Automatically Refining Partial Specifications for Program Verification [⋆]

Shengchao Qin[1], Chenguang Luo[2], Wei-Ngan Chin[3], and Guanhua He[1,2]

[1]Teesside University [2]Durham University [3]National University of Singapore

Abstract. Automatically verifying heap-manipulating programs is a challenging task, especially when dealing with complex data structures with strong invariants, such as sorted lists and AVL/red-black trees. The verification process can greatly benefit from human assistance through specification annotations, but this process requires intellectual effort from users and is error-prone. In this paper, we propose a new approach to program verification that allows users to provide only partial specification to methods. Our approach will then refine the given annotation into a more complete specification by discovering missing constraints. The discovered constraints may involve both numerical and multi-set properties that could be later confirmed or revised by users. We further augment our approach by requiring only partial specification to be given for primary methods. Specifications for loops and auxiliary methods can then be systematically discovered by our augmented mechanism, with the help of information propagated from the primary methods. Our work is aimed at verifying beyond shape properties, with the eventual goal of analysing full functional properties for pointer-based data structures. Initial experiments have confirmed that we can automatically refine partial specifications with non-trivial constraints, thus making it easier for users to handle specifications with richer properties.

1 Introduction

Human assistance is often essential in (semi-) automated program verification. The user may supply annotations at certain program points, such as loop invariants and/or method specifications. These annotations can greatly narrow down the possible program states at that point, and avoid fixed-point calculation which could be expensive and may be less precise than the user's insight.

However, an obvious disadvantage of user annotation concerns its scalability, since programs to be analysed may be complicated and with significant diversity. Therefore, it may be unreasonable to expect user to provide specification for every method and invariant for every loop when verifying larger software systems. Furthermore, to err is human. A programmer may under-specify with too weak a precondition or over-specify with too strong a postcondition. Such mistakes could lead to failed verification, and it may be difficult to distinguish between a real bug or an inappropriate annotation.

[⋆] This work is supported in part by EPSRC project EP/G042322 and MoE ARF grant R-252-000-411-112.

To balance verification quality and human effort, we provide a novel approach to the verification of heap manipulating programs, which has long since been a challenging problem. To deal with such programs, which manipulate heap-allocated shared mutable data structures, one needs to keep track of not only "shape" information (for deep heap properties) but also related "pure" properties, such as structural numerical information (size and height), relational numerical information (balanced and sortedness properties), and content information (multi-set of symbolic values). Under our framework, the user is expected to provide partial specifications for *primary* methods with only *shape* information. Our verification will then take over the rest of the work to refine those partial specifications with derived (pure) constraints which should be satisfied by the program, or report a possible program bug if the given specifications are rejected by our verifier. This is an improvement over previous works [23], where users must provide full specifications for each method and invariants for each loop. This is also significantly different from the compositional shape analysis [5, 9, 32]. In spite of a higher level of automation, their analysis focuses on pointer safety only and deals primarily with a few built-in predicates over the shape domain only. Our work targets at both memory safety and functional correctness and supports user-defined predicates over several abstract domains (such as shape, numerical, multi-set).

Our approach allows the user to design their predicates for shapes and relative properties, to capture the desired level of program correctness to be verified. For example, with a singly-linked list structure `data node { int val; node next; }`, a user interested in pointer-safety may define a list shape predicate (as in [5, 9]):

$$\texttt{list(p)} \equiv (\texttt{p=null}) \vee (\exists \texttt{i}, \texttt{q} \cdot \texttt{p} \mapsto \texttt{node(i,q)} * \texttt{list(q)})$$

Note that in the inductive case, the separation conjunction $*$ ([28]) ensures that two heap portions (the head node and the tail list) are domain-disjoint.

Yet another user may be interested to track also the length of a list to analyse quantitative measures, such as heap/stack resource usage, using

$$\texttt{ll(p,n)} \equiv (\texttt{p=null} \wedge \texttt{n=0}) \vee (\texttt{p} \mapsto \texttt{node(\_,q)} * \texttt{ll(q,m)} \wedge \texttt{n=m+1})$$

Note that unbound variables, such as `q` and `m`, are implicitly existentially quantified, and _ is used to denote an existentially quantified anonymous variable. This predicate may be extended to capture the content information, to support a higher-level of correctness with multi-set (bag) property:

$$\texttt{llB(p,S)} \equiv (\texttt{p=null} \wedge \texttt{S}=\emptyset) \vee (\texttt{p} \mapsto \texttt{node(v,q)} * \texttt{llB(q,S}_1\texttt{)} \wedge \texttt{S}=\{\texttt{v}\} \sqcup \texttt{S}_1)$$

where the length of the list is implicitly captured by the cardinality $|\texttt{S}|$. A further strengthening can capture also the sortedness property:

$$\texttt{sllB(p,S)} \equiv (\texttt{p=null} \wedge \texttt{S}=\emptyset) \vee (\texttt{p} \mapsto \texttt{node(v,q)} * \texttt{sllB(q,S}_1\texttt{)} \wedge \texttt{S}=\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x} \in \texttt{S}_1 \cdot \texttt{v} \leq \texttt{x}))$$

Therefore, the user can provide predicate definitions w.r.t. various correctness level and program properties, which can be as simple as normal lists or as complicated as AVL trees, depending on their requirements. These predicates are non-trivial but can be reused multiple times for specifications of different methods. We have also built a library of predicates with respect to commonly-used data structures and useful program properties.

Based on these predicates, the user is expected to provide partial specifications for some primary methods which are the main objects of verification. Say, for a sorting algorithm taking $\texttt{x}$ as input parameter that is expected to be non-null, the user may provide $\texttt{llB}(\texttt{x}, \texttt{S}_1)$ as precondition and $\texttt{sllB}(\texttt{x}, \texttt{S}_2)$ as postcondition, and our approach will refine the specification as $\texttt{llB}(\texttt{x}, \texttt{S}_1) \wedge \texttt{x}{\neq}\texttt{null}$ for pre, and $\texttt{sllB}(\texttt{x}, \texttt{S}_2) \wedge \texttt{S}_1{=}\texttt{S}_2$ for post. Here we need user annotations as the initial specification, because we reserve the flexibility of verification w.r.t. different program properties at various correctness levels. For example, our approach can verify the same algorithm, but for different refined specifications, such as:

$$\begin{array}{llllll}
\texttt{requires} & \texttt{list}(\texttt{x}) \wedge & \texttt{x}{\neq}\texttt{null} & \texttt{ensures} & \texttt{list}(\texttt{x}) & \\
\texttt{requires} & \texttt{ll}(\texttt{x},\texttt{n}_1) \wedge & \texttt{n}_1{>}0 & \texttt{ensures} & \texttt{ll}(\texttt{x},\texttt{n}_2) \wedge & \texttt{n}_1{=}\texttt{n}_2 \\
\texttt{requires} & \texttt{llB}(\texttt{x},\texttt{S}_1) \wedge & \texttt{x}{\neq}\texttt{null} & \texttt{ensures} & \texttt{llB}(\texttt{x},\texttt{S}_2) \wedge & \texttt{S}_1{=}\texttt{S}_2 \\
\texttt{requires} & \texttt{llB}(\texttt{x},\texttt{S}) \wedge & \texttt{x}{\neq}\texttt{null} & \texttt{ensures} & \texttt{ll}(\texttt{x},\texttt{n}) \wedge & |\texttt{S}|{=}\texttt{n}
\end{array}$$

where the discovered missing constraints are shown in shaded form.

To summarise, our proposal for refining partial specification is aimed at harnessing the synergy between human's insights and machine's capability at automated program analysis. In particular, human's guidance can help narrow down on the most important of the different specifications that are possible with each program code, while automation by machine is important for minimising on the tedium faced by users. Our proposal has the following characteristics:

– *Specification completion*: We discover three types of constraints added into the user-given incomplete specification: constraints in the precondition for memory safety, (relational) constraints in postcondition to link the method's pre- and post-states, and constraints that the method's post-state satisfies.
– *Flexibility*: We allow the user to define their own predicates for the program properties they want to verify, so as to provide different levels of correctness. Meanwhile we aim at, and have covered much of, full functional correctness of pointer-manipulating programs such as data structure shapes, pointer safety, structural/relational numerical constraints, and bag information.
– *Reduction of user annotations*: Our approach uses program analysis techniques effectively to reduce users' annotations. As for our experiments, the user only has to supply the partial specifications for primary methods, and the analysis will compute pre- and postconditions for loops and auxiliary methods as well as refine primary methods' specifications.
– *Semi-Automation*: We classify our approach as semi-automatic, because the user is allowed to interfere and guide the verification at any point. For instance, they may provide invariant for a loop instead of our automated invariant generation, or choose some other constraints as refinement from what the verification has discovered.

We have built a prototype implementation and carried out a number of experiments to confirm the viability of the approach as described in Section 5. In what follows, we will first depict our approach informally using a motivating example and present technical details thereafter. More related works and concluding remarks come after the experimental results.Technical details not covered here can be found in our report [27].

2 Illustrative Example

We illustrate our approach with an example. We show how our analysis infers missing constraints to improve the user-supplied incomplete specification, and how it analyses the while loop without user-annotations.

```
0 data node2 { int val; node2 prev; node2 next; }
1 node2 sdl2nbt(node2 head,
                node2 tail)
2  requires  sdlB(head,p,q,S)
3  ensures  nbt(res,S_res)
4 {node2 root = head;
5  node2 end = head;
6  while(end != tail) {
7    end = end.next;
8    if (end != tail) {
9      end=end.next; root=root.next;}
10 }
11 if (head == root) root.prev = null;
12 else root.prev = sdl2nbt(head,root);
13 node2 tmp = root.next;
14 if (tmp == tail) root.next = null;
15 else { tmp.prev = null;
16   root.next = sdl2nbt(tmp, tail);}
17 root;}
```

Fig. 1. The method to convert a sorted doubly-linked list to a node-balanced tree.

The method `sdl2nbt` (Fig 1) converts a doubly-linked sorted list into a node-balanced binary search tree, as indicated by the shape-only specification in lines 2 and 3. It first finds the "centre" node in the list (`root`), where the difference between numbers of nodes to the left and to the right of the centre node is at most one (lines 5-10), as Fig 2 (a) shows. It then applies the algorithm recursively on both list segments to the left and to the right of the centre node, and regards the centre node as the tree's root, whose left and right children are the resulted subtrees' roots from the recursive calls (lines 11-17), as in Fig 2 (b) and (c). As the data structures of doubly-linked list and binary tree are homomorphic (line 0), the algorithm reuses the nodes in the input instead of creating a new tree, making itself in-place. The parameter `head` in line 1 denotes the first node of the input list, and `tail` is where the last node's `next` field points to. When using this method `tail` should be set as `null` initially.

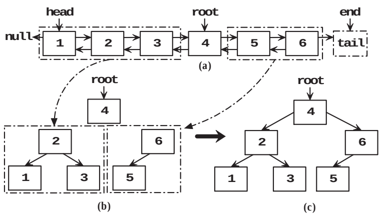

Fig. 2. sdl2nbt

The predicate for doubly-linked sorted list (segment) is defined as follows:

$$\begin{aligned}&\mathtt{sdlB}(\mathtt{root},\mathtt{p},\mathtt{q},\mathtt{S}) \equiv (\mathtt{root}{=}\mathtt{q} \wedge \mathtt{S}{=}\emptyset) \ \vee\ (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{r}) *\\ &\quad \mathtt{sdlB}(\mathtt{r},\mathtt{root},\mathtt{q},\mathtt{S_1}) \wedge \mathtt{root}{\neq}\mathtt{q} \wedge \mathtt{S}{=}\{\mathtt{v}\} \sqcup \mathtt{S_1} \wedge (\forall \mathtt{x}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\leq}\mathtt{x}))\end{aligned}$$

where the parameters `p` and `q` denote resp. the `prev` field of `root` and the `next` field of the last node in the list, and `S` represents the list's content. And below is the predicate specification for node-balanced binary search trees:

$$\begin{aligned}&\mathtt{nbt}(\mathtt{root},\mathtt{S}) \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee\\ &\quad (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}) * \mathtt{nbt}(\mathtt{p},\mathtt{S_p}) * \mathtt{nbt}(\mathtt{q},\mathtt{S_q}) \wedge \mathtt{S}{=}\{\mathtt{v}\} \sqcup \mathtt{S_p} \sqcup \mathtt{S_q} \wedge\\ &\qquad (\forall \mathtt{x}{\in}\mathtt{S_p}{\cdot}\mathtt{x}{\leq}\mathtt{v}) \wedge (\forall \mathtt{x}{\in}\mathtt{S_q}{\cdot}\mathtt{v}{\leq}\mathtt{x}) \wedge -1{\leq}|\mathtt{S_p}|{-}|\mathtt{S_q}|{\leq}1)\end{aligned}$$

where `S` captures the content of the tree. We require the difference in node numbers of the left and right sub-trees be within one, as the node-balanced property indicates.

To refine sdl2nbt's specification, our analysis proceeds in two steps. Firstly, starting from the partial precondition (line 2 of Fig 1), a forward analysis is conducted to compute the postcondition of the method in the form of a *constraint abstraction* [15]. This constraint abstraction is effectively a transfer function for the method, which may be recursively defined (e.g. in this example). Secondly, instead of a direct fixpoint computation in the combined abstract domain (with shape, numerical and bag information), a "pure" constraint abstraction (without heap shape information) is derived from the generated constraint abstraction and the user-given partial postcondition. This pure constraint abstraction is then solved by fixpoint solvers in pure (numerical/bag) domains, such as [24, 25].

As for the example, when the forward analysis reaches the while loop at line 6, it discovers that the loop has no user-supplied annotations. In that case, it uses an augmented technique (details follow slightly later) to synthesise the loop's pre- and post-shapes, and invoke the analysis procedure recursively to find additional pure constraints. In this way, we can infer the while loop's postcondition as

$$\begin{array}{c}\mathtt{sdlB}(\mathtt{head},\mathtt{null},\mathtt{root},\mathtt{S_h}) * \mathtt{sdlB}(\mathtt{root},\mathtt{p},\mathtt{tail},\mathtt{S_r}) \wedge \\ \mathtt{end}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_h}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h}, \mathtt{y}{\in}\mathtt{S_r}\cdot \mathtt{x}{\leq}\mathtt{y}) \wedge \underline{0{\leq}|\mathtt{S_r}|{-}|\mathtt{S_h}|{\leq}1}\end{array} \tag{1}$$

which indicates that the original list starting from head is cut into two sorted pieces with a cutpoint root. Meanwhile, the essential constraint (the underlined part, saying the list segment beginning with head is at most one node shorter than that with root) to ensure the node-balanced property is derived as well.

When the symbolic execution finishes, it generates the following constraint abstraction as the postcondition of the method:

$$\begin{array}{l}\mathtt{Q}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}}) ::= \qquad\qquad\qquad\qquad\qquad\qquad (\dagger)\\ \quad \mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{null},\mathtt{null}){\wedge}\mathtt{head}{=}\mathtt{root}{=}\mathtt{res}{\wedge}\mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail}{\wedge}\mathtt{p}{=}\mathtt{null}{\wedge}\mathtt{S}{=}\{\mathtt{v}\}\\ \vee\ \mathtt{head}{\mapsto}\mathtt{node2}(\mathtt{s},\mathtt{null},\mathtt{root}) * \mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{res_h},\mathtt{null}) \wedge \mathtt{res}{=}\mathtt{root} \wedge \\ \qquad \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\mathtt{s},\mathtt{v}\} \wedge \mathtt{s}{\leq}\mathtt{v}\\ \vee\ \mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{res_h},\mathtt{res_r}) * \mathtt{Q}(\mathtt{head},\mathtt{p},\mathtt{root},\mathtt{S_h},\mathtt{res_h},\mathtt{S^h_{res}}) * \\ \qquad \mathtt{Q}(\mathtt{tmp},\mathtt{null},\mathtt{tail},\mathtt{S_r},\mathtt{res_r},\mathtt{S^r_{res}}) \wedge \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{\neq}\mathtt{tail} \wedge \\ \qquad \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_h}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h}, \mathtt{y}{\in}\mathtt{S_r}\cdot \mathtt{x}{\leq}\mathtt{v}{\leq}\mathtt{y}) \wedge 0{\leq}|\mathtt{S_r}|{-}|\mathtt{S_h}|{\leq}1\end{array}$$

where the first two disjunctive branches are base cases of the method's invocation (where there are only one and two nodes in the returned tree res, respectively), and the last denotes the effect of recursive calls combined into the postcondition (where root's both branches are node-balanced trees). Note that the two Q's in the last branch correspond to the invocations of sdl2nbt in lines 12 and 16. Constraints of some logical variables (like $\mathtt{S_{res}}$) will not show up until the next step.

In the second step, to derive the definition of the pure constraint abstraction P from the above post-state Q, we use each disjunctive branch of Q to entail the user-given post-shape (with appropriate instantiations of the parameters). During this process, all occurrences of Q are replaced by the post-shape conjoined with the P according to the entailment relation

$$\mathtt{Q}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}}) \vdash \mathtt{nbt}(\mathtt{res},\mathtt{S_{res}}) \wedge \mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}})$$

The obtained frames (from the Sleek prover [23]) are used to form (via disjunction) the definition of P:

$$\begin{aligned}&\mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S}_{\mathtt{res}}) ::= \qquad (\ddagger)\\ &\quad \mathtt{head}{=}\mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{S}{=}\mathtt{S}_{\mathtt{res}}{=}\{\mathtt{v}\}\\ &\vee\ \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{res}{=}\mathtt{root} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} {\wedge} \mathtt{S}{=}\mathtt{S}_{\mathtt{res}}{=}\{\mathtt{s},\mathtt{v}\} \wedge \mathtt{s}{\leq}\mathtt{v}\\ &\vee\ \mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{root},\mathtt{S}_{\mathtt{h}},\mathtt{res}_{\mathtt{h}},\mathtt{S}^{\mathtt{h}}_{\mathtt{res}}) \wedge \mathtt{P}(\mathtt{tmp},\mathtt{null},\mathtt{tail},\mathtt{S}_{\mathtt{r}},\mathtt{res}_{\mathtt{r}},\mathtt{S}^{\mathtt{r}}_{\mathtt{res}}) \wedge\\ &\qquad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{\neq}\mathtt{tail} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S}_{\mathtt{h}}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S}_{\mathtt{r}} \wedge\\ &\qquad \mathtt{S}_{\mathtt{res}}{=}\mathtt{S}^{\mathtt{h}}_{\mathtt{res}}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S}^{\mathtt{r}}_{\mathtt{res}} \wedge (\forall \mathtt{x}{\in}\mathtt{S}_{\mathtt{h}},\mathtt{y}{\in}\mathtt{S}_{\mathtt{r}}\cdot\mathtt{x}{\leq}\mathtt{v}{\leq}\mathtt{y}) \wedge 0{\leq}|\mathtt{S}_{\mathtt{r}}|{-}|\mathtt{S}_{\mathtt{h}}|{\leq}1\end{aligned}$$

We then use pure fixpoint solvers to obtain a closed-form formula $\mathtt{p}{=}\mathtt{null} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S}_{\mathtt{res}} \wedge |\mathtt{S}|{\geq}1$ for `P`, and refine the method's specifications as

$$\begin{aligned}&\mathtt{requires}\ \ \mathtt{sdlB}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S}) \wedge\ \mathtt{p}{=}\mathtt{null} \wedge \mathtt{q}{=}\mathtt{tail} \wedge |\mathtt{S}|{\geq}1\\ &\mathtt{ensures}\ \ \ \mathtt{nbt}(\mathtt{res},\mathtt{S}_{\mathtt{res}}) \wedge\ \mathtt{S}{=}\mathtt{S}_{\mathtt{res}}\end{aligned}$$

which proposes more requirements in the precondition, as the `head`'s `prev` field should be `null`, and the whole list's last node's `next` field must point to `tail` for termination. Meanwhile, there should be at least one node in the list for memory safety. With those obligations, the method guarantees that the result is a node-balanced binary search tree, with the same content as the input list.[1]

Analysis for the while loop. The while loop in `sdl2nbt` (lines 6-10) discovers the centre node of the given list segment referenced by `head`. It traverses the list segment with two pointers `root` and `end`. The `end` pointer goes towards the list segment's tail twice as fast as `root`. When `end` arrives at the tail of the segment (`tail`), `root` will point to the list segment's centre node.

Instead of requiring users to supply the loop invariant, our analysis regards the loop as a tail-recursive method and computes its specifications based on the program state in which the loop starts. Our analysis first synthesises its pre- and post-shapes, and then continues the analysis in the same way as for the main method. The pre-shape can be abstracted from the program state in which the loop starts. The post-shape synthesis is done by checking the symbolic execution result of the loop body (unrolled once) against possible abstracted shapes. For this example, we first generate shape candidates according to the variables accessed by the loop, such as (a) $\mathtt{sdlB}(\mathtt{head},\mathtt{p}_{\mathtt{h}},\mathtt{q}_{\mathtt{h}},\mathtt{S}_{\mathtt{h}}) * \mathtt{sdlB}(\mathtt{root},\mathtt{p}_{\mathtt{r}},\mathtt{q}_{\mathtt{r}},\mathtt{S}_{\mathtt{r}})$, and (b) $\mathtt{sdlB}(\mathtt{head},\mathtt{p}_{\mathtt{h}},\mathtt{q}_{\mathtt{h}},\mathtt{S}_{\mathtt{h}}) {*} \mathtt{nbt}(\mathtt{root},\mathtt{h}_{\mathtt{r}},\mathtt{b}_{\mathtt{r}},\mathtt{S}_{\mathtt{r}})$. Then the unrolled loop body is symbolically executed to filter out those shapes that are not valid to be an abstraction of postcondition. For this example, executing the loop body yields

$$\begin{aligned}&\mathtt{head}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{end}) \wedge \mathtt{head}{=}\mathtt{root} \wedge \mathtt{end}{=}\mathtt{tail}\ \vee\\ &\quad \mathtt{head}{\mapsto}\mathtt{node2}(\mathtt{v}_{\mathtt{h}},\mathtt{p},\mathtt{root}) * \mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v}_{\mathtt{r}},\mathtt{head},\mathtt{end}) \wedge \mathtt{end}{=}\mathtt{tail}\end{aligned} \qquad (2)$$

where (b) is directly filtered out since $(2) \vdash (b) * \mathtt{true}$ fails. However (a) remains a candidate, as $(2) \vdash (a) * \mathtt{true}$ holds. Therefore, regarding (a) as a possible post-shape, we can employ the same approach to generate a constraint abstraction for the while loop, and solve it to obtain formula (1) in page 5.

3 Language and Abstract Domain

We focus on a strongly-typed C-like imperative language in Fig 3. A program *Prog* consists of type declarations *tdecl*, which can define either data type *datat* (e.g. `node`) or predicate *spred* (e.g. `llB`), and some method declarations *meth*. The language is expression-oriented, so the body of a method is an expression

[1] We will explain how to attach the fixpoint result to both pre and post in Sec 4.

$$
\begin{array}{lll}
\mathit{Prog} ::= \mathit{tdecl}^* \ \mathit{meth}^* & \mathit{tdecl} ::= \mathit{datat} \mid \mathit{spred} & \\
\mathit{datat} ::= \texttt{data}\ c\ \{\ \mathit{field}^*\ \} & \mathit{field} ::= t\ v & t ::= c \mid \tau \\
\mathit{meth} ::= t\ mn\ ((t\ v)^*; (t\ v)^*)\ \mathit{mspec}^*\ \{e\} & & \tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} \\
e \quad ::= d \mid d[v] \mid v{=}e \mid e_1; e_2 \mid t\ v;\ e \mid \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ (v)\ \{e\} & & \\
d \quad ::= \texttt{null} \mid k^{\tau} \mid v \mid \texttt{new}\ c(v^*) \mid mn(u^*; v^*) & & \\
d[v] \quad ::= v.f \mid v.f{:=}w \mid \texttt{free}(v) & &
\end{array}
$$

Fig. 3. A Core (C-like) Imperative Language.

composed of e (the recursively defined program constructor) and d and $d[v]$ (atom instructions/expressions). We also allow both call-by-value and call-by-reference method parameters (which are separated with a semicolon ;).

$$
\begin{array}{ll}
\mathit{spred} & ::= \mathit{pred}(v^*) \equiv \Phi \\
\mathit{mspec} & ::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \\
\Delta & ::= \mathtt{Q}(v^*) \mid \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \\
\Phi & ::= \bigvee \sigma^* \qquad \sigma ::= \exists v^* \cdot \kappa \wedge \pi \\
\Upsilon & ::= \mathtt{P}(v^*) \mid \bigvee \omega^* \mid \Upsilon_1 \wedge \Upsilon_2 \mid \Upsilon_1 \vee \Upsilon_2 \mid \exists v \cdot \Upsilon \\
\kappa & ::= \texttt{emp} \mid v \mapsto c(v^*) \mid \mathit{pred}(v^*) \mid \kappa_1 * \kappa_2 \\
\omega & ::= \exists v^* \cdot \pi \qquad \pi ::= \gamma \wedge \phi \\
\gamma & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1 \wedge \gamma_2 \\
\phi & ::= \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 \qquad a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \mid |\mathtt{B}| \\
\varphi & ::= v \in \mathtt{B} \mid \mathtt{B}_1{=}\mathtt{B}_2 \mid \mathtt{B}_1 \sqsubset \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubseteq \mathtt{B}_2 \mid \forall v \in \mathtt{B} \cdot \phi \mid \exists v \in \mathtt{B} \cdot \phi \\
\mathtt{B} & ::= \mathtt{B}_1 \sqcup \mathtt{B}_2 \mid \mathtt{B}_1 \sqcap \mathtt{B}_2 \mid \mathtt{B}_1 - \mathtt{B}_2 \mid \{\} \mid \{v\}
\end{array}
$$

Fig. 4. The Specification Language.

Our specification language (in Fig 4) allows (user-defined) shape predicates to specify both separation and pure properties. The shape predicates *spred* are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [23]. A conjunctive abstract program state, σ, is composed of a heap (shape) part κ and a pure part π, where π consists of γ, ϕ and φ as aliasing, numerical and bag information, respectively. We use SH to denote the set of such conjunctive states. During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ. Note that constraint abstractions (e.g. $\mathtt{Q}(\mathtt{v}^*)$) may occur in Δ during the analysis. A closed-form Δ (containing no constraint abstractions) can be normalised to the Φ form [23]. Pure constraint abstraction $\mathtt{P}$ is analogously defined to $\mathtt{Q}$.

Our memory model is adapted from that of separation logic [28], except that we consider memory cells to be structured records. The detailed model definitions can be found in Nguyen et al. [23]. Meanwhile, for program variables in abstract states, we use unprimed ones to denote their initial values and primed ones for current values [23, 27].

4 The Analysis

In this section, we first formulate the main analysis for (primary) methods with given shape specifications. We then show how the analysis is extended to handle auxiliary methods (including loops) without user annotations.

4.1 Refining Specifications for Primary Methods

The algorithm for refinement (CA_Gen_Solve) is given in Fig 5. As illustrated in Section 2, the analysis proceeds in two steps for a primary method with shape information given in specification, namely (1) forward analysis (at lines 1-2) and (2) pure constraint abstraction generation and solving (at lines 3-10).

```
Algorithm CA_Gen_Solve($\mathcal{T}, mn, e, \Phi_{pr}, \Phi_{po}, u^*, v^*$)
 1   $\Delta$ := Symb_Exec($\mathcal{T}, mn, e, \Phi_{pr}$)
 2   if $\Delta$ = fail then return fail end if
 3   Normalise $\Delta$ to DNF, and denote as $\bigvee_{i=1}^{m} \Delta_i$
 4   $w^* := \{u^*, v^*, v'^*\} \cup$ pureV($\{u^*, v^*, v'^*\}, \Phi_{pr} \vee \Phi_{po}$)
 5   $\Delta_P$ := Pure_CA_Gen($\Phi_{po}$, Q($w^*$)::= $\bigvee_{i=1}^{m} \Delta_i$)
 6   if $\Delta_P$ = fail then return fail end if
 7   $\pi$ := Pure_CA_Solve(P($w^*$)::=$\Delta_P$)
 8   $R := t\ mn\ ((t\ u)^*; (t\ v)^*)$ requires
         ex_quan($\Phi_{pr}, \pi$) ensures ex_quan($\Phi_{po}, \pi$)
 9   if Verify($\mathcal{T}, mn, R$) then return $\mathcal{T} \cup \{R\}$ \
       { $t\ mn\ ((t\ u)^*; (t\ v)^*)$ requires $\Phi_{pr}$ ensures $\Phi_{po}$ }
10   else return fail end if
end Algorithm
```

```
Algorithm Symb_Exec
   ($\mathcal{T}, mn, e, \Phi_{pr}$)
11 $errLbls := \emptyset$
12 do
13   $(\Delta, l) := [\![e]\!]_{\mathcal{T}}^{mn}(\Phi_{pr}, 0)$
14   if $l{>}0 \wedge l \notin errLbls$ then
15     $\Phi_{pr}$:=ex_quan($\Phi_{pr}, \Delta$);
16     $errLbls := errLbls \cup \{l\}$
17   else if $l{>}0 \wedge l \in errLbls$
     then return fail
18   end if
19 while $l > 0$
20 return $\Delta$
end Algorithm
```

Fig. 5. Refining method specifications.

The forward analysis is captured as algorithm Symb_Exec to the right of Fig 5. Starting from a given pre-shape Φ_{pr}, it analyses the method body e to compute the post-state in constraint abstraction form. Due to space constraints, the symbolic execution rules are given in our technical report [27]. They are similar to symbolic rules used in [23], except for a novel mechanism to derive pure precondition, which we refer to as *pure abduction*.

This pure abduction mechanism is invoked whenever symbolic execution fails to prove memory safety based on the current prestate. For example, if the current state is `ll(x,n)` (a list that is possibly empty) but `x`$\mapsto$`node(_,p)` is required by the next program instruction, our pure abduction mechanism will infer `n`$\geq$`1` to add to the current state to satisfy the requirement. The variable *errLbls* (initialised at line 11) is to record the program locations in which previous pure abductions occurred. Whenever the symbolic execution fails, it returns a state Δ that contains the pure abduction result and the location l where failure was detected, as shown in line 13. If the current abduction location l is not recorded in *errLbls*, it indicates that this is a new failure. The abduction result is added to the precondition of the current method to obtain a stronger Φ_{pr}, before the algorithm enters the symbolic execution loop with variable *errLbls* updated to add in the new failure location l. This loop is repeated until symbolic execution succeeds with no memory error, or a previous failure point was re-encountered. The latter may indicate a program bug or a specification error, or may be due to the possible incompleteness of the underlying SLEEK prover we use.

Back to the main algorithm CA_Gen_Solve, the analysis next builds a heap-based constraint abstraction mechanism, named Q(w^*), for the post-state in steps 3-4. This constraint abstraction is possibly recursive. (Definition † in page 5 is an example of this heap-based abstraction.) We then make use of another algorithm in Fig 6, named Pure_CA_Gen, to extract a pure constraint abstraction, named $\mathtt{P}(w^*)$, without any heap property. (Definition ‡ in page 6 is an instance of this pure abstraction.) This algorithm tries to derive a branch $\mathtt{P}_i$ for each branch Δ_i of $\mathtt{Q}$. For every Δ_i it proceeds in two steps. In the first step (lines 22-24), it replaces the recursive occurrence of $\mathtt{Q}$ in Δ_i with $\sigma * \mathtt{P}(w^*)$. In the second step (lines 25-26) it tries to derive $\mathtt{P}_i$ via the entailment. If the entailment fails, then pure abduction is used to discover any missing pure constraint σ'_i for $\rho\Delta_i$ to allow the entailment to succeed. In this case, σ'_i is incorporated into σ_i (and eventually $\mathtt{P}_i$). Once this is done, we use some existing fixpoint analysis (e.g. [25]) inside Pure_CA_Solve to derive non-recursive constraint π, as a simplification of $\mathtt{P}(w^*)$. This result is then incorporated into the pre/post specifications in line 8, before we perform a post verification in line 9 using the Hip verifier [23], to ensure the strengthened precondition is strong enough for memory safety.

Two auxiliary functions used in the algorithm are described here. The function $\mathsf{pureV}(V, \Delta)$ retrieves from Δ the shapes referred to by all pointer variables from V, and returns the set of logical variables used to record numerical (size and bag) properties in these shapes, e.g. $\mathsf{pureV}(\{\mathtt{x}\}, \mathtt{ll}(\mathtt{x},\mathtt{n}))$ returns $\{\mathtt{n}\}$. This function is used in the algorithm to ensure that all free variables in Φ_{pr} and Φ_{po} are added into the parameter list of the constraint abstraction $\mathtt{Q}$. The function $\mathsf{ex\_quan}(\Delta, \pi)$ is to strengthen the state Δ with the abduction result π: $\mathsf{ex\_quan}(\Delta, \pi) =_{df} \Delta \wedge \exists(\mathsf{fv}(\pi) \setminus \mathsf{fv}(\Delta)) \cdot \pi$. It is used to incorporate the discovered missing pure constraints into the original specification. For example, $\mathsf{ex\_quan}(\mathtt{ll}(\mathtt{x},\mathtt{n}), \mathtt{0{<}m} \wedge \mathtt{m{\leq}n})$ returns $\mathtt{ll}(\mathtt{x},\mathtt{n}) \wedge \mathtt{0{<}n}$.

```
**Algorithm** Pure_CA_Gen($\sigma$, $\mathtt{Q}(w^*)$::= $\bigvee_{i=1}^{m} \Delta_i$)
21 **for** $i = 1$ **to** $m$
22     Denote all appearances of $\mathtt{Q}(w^*)$ in $\Delta_i$ as $\mathtt{Q}_j(w_j^*), j = 1, ..., p$
23     Denote substitutions $\rho_j = [(([w_j^*/w^*]\sigma * \mathtt{P}(w_j^*))/\mathtt{Q}_j(w_j^*)]$
24     Let substitution $\rho := \rho_1 \circ \rho_2 \circ ... \circ \rho_p$ as applying all substitutions
         defined above in sequence
25     **if** $(\rho\Delta_i \vdash \sigma * \sigma_i$ **or** $\rho\Delta_i \wedge [\sigma'_i] \rhd \sigma * \sigma_i)$ **and** $ispure(\sigma_i)$ **then** $\mathtt{P}_i := \sigma_i$
26     **else return** fail **end if**
27 **end for**
28 **return** $\bigvee_{i=1}^{m} \mathtt{P}_i$
**end Algorithm**
```

Fig. 6. Pure constraint abstraction generation algorithm.

Pure abduction mechanism. We use the Sleek prover [23] to check Δ_1 entails Δ_2. If the entailment holds it also derives the frame Δ_3 such that $\Delta_1 \vdash \Delta_2 * \Delta_3$. However, if it fails, we assume that the shape information is sufficiently provided, and use our pure abduction mechanism ($\sigma_1 \wedge [\sigma'] \rhd \sigma_2 * \sigma_3$ in Fig 7) to discover missing pure constraints σ' so that $\sigma_1 \wedge \sigma' \vdash \sigma_2 * \sigma_3$.

Our pure abduction deals with three different cases. The first rule (**R1**) applies when the LHS (σ) does not entail the RHS (σ_1) but the RHS entails the LHS with some pure formula (σ') as the frame; e.g. in $\mathtt{ll}(\mathtt{x},\mathtt{n}) \nvdash \mathtt{x} {\mapsto} \mathtt{node}(\_, \mathtt{null})$, the RHS can entail the LHS with pure frame $\mathtt{n{=}1}$. The abduction then checks to ensure $\texttt{ll(x,n)} \wedge \texttt{n=1} \vdash \texttt{x}{\mapsto}\texttt{node(\_,null)}{*}\sigma_2$ for some σ_2, and returns the result $\texttt{n=1}$. Note the check $ispure(\sigma')$ ensures that σ' contains no heap information.

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad ispure(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2} \;\textbf{(R1)}$$

$$\frac{\begin{array}{c}\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \\ (\sigma_0 \vdash \sigma_1 * \sigma' \text{ or } \sigma_0 \wedge [\sigma_0'] \rhd \sigma_1 * \sigma') \quad ispure(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2\end{array}}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2} \;\textbf{(R2)}$$

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma_1 \wedge [\sigma_1'] \rhd \sigma * \sigma' \quad ispure(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2} \;\textbf{(R3)}$$

Fig. 7. Pure abduction rules.

In the second rule (**R2**), neither side entails the other but the LHS term could be unfolded. An example is $\sigma = \texttt{sllB(x,S)}$, $\sigma_1 = \texttt{x}{\mapsto}\texttt{node(u,p)} * \texttt{p}{\mapsto}\texttt{node(v,null)}$. As the shape predicates on the LHS are of disjunctive forms (i.e. $\texttt{sllB}$ in σ), certain branches of σ may entail σ_1. As the rule suggests, to accomplish abduction $\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2$, we first unfold σ and try entailment or further abduction with the results (σ_0) against σ_1. If it succeeds with a pure frame σ', then we confirm the abduction by checking $\sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2$. For the example above, the abduction returns $|\texttt{S}|{=}2$ (σ') and discovers the nontrivial frame $\texttt{S}{=}\{\texttt{u},\texttt{v}\} \wedge \texttt{u}{\leq}\texttt{v}$ (σ_2). Note that function $\mathsf{data\_no}$ returns the number of data nodes in a state, e.g. it returns one for $\texttt{x}{\mapsto}\texttt{node(v,p)} * \texttt{ll(p,m)}$. (This syntactic check is important for the termination of the abduction.) The unroll operation unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{u} \sigma^i$), and returns the result as a set of formulae ($\{\sigma^1, ..., \sigma^u\}$).

In the third rule (**R3**), neither side entails the other and the LHS term cannot be unfolded. e.g., $\sigma = \texttt{x}{\mapsto}\texttt{node(u,p)} * \texttt{p}{\mapsto}\texttt{node(v,null)}$, $\sigma_1 = \exists \texttt{S} \cdot \texttt{sllB(x,S)}$. In this case, the rule swaps the two sides of the entailment and applies the second rule to uncover the pure constraints σ_1' and σ'. It checks that adding σ' to the LHS (σ) entails the RHS (σ_1) before it returns σ'. For the example, the abduction returns $\texttt{u}{\leq}\texttt{v}$ which is essential for the two nodes to form a sorted list (RHS).

4.2 Inferring Specifications for Auxiliary Methods and Loops

For auxiliary methods[2], we conduct a pre-analysis (Fig 8) to synthesise the pre- and post-shapes before we conduct the refinement analysis from Fig 5. Loops are dealt with by analysing their tail-recursive versions in the same way. This approach alleviates the need for users to provide specification annotations for both loops and auxiliary methods.

The pre-shape synthesis algorithm SynPre (Fig 8 left) takes in as input the set of shape predicates ($\mathcal{S}$), the auxiliary method name (f), its formal parameters (u^*, v^*), the current symbolic state in which f is called (σ), and the corresponding actual parameters (x^*, y^*) of the invocation. The algorithm first obtains possible shape candidates from the parameters u^*, v^* with $\mathsf{ShpCand}$ (line 1),

[2] In practice, we treat methods without user-specified shape specifications as auxiliary.

```
Algorithm SynPre
  ($\mathcal{S}, f, u^*, v^*, \sigma, x^*, y^*$)
 1  $C :=$ ShpCand($\mathcal{S}, u^*, v^*$)
 2  for $\sigma_C \in C$ do
 3    if $\sigma \nvdash [x^*/u^*, y^*/v^*]\sigma_C$
 4    then $C{:=}C \backslash \{\sigma_C\}$ end if
 5  end for
 6  return $C$
end Algorithm
```

```
Algorithm SynPost ($\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, u^*, v^*$)
 7  $C :=$ ShpCand($\mathcal{S}, u^*, v^*$)
 8  $\mathcal{T}'{:=}\mathcal{T} \cup \{f(u^*{,}v^*)$ requires $\Phi_{pr}$ ensures false $\{e\}\}$
 9  $\Delta :=$ Symb_Exec($\mathcal{T}', f,$ syn_unroll($f, e$), $\Phi_{pr}$)
10  for $\sigma_C \in C$ do
11     if $\Delta \wedge [\sigma] \not\triangleright \sigma_C$ then $C := C \backslash \{\sigma_C\}$ end if
12  end for
13  return pair_spec_list($\Phi_{pr}, C$)
end Algorithm
```

Fig. 8. Shape synthesis algorithms.

then picks up a sound abstraction for the method's pre-shape with entailment, and filter out the ones which fail (line 4). Finally the pre-shape abstraction is returned. While we use an enumeration strategy here, the number of possible shape candidates per type is small as it is strictly limited by what the user provides in the primary methods, and then filtered and prioritised by our system.

To synthesise post-shapes (SynPost, Fig 8 right), we also assign C as possible shape candidates (line 7). We unroll f's body e once (i.e. replace recursive calls to f in e with a substituted e) and symbolically execute it (line 9), assuming f has a specification *requires* Φ_{pr} *ensures* `false` (line 8). The postcondition `false` is used to ensure that the execution only considers the effect of the program branches with no recursive calls (to f itself). We then use Δ to find out appropriate abstraction of post-shape (line 11), which is paired with Φ_{pr} and returned as result. Here we use pure abduction to filter post-shapes to preserve as many shapes that are potentially refinable as possible. The function pair_spec_list(Φ_{pr}, C) forms an ordered list of pre-/post-shape pairs, each of which has Φ_{pr} as pre-shape and a Φ_{po} in C as post-shape.

We illustrate our procedure to generate and confirm candidate shape abstractions (ShpCand) with an example. If we have two parameters `x` and `y` with type `node`, and two compatible shape predicates `llB` and `sllB`, then the list of all possible shape candidates for the two variables (C) will be [`sllB(x,S)` $*$ `sllB(y,T)`, `llB(x,S)` $*$ `sllB(y,T)`, `sllB(x,S)` $*$ `llB(y,T)`, `llB(x,S)` $*$ `llB(y,T)`, `sllB(x,S)`, `sllB(y,S)`, `llB(x,S)`, `llB(y,S)`, `emp`]. Elements of this list will be checked against appropriate abstract states (line 4 in Fig 8 left and line 11 in Fig 8 right) where most elements should be reduced because they are not sound abstractions. For example, in the previous list, only `llB(x,S)` $*$ `llB(y,T)` remains in the list and participates in further verification.

The initial experimental results confirm that our shape synthesis keeps only highly relevant abstractions. For the while loop in Section 2, we filtered out 24 (of 26) abstractions. Generally, in case that there are several abstractions as candidate specifications, we employ some other mechanisms to reduce them further. Firstly, we prioritise post-shapes with same (or stronger) predicates as in precondition since it is more likely that the output will have the same or similar shape predicates as the input, e.g. `x` is expected to remain as `sllB` (or stronger) if it points to `sllB` as input. Secondly, we employ a lazy scheme when refining the synthesised pre/post-shapes (to complete specifications). We retrieve (and remove) the pre/post-shape pair from the head of the list, (1) use the refinement algorithm (Fig 5) to obtain a specification for the auxiliary method, and (2) continue the analysis for the primary method. If the analysis for the primary method succeeds, we will ignore all other synthesised pre/post-shapes from the list. These mechanisms help to keep attempts over candidate specifications at a minimum level.

Soundness. Based on the soundness of the following: the entailment prover [23], the abstract semantics (w.r.t. the concrete semantics), the pure constraint abstraction generation, and the fixpoint calculation [24, 25], we have

Theorem 1 (Soundness). *Our analysis is sound with respect to the underlying operational semantics.*

The proof and more details can be found in the technical report [27].

5 Experiments and Evaluation

We have implemented a prototype system for evaluation. Our experimental results were achieved with an Intel Core 2 CPU 2.66GHz with 8Gb RAM. The four columns in Fig 9 describe, resp., the analysed programs, the analysis time in seconds, and the primary methods' (given and inferred) preconditions and postconditions. All formulae with a grey background are inferred by our analysis. For some programs, we have verified them with different pre/post shape templates. More results and details are available in the report [27].

The results highlight the refinement of both pre- and postconditions based on user-provided shape specifications, even for complicated data structures such as AVL and red-black trees. Firstly, our approach can compute non-trivial pure constraints for postcondition, e.g. for `delete` we know the content of the result list is subsumed by that of the input list, for list-sorting algorithms we confirm the content of the output is the same as that of the input, and for tree-processing programs (`insert`, `delete` and `avl_ins`), we obtain that the height difference between the input and output trees is at most one. Meanwhile, we can calculate non-trivial requirements in precondition for memory safety or functional correctness. As an example, the `travrs` method, taking in a list with length `m` and an integer `n`, traverses towards the tail of the list for `n` steps. the analysis discovers `m`$\geq$`n` in the precondition to ensure memory safety. Another example is the `append` method concatenating two sorted lists into one. To ensure that the result list is sorted, the analysis figures out that the minimum value in the second list must be no less than the maximum value in the first list.

A second highlight is our flexibility by supporting multiple predicates. Our analysis tries to refine different specifications for the same program at various correctness levels (with different predicates), e.g. `sort_insert` and `append`. For `rand_insert`, which inserts a node into a random place (after the head) of a list, we confirm that the list's length is increased by one, but cannot verify the list is kept sorted if it was before the insertion, as the result indicates.

Another highlight is that we can reduce user annotations by synthesising specifications for auxiliary methods, given raw specifications of primary methods. For example, we have analysed a number of list-sorting algorithms with at least one auxiliary method each. We list two auxiliary methods (`merge` for `merge_sort`

Prog.	Time	Pre	Post
List processing programs			
sort_insert	0.591	$\mathtt{ll(x,n)} \wedge \mathtt{n \geq 1}$	$\mathtt{ll(x,m)} \wedge \mathtt{m=n+1}$
	0.504	$\mathtt{sll(x,n,xs,xl)} \wedge \mathtt{v \geq xs}$	$\mathtt{sll(x,m,mn,mx)} \wedge \mathtt{xs=mn} \wedge \mathtt{mx=max(xl,v)} \wedge \mathtt{m=n+1}$
rand_insert	0.522	$\mathtt{ll(x,n)} \wedge \mathtt{n \geq 1}$	$\mathtt{ll(x,m)} \wedge \mathtt{m=n+1}$
	—	$\mathtt{sll(x,n,xs,xl)} \wedge$ (fail)	$\mathtt{sll(x,m,mn,mx)} \wedge$ (fail)
delete	1.024	$\mathtt{sllB(x,S)} \wedge \lvert\mathtt{S}\rvert \geq 2$	$\mathtt{sllB(x,T)} \wedge \exists \mathtt{a.S=T} \sqcup \{\mathtt{a}\}$
travrs	0.296	$\mathtt{ll(x,m)} \wedge \mathtt{n \geq 0} \wedge \mathtt{m \geq n}$	$\mathtt{ls(x,p,k)} * \mathtt{ll(res,r)} \wedge \mathtt{p=res} \wedge \mathtt{k=n} \wedge \mathtt{m=n+r}$
append	0.512	$\mathtt{ll(x,xn)} * \mathtt{ll(y,yn)} \wedge \mathtt{xn \geq 1}$	$\mathtt{ll(x,m)} \wedge \mathtt{m=xn+yn}$
	0.948	$\mathtt{sll(x,xn,xs,xl)} \wedge \mathtt{xl \leq ys} * \mathtt{sll(y,yn,ys,yl)}$	$\mathtt{sll(x,m,rs,rl)} \wedge \mathtt{yl=rl} \wedge \mathtt{m \geq 1+yn} \wedge \mathtt{m=xn+yn}$
Sorting (main)		$\mathtt{llB(x,S)} \wedge \lvert\mathtt{S}\rvert \geq 1$	$\mathtt{sllB(res,T)} \wedge \mathtt{T=S}$ $(\star)$
merge	4.107	$\mathtt{sllB(x,S_x)} * \mathtt{sllB(y,S_y)}$	$\mathtt{sllB(res,T)} \wedge \mathtt{T=S_x} \sqcup \mathtt{S_y}$
flatten	2.693	$\mathtt{bstB(x,S)}$	$\mathtt{sllB(res,T)} \wedge \mathtt{T=S}$
Binary tree, binary search tree, AVL tree and red-black tree processing programs			
insert	1.276	$\mathtt{bt(x,S,h)} \wedge \lvert\mathtt{S}\rvert \geq 1 \wedge \mathtt{h \geq 1}$	$\mathtt{bt(x,T,k)} \wedge \mathtt{T=S} \sqcup \{\mathtt{v}\} \wedge \mathtt{h \leq k \leq h+1}$
delete	0.970	$\mathtt{bt(x,S,h)} \wedge \lvert\mathtt{S}\rvert \geq 2 \wedge \mathtt{h \geq 2}$	$\mathtt{bt(x,T,k)} \wedge \exists \mathtt{a.S=T} \sqcup \{\mathtt{a}\} \wedge \mathtt{h-1 \leq k \leq h}$
search	1.583	$\mathtt{bst(x,sm,lg)}$	$\mathtt{bst(x,mn,mx)} \wedge \mathtt{sm=mn} \wedge \mathtt{lg=mx} \wedge \mathtt{0 \leq res \leq 1}$
bst_insert	1.720	$\mathtt{bst(x,sm,lg)}$	$\mathtt{bst(x,mn,mx)} \wedge (\mathtt{v<sm} \wedge \mathtt{v=mn} \wedge \mathtt{lg=mx} \vee \mathtt{lg<v} \wedge \mathtt{v=mx} \wedge \mathtt{sm=mn} \vee \mathtt{sm=mn} \wedge \mathtt{lg=mx})$
avl_ins	11.12	$\mathtt{avl(x,S,h)}$	$\mathtt{avl(res,T,k)} \wedge \mathtt{T=S} \sqcup \{\mathtt{v}\} \wedge \mathtt{h \leq k \leq h+1}$
rbt_ins	8.76	$\mathtt{rbt(x,S)}$	$\mathtt{rbt(res,T)} \wedge \mathtt{T=S} \sqcup \{\mathtt{v}\}$

Fig. 9. Selected Experimental Results.

and `flatten` for `tree_sort`) and their discovered specifications. Note that these sorting algorithms have the same specification for their primary methods (line $\star$). As another example, `avl_ins` also has some auxiliary (recursive) methods such as calculation of tree's height, which are automatically analysed as well.

We have also tried our approach over part of the FreeRTOS kernel [2]. For its list processing programs `list.h` and `list.c` (472 lines with intensive manipulation over composite sorted doubly-linked lists) it took 2.85 seconds for our prototype to refine all the specifications given for the main functions, which further confirms the viability of our approach.

6 Related Work and Conclusion

Related works. The local shape analysis [9] infers loop invariants for list-processing programs, followed by the SpaceInvader tool to verify larger industrial codes [5, 32]. Gulavani et al. [12] propose a stronger bi-abduction algorithm to compute the shape pre/post-condition at the same time. The SLAyer tool [11] implements an interprocedural shape analysis. To infer also size information, THOR [19, 20] is armed with additional numerical analysis to gain better precision. Gulwani et al. [13] combine a set domain with its cardinality domain in a general framework. Magill et al. [21] instrument programs with numerical instructions from which pure numerical programs are generated for further analysis. Compared with these, our approach can handle additional data structures with stronger invariants like sortedness, height-balanced and bag-related invariants. Relational inductive shape analysis [6] employs inductive checkers to express shape and numerical information, where they only demonstrate how to analyse a program with one particular shape. Our previous loop invariant synthesis [26] also infers strong loop invariants. Compared with them, this work is inter-procedural and addresses specification refinement with pure properties in both pre- and postconditions in two phases (for shape and pure resp.) with pure abduction.

There are also other approaches to expressing heap-based domains than separation logic. Hackett and Rugina [16] can deal with AVL-trees but is customised to handle only tree-like structures with height property. TVLA [30] can handle complicated data structure properties like sortedness. Bouajjani et al. [3] synthesise list-related invariants over infinite data domains using graph heap representation. Comparatively, separation logic based approach benefits from the frame rule and local reasoning. Meanwhile, our approach aims at full functional correctness including both quantitative and content properties of shapes.

Automated assertion discovery techniques [8, 14, 31] mainly find numerical program properties. Our work is complementary to them as we focus more on refining specifications for heap-manipulating programs. Semi-automatic approaches [17, 29] are also used to infer numerical constraints for given type templates in functional programs, where data structures are mostly immutable.

On the verification side, the HIP/SLEEK verification system [23] supports user-defined shape predicates over a combined domain. The PALE system [22] transforms constraints in the pointer assertion logic (PAL) into monadic second-order logic (MSOL) and discharge them with MONA. JML [4] uses model/ghost fields and model methods to specify/model Java program properties. Jahob [18] also verifies Java and focuses more on heap shape. Spec$^\#$ [1] is for C$^\#$ by enforcing object invariants and method specifications. Havoc [7] is another verification tool for C language about heap-allocated data structures, using a novel reachability predicate. Compared with these works, we can free users from writing whole specifications by requiring only partial specifications, and omit annotations for loops and auxiliary methods.

Concluding remarks. We have reported a new approach to program verification that accepts partial specifications of methods, and refines them by discovering missing constraints for numerical and bag properties, aiming at full functional correctness for pointer-based data structures. We further augment our approach by requiring only partial specification for primary methods. Specifications for loops and auxiliary methods can then be systematically discovered. We have built a prototype system and the initial experimental results are encouraging.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\#$ programming system: An overview. In *CASSIS*, 2004.
2. R. Barry. FreeRTOS — a free RTOS for small embedded real time systems. 2006.

3. A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant Synthesis for Programs Manipulating Lists with Unbounded Data. In *CAV*, 2010.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino. An Overview of JML Tools and Applications. STTT. 7(3):212–232. 2005.
5. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
6. B. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
7. S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, 2007.
8. P. Cousot and R. Cousot. On abstraction in software verification. In *CAV*, 2002.
9. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
10. R. Giacobazzi. Abductive analysis of modular logic programs. In *ILPS*, 1994.
11. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
12. B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis. In *SAS*, 2009.
13. S. Gulwani, T. Lev-Ami, and M. Sagiv. A Combination Framework for Tracking Partition Sizes. In *POPL*, 2009.
14. A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS09*.
15. J. Gustavsson and J. Svenningsson. Constraint abstractions. In *Programs as Data Objects II*, Denmark, May 2001.
16. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
17. M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.
18. V. Kuncak. *Modular Data Structure Verification.* PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
19. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.
20. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.
21. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, 2010.
22. A. Møller and M. Schwartzbach. The pointer assertion logic engine. *ACM SIGPLAN Notices*, 36(5):221–231, 2001.
23. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, 2007.
24. T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL — a proof assistant for higher-order logic, volume 2283 of LNCS. Springer, 2002.
25. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *Proceedings of 11th Asian Computing Science Conference*, 2006.
26. S. Qin, G. He, C. Luo, and W.-N. Chin. Loop invariant synthesis in a combined domain. In *ICFEM*, 2010.
27. S. Qin, C. Luo, W.-N. Chin, and G. He. Automatically Refining Partial Specification for Program Verification. Technical Report, Teesside University, `http://www.scm.tees.ac.uk/s.qin/papers/refine.pdf`, 2010.
28. J. Reynolds. Separation logic: a logic for shared mutable data structures. *LICS02*.
29. P. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, 2008.
30. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
31. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.
32. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.