same `malloc` call located at Line i in Listing 1. However, a single abstract dynamic object is not sufficient to represent *all* concrete dynamic objects allocated by a given `malloc`. The reason for this is that the program may use several independent objects created at an allocation site at the same time. Typically, this issue is solved by the analysis algorithm materialising dynamic objects on-demand. We take a different approach and statically over-approximate the maximum number n_i of concrete objects required (see next section below). Hence, we use a *set* $AO_i = \{ao^k_i \mid 1 \leq k \leq n_i\}$ of abstract dynamic objects for that purpose. We let $AO = \cup_i AO_i$ and require $Var \cap AO = \emptyset$ and $AO_i \cap AO_j = \emptyset$ for $i \neq j$. The set of all objects of our program abstraction is then $Obj = AO \cup Var$.

Pairs consisting of an abstract dynamic object and a field, i.e. elements of the set $AO \times Fld$, represent an abstraction of the appropriate *fields* of all the represented concrete objects. We use the "dot" notation to represent such pairs: e.g. $ao_i.next$ denotes the abstraction of the $next$ field of all the concrete dynamic objects represented by ao_i.

We define $Ptr = PVar \cup ((SVar \cup AO) \times PFld)$ to be the set of all *pointers* of the given program abstraction. Pointers can be assigned addresses of objects. Since we currently do not support pointer arithmetic, the only addresses that we consider are *symbolic addresses* of static and dynamic objects together with the special address null. The symbolic address of an abstract dynamic object ao_i is an abstraction of the symbolic addresses of the concrete dynamic objects represented by ao_i. To get the address of both static and dynamic objects, we use the &-operator. Hence, the set $Addr$ of addresses that we consider is defined as $Addr = \{\&o \mid o \in Obj\} \cup \{\mathsf{null}\}$.[1]

2) Pre-Materialisation: As mentioned above, instead of materialising dynamic objects on-demand, we pre-materialise a sufficient number n_i of them for each allocation site i and encode them into our SSA representation. In order for this abstraction to be sound, it is sufficient that the number n_i equals the maximal number of distinct concrete objects allocated at i that are simultaneously pointed to by some pointer at any location of the analysed program.

For each allocation site i, we compute the number n_i as follows. First, using a standard static may-alias analysis, we over-approximate, for each program location j, the set P^i_j of all pointer expressions of the source program that *may point* to some object allocated at i. These might be pointer variables from $PVar$, pointer-typed fields of static objects from $SVar \times PFld$, or pointer-typed fields of dynamic objects accessed through dereferences of pointers—i.e. elements of $PVar \times PFld$. For simplicity, we assume that all chained dereferences of the form $p \rightarrow f_1 \rightarrow f_2$ with $f_1, f_2 \in PFld$ are broken into two expressions using an intermediate variable. Overall, $P^i_j \subseteq PVar \cup ((SVar \cup PVar) \times PFld)$. Next, we compute the *must-alias relation* $\sim_j$. For each pair of pointers p and q and for each program location j, $p \sim_j q$ iff p and q must point to the same concrete dynamic object at j. Finally, we partition the set P^i_j into equivalence classes by $\sim_j$, and n_i is given by the maximal number of such classes at any j.

B. Operations over the Abstract Memory Representation

1) Dynamic Memory Allocation: We represent a call to `malloc` at program location i by a non-deterministic choice among the addresses of objects from the set AO_i. Hence, a statement $p = \texttt{malloc}(\ldots)$ at i is translated to the formula $p_i = g^{os}_{i,1} \, ? \, \&ao^1_i \, : \, (g^{os}_{i,2} \, ? \, \&ao^2_i \, : \, (\ldots(g^{os}_{i,n_i-1} \, ? \, \&ao^{n_i-1}_i \, : \, \&ao^{n_i}_i)))$ where $g^{os}_{i,j}$, $1 \leq j < n_i$ are free Boolean variables, so-called *object-select guards*.

[1] We currently assume that addresses of newly allocated objects are fresh. Hence, we can miss behaviours where some memory space is recycled while some pointers are still pointing to it, which is undefined according to the C standard, but sometimes used in practice. If that was a problem, we could, e.g., extend our preliminary static analysis to detect objects that can possibly be in that form and add them among possible returns from the allocation.

Example. In Listing 1, two calls of `malloc` occur on Lines 7 and 13. For Line 7, a single abstract dynamic object ao_7 is created as there is just one concrete object allocated.[2] The `malloc` on Line 13 must be represented by two objects ao^1_{13} and ao^2_{13} as, e.g. on Line 14, variables `tail` and `p` may point to different concrete objects allocated by this `malloc` call. Specifically, the statement on Line 13 will be translated into the equality $p_{13} = g^{os}_{13} \,?\, \&ao^1_{13} : \&ao^2_{13}$. Abstract dynamic objects ao^1_{13} and ao^2_{13} then collectively represent all concrete dynamic objects allocated in the loop.

2) Reading through Dereferenced Pointers: We handle expressions of the form $p \to f$ for $p \in PVar$, $f \in Fld$ appearing on the right-hand side of assignments or in conditions as follows. We first perform a *may-points-to analysis*, which overapproximates for each pointer $p \in Ptr$ and each program location i the set of objects from Obj that p may point to at i. Using the result of the analysis, we can replace the pointer dereference $p \to f$ by a choice among the values of the field f of the objects possibly pointed to by p.

To facilitate the replacement, we introduce purely logical *dereference variables*. Assume that at program location i there appears an R-expression $p \to f$ and that the pointer p may point to a set of objects $O \subseteq Obj$ at i. We replace the use of $p \to f$ by using a fresh variable $drf(p).f_i$ whose value is defined by the formula $(\bigwedge_{o \in O} p_j = \&o \implies drf(p).f_i = o.f_k) \wedge ((\bigwedge_{o \in O} p_j \neq \&o) \implies drf(p).f_i = o_\perp)$ where j, k are the relevant versions of the concerned variables at program location i and $o_\perp$ denotes a special "unknown object" (a result of a dereference of an unknown or invalid (null) address).[3]

Example. We give the translation of the assignment $p = p \to next$ from Line 18 in Listing 1. Since the assignment is executed at the end of each loop iteration, we define its program location to be Line 22. At this program location, p may point to the set of objects $\{ao_7, ao^1_{13}, ao^2_{13}\}$. Hence, the assignment will be represented by the following formula.

$$
\begin{aligned}
&p_{22}{=}drf(p).next_{22} \wedge \\
&\Big(p^{phi}_{18}{=}\&ao_7 \Rightarrow drf(p).next_{22}{=}ao_7.next^{phi}_{18}\Big) \wedge \\
&\bigwedge_{l=1,2} \Big(p^{phi}_{18}{=}\&ao^l_{13} \Rightarrow drf(p).next_{22}{=}ao^l_{13}.next^{phi}_{18}\Big) \wedge \\
&\Big((p^{phi}_{18}{\neq}\&ao_7 \wedge \bigwedge_{l=1,2} p^{phi}_{18}{\neq}\&ao^l_{13}) \Rightarrow drf(p).next_{22}{=}o_\perp\Big)
\end{aligned}
$$

The first conjunct represents the transformed assignment, and the following conjuncts define the value of the dereference variable. The value of p entering program location 22 is the value from the loop head p^{phi}_{18}. If it equals the address of ao_7, ao^1_{13}, or ao^2_{13}, the value of $drf(p).next_{22}$ is $ao_7.next^{phi}_{18}$, $ao^1_{13}.next^{phi}_{18}$, or $ao^2_{13}.next^{phi}_{18}$, otherwise, it equals $o_\perp$.

As an optimisation, if the dereference variable is once created and the value of the concerned expression does not change, we reuse the existing dereference variable. Second, when dealing with a statement like $v = p \to f$, the use of the dereference variable may seem unnecessary as one can plug v_i instead of $drf(p).f_i$ into the formula defining the value of $drf(p).f_i$. This can be done, but, as explained below, the use of dereference variables can give us more precision when dealing with sequences of reading and writing operations.

3) Writing through a Dereference: When writing into an abstract dynamic object ao_i, we need to respect the fact that only one concrete object abstracted by ao_i is actually written to, and the others keep the original value. Hence, we need to make a join of the original and new value. We again use dereference variables to facilitate the transformation.

Assume that at program location i, we have an assignment $p \to f = v$, $p \in PVar$, $f \in Fld$, $v \in Var$, and that p may point to a set of objects $O \subseteq Obj$ at the entry to i.[4] We replace the L-expression $p \to f$ by a fresh variable $drf(p).f_i$ whose value is defined by the value of v, i.e. we assert that $drf(p).f_i = v_l$ where v_l is the version of v valid at program location i. We then use $drf(p).f_i$ to update the value of the field f of the referenced object, using the formula $\bigwedge_{o \in O} o.f_i = (p_j = \&o \wedge g^{os}_i) \,?\, drf(p).f_i : o.f_k$ where j, k are the relevant versions of the variables p and $o.f$ at program location i.[5] The formula expresses the fact that $o.f_i$ gets updated if p equals the address of o, otherwise its value remains unchanged; k is the last program location before i where the value of $o.f$ was changed. The object-select guard g^{os}_i, which is a freshly introduced unconstrained Boolean variable, enforces that the value of field f is changed in only one of the concrete objects abstracted by o while it remains unchanged in the other objects abstracted by o. If o is not allocated in a loop (and hence representing a single instance), g^{os}_i may be omitted.

Example. For illustration, the assignment `tail->next=p` from Line 15 of Listing 1 will be translated into the formula:

$$
\begin{aligned}
&(drf(list).next_{15} = p_{13}) \wedge \\
&\Big(ao_7.next_{15} = (list^{phi}_{10}{=}\&ao_7) \;? \\
&\qquad\qquad drf(list).next_{15} \;:\; ao_7.next^{phi}_{10}\Big) \wedge \\
&\bigwedge_{l=1,2} (ao^l_{13}.next_{15} = (list^{phi}_{10}{=}\&ao^l_{13} \wedge g^{os}_{15}) \;? \\
&\qquad\qquad drf(list).next_{15} \;:\; ao^l_{13}.next^{phi}_{10})
\end{aligned}
$$

As mentioned above, the use of dereference variables may increase the precision of our analysis. This happens in particular when we write into an abstract object through some pointer and later read the written value back through the same pointer (or a pointer aliased with it) without any change of the pointers and the concerned value in between. Then, we get back exactly the value that we wrote, which would otherwise not happen due to the joins involved.

4) Memory Free: Since the `free` operation has no effect on the heap reachability itself, we defer its discussion to Section V devoted to checking memory safety.

[2] In fact, we should write ao^1_7, but we omit the superscript when a single abstract object suffices. Likewise for the object-select guards below.

[3] A dereference of the form $*p$ for a non-structured object can be handled analogously, just without the field f in the above formula.

[4] More complex assignments can be transformed into this form.

[5] A write to a dereference of the form $*p$ to a non-structured object can be handled analogously, omitting field f from the formula.

IV. An Abstract Domain for Heap Analysis

We will now work towards our template-based abstract domain suitable for reasoning about properties of heap-manipulating programs, starting from a base shape domain and refining it. We will show that, due to the fact that all domains in the considered approach are based on templates, the new domain can be easily combined with other domains, e.g. for inferring properties about numerical data of data structures.

A. Base Abstract Shape Domain

In the considered approach, an abstract domain needs to have the form of a *template*—a fixed, parametrised, quantifier-free first-order logic formula describing the desired property of a program. As described in Section II, templates are used to efficiently compute *loop invariants* of the analysed program. These are used to constrain values of the *loop-back variables* that are used in the SSA-based program encoding to over-approximate values returning from the end of the loop to the loop head. Hence, a loop invariant describes a property that holds for some program variables at the end of the loop body after any iteration of the loop. Hence, we limit our shape domain to the set Ptr^{lb} of all *loop-back pointers*. Let L be the set of all loops in the program. Since there is one loop-back pointer variable for each pointer variable and each loop, we define $Ptr^{lb} = Ptr \times L$. We denote elements $(p, l) \in Ptr^{lb}$ by p_i^{lb} where i is the program location of the end of the loop l. Intuitively, the value of p_i^{lb} is an abstraction of the value of the pointer p coming from the end of the body of the loop l. The property that our base shape domain describes is the *may-point-to* relation between the set Ptr^{lb} and the set $Addr$.[6]

The template of our base shape domain has the form of the formula $\mathcal{T}^S \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$. It is a conjunction of so-called *template rows* $\mathcal{T}^S_{p_i^{lb}}$, each row corresponding to one loop-back pointer from the set Ptr^{lb}. A template row $\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$ describes the may-point-to relation for the loop-back pointer p_i^{lb}. The parameter $d_{p_i^{lb}} \subseteq Addr$ of the row (a so-called *abstract value of the row*) specifies the set of all addresses from the set $Addr$ that p may point to at the location i. The template row can thus be expressed as the quantifier-free formula $\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \equiv (\bigvee_{a \in d_{p_i^{lb}}} p_i^{lb} = a)$.

Abstract values of template rows corresponding to pointer fields of abstract dynamic objects allow the domain to describe unbounded linked paths in the heap, such as list segments.

Example. In Listing 1, a list segment is created by the first loop. Objects in the segment are linked through the pointer field `next`, and they are represented by the abstract dynamic objects ao_{13}^1 and ao_{13}^2. In our base shape domain, the shape of this segment will be described by an invariant for the first loop, specifically by the two template rows for $ao_{13}^1.next_{16}^{lb}$ and $ao_{13}^2.next_{16}^{lb}$. They will give us the formula $\wedge_{l=1,2}\mathcal{T}^S_{ao_{13}^l.next_{16}^{lb}}(\{\&ao_{13}^1, \&ao_{13}^2, \mathsf{null}\})$ where the rows $\mathcal{T}^S_{ao_{13}^l.next_{16}^{lb}}$ are the formulae $ao_{13}^l.next_{16}^{lb} = \&ao_{13}^1 \vee ao_{13}^l.next_{16}^{lb} = \&ao_{13}^2 \vee ao_{13}^l.next_{16}^{lb} = \mathsf{null}$. These formulae say that the `next` fields of both ao_{13}^1 and ao_{13}^2 may either point to one of the objects themselves or to null. This describes an unbounded linked path in the heap composed of objects abstracted by ao_{13}^1 or ao_{13}^2 and terminated by null.

B. Guarded Shape Templates

In order to use the base shape domain in our approach, we have to augment it with information about the guard variables that encode the program's control flow in the SSA. The guards express when an appropriate loop-back control edge is executed and the loop-back pointer has a defined value[7]. A row of a *guarded shape template* is defined as a formula $\mathcal{T}^G_{p_i^{lb}}(d_{p_i^{lb}}) \equiv G_{p_i^{lb}} \Rightarrow \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$ where $G_{p_i^{lb}}$ is a conjunction of SSA guards associated with the definition of the variable p_i^{lb} and $\mathcal{T}^S_{p_i^{lb}}$ is as in the base shape domain. If $G_{p_i^{lb}}$ is true for a program run, the definition of p_i^{lb} was reached in the run. A shape template $\mathcal{T}^G$ with guards is then a conjunction $\mathcal{T}^G \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^G_{p_i^{lb}}(d_{p_i^{lb}})$.

Let p_i^{lb} be a loop-back pointer abstracting the value of a pointer $p \in Ptr$ coming from the end of a loop $l \in L$. The row guard $G_{p_i^{lb}}$ is a conjunction of the following guards:

- The guard g_j^{lh} linked with the head of the loop l located at program location j, encoding that the loop l is reachable.
- The guard g_i^{ls} linked with the use of p_i^{lb}. The value of g_i^{ls} is true if p_i^{lb} is chosen as the value of the corresponding *phi* variable at the head of l (see Section II-C).
- If p_i^{lb} describes a pointer field of some abstract dynamic object (i.e. it has the form $ao_j^k.f_i^{lb}$ for some $ao_j^k \in AO, f \in Fld$), we also use the guard $g^{ao_j^k}$ linked with the allocation of ao_j^k at program location j. This guard conjoins the guard expressing reachability of program location j with the object-select guards $g_{j,l}^{os}$ and their negations denoting allocation of the k-th materialisation ao_j^k of the object allocated at j.

Example. In Section IV-A, we presented a shape invariant describing the linked segment created by the first loop from Listing 1. The corresponding guards for the two template rows of that invariant are $G_{ao_{13}^1.next_{16}^{lb}} = g_{10} \wedge g_{16}^{ls} \wedge (g_{13} \wedge g_{13}^{os})$ and $G_{ao_{13}^2.next_{16}^{lb}} = g_{10} \wedge g_{16}^{ls} \wedge (g_{13} \wedge \neg g_{13}^{os})$. Here, the loop head guard is g_{10}, the loop-select guard is g_{16}^{ls}, and the allocation guard is given by the guard of the reachability of the allocation site g_{13} and by the appropriate object-select guards (g_{13}^{os} for ao_{13}^1 and $\neg g_{13}^{os}$ for ao_{13}^2, respectively).

C. Shape Domain with Symbolic Loop Paths

Unfortunately, guarded shape templates are not precise enough for many heap-manipulating programs. One often needs to allow the invariant of a loop to be able to distinguish which loops were or were not executed while reaching the given loop. This can, e.g. distinguish which objects were allocated and can hence be processed in the given loop.

[6] Note that unlike the previously mentioned point-to relations, this relation is computed not just syntactically but using the considered abstract semantics.

[7] Using the base domain without the guard variables would be sound. However, it would produce very imprecise results since the abstract value would need to cover even states in which the loop-back edge was not taken.

To deal with the above problem, we introduce the concept of *symbolic loop paths* and compute different invariants for different paths. Since we use loop-select guards to express the control flow through the loops (see Section II-C), a symbolic loop path is simply a conjunction of loop-select guards.[8] Let G^{ls} be the set of all loop-select guards of all loops in a program. A symbolic loop path π is then formally defined as $\pi = \bigwedge_{g \in G^{ls}} l_g$ where l_g is a literal of the variable g, i.e. either g or $\neg g$. We use Π to denote the set of all symbolic loop paths of a given program. A *shape template extended with symbolic loop paths* is then given by a formula $\mathcal{T}^L \equiv \bigwedge_{\pi \in \Pi} \pi \implies \mathcal{T}^G_\pi$ where the $\mathcal{T}^G_\pi$ formulae are guarded shape templates as defined in Section IV-B. Here, $\pi_\perp$ a special path containing negative literals only. On that path no loop invariants are computed.

Example. We now show invariants for the pointer p for the second loop of the program in Listing 1. Using our (trace-insensitive) guarded shape domain, the corresponding template row would be $\mathcal{T}^G_{p^{lb}_{22}}(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\})$. In other words, p would be understood as possibly pointing to ao^1_{13} or ao^2_{13} even on paths where they were not allocated. However, symbolic loop paths allow us to obtain two different invariants depending on the execution of the first loop (for simplicity, we only provide the appropriate template row): namely, $g^{ls}_{16} \wedge g^{ls}_{22} \Rightarrow \mathcal{T}^G_{p^{lb}_{22}}(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\})$ for the case when the body of the first loop is executed and $\neg g^{ls}_{16} \wedge g^{ls}_{22} \Rightarrow \mathcal{T}^G_{p^{lb}_{22}}(\{\mathsf{null}\})$ for the case when the body of the first loop is not executed.

D. Combinations of Domains

The true power of the template-based verification approach lies in the simplicity of domain combinations. Since templates are general logical formulae, they can be easily composed, forming abstract domains capable of describing more complex properties of programs while relying on the solver to do the heavy-lifting on the combination of the domain operations and the mutual reduction of their abstract values.

1) Power Templates: The definition of shape templates with symbolic loop paths shows one way how a complex template can be formed from a simpler one. In this case, the template parameter, i.e. the abstract value, maps particular symbolic loop paths to sets of parameters of the original shape template. In fact, the shape domain could be replaced by any other abstract domain. The symbolic paths template can hence be viewed as a *power template*—in the sense of power domains [15]—which assigns to each element of the base domain an element of the exponent domain.

2) Product Templates: From the perspective of program analysis, a very interesting possibility is the combination of the shape domain with an abstract domain capable of describing values of variables of non-pointer types, e.g. numerical variables (such as the well-known interval or octagon domains). The simplest way to achieve such a combination is to use a *Cartesian product template* that combines templates of different kinds to be used independently side-by-side. The proposed shape template with loop-back guards $\mathcal{T}^G$ from Section IV-C can be combined with a template for analysis of numerical values $\mathcal{T}^V$ by simply taking their conjunction, i.e. $\mathcal{T}^G \wedge \mathcal{T}^V$. This not only allows us to analyse programs that use pointer and numerical variables simultaneously, but also to reason about the contents of data structures on the heap. We achieve this by analysing numerical fields of abstract dynamic objects using the value part of the template.

In addition, we use this product template as the inner template of the template with symbolic loop paths, forming an even stronger abstract domain: $\mathcal{T}^{LV} \equiv \bigwedge_{\pi \in \Pi} \pi \implies \mathcal{T}^G_\pi \wedge \mathcal{T}^V_\pi$. Using this domain for the running example allows us to analyse the shape and the contents of the linked list at the same time, obtaining the invariants described in Section I that enable us to prove the given property of interest.

V. Memory Safety Analysis

Apart from checking user-defined assertions, we can also verify memory safety. This includes a number of properties: * (1) pointer dereferencing safety, (2) `free` safety, and (3) absence of memory leaks.

A. Dereferencing a null Pointer

Since our invariants are over-approximating the reachable program states, we can soundly verify *may* (or better called *must-not*) properties. To check dereferences of null, for each expression $*p$ occurring in a program location i, we verify the assertion $p_j \neq \mathsf{null}$ where p_j is the version of p valid at i.

B. `Free` Safety

`Free` safety includes the absence of dereferencing a freed pointer and freeing an already freed pointer (a so-called “double free”). To prove absence from these errors, we introduce a new special variable fr initialised to null, which is then non-deterministically set to the address of the object to be freed in a `free` call. We replace each call of the form `free(p)` at program location i by a formula $fr_i = g^{fr}_i ? p_j : fr_k$, where p_j and fr_k are the versions of p and fr, respectively, valid in i, and g^{fr}_i is a free Boolean variable (a so-called *free guard*). Treating fr as a standard pointer-typed variable allows us to over-approximate the set of all freed addresses with the help of our shape domain. Then, in each program location i where either $*p$ or `free(p)` occurs, we can check for the assertion $p_j \neq fr_k$ to prove `free` safety (here, p_j and fr_k are again versions of p and fr, respectively, valid at i).

Even though this approach is sound, it is often too imprecise. Freeing one of the concrete objects does not mean that all objects were freed and that it is not safe any more to dereference/free the abstract object. To improve precision, we modify the representation of `malloc` calls. At each allocation site i, we add one more object ao^{co}_i to the set $\{ao^k_i\}$. The object can be chosen as the result of the allocation non-deterministically like any other ao^k_i, but it is guaranteed to be allocated only once (by an additional condition checking that, upon its allocation, no loop-back pointer can point to it). Hence, ao^{co}_i represents a concrete object. Then, for each

[8] The notion of symbolic loop paths can be easily generalised to program path sensitivity by including branches of conditional statements too.

meaning wht ??

allocation site i, we only allow $\&ao_i^{co}$ to be assigned to fr. The checks for free safety described above are done on concrete objects only, avoiding possible imprecision stemming from dealing with multiple objects represented by a single abstract object which would join the possibly different values of these objects. Also, as ao_i^{co} represents an arbitrary concrete object allocated at i, if safety can be proven for it, it can be assumed to hold for any other object allocated at i.

C. Absence of Memory Leaks

Using fr, we then check whether some ao_i^{co} object may be not freed at the end of the program (if there is a leak, it must be possible to show it on some concrete object). Unfortunately, as we do not track the sequencing of abstract objects representing a set of objects allocated at an allocation site (even when they form a list segment), our analysis typically sees that ao_i^{co} may be skipped in the deallocation loops, and hence remains inconclusive on the memory leaks.

VI. Implementation

We implemented[9] the proposed shape domain within the 2LS framework [35] that uses the template-based verification method described in Section II. We extended the SSA form generated by the framework to handle dynamic memory allocation. 2LS is based on the CPROVER framework [13], which includes an SMT solver based on reduction to propositional logic. We used Glucose 4.0 as the back-end solver in our experiments. We let 2LS inline all functions before running our analysis. For combination with numerical domains described in Section IV-D, we use the template polyhedra domain that is already a part of 2LS. Our approach handles any sequential C program, however, invariants are not inferred for array contents and memory manipulation using pointer arithmetic.

VII. Experiments

We performed the experiments to show how our approach improves the performance of 2LS and also how it compares to other state-of-the-art software verification tools.[10] We used BenchExec [4] to run the experiments with time limit set to 900 s and memory limit to 15 GB. The first comparison was done on the subcategories of the SV-COMP benchmarks [36] related to memory safety, particularly *ReachSafety-ControlFlow*, *ReachSafety-Heap*, *MemSafety-Heap*, *MemSafety-LinkedLists*, *MemSafety-Others*. Tasks in *ReachSafety* are checked for reachability of an error condition, tasks in *MemSafety* for absence of invalid pointer dereference, invalid free, and memory leaks. We compared our implementation to the version of 2LS from SV-COMP'17 without the proposed shape analysis.

The results are shown in Table I. The proposed method significantly improves the performance of the tool. Due to missing heap analysis support, the old version of 2LS often reported wrong results and therefore it had a negative score in three subcategories. 2LS with our analysis obtained a positive score in all subcategories and it is also faster in some of them.

Although the results show an improvement, we are still unable to compete with the best tools of SV-COMP'18 in the heap categories. This is mainly because our analysis does not yet support pointer arithmetic and is not yet expressive enough to handle various kinds of trees or nested lists. ??

However, the main purpose of our work was to extend possibilities of analysing combined shape and value properties of programs. To evaluate, we performed an experiment comparing our tool with the leaders of SV-COMP'18 in the heap-related categories, on tasks combining manipulation of unbounded data structures with a need to reason about the data stored in these structures. All these tasks[11] are correct programs created by our team, since no such programs are part of the SV-COMP benchmarks yet. For each task, we verify that no error state is reachable. The results of the evaluation are shown in Table II. Numbers in the table represent CPU time in seconds needed for the analysis of the example. The value *unknown* means that the tool was not able to analyse the task.

On these benchmarks, 2LS outperforms the other tools significantly. Even tools specialised in shape analysis, *Forester* [17] and *Predator* [16], often report unknown, timeout or even find a false error. This is probably caused by their inability to reason about the data stored in the lists. More general tools such as *Symbiotic* [9] or *Ultimate Automizer* [18] often time out since they probably lack an efficient abstraction for combination of shape and value properties. *CPAChecker* [3] (in the *CPA-Seq* configuration from SV-COMP'18) solved four tasks but times out on the rest.

VIII. Related Work

There is a vast body of work on shape analysis. We can only give an overview of the main lines of research in this section. For a more complete survey, we refer to [25].

Many of the existing approaches to shape analysis are based on abstract interpretation [14], some of them dating back to 1980s [23]. In particular, the TVLA engine [34] came with a flexible approach based on abstract interpretation over a set of user-supplied predicates. In comparison, our approach can be viewed as using a set of parametrised predicates.

Several further approaches based on abstract interpretation and various underlying formalisms (logics, automata, graphs) are mentioned below. In general, our approach differs in that it uses inductive invariant synthesis based on gradually refining parameters of templates via SMT solving on the SSA form (with no iterative execution), instead of iteratively executing the program using abstract transformers and widening until a fixed point is reached. Hence, our approach does not use widening over gradually growing instances of dynamic data structures to capture unbounded sets of instances of such structures. Also, it does not use on-demand materialisation of a concrete memory node from an abstract representation of a set of such nodes followed by again abstracting the resulting

[9] Available at https://github.com/diffblue/2ls/releases/tag/2ls-0.7.

[10] All tools, benchmarks, and results are available here: https://pschrammel.bitbucket.io/schrammel-it/research/2ls/fmcad18_exp.tar.xz.

[11] See https://github.com/diffblue/2ls/tree/2ls-0.7/regression/heap-data.

static better. can we use dyn to help? (ML)

TABLE I: Comparison of 2LS using the proposed method with the previous version of the tool over the SV-COMP benchmark.

	RS-ControlFlow		RS-Heap		MS-Heap		MS-LinkedLists		MS-Other	
	cpu (s)	score	cpu (s)	score	cpu (s)	score	cpu (s)	score	cpu (s)	score
2LS	252	64	41	106	17.5	59	107	7	29	46
2LS-old	1400	45	53	-161	190	-194	96	-182	23	46

TABLE II: Comparison of 2LS with other tools on examples combining unbounded data structures and their stored data.

	2LS	CPA-Seq	PredatorHP	Forester	Symbiotic	UAutomizer
Calendar	2.88	timeout	false	unknown	timeout	timeout
Cart	23.70	timeout	false	unknown	timeout	timeout
Hash Function	3.65	8.51	unknown	unknown	unknown	timeout
MinMax	5.14	timeout	false	unknown	timeout	timeout
Packet Filter	431.00	timeout	timeout	unknown	unknown	timeout
Process Queue	6.62	7.68	timeout	unknown	timeout	timeout
Quick Sort	18.20	3.50	timeout	unknown	unknown	5.75
Running Example	1.24	timeout	timeout	unknown	timeout	unknown
SM1	0.53	timeout	0.31	false	timeout	timeout
SM2	0.55	5.41	false	false	timeout	14.50

memory configuration. These aspects are handled by our encoding into guarded templates and representing `malloc` calls by choosing abstract objects from a predefined pool.

Various extensions of Hoare logic have been developed to cope with heap-manipulating programs. E.g., [22] proposed a way to reason about lists using the Mona tool, which was then extended to more complex data structures [29] and their contents [27]. Another program logic is separation logic [32], which enables reasoning about local memory modifications, rather than looking at the memory as a whole. It has been used for deductive program verification based on user-provided annotations [11]. Fully automated approaches based on separation logic and abstract interpretation have also been proposed and used, e.g., in the Space Invader [37] and SLAyer [2] tools.

More recently, automation of separation logic using SMT solvers by reduction to effectively propositional logic has been proposed by [31], [20], [21]. A different approach [30] uses the Houdini algorithm to find inductive invariants over heap predicates generated from grammars. These works share the common approach with our method to use SMT solvers to reason about heap properties; however, each of them uses different techniques for synthesising the invariant predicates. For an overview on template-based analysis techniques for numerical properties, we refer to [8].

Other fully automated approaches based on abstract interpretation build on shape graphs [26], such as the Predator tool [16], or tree automata and regular tree model checking, such as [6] or the Forester tool [17]. These approaches primarily aim at handling unbounded heap structures. Their combination with reasoning about value properties is not easy as shown in the works [1], [19] that extended Forester with reasoning about finite data and a specialised support for handling ordered list segments. As our experiments showed, Forester and Predator could handle almost none of our examples.

Several further abstract domains have been proposed for combining shape and data domains (e.g. [10], [5]). Our approach has the advantage that such domain combinations need not be designed from scratch.

Beyond the mentioned tools, several participants in SV-COMP, such as CPAChecker [3], Symbiotic [9], Ultimate Automizer [18], or CBMC [13], provide support for dealing with dynamic data structures and their content. However, they cannot handle data structures of unbounded size.

All the above methods are *store-based*, i.e., they describe the heap explicitly by a graph encoded in different ways. Other approaches are inspired by *storeless semantics* [24] using pointer access paths [12], [33], [28], [7] to describe reachability properties on the heap. This idea proved most suitable for our purposes. A *pointer access path* does not concretely express the heap state, it only describes which dynamic objects are reachable from a pointer. Using a set of access paths for each pointer, one can efficiently describe the shape of the heap. Compared with our method, the above approaches, however, use abstract interpretation over CFGs, and their support of dealing with the data content is limited [28].

IX. Conclusions and Future Work

We present a verification approach for heap-manipulating programs based on template-based invariant synthesis. We propose an abstract template domain capable of expressing reachability in dynamic data structures. We show that the domain can easily be combined with other domains to form power and product domains that are able to express complex properties about the shape and the contents of data structures. We experimentally evaluate our approach by within the 2LS framework. We plan to extend the technique to support pointer arithmetic and to develop templates that can express more complex data structure shapes, such as trees, skip-lists, or nested lists. Moreover, we work on using our method to infer function summaries to enable a modular verification approach.

Acknowledgement: The Czech authors were supported by the Czech Science Foundation project 17-12465S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the FIT BUT internal project FIT-S-17-4014. Viktor Malík and Martin Hruška are holders of the Brno Ph.D. Talent Scholarship, funded by the Brno City Municipality.

REFERENCES

[1] Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. In: Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 8172, pp. 224–239. Springer (2013)

[2] Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory Safety for Systems-Level Code. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 6806, pp. 178–183. Springer (2011)

[3] Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 6086, pp. 184–190. Springer (2011)

[4] Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: SPIN. Lecture Notes in Computer Science, vol. 9232, pp. 160–178. Springer (2015)

[5] Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: Programming Language Design and Implementation. pp. 578–589. ACM (2011)

[6] Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 4134, pp. 52–70. Springer (2006)

[7] Brain, M., David, C., Kroening, D., Schrammel, P.: Model and proof generation for heap-manipulating programs. In: Proceedings of the 23rd European Symposium on Programming. pp. 432–452. Springer (2014)

[8] Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety Verification and Refutation by k-Invariants and k-Induction. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 9291, pp. 145–161. Springer (2015)

[9] Chalupa, M., Vitovská, M., Strejcek, J.: SYMBIOTIC 5: Boosted instrumentation - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 10806, pp. 442–446. Springer (2018)

[10] Chang, B.E., Rival, X.: Relational inductive shape analysis. In: Principles of Programming Languages. pp. 247–260. ACM (2008)

[11] Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming 77(9), 1006–1036 (2012)

[12] Chong, S., , Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Proceedings of the 10th Static Analysis Symposium. pp. 463–482. Springer (2003)

[13] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)

[14] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages. pp. 238–252 (1977)

[15] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages. pp. 269–282 (1979)

[16] Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Static Analysis Symposium. pp. 215–237. Springer (2013)

[17] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Computer-Aided Verification. pp. 424–440. Springer (2011)

[18] 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: Tools and Algorithms for the Construction and Analysis of Systems. pp. 447–451 (2018)

[19] Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Vojnar, T.: Counterexample validation and interpolation-based refinement for forest automata. In: Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 10145, pp. 288–309. Springer (2017)

[20] Itzhaky, S., Banerjee, A., Immerman, N., Lahav, O., Nanevski, A., Sagiv, M.: Modular reasoning about heap paths via effectively propositional formulas. In: Principles of Programming Languages. pp. 385–396. ACM (2014)

[21] Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 8559, pp. 35–51. Springer (2014)

[22] Jensen, J.L., Jørgensen, M.E., Klarlund, N., Schwartzbach, M.I.: Automatic verification of pointer programs using monadic second-order logic. In: Programming Language Design and Implementation. pp. 226–236. ACM (1997)

[23] Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Principles of Programming Languages. pp. 66–74. ACM (1982)

[24] Jonkers, H.B.M.: Abstract storage structures. In: Algorithmic Languages. pp. 321–343. IFIP (1981)

[25] Kanvar, V., Khedker, U.P.: Heap abstractions for static analysis. ACM Comput. Surv. 49(2), 29:1–29:47 (2016)

[26] Laviron, V., Chang, B.E., Rival, X.: Separating shape graphs. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 6012, pp. 387–406. Springer (2010)

[27] Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Principles of Programming Languages. pp. 611–622. ACM (2011)

[28] Matosevic, I., Abdelrahman, T.S.: Efficient bottom-up heap analysis for symbolic path-based data access summaries. In: International Symposium on Code Generation and Optimisation. pp. 252–263. ACM (2012)

[29] Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Programming Language Design and Implementation. pp. 221–231. ACM (2001)

[30] Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Invariant synthesis for incomplete verification engines. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 10805, pp. 232–250. Springer (2018)

[31] Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013)

[32] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science. pp. 55–74. IEEE Computer Society (2002)

[33] Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: Proceedings of the 32nd Principles of Programming Languages. pp. 296–309 (2005)

[34] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Principles of Programming Languages. pp. 105–118. ACM (1999)

[35] Schrammel, P., Kroening, D.: 2LS for Program Analysis - (Competition Contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9636, pp. 905–907. Springer (2016)

[36] Software Verification Competition: Benchmarks (2017), https://github.com/sosy-lab/sv-benchmarks/

[37] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Computer-Aided Verification. Lecture Notes in Computer Science, vol. 5123, pp. 385–398. Springer (2008)