Shape Refinement through Explicit Heap Analysis*

Dirk Beyer[1,2], Thomas A. Henzinger[3],
Grégory Théoduloz[4], and Damien Zufferey[3]

[1] Simon Fraser University, B.C., Canada
[2] University of Passau, Germany
[3] IST Austria (Institute of Science and Technology Austria)
[4] EPFL, Switzerland

Abstract. Shape analysis is a promising technique to prove program properties about recursive data structures. The challenge is to automatically determine the data-structure type, and to supply the shape analysis with the necessary information about the data structure. We present a stepwise approach to the selection of instrumentation predicates for a TVLA-based shape analysis, which takes us a step closer towards the fully automatic verification of data structures. The approach uses two techniques to guide the refinement of shape abstractions: (1) during program exploration, an explicit heap analysis collects sample instances of the heap structures, which are used to identify the data structures that are manipulated by the program; and (2) during abstraction refinement along an infeasible error path, we consider different possible heap abstractions and choose the coarsest one that eliminates the infeasible path. We have implemented this combined approach for automatic shape refinement as an extension of the software model checker BLAST. Example programs from a data-structure library that manipulate doubly-linked lists and trees were successfully verified by our tool.

1 Introduction

Proving the safety of programs that use dynamically-allocated data structures on the heap is a major challenge due to the difficulty of finding appropriate abstractions. For cases where the correctness property intimately depends on the shape of the data structure, researchers have over the last decade designed abstractions that are collectively known as shape analysis. One approach that has been particularly successful is based on the representation of heaps by three-valued logical structures [17]. The abstraction is specified by a set of predicates over nodes (unary and binary) representing core facts (e.g., points-to and field predicates) and derived facts (e.g., reachability). The latter category of predicates is called instrumentation predicates. Instrumentation predicates are crucial to control the precision of the analysis. First, they can keep track of relevant properties; second, they allow for more precise successor computations; and third, when used as abstraction predicates, they can control node summarization.

* Supported in part by the Canadian NSERC grant RGPIN 341819-07, by the SFU grant PRG 06-3, by the Swiss National Science Foundation, and by Microsoft Research through its PhD scholarship program.

In our previous work, we combined shape analysis with an automatic abstraction-refinement loop [4]. If a chosen abstraction is too coarse to prove the desired correctness property, a spurious counterexample path is identified, i.e., a path of the abstract program which witnesses a violation of the property but has no concrete counterpart. We analyzed such counterexample paths in order to determine a set of additional pointers and field predicates which, when tracked by the abstraction, remove the spurious counterexample. These core predicates are then added to the analysis, and a new attempt is made at proving the property. A main shortcoming of that work is that the refinement loop never automatically discovers the shape class (e.g., doubly-linked list, binary tree) that is suitable for proving the desired property, and it never adds new instrumentation predicates to the analysis. Consequently, programs can only be verified if all necessary shape classes and instrumentation predicates are "guessed" by the verification engineer when an abstraction is seeded. In the absence of such a correct guess, the method will iteratively track more and more core predicates, until either timing out or giving up because no more relevant predicates can be found.

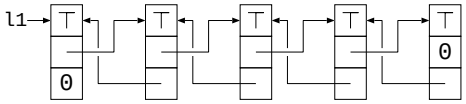
In this work, we focus on the stepwise refinement of a TVLA-based shape analysis by automatically increasing the precision of the shape classes via instrumentation predicates. Suppose that counterexample analysis (e.g., following [4]) indicates that we need to track the heap structure to which a pointer p points, in order to verify the program. We can encounter two situations: (1) we do not yet track p and we do not know to which kind of data structure p points; or (2) we already track the shape of the heap structure to which p points but the tracked shape class is too coarse and may lack some necessary instrumentation predicates. We address situation (1) by running an explicit heap analysis in order to identify the shape of the data structure from samples, and situation (2) by selecting the coarsest refinement from a lattice of plausible shape classes. Our implementation provides such plausible shape classes by default for standard data structures like lists and trees, but also supports a flexible way to extend the existing shape classes.

Example. We illustrate our method on a simple program that manipulates doubly-linked lists (cf. Fig. 1(a)). First, two (acyclic) doubly-linked lists of arbitrary length are generated (`alloc_list`); then the two lists are concatenated; finally, the program checks if the result is a valid doubly-linked list (`assert_dll`). Our algorithm automatically verifies that no assertion in this program is violated. The algorithm starts with a trivial abstraction, where no predicates are tracked, and the reachability analysis using this abstraction finds an abstract error path. The algorithm checks whether this abstract error path corresponds to a concrete error path of the program by building a path formula (i.e., a formula which is satisfiable iff the path is a concrete error path). The path formula of the first abstract error path is unsatisfiable; therefore, this is an infeasible error path (also called spurious counterexample), and the abstraction is refined using an interpolation-guided refinement process. The following atoms occur in interpolants for the first path formula: pointer equalities among `l1`, `l2`, and `p`; `l1->succ` $=$ `p`; and `l2->pred` $=$ `p`. Since the interpolants mention pointers of a recursive data structure, we need to observe them via a shape analysis tracking `l1`, `l2`, and `p` (and their aliases).

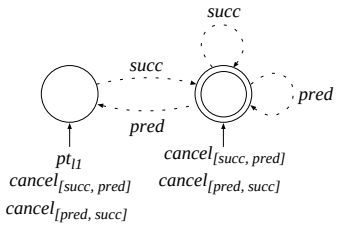
But it is not enough to know which pointers to analyze; we also need to know their data structures, in order to determine the shape abstraction (so-called shape class), because different data structures require different instrumentation predicates. Since it

```
1 typedef struct node {
2   int data;
3   struct node *succ, *prev;
4 } *List;
5 List alloc_list() {
6   List r = (List) malloc(...);
7   List p = r;
8   if (r == 0) exit(1);
9   while (*) {
10    List t = (List) malloc(...);
11    if (t == 0) exit(1);
12    p->succ = t; t->pred = p;
13    p = p->succ;
14  }
15  return r;
16 }
17 void assert_dll(List p) {
18   while ((p != 0) && (p->succ != 0)) {
19     assert(p->succ->pred == p);
20     p = p->succ;
21   }
22 }
23 void main() {
24   List l1 = alloc_list();
25   List l2 = alloc_list();
26
27   List p = l1;
28   while (p->succ != 0) p = p->succ;
29   p->succ = l2;  l2->pred = p;
30
31   assert_dll(l1);
32 }
```

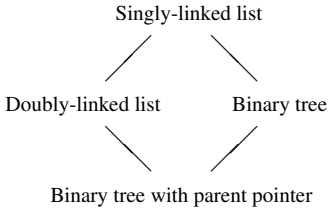
(a) Example C program



(b) Sample explicit heap



(c) Sample shape graph



(d) Example hierarchy of data structures, finer data structures are drawn lower

Fig. 1. Example program, two list abstractions, and hierarchy of data structures

is the first time we encounter this data structure, our algorithm uses an explicit heap analysis to collect explicit heap samples that would occur during program execution. We graphically illustrate an explicit heap that is collected by the explicit heap analysis in Fig. 1(b). A node (rectangle with three boxes) represents one structure element; the first box represents the integer value for the field `data`; the second and third box represent the pointer values of the fields `succ` and `prev`, respectively. An arrow represents a pointer valuation. A symbol $\top$ in a box represents an unknown value. When a threshold is hit (e.g., once we have collected explicit heaps with at least 5 nodes each), we stop the explicit heap analysis, and extract the shape class from the explicit heap samples by checking which data structure invariants they satisfy. In the example heap, all nodes satisfy the invariant for acyclic singly-linked lists for each field individually, and the invariant for doubly-linked lists (for every node n, the predecessor of the successor of n is n itself), but not the invariant for binary trees (acyclic graph formed by the two field pointers). Knowing that the data structure is not a tree, and because both fields `pred` and `succ` occur in interpolants, we restrict the search for a shape abstraction to those suitable for doubly-linked lists. We refine the shape abstraction by choosing the coarsest shape class for doubly-linked lists, i.e., in addition to points-to predicates, we track two binary predicates for the fields `pred` and `succ`, and no instrumentation predicates.

The refined abstraction is still not fine enough to prove the program safe, because we find a new abstract error path. Its path formula is unsatisfiable, but the interpolant-based analysis of the abstract error path does not yield any new predicates. Therefore, we have

to search for a finer shape class that contains instrumentation predicates as well. From the previous analysis we know that we have a doubly-linked list. We use a binary search to find, in the given lattice, the coarsest abstraction specification that eliminates the abstract error path. In our example, the tool discovers the necessity to track the unary instrumentation predicates $cancel[\texttt{succ}, \texttt{pred}]$ and $cancel[\texttt{pred}, \texttt{succ}]$ in addition to previously tracked predicates. For a node v, the predicate $cancel[f_1, f_2](v)$ holds if the following condition is fulfilled: if the field f_1 of an element represented by v points to an element represented by some node v', then the field f_2 of the element represented by v' points back to the element represented by v. After this last refinement step, the abstract reachability analysis proves that no assertion is violated. Figure 1(c) shows a shape graph that is reachable at the entry point of function `assert_dll`. A node represents a single structure element, and a summary node (drawn as a double circle) represents one or more structure elements. Unary predicate valuations are represented by arrows (or the absence of arrows) from predicates to nodes; binary predicate valuations are represented by arrows between nodes, labeled with the predicate. We can observe that the instrumentation predicates $cancel[\texttt{succ}, \texttt{pred}]$ and $cancel[\texttt{pred}, \texttt{succ}]$ have a valuation of 1 for all nodes in the data structure. Due to the information carried by those instrumentation predicates, we are able to prove the program safe.

Related Work. Counterexample-guided abstraction refinement (CEGAR) [7] is used in several predicate-abstraction based verifiers [1,3,6]. Attempts to apply CEGAR to other abstract domains exist. For instance, Gulavani and Rajamani proposed CEGAR-based widening operators in the general context of abstract interpretation [9]. Refinement of shape analysis in particular has also been studied: Loginov et al. proposed a technique to learn new instrumentation predicates from imprecise verification results [13]. In our previous work [4], we studied how to combine nullary predicate abstraction and shape analysis, and how to refine shape analysis by discovering new core predicates.

Our current work is also in the tradition of combining symbolic and explicit analyses for program verification. In particular, combinations of symbolic abstraction methods with concrete program execution (testing) to build safety proofs have received much attention recently. Such techniques have been applied in the context of predicate abstraction-based model checkers to accelerate the state construction and guide the refinement [2,8,12,18], and in the context of constraint-based invariant generation [10]. We explored in previous work the use of precision adjustment to switch between explicit and symbolic steps during a reachability analysis [5]. To the best of our knowledge, no existing technique uses explicit heaps to guide the refinement of a shape abstraction.

2 Preliminaries

2.1 Programs

In this exposition, we consider flat programs (i.e., programs with a single function). Our tool implementation supports interprocedural analysis [11, 15, 16]. We formalize programs using control-flow automata. A *control-flow automaton* (CFA) is a directed, labeled graph (L, E), where the set L of nodes represents the control locations of the program (program-counter values), and the set $E \subseteq L \times \mathit{Ops} \times L$ of edges represents the program transfers. Each edge is labeled with a program operation that can be either

an assignment or an assume predicate. The program operations are based on a set X of identifiers to identify program variables, and a set F of identifiers to identify fields. Variable identifiers and field identifiers can be either of type *integer* (denoted by `int`) or of type *pointer* to a (possibly recursive) structure (denoted by a C `struct` type). A *structure* is a set of field identifiers. We use a C-like syntax to denote program operations; in particular, p->*field* denotes the content of the field *field* in the structure pointed to by variable p. A *program* (G, l_0) consists of a CFA $G = (L, E)$ and an initial control location $l_0 \in L$. A *program path* t of length *n* is a sequence $(op_1 : l_1); \ldots; (op_n : l_n)$ of operations, such that $(l_{i-1}, op_i, l_i) \in E$ for all $1 \leq i \leq n$. A program path is *feasible* if there exists a concrete program execution with matching locations. The *verification problem* (G, l_0, l_{err}) is constituted by a program (G, l_0) and an error location l_{err}. The answer to the verification problem is SAFE if there exists no feasible path t that ends in location l_{err}, and UNSAFE otherwise. In the following two subsections we present the two abstract domains that our model-checking algorithm uses to compute an over-approximation of reachable states: explicit-heap abstraction and shape abstraction.

2.2 Explicit-Heap Abstraction

Explicit heap analysis stores concrete instances of data structures in its abstract states. Each abstract state represents an explicit, finite part of the memory. An *abstract state* $H = (v, h)$ of explicit heap analysis consists of the following two components: (1) the variable assignment $v : X \to \mathbb{Z}_\top$ is a total function that maps each variable identifier (integer or pointer variable) to an integer (representing an integer value or a structure address) or the special value $\top$ (representing the value 'unknown'); and (2) the heap assignment $h : \mathbb{Z} \rightharpoonup (F \to \mathbb{Z}_\top)$ is a partial function that maps every valid structure address to a field assignment, also called *structure cell* (memory content). A field assignment is a total function that maps each field identifier of the structure to an integer, or the special value $\top$. We call H an *explicit heap*. The initial explicit heap $H_0 = (v_0, \emptyset)$, with $v_0(x) = \top$ for every program variable x, represents all program states. Given an explicit heap H and a structure address a, the *depth* of H from a, denoted by $depth(H, a)$, is defined as the maximum length of an acyclic path whose nodes are addresses and where an edge from a_1 to a_2 exists if $h(a_1)(f) = a_2$ for some field f, starting from $v(a)$. The *depth* of H, denoted by $depth(H)$, is defined as $\max_{a \in X} depth(H, a)$.

The *explicit-heap abstraction* is a mapping $\Theta : L \to 2^X$, which assigns to each program location a subset of variables from X. Only the variables in the subset are tracked by the explicit heap analysis, i.e., the variable assignment of an abstract heap at location l maps every variable not in $\Theta(l)$ to $\top$. The abstract post operator reflects the effect of applying an operation on the explicit heap, provided it affects a data structure pointed to by a variable in the explicit-heap abstraction. Figure 1(b) graphically depicts an explicit heap (v, h) with $v = \{l1 \mapsto 1\}$ and $h = \{1 \mapsto \{data \mapsto \top, prev \mapsto 0, succ \mapsto 2\}, 2 \mapsto \{data \mapsto \top, succ \mapsto 3, prev \mapsto 1\}, 3 \mapsto \{data \mapsto \top, succ \mapsto 4, prev \mapsto 2\}, 4 \mapsto \{data \mapsto \top, succ \mapsto 5, prev \mapsto 3\}, 5 \mapsto \{data \mapsto \top, prev \mapsto 4, succ \mapsto 0\}\}$.

2.3 Shape Abstraction

Shape abstraction *symbolically* represents instances of data structures in its abstract states. We use a shape abstraction that is based on three-valued logic [17]. The notions of shape class, tracking definition, and shape-class generator are taken from *lazy shape analysis* [4]. We model the memory content by a set V of *heap nodes*. Each heap node represents one or more structure cells. Properties of the heap are encoded by predicates over nodes. The number of nodes that a predicate constrains is called the arity of the predicate, e.g., a predicate over one heap node is called *unary predicate* and a predicate over two heap nodes is called *binary predicate*. A *shape class* $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of three sets of predicates over heap nodes: (1) a set P_{core} of core predicates, (2) a set P_{instr} of instrumentation predicates with $P_{core} \cap P_{instr} = \emptyset$, where each instrumentation predicate $p \in P_{instr}$ has an associated *defining formula* φ^p over predicates, and (3) a set $P_{abs} \subseteq P_{core} \cup P_{instr}$ of abstraction predicates [17]. We denote the set of shape classes by $\mathcal{S}$. A shape class $\mathbb{S}$ *refines* a shape class $\mathbb{S}'$, written $\mathbb{S} \preccurlyeq \mathbb{S}'$, if (1) $P'_{core} \subseteq P_{core}$, (2) $P'_{instr} \subseteq P_{instr}$, and (3) $P'_{abs} \subseteq P_{abs}$. The partial order $\preccurlyeq$ induces a lattice of shape classes. We require the set P_{core} of core predicates to contain the (special) unary predicate *sm*. For a heap node v, the predicate *sm*(v) has the value $false$ if v represents exactly one structure cell, and the value $1/2$ if v represents one or more structure cells. In the latter case, the heap node is called *summary node*. In the following, we make use of the following two families of core predicates. A *points-to predicate* $pt_x(v)$ is a unary predicate that is $true$ if pointer variable x points to a structure cell that is represented by v, and $false$ otherwise. A *field predicate* $fd_\phi(v)$ is a unary predicate that is $true$ if field assertion ϕ holds for all structure cells that are represented by heap node v, and $false$ otherwise. A field assertion is a predicate over the field identifiers of a structure. Therefore, field predicates represent the data content of a structure, rather than the shape of the structure. A *shape graph* $s = (V, val)$ for a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of a set V of heap nodes and a valuation val in three-valued logic of the predicates of $\mathbb{S}$: for a predicate $p \in P_{core} \cup P_{instr}$ of arity n, $val(p) : V^n \to \{0, 1, 1/2\}$.

The *shape abstraction* is a function $\Psi : L \to 2^{\mathcal{S}}$ that maps each control location to a set of shape classes (different shape classes can be used to simultaneously track different data structures). The Ψ-abstraction, i.e., the result of applying a shape abstraction Ψ, is an abstract state, called shape region. A *shape region* $G = \{(\mathbb{S}_1, S_1), ..., (\mathbb{S}_n, S_n)\}$ consists of a set of pairs $(\mathbb{S}_i, S_i)$ where $\mathbb{S}_i$ is a shape class and S_i is a set of shape graphs for $\mathbb{S}_i$. The abstract post operator for shape graphs is defined as in TVLA [17].

Tracking definitions and shape-class generators. Instead of directly considering shape classes, we separate two aspects of shape classes. First, a tracking definition provides information about which pointers and which field predicates need to be tracked on a syntactic level. Second, given a tracking definition, a shape-class generator determines which predicates are actually added to the shape class.

A *tracking definition* $D = (T, T_s, \Phi)$ consists of (1) a set T of *tracked pointers*, which is the set of variable identifiers that may be pointing to some node in a shape graph; (2) a set $T_s \subseteq T$ of *separating pointers*, which is the set of variable identifiers for which we want the corresponding predicates (e.g., points-to, reachability) to be abstraction predicates (i.e., precisely tracked, no value $1/2$ allowed); and (3) a

set Φ of field assertions. A tracking definition $D = (T, T_s, \Phi)$ *refines* a tracking definition $D' = (T', T'_s, \Phi')$, if $T' \subseteq T$, $T'_s \subseteq T_s$ and $\Phi' \subseteq \Phi$. We denote the set of all tracking definitions by $\mathcal{D}$. The coarsest tracking definition $(\emptyset, \emptyset, \emptyset)$ is denoted by D_0.

A *shape-class generator* (SCG) is a function $m : \mathcal{D} \to \mathcal{S}$ that takes as input a tracking definition and returns a shape class, which consists of core predicates, instrumentation predicates, and abstraction predicates. While useful SCGs contain points-to and field predicates for pointers and field assertions from the tracking definition, and the predicate *sm*, other predicates need to be added by appropriate SCGs. An SCG m *refines* an SCG m' (denoted by $m \sqsubseteq m'$) if $m(D) \preccurlyeq m'(D)$ for every tracking definition D. We require that the set of SCGs contains at least the coarsest element m_0, which is a constant function that generates for each tracking definition the shape class $(\emptyset, \emptyset, \emptyset)$. Furthermore, we require each SCG to be monotonic: given an SCG m and two tracking definitions D and D', if $D \preccurlyeq D'$, then $m(D) \preccurlyeq m(D')$.

A *shape type* $\mathbb{T} = (\sigma, m, D)$ consists of a structure type σ, an SCG m, and a tracking definition D. For example, consider the type `struct node {int data; struct node *succ;};` and the tracking definition $D = (\{l1, l2\}, \{l1\}, \{data = 0\})$. To form a shape type for a singly-linked list, we can choose an SCG that takes a tracking definition $D = (T, T_s, \Phi)$ and produces a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ with the following components: the set P_{core} of core predicates contains the default unary predicate *sm* for distinguishing summary nodes, a binary predicate $succ$ for representing links between nodes in the list, a unary points-to predicate for each variable identifier in T, and a unary field predicate for each assertion in Φ. The set P_{instr} of instrumentation predicates contains for each variable identifier in T a reachability predicate. The set P_{abs} of abstraction predicates contains all core and instrumentation predicates about separating pointers from T_s. More precise shape types for singly-linked lists can be defined by providing an SCG that adds more instrumentation predicates (e.g., cyclicity).

A *shape-abstraction specification* is a function $\widehat{\Psi}$ that assigns to each control location a set of shape types. The specification $\widehat{\Psi}$ defines a shape abstraction Ψ in the following way: a pair $(l, \{\mathbb{T}_1, \ldots, \mathbb{T}_k\}) \in \widehat{\Psi}$ yields a pair $(l, \{\mathbb{S}_1, \ldots, \mathbb{S}_k\}) \in \Psi$ with $\mathbb{S}_i = \mathbb{T}_i.m(\mathbb{T}_i.D)$ for all $1 \leq i \leq k$. (We use the notation $X.y$ to denote the component y of a structure X.) Given a program P, the initial shape-abstraction specification $\widehat{\Psi}_0$ is defined as the set $\{(\sigma, m_0, D_0) \mid \sigma \text{ is a structure type occurring in } P\}$; the initial shape region G_0 consists of one pair $(\emptyset, \emptyset)$ for every shape type in $\widehat{\Psi}_0$. Region G_0 does not constrain the state space; it represents all program states.

3 Shape Analysis with Abstraction and Refinement

We introduce a new verification algorithm that is based on abstraction *and* refinement. Shape types can be refined in two different ways: either we refine the shape type's tracking definition, or we refine the shape type's SCG. In both cases, the resulting shape class is guaranteed to be finer, because SCGs are monotonic. Previous work has shown how tracking definitions can be refined, by extracting information from infeasible error paths using interpolation [4]. Our approach is based on this algorithm, and proposes a novel technique to refine SCGs, by combining information from two sources. The first

source of information is explicit heaps and is used to restrict the refinement to SCGs that are designed to support the kind of data structure (e.g., doubly-linked list, binary tree) that the program manipulates. When we discover pointers to data structures for the first time, we run an explicit heap analysis of the program until we encounter explicit heaps with a depth that exceeds a given threshold. The explicit heaps that have been computed are queried for data structure invariants, and are then abstracted to shape graphs. The second source of information is infeasible error paths. We simulate shape analysis with different SCGs along the path to determine the coarsest SCG that is able to eliminate the infeasible path. A library of SCGs that supports standard data structures like lists and trees is available in BLAST.

3.1 Model-Checking Algorithm (*ModelCheck*)

Our analysis algorithm operates on an abstract reachability tree (ART), whose nodes contain two abstract states: one abstract state models the heap memory explicitly (using explicit heaps), and the other abstract state models the heap memory symbolically (using shape graphs). Formally, an abstract reachability tree (ART) [3] is a tree that fulfills the following properties. Every node n is a tuple $n = (l, H, G)$ which consists of a control-flow location l, an explicit heap H, and a shape region G. The root node $n_0 = (l_0, H_0, G_0)$ consists of the initial control-flow location l_0, the initial explicit heap H_0, and the initial shape region G_0. An edge (n, n') in the ART means that node n' is the abstract successor of node n, i.e., the edge $((l, H, G), (l', H', G'))$ exists in the ART if l' is a successor location of l in the CFA, H' is the abstract explicit-heap successor of explicit heap H, and G' is the abstract shape successor of shape region G. A node n is covered if there exists another node n' in the ART for the same location and all concrete states represented by n are represented by n'.

Algorithm *ModelCheck* (Alg. 1) takes as input a program P, an error location l_{err} of P, and a lattice M of SCGs. The algorithm tries to prove (or disprove) that l_{err} is not reachable in any concrete program execution. It keeps track of the current abstraction, i.e., an explicit-heap abstraction and shape-abstraction specification. In addition, it maintains a mapping from program types to sets of enabled SCGs (subsets of M). Only enabled SCGs are considered during refinement. In a first step, the algorithm initializes the abstractions for each control location of the input program P with trivial abstractions. All SCGs are initially enabled, and the ART A is initialized as a tree with a single node representing the initial program states. Then a check-refine loop is executed until either the program is declared safe or a feasible path to the error location is found.

In each iteration, we first call procedure *BuildART* to extend the given ART A for the given program P and the current abstractions Θ and $\widehat{\Psi}$, towards a resulting ART that is closed under abstract successors. Procedure *BuildART* (not shown in pseudocode) takes as input a program P, an error location l_{err}, an ART A, an explicit-heap abstraction Θ, and a shape abstraction specification $\widehat{\Psi}$. If the procedure stops, it returns a pair (A, n) consisting of the ART and its last processed (leaf) node. It operates on the ART nodes and performs a waitlist-based reachability analysis to explore the abstract state space that Θ and $\widehat{\Psi}$ define. Children of nodes are computed until every leaf of the ART is covered, i.e., the ART is *complete*. The procedure stops if one of the following conditions is fulfilled: (a) The reachability analysis encounters a node n whose location

Algorithm 1 $ModelCheck(P, l_{err}, M)$

Input: a program P, an error location l_{err} of P,
a lattice M of SCGs with finite height
Output: either an ART to witness safety,
or an error path to witness the existence of a feasible error path
Variables: an explicit-heap abstraction Θ, a shape-abstraction specification $\widehat{\Psi}$, an ART A,
a mapping E from types to sets of enabled SCGs

```
for each location l of P do
  Ψ̂(l) := Ψ̂_0;   Θ(l) := ∅;
for each pointer type σ in P do
  E(σ) := M
A = {(l_0, H_0, G_0)};
while true do
  (A, n) := BuildART(P, l_err, A, Θ, Ψ̂);
  if n is not an error node then // ART A is safe, i.e., A contains no error node
    if A is complete then
      print "Yes. The program is safe. Certificate:" A; stop;
    else // threshold exceeded, switch off explicit tracking
      (A, Θ, Ψ̂, E) := Abstract(A, n, Θ, Ψ̂, M, E);
  else // n is an error node, i.e., n = (l_err, ·, ·)
    let t be the path in A from the root to n
    if PathFormula(t) is satisfiable then // t is feasible; the error is really reachable
      print "No. The program is unsafe. Counterexample path:" t; stop;
    else // t is infeasible due to a too coarse abstraction
      (A, Θ, Ψ̂, E) := Refine(A, n, Θ, Ψ̂, M, E);
```

is the error location. Then the last computed node contains the error location. (b) The reachability analysis completes the ART, i.e., all leaf nodes of the ART are covered and the ART does not contain any node with the error location — the ART is *safe*, and complete. (c) The depth of the last explicit heap that the procedure has computed exceeds a given threshold. The last computed node contains an explicit heap suitable for abstraction.

Algorithm $ModelCheck$ distinguishes the different outcomes of ***BuildART*** based on the ART properties safe and complete. (1) If the ART is safe and complete, the overall algorithm can stop and report that the program is safe. (2) If the ART is safe but not complete, then the threshold for the explicit heap analysis was reached at node n, in other words, the explicit heap analysis has collected enough information to guide the refinement of the shape-abstraction specification. Procedure $Abstract$ is called to analyze explicit heaps to restrict enabled SCGs, refine SCGs in the shape-abstraction specification, and replace explicit heaps in the ART by shape graphs. (3) If n represents an error location and the path from the root of A to n is feasible, then the overall algorithm can stop and report an error. (4) If n represents an error location but the path from the root of A to n is infeasible, then the path was encountered due to a too coarse abstraction, and procedure ***Refine*** will try to find a more suitable abstraction. Procedure ***Refine*** may fail due to the absence of a suitable, fine-enough SCG in the lattice of SCGs. Note that Algorithm ***ModelCheck*** may not terminate, in case it produces finer and finer abstractions to rule out longer and longer infeasible error paths.

Algorithm 2 $Abstract(A, n, \Theta, \widehat{\Psi}, M, E)$

Input: an ART A, an ART node n, an abstraction consisting of Θ and $\widehat{\Psi}$,
a set M of SCGs, and a type-to-SCGs mapping E
Output: an ART, an abstraction consisting of Θ and $\widehat{\Psi}$, and a type-to-SCGs mapping E

```
let n = (l, H, G)
let pointer p ∈ Θ(l) s.t. depth(H, p) > k
let σ = type(p);  choose (σ, m, D) ∈ Ψ̂(l)
// evaluate invariants on explicit heap, and update abstractions
E(σ) := E(σ) ∩ SCGsFromExplicit(H, p)
let m' be the coarsest SCG in E(σ)
replace (σ, m, D) by (σ, m', D) in Ψ̂(l)
remove all x from Θ(l) s.t. type(x) = type(p)
// remove explicit heap info and update shape graphs in ART
for each node n = (l, H, G) in A do
  n' = (l, H_0, G') with G' = HeapToShape(H, Ψ̂(l))
  replace n by n' in A
return (A, Θ, Ψ̂, E)
```

3.2 Algorithm for Abstraction from Explicit Heaps (*Abstract*)

When the explicit heap analysis has generated sufficiently large explicit heaps, Algorithm $Abstract$ (Alg. 2) is called to extract information from explicit heaps in order to choose a suitable SCG, and explicit heaps are abstracted to shape graphs. The algorithm takes as input an ART A, a leaf node n of the ART, the current abstraction specified by an explicit-heap abstraction Θ and a shape-abstraction specification $\widehat{\Psi}$, a lattice of SCGs, and a mapping E from types to sets of enabled SCGs. Upon termination, the algorithm returns the updated ART, abstraction, and mapping.

The algorithm first determines a pointer to the data structure whose depth exceeds the threshold k. Function $SCGsFromExplicit$ analyzes an explicit heap and returns all relevant SCGs: Every SCG is annotated with a set of invariants that must be fulfilled by explicit heaps for the SCG to be relevant (e.g., all SCGs generating instrumentation predicates for trees are annotated with the tree-ness invariant). For each SCG m, function $SCGsFromExplicit$ evaluates the invariants of m on explicit heap H, and if all those invariants are fulfilled, the function enables m for its structure type. Then the abstraction is updated: pointer p and all other pointers of the same type are removed from the explicit-heap abstraction, and we refine the SCG of the chosen shape type to be the coarsest enabled SCG for the structure type. After the refinement of the SCG, we erase the explicit heap in the ART node, and replace the corresponding shape region by the result of abstracting the explicit heap to shape graphs (function $HeapToShape$). The result of $HeapToShape$ has a single shape graph for each shape class that results from applying the newly refined SCG to the current tracking definitions. For example, the shape graph represented in Fig. $1(c)$ is a possible abstraction of the explicit heap represented in Fig. $1(b)$. In the next iteration of reachability, the construction of the ART continues from the newly computed shape graphs. Note that converting an explicit heap to a shape graph is significantly less expensive than obtaining the shape graph via abstract post computations, and is similar to dynamic precision adjustment [5].

Algorithm 3 $Refine(A, n, \Theta, \widehat{\Psi}, M, E)$

Input: an ART A, an ART node n, an abstraction consisting of Θ and $\widehat{\Psi}$,
a set M of SCGs, and a type-to-SCGs mapping E
Output: an ART, an abstraction consisting of Θ and $\widehat{\Psi}$, and a type-to-SCGs mapping E
Variables: an interpolant map Π

```
let t = (op_1 : l_1); ...; (op_k : l_k) be the program path from n to the root of A;
Π := ExtractInterpolants(t);
for i := 1 to k do
  choose (σ, m, D) from Ψ̂(l_i), with D = (T, T_s, P)
  // Step 1: Refine the tracking definitions
  for each atom φ ∈ Π(l_i) do
    if some pointer p occurs in φ, and type(p) matches σ then
      add p and all elements of alias(p) to D.T
      add p to D.T_s
      if pointer p is dereferenced in φ then
        add to D.P the field assertion corresponding to φ
  // Step 2: Start explicit heap analysis or refine the SCG
  for each pointer p in D.T do
    if p ∉ Θ(l_i) and m = m_0 then
      // p was not analyzed before, switch to explicit heap analysis mode
      add p to Θ(l_i)
    if p ∉ Θ(l_i) and m ≠ m_0 then
      // in shape analysis mode: binary-search refinement
      m' := FineTune(t, m, E(σ))
      if m = m' then // the binary search cannot refine; extend the search
        add to E(σ) every m'' ∈ M s.t. m ⋤ m''
        m' := FineTune(t, m, E(σ))
      replace (σ, m, D) by (σ, m', D) in Ψ̂(l_i)
    if Θ(l_i) or Ψ̂(l_i) was changed then
      remove from A all nodes with location l_i and their children
if Ψ̂ and Θ did not change then
  print "Refinement failed on path:" t; stop;
return (A, Θ, Ψ̂, E)
```

3.3 Algorithm for Shape Refinement (*Refine*)

When an infeasible error path is found in the ART, it is due to a shape abstraction that is not fine enough. Algorithm *Refine* tries to produce a finer shape abstraction such that the infeasible error path does not occur in the ART built using the refined abstraction. Algorithm *Refine* (Alg. 3) takes as input an ART A, a leaf node n of the ART, the current abstraction specified by an explicit heap abstraction Θ and a shape-abstraction specification $\widehat{\Psi}$, a lattice of SCGs, and a mapping from types to set of enabled SCGs. The algorithm assumes that the location of n is the error location and that the path from the root of A to n is infeasible. Upon termination, a refined ART, a refined abstraction, and a (possibly updated) mapping from types to set of enabled SCGs is returned.

The first step of the algorithm analyzes the infeasible error path. We compute the (inductive) interpolants of the (unsatisfiable) path formula corresponding to the path from the root to node n, for every location on the path (*ExtractInterpolants*). We use

the interpolants to check whether we can find new pointers or field assertions to track by analyzing all atoms occurring in interpolants. If we find a pointer that we have to track, we add it to the set of tracked separating pointers, and add all its aliases to the set of tracked pointers. If it is the first time we encounter a pointer, we need to know which kind of data structure it is pointing to in order to enable only a subset of SCGs. To discover this information, we cannot rely exclusively on syntactical type information. For example, the types for doubly-linked lists and binary trees (without parent pointers) have the same syntactical structure. We enable an explicit heap analysis of the data structure by adding the pointer to the abstraction of the explicit heap analysis, and the SCG is the trivial SCG m_0. If we considered the pointer before, then the explicit analysis was switched on, and we refined the SCG to a non-trivial SCG. In this case, the explicit heap analysis need not be run again because it will not provide new information. Instead, we decide to fine-tune the SCG by using a binary-search-like exploration of the lattice of enabled SCGs. If the fine-tuning fails to yield a finer SCG, it may still be the case that there exists a fine-enough SCG in the lattice of all SCGs that is prevented to be found because the explicit heap analysis over-restricted the set of enabled SCGs. In this case, we extend the set of enabled SCGs to include all SCGs from the set M of SCGs that are not coarser than the current SCG.

Procedure $FineTune$ takes as input an infeasible program path t, the current SCG m and a lattice M of SCGs. The procedure searches for the coarsest SCG m' such that m' *rules out* path t, i.e., the abstract strongest postcondition of the program path represents no states when SCG m is replaced by m' in the shape-abstraction specification. Note that we only compute shape regions along the given path t at this point, not along any other program path. To make the search more efficient, we try to prune in each iteration approximately half of the candidate SCGs. Because of the monotonicity of SCGs, if a given SCG cannot rule out t, then no coarser SCG can. The algorithm maintains a set C of candidates. The set C is initialized with all SCGs in M that are finer than m. We repeat the following steps until no more SCGs can be removed from C. We select a subset S of SCGs as small as possible such that the set of SCGs coarser than some SCG in S contains as many elements as the set of SCGs finer than some SCG in S. If no SCG in S rules out t, we remove from C all SCGs coarser or equal to a SCG in S; otherwise, we keep in C only those SCGs that are coarser or equal to some SCG in S that rules out t. When the loop terminates, if $C = \emptyset$, then the fine-tuning failed and we return m; otherwise, we choose one SCG m' in C that generates the fewest predicates when applied to the current tracking definition, and return m'.

4 Experimental Evaluation

Implementation. Our new algorithm is implemented as an extension of BLAST 3.0, which integrates TVLA for shape transformations and the FOCI library [14] for formula interpolation. In addition to the algorithm discussed in this paper, our implementation supports nullary-predicate abstraction and refinement based on interpolants.

The SCG library provided with BLAST supports singly-linked lists, doubly-linked lists, and trees with and without parent pointers. The library is based on well-known instrumentation predicates from the literature [17]: for singly-linked lists, reachability

(unary) and cyclicity (unary); for doubly-linked lists, reachability (unary and binary), cyclicity (unary), and cancellation (unary, holds for a given node when the node pointed to by the forward pointer has its backward pointer pointing to the given node); for trees (with and without parent pointers), down pointer and its transitive closure (binary), downward reachability (unary), downward acyclicity (binary), and in addition, for trees with a parent pointer, cancellation for left and right (unary, holds for a given node when the node pointed to by the left, respectively right, pointer has its parent pointer pointing to the given node).

The library of SCGs is implemented in BLAST using a domain-specific language (DSL), in order to decouple the specification of SCGs from the verification and refinement engine. Should the verification engineer need to verify a program that uses a data structure that is not yet supported in BLAST's default SCG lib, the DSL makes it easy to add support for different data structures and other instrumentation predicates. Each DSL entry corresponds to a data structure. Instead of specifying all SCGs in the lattice, the DSL entry specifies the most refined SCG, and coarser SCGs are derived by considering subsets of predicates. Moreover, a refinement relation between different data structures is specified separately.

Example Programs. We evaluate our technique on the open-source C library for data-structures GDSL 1.4 [5]. We consider non-trivial low-level functions operating on doubly-linked lists and trees. Each function is inserted in client code, non-deterministically simulating valid uses of the function. The client code inputs arbitrary valid data structures to the function, and on return, checks that a given property is preserved. The benchmarks `cancel_*` and `acyclic_*` operate on doubly-linked lists, and check, respectively, for the preservation of the structure of a doubly-linked list (i.e., the backward pointer of the node pointed to by a given node's forward pointer points back to the given node, and vice versa), and for acyclicity following forward pointers. The benchmarks `bintree_*` and `treep_*` operate on binary trees, and check, respectively, for the preservation of acyclicity following left and right pointers, and for the validity of parent pointers with respect to left and right pointers.

Results. All examples could be proved safe by BLAST after a few refinement steps. Table 1 reports the execution time of BLAST on a GNU/Linux machine with an Intel Core Duo 2 6700 and 4 GB of memory. The first part of the table reports the results with the most refined (maximal) SCGs used for all pointers in the program, and therefore no refinement is needed. The first column reports the kind of data structure and the number of instrumentation predicate families used by the SCG. The second column reports the verification time. The second part of the table reports the results when refinement is used. The first column of this part of the table reports the SCG and number of enabled instrumentation predicates families (compared to maximum). The second column reports the number of each kind of refinements: the first kind (td) corresponds to the refinement of a tracking definition (i.e., a new pointer or a new field predicate is discovered), and the second kind (scg) corresponds to the refinement of SCGs (i.e., new instrumentation predicates are introduced). The information in the first and second columns is identical for both configurations with refinement. To evaluate the impact of the explicit heap analysis on performance, we replace in one experimental setting the

[5] Available at `http://home.gna.org/gdsl/`

Table 1. Runtime of BLAST on functions from the GDSL library, using (a) maximal SCG, or shape refinement with (b) program annotations or (c) explicit heap analysis to determine the SCG

	Maximal SCG		SCG / #instr. pred.	**With refinement**		
Program	SCG	Time	families/max	#refines	Annotation	Explicit
`cancel_list_link`	dll/3	10.04 s	dll / 1/3	1 td, 1 scg	12.65 s	13.76 s
`cancel_list_insert_after`	dll/3	23.62 s	dll / 1/3	1 td, 1 scg	24.41 s	26.82 s
`cancel_list_insert_before`	dll/3	30.90 s	dll / 3/3	2 td, 2 scg	69.01 s	77.22 s
`cancel_list_remove`	dll/3	4.42 s	dll / 2/3	1 td, 1 scg	28.49 s	29.05 s
`acyclic_list_link`	dll/3	11.57 s	sll / 2/2	1 td, 1 scg	6.32 s	6.49 s
`acyclic_list_insert_after`	dll/3	24.21 s	sll / 2/2	1 td, 1 scg	23.57 s	26.06 s
`acyclic_list_insert_before`	dll/3	34.53 s	dll / 3/3	2 td, 2 scg	80.81 s	88.21 s
`acyclic_list_remove`	dll/3	4.23 s	sll / 2/2	1 td, 2 scg	96.77 s	99.75 s
`bintree_rotate_left`	tree+p/5	>9000 s	tree / 2/4	3 td, 2 scg	414.28 s	521.31 s
`bintree_rotate_right`	tree+p/5	>9000 s	tree / 2/4	3 td, 1 scg	419.24 s	437.30 s
`bintree_rotate_left_right`	tree+p/5	>9000 s	tree / 2/4	2 td, 2 scg	7023.41 s	7401.74 s
`treep_rotate_left`	tree+p/5	>9000 s	tree+p / 2/5	4 td, 2 scg	180.58 s	66.63 s
`treep_rotate_right`	tree+p/5	>9000 s	tree+p / 2/5	4 td, 2 scg	402.70 s	384.19 s
`treep_rotate_left_right`	tree+p/5	>9000 s	tree+p / 2/5	4 td, 2 scg	1175.14 s	1189.42 s

procedure $Abstract$ by a procedure that enables the suitable set of SCGs based on our knowledge of the data structures, encoded as annotations for BLAST in the code. Therefore, the third column reports verification times for the experiments when using annotations to determine the type of data structures (explicit heap analysis disabled), and the fourth column, when using the explicit heap analysis to infer the type of data structures. We run the explicit heap analysis until five different samples of data structures containing (at least) four structure nodes are collected. In all examples, both tracking definitions and SCGs are refined. In most examples, the finest SCG is not needed (only a subset of available predicates is used). Note that for three out of four `acyclic_*` benchmarks, a shape class for singly-linked lists (considering only the forward pointer) is sufficient to prove safety.

The explicit heap analysis correctly identifies the data-structure in every example. The run time for explicit-heap based refinement is comparable to annotation-guided refinement. The variations between the two result from two sources: (1) the overhead of performing the explicit heap analysis, and (2) the abstraction from explicit heaps to shape graphs and the subsequent ART extension. On all examples, the explicit heap analysis accounts for a negligible fraction of the execution time. Most of the runtime is consumed by (symbolic) shape operations in TVLA. On the one hand, some shape-graph computations are saved. But on the other hand, depending on how large the ART is when *Abstract* is executed, many explicit heaps may abstract to the same shape graph, subsequently causing an overhead. Infeasible error paths may also have different lengths resulting in different interpolation and refinement timings. On small examples, the refinement contributes most of the total execution time (up to nearly 50%): most of the time is spent in the path simulations of $FineTune$. On larger examples, most of the time is spent in the final iteration of the reachability analysis, in particular, while computing abstract shape successors using TVLA. Overall, we conclude that the explicit heap analysis provides reliable information for the refinement, for a reasonable overhead.

Our refinement strategy outperforms the direct use of the most refined SCG on large examples (involving trees), because the refinement allows for the use of significantly less instrumentation predicates, compared to the most refined SCGs. On smaller examples, though, the run time can be larger if refinement is used, due to the high portion of time spent on refinement and the high number of instrumentation predicates we need, compared to the most refined case. The final reachability analysis sometimes takes significantly less time if the most refined SCG is used; one particular case is the two `list_remove` examples. The reason is that the SCG discovered by our refinement strategy (which only tracks the forward pointer) happens to generate more different shape graphs than the most refined SCG (which tracks both pointers), although the former generates less predicates than the latter.

References

1. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
2. N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proc. ISSTA*, pages 3–14. ACM, 2008.
3. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
4. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Proc. CAV*, LNCS 4144, pages 532–546. Springer, 2006.
5. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
6. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
8. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006.
9. B. S. Gulavani and S. K. Rajamani. Counterexample-driven refinement for abstract interpretation. In *Proc. TACAS*, LNCS 3920, pages 474–488. Springer, 2006.
10. A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *Proc. ACAS*, LNCS 5505, pages 262–276. Springer, 2009.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM Press, 2004.
12. D. Kröning, A. Groce, and E. M. Clarke. Counterexample-guided abstraction refinement via program execution. In *Proc. ICFEM*, LNCS 3308, pages 224–238. Springer, 2004.
13. A. Loginov, T. W. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Proc. CAV*, LNCS 3576, pages 519–533. Springer, 2005.
14. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
15. T. W. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural data-flow analysis via graph reachability. In *Proc. POPL*, pages 49–61. ACM, 1995.
16. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Technical Report TAU-CS-26/04, Tel-Aviv University, 2004.
17. M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
18. G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: Better together! In *Proc. ISSTA*, pages 145–156. ACM, 2006.