Learning Assertions to Verify Linked-List Programs

Jan Tobias Mühlberg[1], David H. White[2], Mike Dodds[3], Gerald Lüttgen[2] and Frank Piessens[1]

[1] iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium
[2] Software Technologies Research Group, University of Bamberg, 96045 Germany
[3] Dept. of Computer Science, The University of York, YO10 5GH United Kingdom

Abstract. C programs that manipulate list-based dynamic data structures remain a challenging target for static verification. In this paper we employ the dynamic analysis of dsOli to locate and identify data structure operations in a program, and then use this information to automatically annotate that program with assertions in separation logic. These annotations comprise candidate pre/post-conditions and loop invariants suitable to statically verify memory safety with the verification tool VeriFast. By using both textbook and real-world examples on our prototype implementation, we show that the generated assertions are often discharged automatically. Even when this is not the case, candidate invariants are of great help to the verification engineer, significantly reducing the manual verification effort.

1 Introduction

Handling dynamically allocated linked-list data structures presents a major challenge in the static verification of C-like programs. Separation logic [15] has been proposed as a way to tackle this challenge. It extends Hoare logic with assertions to describe the structure of the heap and allows for local reasoning through the frame rule, which informally states that, when reasoning about the behaviour of a command, it is safe to ignore memory locations not accessed by that command.

A well-known tool that applies separation logic is *VeriFast* [12], a sound static verifier for C and Java. It modularly checks via symbolic execution [2] that each function in a program satisfies its contract, i.e., its pre- and post-condition, which are given as code annotations in separation logic. Through the frame rule, a program that passes VeriFast verification is guaranteed not to have memory safety errors such as buffer overflows or overreads, accesses to uninitialised memory, dereferences of dangling or null pointers and double frees.

The tool has been successfully applied to industrial verification projects, but it focuses on speed, expressiveness and error diagnosis rather than automation. In particular, source code annotations must be provided by a skilled verification engineer so that VeriFast can discharge contracts and invariants automatically. This limits the tool's use since it has been estimated that it takes a skilled verification engineer about one hour to provide the necessary annotations to

verify two lines of source code [16]. Many of the required annotations have little to do with the functional behaviour to be verified, but instead refer to data structures, e.g., to ensure that data structure shape is preserved and memory safety is enforced by a data structure manipulating operation.

In this paper we aim to generate these more trivial annotations for data structure manipulating code automatically, so as to reduce the burden on the verification engineer. We do this by utilising information produced as a by-product of the dynamic analysis tool dsOli (Data Structure Operation Location and Identification) [21]. dsOli combines machine learning and pattern matching to automatically locate and identify operations on linked-list data structures in C programs (Sec. 2) and outputs a set of instantiated *operation templates*, where each template describes a data structure operation performed by the program, e.g., inserting to the front of a singly-linked-list (SLL).

We provide *annotation templates* that are instantiated and injected into the program's source code by selecting appropriate information from a corresponding instantiated operation template provided by dsOli (Sec. 3). Such information is made available by a new XML-based dsOli output, which permits extraction of data structure shape transformations and the responsible source code locations. Collectively, this enables the generation of the following kinds of annotations required by VeriFast: *function contracts*, which specify data structure shape transformations and the associated memory safety properties; *recursive predicates*, which describe the recursive shape of data structures such as linked-lists; *in-line annotations*, which show where to fold and unfold recursive predicates; and *loop invariants*, which specify behavior during list traversal.

In contrast to other approaches for discovering data structure behaviour [1, 13], dsOli does not require the usage of well defined interfaces for data structure operations. Thus, our approach can detect and annotate operations even if they are tightly interwoven with other aspects of a program. Moreover, prior knowledge about the program's structure or behaviour can be used to select "interesting" execution traces for an efficient and effective analysis, while detection results may alleviate the automated exploration of related program behaviour.

We have implemented our approach in a prototypic tool-chain and use this to evaluate its utility to the verification engineer by applying it on textbook and real-world examples (Sec. 4), of which the latter comprises parts of a web-server [4] and a key-value store [17]. Overall our findings are very encouraging: our approach is able to automatically generate the vast majority of annotations required to verify the list manipulating functions of our examples. Thus, a verification engineer can spend their time on the more intellectually demanding points of verification rather than having to specify function contracts for data structure shape and the numerous required auxiliary annotations. While the generated annotations may not necessarily be discharged automatically by VeriFast, our experience shows that they still encapsulate useful a-priori knowledge about the data structure under analysis; indeed, they can often be automatically discharged after few minor manual revisions.

Related Work. Existing separation logic tools typically generate candidate invariants by shape analysis [18], such as Space Invader [23], jStar [8], Hip/Sleek [7], and SLAyer [3]. Invariants are computed by a combination of (forward) symbolic execution and abstraction at loop heads. A disadvantage of these tools is that the analysis does not scale and recovers poorly from over-abstraction. To mitigate this, recent tools have added a forwards-backwards analysis called *abduction* [6], which has been studied in the context of VeriFast [19], and counterexample elimination using external solvers [3].

Our approach, which builds upon an improved version of dsOli when compared to [21], differs from these related works in that we do not symbolically execute the program; rather we generate concrete executions and apply a heuristic machine-learning process to guess candidate invariants. Therefore, we expect our technique to increase scalability over symbolic execution when being applied to large programs or in the presence of concurrency. Improvements over the prior version of dsOli concern functional unit detection (Sec. 2.2), a new output exchange format in XML (Sec. 2.3) and template matching (Sec. 2.3); the latter has been reimplemented in Prolog, resulting in faster matching and more expressive templates. Of course, dsOli can only observe behaviour that a program exhibits when executing. An extensive set of test cases or techniques such as dynamic symbolic execution [9] may be used to expose interesting behaviour to dsOli automatically.

In Guo et al. [10], the problem of generating program invariants for data structure manipulating programs is addressed by means of static shape analysis rather than dynamic analysis and machine learning. While Guo et al. focus on generating invariants that hold for programs pruned of code that has no effect on shape properties, we produce assertions that are meant to be extended by a verification engineer with the intent to verify properties of the entire program, e.g., functional correctness. Our work may benefit from adopting the algorithm for unfolding and folding back recursive predicates presented in [10].

Active register automata learning [11] is used to determine a protocol for interaction with a data structure or API in situations where suitable example interactions may be generated. Closer to our work is specification mining [1], which generates specifications from arbitrary program executions. However, these approaches assume that interaction takes place through a well-defined interface and aim to generate a specification at that level of abstraction, representing, e.g., functional correctness. Here, lower level specifications are of interest, with the goal of proving memory safety properties in the context of VeriFast.

DDT [13] is the closest related work to dsOli and works by exploiting the coding structure in standard library implementations to identify interface functions for data structures. As such, it shares similar assumptions with [1] and is thus not designed for the customised interfaces employed in OS/legacy software and C programs, or the replicated interfaces that appear due to function in-lining. In contrast, the machine learning approach of dsOli is more tolerant of how the code implementing operations is structured (Sec. 2.2).

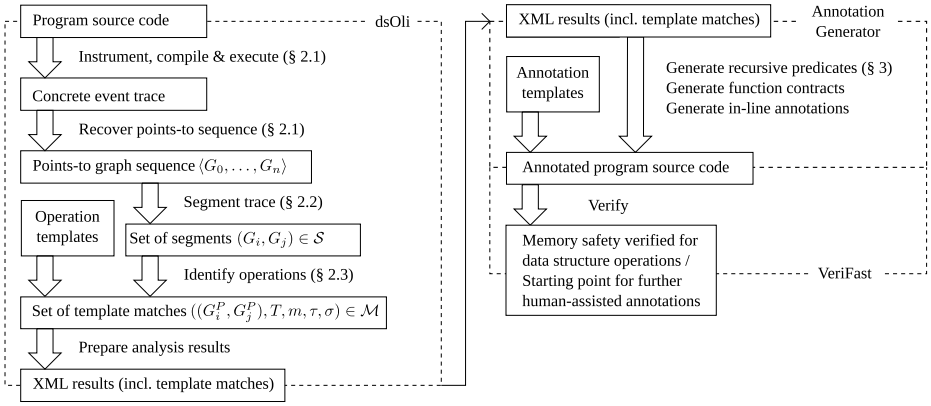

Fig. 1: An overview of our approach, which comprises dsOli, the annotation generator and VeriFast.

2 Data Structure Operation Identification

This section presents dsOli, which is responsible for discovering data structure operations in C source code. The discovered operations will be passed to our annotation generator (Sec. 3) which inserts source code annotations suitable for verifying memory safety properties of the operations. An overview of the tool chain, including the annotation generator, is given in Fig. 1. We illustrate each stage of the approach by the running example shown in Fig. 2.

2.1 Instrumentation and Preparation

We consider a dynamic data structure to be a set of *objects* (instances of C `structs`) linked by pointers. To locate and identify *operations* on data structures we reconstruct a sequence of *points-to graphs* $\langle G_0, \ldots, G_n \rangle$ from an execution of the program under analysis [21]. This reconstruction is enabled by first instrumenting the program, which results in the runtime capture of *program events* such as pointer writes and dynamic memory (de)allocation. The result of program event i is captured by G_i, where $1 \leq i \leq n$ and G_0 is empty. By default we instrument pointer writes where the unwound target is a struct with a self-reference; however, instrumentation of user-specified types is also possible.

Formally, a points-to graph $G = (\mathcal{V}, \mathcal{E})$ is a directed graph comprising a vertex set $\mathcal{V}$ and an edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathbb{N}$. Vertices in the graph represent either heap allocated objects or global/stack allocated objects that contain pointer variables, while edges represent points-to relationships. The key abstraction presented by a points-to graph is the grouping of related adjacent memory cells into a single vertex, i.e., using one vertex to represent a struct object. It is for this reason that we record the offset within an object at which the pointer originates in the third element of edge tuples. We require a pointer's target to be the start address of an object, and hence we do not record the offset into a target vertex.

Vertices are added by two means: either a dynamic memory allocation takes place, or a pointer is written in a non-dynamically allocated variable. It is necessary to include variables of the latter type since, in addition to forming part of the points-to structure, they commonly represent entry points to data structures. Every vertex v added to the graph is tagged with an attribute $v.\text{eid} = i$ recording the event i responsible for its creation, and a unique id $v.\text{cid}$ that is used to track the object represented by the vertex over multiple points-to graphs. A vertex is removed from the graph when a deallocation event occurs, or when a stack allocated variable leaves scope; the cids used for removed vertices are never reused. If applicable, a vertex has attributes $v.\text{allocLoc}$ and $v.\text{freeLoc}$ referring to the source code location responsible for the dynamic memory allocation and deallocation, respectively. Lastly, $v.\text{ctype}$ records the concrete type of the object represented by the vertex (i.e., a C type). An edge e also has an attribute $e.\text{eid}$ recording the corresponding event and, additionally, an attribute $e.\text{setLoc}$ recording the source location of the pointer write. Referring to the example of Fig. 2, G_i and G_j are points-to graphs corresponding to program states before and after function `push()` (line 21) has been invoked.

2.2 Trace Segmentation

The identification of a data structure operation is performed by analysing the change between the points-to graph *before* and *after* the operation. Therefore, the next task is to determine which segments $\langle G_{i+1}, \ldots, G_j \rangle \subseteq \langle G_1, \ldots, G_n \rangle$, where $0 \leq i < j \leq n$, of the event sequence potentially constitute operations. Later, the set of segments $\mathcal{S}$ specified in terms of points-to graph pairs (G_i, G_j) will be passed to the classification stage to identify the operations. In our running example of Fig. 2, the segment (G_i, G_j) captures the behavior of `push()`.

If dsOli operates in user-assisted mode, the user may manually mark the start and end of data structure operations and have this information used to compute the segments. Alternatively, if the approach operates on the assumption that functions will always perfectly encapsulate data structure operations, then the start and end of functions can be used to compute the segments. Clearly, this will include segments that do not correspond to data structure operations, but these will be filtered later by the classification stage.

The final and most interesting operation location approach alleviates the function encapsulation assumption, i.e., data structure operations may appear anywhere in the program, e.g., in multiple locations due to in-lining or through ad-hoc implementations commonly used for low-level optimisation, e.g., device driver software. To identify such operations we employ the observation that programs are, by nature, highly repetitive due to function calls and iterative structures. We exploit this property to identify the *functional units* of a program by their repeated invocation. The key idea is that, although the concrete addresses being operated on are different in each invocation, the points-to topology around those addresses and the sequence of changes remains similar, and hence recognisable. More details on this approach may be found in [21].

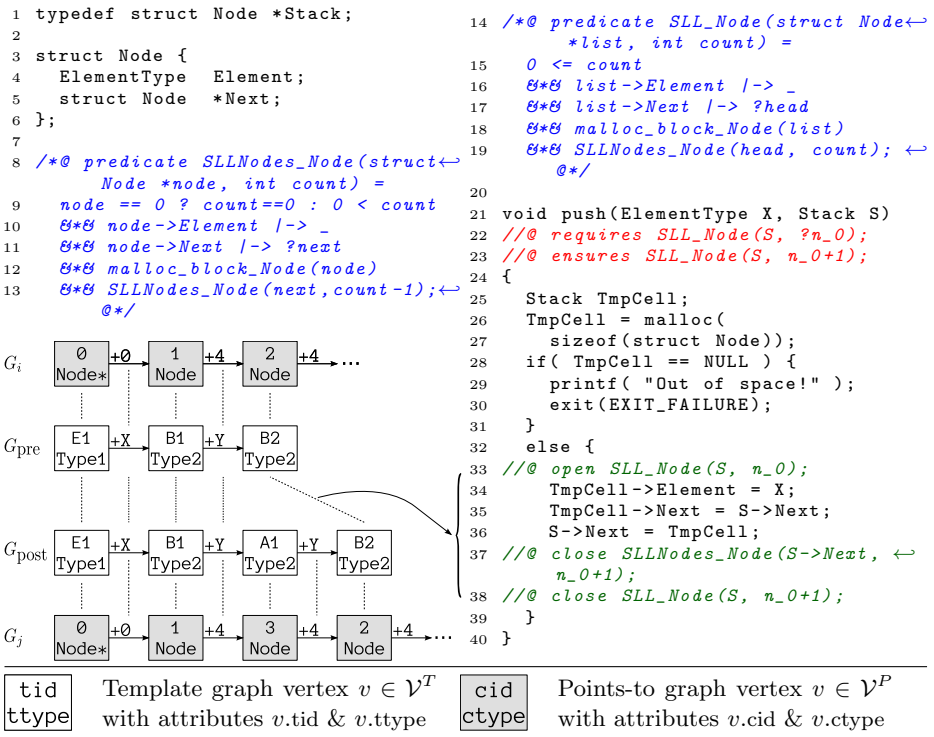

Fig. 2: An example of the annotation process for the `push()` operation from the Weiss Stack Example [20], which employs an SLL with a header node. The left drawing shows a template being matched to an invocation of `push()`. Annotations in *italics* are constructed automatically from this match as follows: blue (i.e., lines 8–19) recursive predicates, red (i.e., lines 22 & 23) function contracts and green (i.e., lines 33, 37 & 38) inline annotations.

2.3 Classifying Data Structure Operations

With the set $\mathcal{S}$ of segments to hand we may now proceed to classify the behaviour observed during a segment. The expected behaviour for each data structure operation of interest is specified via a manually defined operation template. Templates for standard data structure operations on lists are included in dsOli by default, but the user can easily add further templates by specifying them in an XML syntax. For each segment $S \in \mathcal{S}$, a match of each operation template is attempted and is considered a success if a suitable instantiation of the template's elements can be found. Successful instantiations are output in XML format to be used as input for our annotation generator (Fig. 1). If no match is possible for a segment, then it is ignored as "noise"; such segments either result from non-user assisted functional unit identification, where the fact that many segments will not correspond to data structure operations is an expected artifact of dsOli's machine learning approach, or incomplete template coverage, in which case additional templates may be specified by the user.

Table 1: Operation template attributes exposed to external programs for interpreting the associated memory transformation. Example values are taken from the template $(G_{\text{pre}}, G_{\text{post}})$ in Fig. 2.

T.dataStructureKind $\in \{\text{SLL}, \text{DLL}\}$	Example: SLL
– describes the kind of data structure that the template is intended to identify.	
T.manipulationKind $\in \{\text{Insert}, \text{Remove}\}$	Example: Insert
– determines if the template is designed to identify a node being inserted to or removed from the data structure.	
T.manipulationPosition $\in \{\text{FrontDH}, \text{Front}, \text{Middle}, \text{End}\}$	Example: FrontDH
– describes the position at which a node is inserted/removed. DH indicates a dummy-head node, so that the 2nd element in the list is semantically the front.	
T.dataStructureNodeType $\in \{v.\text{ttype} : v \in (\mathcal{V}^T_{\text{pre}} \cup \mathcal{V}^T_{\text{post}})\}$	Example: Type2
– the abstract type name for all data structure nodes, which will be mapped to a concrete C struct type after matching is performed.	
T.stableVertices $\subset \{v.\text{tid} : v \in (\mathcal{V}^T_{\text{pre}} \cup \mathcal{V}^T_{\text{post}})\}$	Example: $\{\text{E1}, \text{B1}, \text{B2}\}$
– the set of template vertex "tid"s that represent data structure nodes that remain unchanged by the operation. These sufficiently define the neighborhood around the vertex to be inserted/removed such that we may recognise the operation.	
T.differenceVertex $\in \{v.\text{tid} : v \in (\mathcal{V}^T_{\text{pre}} \cup \mathcal{V}^T_{\text{post}})\}$	Example: A1
– the template vertex tid that represents the data structure node that is added or removed.	
T.linkageOffset $\subset \{o : (v, w, o) \in (\mathcal{E}^T_{\text{pre}} \cup \mathcal{E}^T_{\text{post}})\}$	Example: $\{\text{Y}\}$
– the set of offsets for pointer(s) that link data structure nodes.	

Operation Templates. An operation template $T = (G^T_{\text{pre}}, G^T_{\text{post}})$ is defined by a pair of graphs that describe the local topological change indicative of the template attribute T.operationName. In the following, we use superscripts P and T to distinguish graphs, vertices and edges describing concrete points-to graphs and template graphs, respectively. To enable automated interpretation of this topological change, as performed in Sec. 3, we expose additional attributes concerning the template's intended usage (Table 1). For automation to be successful, we must constrain our expectation of a linked-list: we define a linked-list to be a series of nodes all of type *DS node type* and connected by pointers that always originate from a node at the same *linkage offset*, or the same offsets in the case of DLLs. Currently we only consider operations that insert or remove one node to or from the list; in other cases there is either nothing to verify as the shape does not change, or there are multiple insertions/removals which are viewed as a series of single node changes.

A match against a segment $(G^P_i, G^P_j) \in \mathcal{S}$ is performed as follows: G^T_{pre} is matched on the points-to graph before the segment starts, i.e., G^P_i, while G^T_{post} is matched after the segment on G^P_j. An attribute T.overrides lists templates less specific than T, and this means that if T is matched then it overrides the match of any template $T' \in T$.overrides. This is necessary to exclude, e.g., an SLL template matching part of a doubly-linked-list (DLL). The attribute T.templateName uniquely identifies a template as multiple templates may recognise the same op-

eration in different contexts, e.g., differentiating between inserting to the front of an empty or a non-empty list.

Each template vertex v^t has an attribute $v^t.\text{tid}$ that describes equivalence between vertices, i.e., if a vertex v in G_{pre} and a v' in G_{post} have the same tid, then v and v' must be matched to the same object in the points-to graphs. Correspondingly, the element o of some template edge (v, w, o) describes equivalence between offsets and allows one to specify that a set of pointers should all originate from their respective vertex at the same offset. Lastly, each v^t has an abstract type $v^t.\text{ttype}$, which allows vertex matches to be constrained based on C types. The graphs $(G^T_{\text{pre}}, G^T_{\text{post}})$ in Fig. 2 show a template capable of recognising inserts to the front of an SLL with a dummy-head node. The mapping between G^T_{pre} and G^T_{post} enforced via tids is displayed by dotted lines.

Operation Template Matching. An operation template match is performed by computing match functions m, τ and σ described below. If a solution to all functions can be found such that the predicates below are satisfied, then the template T is considered matched and is recorded as $((G^P_i, G^P_j), T, m, \tau, \sigma)$ in a set $\mathcal{M}$ used in Alg. 1 in Sec. 3. The match is phrased as a Prolog program, and thus we are instantiating a template's free variables, i.e., the vertices, edges and abstract types from G^T_{pre} and G^T_{post} with concrete values from the segment's points-to graph pair $(G^P_i, G^P_j) \in \mathcal{S}$:

$$G^T_{\text{pre}} = (\mathcal{V}^T_{\text{pre}}, \mathcal{E}^T_{\text{pre}}),\ \ G^T_{\text{post}} = (\mathcal{V}^T_{\text{post}}, \mathcal{E}^T_{\text{post}}) \tag{1}$$

$$G^P_i = (\mathcal{V}^P_i, \mathcal{E}^P_i),\ \ G^P_j = (\mathcal{V}^P_j, \mathcal{E}^P_j) \tag{2}$$

$$m : \{v.\text{tid} : v \in (\mathcal{V}^T_{\text{pre}} \cup \mathcal{V}^T_{\text{post}})\} \to \{v.\text{cid} : v \in (\mathcal{V}^P_i \cup \mathcal{V}^P_j)\} \tag{3}$$

To formalise the matching process, let the template and points-to graphs be written as in (1) and (2). The injective function m (3) then specifies a match from the set of template vertex tids to a subset of points-to vertex cids. Additionally, the injective functions τ, from template types to concrete types, and σ, from template offsets to concrete offsets, enforce consistency over types and offsets, respectively. We require that every template edge is mapped to a suitable points-to edge and that this mapping respects σ. This must be checked for both template graphs, i.e., for $(\mathcal{E}^T, \mathcal{E}^P) \in \{(\mathcal{E}^T_{\text{pre}}, \mathcal{E}^P_i), (\mathcal{E}^T_{\text{post}}, \mathcal{E}^P_j)\}$:

$$\forall (v^t, w^t, o^t) \in \mathcal{E}^T\ \ \exists (v^p, w^p, o^p) \in \mathcal{E}^P :$$
$$m(v^t.\text{tid}) = v^p.\text{cid}\ \wedge\ m(w^t.\text{tid}) = w^p.\text{cid}\ \wedge\ \sigma(o^t) = o^p$$

Note that, since m is injective and each vertex has a unique tid or cid, each template vertex must be matched to a corresponding points-to vertex. Lastly, we must ensure that all vertices mapped by m respect τ:

$$\forall (v^t, v^p) \in (\mathcal{V}^T_{\text{pre}} \times \mathcal{V}^P_i) \cup (\mathcal{V}^T_{\text{post}} \times \mathcal{V}^P_j) : m(v^t.\text{tid}) = v^p.\text{cid} \Rightarrow \tau(v^t.\text{ttype}) = v^p.\text{ctype}$$

An example match is shown in Fig. 2, where m and σ are indicated by dashed lines between graph vertices, $\tau = \{(\text{Type1}, \texttt{struct Node *}), (\text{Type2}, \texttt{struct Node})\}$ and $\sigma = \{(X, 0), (Y, 4)\}$.

3 Annotation Generation

In this section we discuss our annotation generation approach which is motivated by our goal of generating *function contracts* for the data structure operations discovered by dsOli. In order to fully specify such contracts we will need to generate *recursive predicates*, i.e., predicates that describe the recursive nature of a linked-list's shape. Further, to automate verification, we generate *inline annotations* that specify where to fold and unfold the recursive predicates, and additionally generate loop invariants that encapsulate behavior during traversals.

The essence of our approach is to take an instantiated *operation template*, as presented in Sec. 2.3, and use this to instantiate a number of *annotation templates* which we provide for each operation template in our template library. XML is used as the interchange format between the tools; however, for brevity we gloss over this and continue employing the mathematical notation introduced in Sec. 2. By summarising over the output of dsOli, it is possible to specify the annotation generation for any linked-list operation template; thus, this summarisation removes the necessity to define a one-to-one correspondence between operation templates and annotation templates. Typically, this process reduces elements of an operation template instantiation to their corresponding source code locations, or interprets the elements in terms of the template attributes given in Table 1. For example, the structural change described by a template is summarised by the attributes $T.\text{dataStructureKind}$, $T.\text{manipulationKind}$ and $T.\text{manipulationPosition}$.

We now present the essence of our algorithm that generates and injects annotations into the source code of the program under analysis (Alg. 1.I and 1.II), beginning with the generation of recursive predicates. Our algorithm relies on a few functions that are not presented in detail: ANNOTATE inserts VeriFast annotations into a C file at a source location determined by the helper functions BEFORE, AFTER and atFUNCDEF while dfTRACE performs an intra-procedural reaching definition analysis on a C file for a given program variable.

Recursive Predicates. Recall that function contracts for data structure manipulating functions employ recursive predicates to describe data structure shape. Each operation template match found by dsOli provides information about a particular usage of a `struct` type in the program. As shown in Alg. 1.I, aggregating information from template attributes $T.\text{dataStructureNodeType}$ and $T.\text{linkageOffset}$ with that of τ and σ allows us to construct recursive predicates. These describe linked-list data structures by making explicit, e.g., which `struct` field(s) represent linkage(s) in a list and what form the head and tail elements have. We then complete the predicate definition by adding further field names from the C source code, which function as placeholders so that a verification engineer may extend the annotations to model further aspects of the implementation. To the right of the vertical bar in lines 7 and 9 of Alg. 1.I we show the annotation templates predSLLNodes and predSLLDH for an SLL with a fixed head element; instantiations, highlighted with a grey background, are shown for our running example. Here, `SLLNodes` recursively defines the list, while `SLL` represents a handle for that list. We currently provide such predicate annotation templates for

Algorithm 1 Part I: Recursive predicates

1: generateRecursivePredicates($T, \tau, \sigma, \mathcal{M}$)
2: **switch** T.dataStructureKind ▷ Attributes of T are given in Table 1
3: **case** SLL:
4: **let** $t = \tau(T.\text{dataStructureNodeType})$
5: **let** $o = \sigma(T.\text{linkageOffset})$
6: **let** $f =$ getFieldName(t, o)
7: annotate(after(definitionOf(t)), predSLLNodes, t, f)

```
predicate SLLNodes_Node(struct Node *node, int count) =
  node == 0 ? count == 0 : 0 < count
  &*& node->Next |-> ?next
  // &*& other field chunks...
  &*& malloc_block_Node(node)
  &*& SLLNodes_Node(next, count-1);
```

8: **if** $\exists\,(\_, T', \_, \tau', \sigma') \in \mathcal{M}\ :\ T'.\text{dataStructureKind} = \text{SLL} \wedge \tau'(T'.\text{dataStructureNodeType}) = t \wedge \sigma'(T'.\text{linkageOffset}) = o \wedge T'.\text{manipulationPosition} = \text{FrontDH}$
9: annotate(after(definitionOf(t)), predSLLDH, t, f)

```
predicate SLL_Node(struct Node *list, int count) =
  &*& list->Next |-> ?head
  // &*& other field chunks...
  &*& malloc_block_Node(list)
  &*& SLLNodes_Node(head, count);
```

else
10: annotate(after(definitionOf(t)), predSLL, t, f)
11: **case** DLL: ...

SLL and DLL data structures with and without head and tail elements. Note that `&*&` is VeriFast notation for the separating conjunction operator $*$, and `?x` introduces an existentially quantified logic variable x.

dsOli may identify multiple different access patterns for the same data structure. For example, there may be functions in a program that always access elements at the head of a list, making this head element visible, while other functions modify arbitrary elements of the same list. When generating annotations we always pick the more restrictive option, e.g., a list with a head element, if at least one operation exposes this characteristic. We expect this to lead to specifications that more accurately capture program behaviour. This specificity can be seen in line 8 of Alg. 1.I, where we check over all template matches (stored in $\mathcal{M}$) to determine the most restrictive predicate.

Function Contracts. VeriFast employs the concept of permission accounting [5]. Thus, our generated function contracts give permission to a single function, or a group of functions that jointly perform an operation, to access a list and insert or remove an element of that list. Multiple function contracts may be generated for one function, specifying that this function performs operations on multiple lists.

Algorithm 1 Part II: Function contracts and inline annotations

12: GENERATECONTRACTSANDINLINE($T, \tau, \sigma, m, \mathcal{E}_{\text{manipulated}}$)
13: **switch** T.dataStructureKind
14: **case** SLL:
15: **let** $t = \tau(T.\text{dataStructureNodeType})$
16: **let** $f =$ GETFIELDNAME$(t, \sigma(T.\text{linkageOffset}))$
17: **let** $\text{cid}_{\text{diff}} = m(T.\text{differenceVertex})$
18: **let** $\text{cids}_{\text{stable}} = \{m(\text{tid}) : \text{tid} \in T.\text{stableVertices}\}$
19: **let** $e = (v, w, o) \in \mathcal{E}_{\text{manipulated}} : o = \sigma(T.\text{linkageOffset})$
$\wedge((v.\text{cid} = \text{cid}_{\text{diff}} \wedge w.\text{cid} \in \text{cids}_{\text{stable}})$
$\vee(v.\text{cid} \in \text{cids}_{\text{stable}} \wedge w.\text{cid} = \text{cid}_{\text{diff}}))$
20: **if** $v.\text{cid} = \text{cid}_{\text{diff}}$
21: **let** $list =$ DFTRACE(GETVARIABLEONASSIGNMENTRHS(e.setLoc))
22: **else**
23: **let** $list =$ DFTRACE(GETVARIABLEONASSIGNMENTLHS(e.setLoc))
24: **if** T.manipulationKind = Insert
25: ANNOTATE(ATFUNCDEF(e.setLoc), ContractInsert, $list$, t)
```
requires SLL_Node(S, ?n_Node);
ensures SLL_Node(S, n_Node + 1);
```
26: ANNOTATE(BEFORE(e_{first}.setLoc), Open, $list$, t)
```
open SLL_Node(S, n_Node);
```
27: ANNOTATE(AFTER(e_{last}.setLoc), CloseInsert, $list$, t, f)
```
close SLLNodes_Node(S->Next, n_Node + 1);
close SLL_Node(S, n_Node + 1);
```
28: **else if** T.manipulationKind = Remove ...
29: **case** DLL: ...

We first describe the simple case, i.e., where all events that transform a data structure from T_{pre} to T_{post} are located within one function body and where there are no further operation templates that match events caused by this function. If the operation is, e.g., "insert one element into a list", we are able to specify as a pre-condition that the function requires permission to a list predicate with n elements of the type mentioned in the template match. The post-condition will be that the function returns permission to the list with $n + 1$ elements to the caller. A concrete example of each can be seen at line 25 of Alg. 1.II.

To explain Alg. 1.II we introduce the set $\mathcal{E}_{\text{manipulated}}$ that comprises the points-to edges manipulated during the operation that directly contributed to breaking apart structures observed in G^T_{pre} and forming those in G^T_{post}:

$$\mathcal{E}_{\text{manipulated}} = \{(v^p, w^p, o^p) \in \mathcal{E}^P_k : k \in (i..j] \wedge (v^p, w^p, o^p).\text{eid} \in (i..j] \\ \wedge \exists (v^t, w^t, o^t) \in (\mathcal{E}^T_{\text{pre}} \cup \mathcal{E}^T_{\text{post}}) : m(v^t.\text{tid}) = v^p.\text{cid} \wedge \sigma(o^t) = o^p\}$$

i.e., points-to edges created during the segment, where the source vertices and offsets of those pointers map to template edges in either G^T_{pre} or G^T_{post}.

The set $\mathcal{E}_{\text{manipulated}}$ allows us to determine an entry point to the linked-list data structure manipulated by an operation. It relies on the computation at line 19, which locates a stable vertex w that has either an incoming or outgoing

pointer e at the linkage offset to the difference vertex v. As our analysis requires the source code to contain no more than one assignment statement per line of code, we may employ $e.\text{setLoc}$ to determine the location of that pointer write, i.e., the location of the program variable that establishes a points-to relationship between the difference vertex v and some stable vertex w (see Table 1 for details on stable and difference vertices). In lines 20 to 23 we perform a reaching definition analysis to determine the function inputs on which the program variable referring to w is data dependent. There should be one such input variable, either a function parameter or a global variable, that is of the type associated with the `SLL` predicate and contains an `SLLNodes` predicate for v. We assume this input variable to be the entry point to the list that is manipulated by the operation matched. Finally, at line 25, we insert the annotation template ContractInsert, with instantiations shown for `push()` from our running example.

Situations in which an operation spans multiple functions or is interleaved with another operation, are handled by generating contracts that capture the requirements and results of the separate event sequences that comprise a match. A typical example for this would be that the (de)allocation site of the difference vertex is located outside of the function that performs the insert or remove operation on the list. In that case, permissions for the detached node are appended to the contract so as to pass these permissions to the (de)allocation site.

Inline Annotations. Inline annotations such as loop invariants and `open`/`close` statements make transformations on VeriFast's symbolic heap explicit and, thus, provide the proof steps and invariants necessary to automate verification. Alg. 1.II produces inline annotations at lines 26 and 27 for annotation templates Open and CloseInsert. As before, instantiations are shown for our running example; also consult lines 33, 37 and 38 of Fig. 2 to view these in the context of `push()`. By consulting the elements of $\mathcal{E}_{\text{manipulated}}$ that occurred first and last, $e_{\text{first}}, e_{\text{last}} \in \mathcal{E}_{\text{manipulated}}$ with minimum or maximum value of $e.\text{eid}$ respectively, it is possible to determine the most tightly enclosing source lines at which the operation begins and ends. In the case of traversals we generate auxiliary lemmas that can be used to segment the list and re-join the segments in subsequent loop iterations. These are automatically produced from a special type of operation template, which are designed to recognise the memory transformation associated with one iteration of common list traversal implementations.

4 Evaluation

We implemented our approach in a prototypic tool-chain that takes as input C program source files and outputs annotated C source files, which are then passed to VeriFast. The annotation generator is based on LLVM/Clang [14] for parsing and annotating the input program and performing data-flow analyses. Our tool-chain was applied to two examples from textbooks, which we reuse from [21], and two examples from real-world open-source projects. We provide the output of dsOli and the automatically generated annotations for each benchmark program at `http://people.cs.kuleuven.be/~jantobias.muehlberg/sefm15/`.

The textbook examples, Weiss Stack [20] and Wolf Queue [22], employ SLLs with a head element. The key difference is that, in the former example, nodes are always appended and removed at the head position, while the latter example involves list traversal and insertion at the tail; hence, this later example includes an auto-generated loop invariant. Our results for these examples are very encouraging as all employed data structure manipulating functions could be verified by VeriFast based on our automatically generated annotations with very few minor modifications. The generated annotations of Weiss Stack required only one minor edit (moving a valid `open` statement by one line). In Wolf Queue, changes were necessary to correct a variable name in an annotation ("major edit") and to introduce new `open` and `close` annotations ("added/removed" in Table 2).

For the following two real-world examples, we sliced away code not relating to functions that were labelled by dsOli to be part of data structure operations, since currently VeriFast must verify all source code in the source file; an upcoming release will alleviate this requirement. As a first real-world example, we extracted a part of the hash table implementation from the *Redis* key-value store [17] (`dictAddRaw()` from `src/dict.c`). This component inserts a new key value into a hash bucket, represented by an SLL. The generated annotations reflect the use of the list, yet additional annotations were required to capture accessing the nested structs and arrays that contain the hash buckets. Our second example originates from the *Boa* webserver [4]. The analysed component stores requests in a DLL (`src/queue.c`), of which we verify the `enqueue` and `dequeue` functions. The latter is challenging as an arbitrary element, passed via a pointer, is to be removed from the list. Since our operation templates are based on local changes, sometimes this prevents an association between the removed element and the list head from being recognised. Nevertheless, the generated annotations are valid, but they required us to manually supply assertions to make some linkages explicit.

Table 2 summarises our results for functions in the examples that manipulate data structures only. We distinguish between the total amount of annotations required to verify a function (including those covering, e.g., field initialisation or input validation) vs. their subset that specifies data structure manipulations only (i.e., those that are in-scope of our analysis). Annotations are quantified in terms of separating conjuncts, which loosely correspond to lines of annotations as given in [16]. We also provide an estimate t_{fix} for the time required to correct the auto-generated annotations.

The runtime of our annotation generator is no more than a few seconds for all examples. As dsOli remains a prototype tool, its runtime is in the order of tens-of-minutes and requires a few GBs of RAM; since these factors depend on trace length and average points-to graph size, shorter, more representative traces can significantly reduce the requirements. The repetition-based functional unit identification strategy was employed for the textbook examples, while the real-world examples assume that functions perfectly encapsulate operations (Sec. 2.2).

Overall our findings are very encouraging, showing that our tool-chain automatically generates the majority of annotations required to verify the list manip-

Table 2: Annotation results for four sample programs

Example	LOC	Numbers of Annotations (given in terms of separation conjuncts)						t_{fix} in min
		Annot. req. for verificat.	Annot. for DS Manipul.	Auto-generated	Minor Revision Required	Major Revision Required	Added/ Removed	
Weiss Stack	**36**	**25**	**25**	**25**	**1**	**0**	**0**	2
→ Predicates	7	14	14	14	0	0	0	
→ `push()`	14	6	6	6	0	0	0	
→ `pop()`	15	5	5	5	1	0	0	
Wolf Queue	**40**	**107**	**102**	**99**	**2**	**1**	**3**	15
→ Predicates	7	65	65	65	0	0	0	
→ `get()`	14	12	7	7	1	0	0	
→ `put()`	19	30	30	27	1	1	3	
Redis	**31**	**54**	**22**	**21**	**1**	**0**	**1**	15
→ Predicates	10	28	16	16	0	0	0	
→ `dictAddRaw()`	21	26	6	5	1	0	1	
Boa	**29**	**45**	**45**	**28**	**0**	**0**	**17**	60
→ Predicates	6	9	9	9	0	0	0	
→ `enqueue()`	9	13	13	8	0	0	5	
→ `dequeue()`	14	23	23	11	0	0	12	

ulating functions of our examples with the need of few manual revisions. To assess the potential benefit of our approach for a verification engineer, Philippaerts et. al. [16] reports that the typical annotation overhead for VeriFast varies between 0.69 and 2.5 lines of annotation per line of code, and a verification engineer will verify an average of 2.17 lines of C/low-level Java code per hour. Based on this data we can conclude that our approach has the potential to save a verification engineer significant time. For our simple, albeit realistic examples we estimate time savings between 50% and 80%; our observation is that the auto-generated annotations form a skeleton that can be enriched by a verification engineer to verify functional aspects of a program, such as the ordering of list elements.

5 Conclusions and Future Work

By employing the output of dsOli's dynamic analysis based on machine learning and pattern recognition, we showed that it is possible to automatically generate many candidate annotations for the static verification tool VeriFast, which are suitable for the automated verification of operations that manipulate list-based data structures. We observed very promising initial results for verifying memory safety properties and mainly require manual input from the verification engineer for control paths not affecting data structures, which are out of scope for our analysis. In future work we aim to support a greater variety of data structures, including nested data structures that the next version of dsOli will address.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven. The second and fourth authors are supported by DFG grants LU 1748/4-1 and LU 1748/2-1. The third author is supported by DFG grant LU 1748/2-1.

References

1. Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *POPL 2002*, pp. 4–16. ACM, 2002.
2. Berdine, J., Calcagno, C., and O'Hearn, P. Symbolic execution with separation logic. In *APLAS 2005*, vol. 3780 of *LNCS*, pp. 52–68. Springer, 2005.
3. Berdine, J., Cook, B., and Ishtiaq, S. SLAyer: Memory safety for systems-level code. In *CAV 2011*, vol. 6806 of *LNCS*, pp. 178–183. Springer, 2011.
4. The Boa webserver. `http://www.boa.org/`. Last accessed 2015-06-09.
5. Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. Permission accounting in separation logic. In *POPL 2005*, pp. 259–270. ACM, 2005.
6. Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. Compositional shape analysis by means of bi-abduction. *SIGPLAN Notices*, 44(1):289–300, 2009.
7. Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.
8. Distefano, D. and Parkinson, M. J. jStar: Towards practical verification for Java. In *OOPSLA 2008*, pp. 213–226. ACM, 2008.
9. Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *PLDI 2005*, pp. 213–223. ACM, 2005.
10. Guo, B., Vachharajani, N., and August, D. I. Shape analysis with inductive recursion synthesis. In *PLDI 2007*, pp. 256–265. ACM, 2007.
11. Isberner, M., Howar, F., and Steffen, B. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM 2011*, vol. 6617 of *LNCS*, pp. 41–55. Springer, 2011.
13. Jung, C. and Clark, N. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO 2009*, pp. 56–66. ACM, 2009.
14. Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis transformation. In *CGO 2004*, pp. 75–86. IEEE, 2004.
15. O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *CSL 2001*, vol. 2142 of *LNCS*, pp. 1–19. Springer, 2001.
16. Philippaerts, P., Mühlberg, J. T., Penninckx, W., Smans, J., Jacobs, B., and Piessens, F. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014.
17. The Redis key-value store. `http://www.redis.io/`. Last accessed 2015-06-09.
18. Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
19. Vogels, F., Jacobs, B., Piessens, F., and Smans, J. Annotation inference for separation logic based verifiers. In *FMOODS 2011*, vol. 6722 of *LNCS*, pp. 319–333. Springer, 2011.
20. Weiss, M. A. *Data structures and algorithm analysis in C (Second Edition)*. Addison-Wesley, 1997.
21. White, D. H. and Lüttgen, G. Identifying dynamic data structures by learning evolving patterns in memory. In *TACAS 2013*, vol. 7795 of *LNCS*, pp. 354–369. Springer, 2013.
22. Wolf, J. *C von A bis Z*. Galileo Computing, 2009.
23. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. Scalable shape analysis for systems code. In *CAV 2008*, vol. 5123 of *LNCS*, pp. 385–398. Springer, 2008.