Byte-Precise Verification of Low-Level List Manipulation*

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. We propose a new approach to shape analysis of programs with linked lists that use low-level memory operations. Such operations include pointer arithmetic, safe usage of invalid pointers, block operations with memory, reinterpretation of the memory contents, address alignment, etc. Our approach is based on a new representation of sets of heaps, which is to some degree inspired by works on separation logic with higher-order list predicates, but it is graph-based and uses a more fine-grained (byte-precise) memory model in order to support the various low-level memory operations. The approach was implemented in the Predator tool and successfully validated on multiple non-trivial case studies that are beyond the capabilities of other current fully automated shape analysis tools.

1 Introduction

Dealing with programs with pointers and dynamic linked data structures belongs among the most challenging tasks of formal analysis and verification due to a need to cope with infinite sets of reachable program configurations having the form of complex graphs. This task becomes even more complicated when considering low-level memory operations such as pointer arithmetic, safe usage of pointers with invalid targets, block operations with memory, reinterpretation of the memory contents, or address alignment. Despite the rapid progress in the area of formal program analysis and verification, fully automated approaches capable of efficiently handling sufficiently general classes of dynamic linked data structures in the form used in low-level code are still missing.

In this paper, we propose a new fully automated approach to formal verification of list manipulating programs designed to cope with all of the above mentioned low-level memory operations. Our approach is based on a new representation of sets of heaps, which is to some degree inspired by works on separation logic with higher-order list predicates [1], but it is graph-based and uses a much more fine-grained memory model. In particular, our memory model allows one to deal with *byte-precise* offsets of fields of objects, offsets of pointer targets, as well as object sizes. Together with the new heap representation, we propose original algorithms for all the operations needed for a use of the new representation in a fully automated shape analysis. As our experiments show, these algorithms allow our analysis to successfully handle many programs on which other state-of-the-art fully automated approaches fail (by not terminating or by producing false positives or even false negatives).

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (project MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the BUT FIT project FIT-S-12-1.

F. Logozzo and M. Fähndrich (Eds.): SAS 2013, LNCS 7935, pp. 215–237, 2013.

In particular, we represent sets of heap graphs using the so-called *symbolic memory graphs* (SMGs) with two kinds of nodes: *objects* and *values*. Objects represent allocated memory and are further divided into *regions* representing individual memory areas and *list segments* encoding linked sequences of n or more regions uninterrupted by external pointers (for some $n \geq 0$). Values represent addresses and other data stored inside objects. Objects and values are linked by two kinds of edges: *has-value* edges from objects to values and *points-to* edges from value nodes representing addresses to objects. For efficiency reasons, we represent equal values by a single value node. We explicitly track sizes of objects, byte-precise offsets at which values are stored in them, and we allow pointers to point to objects with an arbitrary offset, i.e., a pointer can point *inside* as well as *outside* an object, not just at its beginning as in many current analyses.

We are capable of handling possibly cyclic, nested (with an arbitrary depth), and/or shared singly- as well as doubly-linked lists (for brevity, below, we concentrate on doubly-linked lists only). Our analysis can fully automatically recognise linking fields of the lists as well as the way they are possibly hierarchically nested. Moreover, the analysis can easily handle lists in the form common in system software (in particular, the Linux kernel), where list nodes are linked through the middle of them, pointer arithmetic is used to get to the beginning of the nodes, pointers iterating through such lists can sometimes safely point to unallocated memory, the forward links are pointers to structures while the backward ones are pointers to pointers to structures, etc.

To reduce the number of SMGs generated for each basic block of the analysed program, we propose a join operator working over SMGs. Our join operator is based on simultaneously traversing two SMGs while trying to merge the encountered pairs of objects and values according to a set of rules carefully tuned through many experiments to balance precision and efficiency (see Section 3.2 for details). Moreover, we use the join operator as the core of our abstraction, which is based on merging neighbouring objects (together with their sub-heaps) into list segments. This approach leads to a rather easy to understand and—according to our experiments—quite efficient abstraction algorithm. In the abstraction algorithm, the join is not applied to two distinct SMGs, but a single one, starting not from pairs of program variables, but the nodes to be merged. Further, we use our join operator as a basis for checking entailment on SMGs too (by observing which kind of pairs of objects and values are merged when joining two SMGs). In order to handle lists whose nodes *optionally* refer to some regions or sub-lists (which can make some program analyses diverge and/or produce false alarms [16]), our join and abstraction support the so-called 0/1 abstract objects.

Since on the low level, the same memory contents can be interpreted in different ways (e.g., via unions or type-casting), we incorporate into our analysis the so-called read, write, and join *reinterpretation*. In particular, we formulate general conditions on the reinterpretation operators that are needed for soundness of our analysis, and then instantiate these operators for the quite frequent case of dealing with blocks of nullified memory. Due to this, we can, e.g., efficiently handle initialization of structures with tens or hundreds of fields commonly allocated and nullified in practice through a single call of `calloc`, at the same time avoiding false alarms stemming from that some field was not explicitly nullified. Moreover, we provide a support for block operations like `memmove` or `memcpy`. Further, we extend the basic notion of SMGs to support pointers

having the form of not just a single address, but an interval of addresses. This is needed, e.g., to cope with address alignment or with list nodes that are equal up to their incoming pointers arrive with different offsets (as common, e.g., in memory allocators).

We have implemented the proposed approach in a new version of our tool Predator [7]. Predator automatically proves absence of various memory safety errors, such as invalid dereferences, invalid free operations, or memory leaks. Moreover, Predator can also provide the user with the derived shape invariants. Due to SMGs provide a rather detailed memory model, Predator produces fewer false alarms compared with other tools, and on the other hand, it can discover bugs that may be undetected by other state-of-the-art tools (as illustrated by our experimental results). In particular, Predator can discover out-of-bound dereferences (including stack smashing or buffer overflows) as well as nasty bugs resulting from dealing with overlapping blocks of memory in operations like `memcpy`. We have successfully validated the new version of Predator on a number of case studies, including various operations on lists commonly used in the Linux kernel as well as code taken directly from selected low-level critical applications (without any changes up to adding a test environment). In particular, we considered the memory allocator from the Netscape portable runtime (NSPR), used, e.g., in Firefox, and the `lvm2` logical volume manager. All of the case studies are available within the distribution of Predator. To the best of our knowledge, many of our case studies are out of what other currently existing fully automated shape analysis tools can handle.

Related Work. Many approaches to formal analysis and verification of programs with dynamic linked data structures have been proposed. They differ in their generality, level of automation, as well as the formalism on which they are based. As said already above, our approach is inspired by the fully automated approaches [1,17] based on separation logic with higher-order list predicates implemented in two well-known tools, namely, Space Invader and SLAyer [2]. Compared with them, however, we use a purely graph-based memory representation. In fact, a graph-based representation was used already in the older version of our tool Predator [7]. However, that representation was a rather straightforward graph-based encoding of separation logic formulae, which is no more the case for the representation proposed in this paper. Our new heap representation is much finer, which on one hand complicates its formalization, but on the other hand, it allows us to treat the different peculiarities of low-level memory manipulation. Moreover, somewhat surprisingly, despite our new heap representation is rather detailed, it still allowed us to propose algorithms for all the needed operations such that they are quite efficient. Indeed, the new version of Predator is much faster than the old one while at the same time producing fewer false positives. Compared with Space Invader and SLAyer, Predator based on the new memory representation and new algorithms is not only faster, but also terminates more often, avoids false positives and, in particular, is able to detect additional classes of program errors that the other tools silently ignore (as illustrated in the section on experiments).

Both Space Invader and SLAyer provide some support for pointer arithmetic, but its systematic description is (to the best of our knowledge) not available, and moreover, the support seems to be rather basic as illustrated by our experimental results. The same is the case with some other fully automated tools for verification of programs with dynamic linked data structures based on other formalisms, such as Forester [9] based

on automata. A support for pointer arithmetic in combination with separation logic appears in [5], which is, however, highly specialised for a particular kind of linked lists with variable length entries used in some memory allocators.

As for the memory model, probably the closest to our work is [11], which uses the so-called separating shape graphs. They support tracking of the size of allocated memory areas, pointers with byte-precise offsets wrt. addresses of memory regions, dealing with offset ranges, as well as multiple views on the same memory contents. A major difference is that [11] and the older work [6], on which [11] is based, use the so-called summary edges annotated by *user-supplied* data structure invariants to summarize parts of heaps of an unbounded size. This approach is more general in terms of the supported shapes of data structures but less automatic because the burden of describing the shape lies on the user. We use abstract objects (list segments) instead, which are capable of encoding various forms of hierarchically nested lists (very often used in practice) and are carefully designed to allow for *fully automatic* and *efficient* learning of the concrete forms of such lists (the concrete fields used, the way the lists are hierarchically nested, their possible cyclicity, possibly shared nodes, optional nodes, etc.). Also, the level of nesting is not fixed in advance—our list segments are labelled by an integral nesting level, which allows us to represent hierarchically nested data structures as flattened graphs. Finally, although [11] points out a need to reinterpret the memory contents upon reading/writing, the corresponding operations are not formalized there. One of our contributions is thus also a definition of read/write reinterpretation operators in a way that can be used by a fully automatic shape analysis algorithm.

A graph-based abstraction of sets of heap configurations is used in [12] too. On one hand, the representation allows one to deal even with tree-like data structures, but on the other hand, the case of doubly-linked lists is not considered. Further, the representation does not consider the low-level memory features covered by our symbolic memory graphs. Finally, the abstraction and join operations used in [12] are more aggressive and hence less precise than in our case.

The work [10], which is based on an instantiation of the TVLA framework [14], focuses on analysis of Linux-style lists, but their approach relies on an implementation-dependent way of accessing list nodes, instead of supporting pointer arithmetics, unions, and type-casts in a generic way. Finally, the work [15] provides a detailed treatment of low-level C features such as alignment, byte-order, padding, type-unsafe casts, etc. in the context of theorem proving based on separation logic. Our reinterpretation operators provide a lightweight treatment of these features designed to be used in the context of a fully automated analysis based on abstraction.

2 Symbolic Memory Graphs

We encode sets of program configurations using the so-called *symbolic memory graphs* (SMGs) together with a mapping from global (static) and local (stack) variables to nodes of the SMGs. In particular, SMGs have a form of node- and edge-labelled directed graphs. Below, we start by an informal description of SMGs, followed by their formalisation. For an illustration of the notions discussed below, we refer the reader to Fig. 1, which shows how SMGs represent cyclic Linux-style DLLs (with a head node

without any data part, other nodes including the head structure as well as custom data, and with the next/prev pointers pointing *inside* list nodes, not at their beginning). Some more examples illustrating the notion of SMGs, including its use for encoding various low-level Linux-style lists, can be found in [8].

2.1 The Intuition behind SMGs

An SMG consists of two kinds of nodes: *objects* and *values* (in Fig. 1, they are represented by boxes and circles, respectively). Objects are further divided to *regions* and (doubly-linked) *list segments* (DLSs)[1]. A region represents a contiguous area of memory allocated either statically, on the stack, or on the heap.

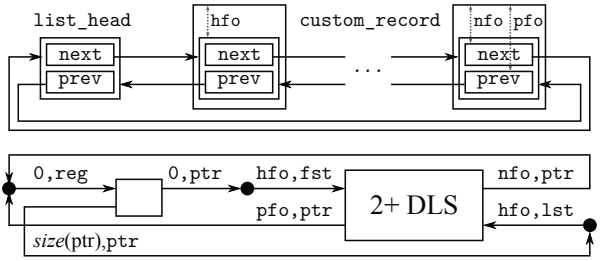

Fig. 1. A cyclic Linux-style DLL (top) and its SMG (bottom), with some SMG attributes left out for readability.

Each consistent SMG contains a special region called the *null object*, denoted #, which represents the target of `NULL`. DLSs arise from abstracting sequences of doubly-linked regions that are not interrupted by any external pointer. For example, in the lower part of Fig. 1, the left box is a region corresponding to the list head from the upper part of the figure whereas the right box is a DLS summarizing the sequence of `custom_record` objects from the upper part. Values are then used to represent *addresses* and other *data* stored in objects. All values are abstract in that we only distinguish whether they represent equal or possibly different concrete values. The only exception is the value 0 that is used to represent sequences of zero bytes of any length, which includes the zeros of all numerical types, the address of the null object, as well as nullified blocks of any size. Zero values are supported since they play a rather crucial role in C programs. In the future, a better distinction of values can be easily added.

SMGs have two kinds of *edges*: namely, *has-value edges* leading from objects to values and *points-to edges* leading from addresses to objects (cf. Fig. 1). Intuitively, the edges express that objects have values and addresses point to objects. Has-value edges are labelled by the *offset* and *type* of the *field* in which a particular value is stored within an object. Note that we allow the fields to overlap. This is used to represent different *interpretations* that a program can assign to a given memory area in order not to have to recompute them again and again. Points-to edges are labelled by an *offset* and a *target specifier*. The offset is used to express that the address from which the edge leads may, in fact, point *before, inside, or behind* an object. The target specifier is only meaningful for list segments to distinguish whether a given edge represents the address (or addresses) of the first, last, or each concrete region abstracted by the segment. The last option is

[1] Our tool Predator supports *singly-linked list segments* too. Such segments can be viewed as a restriction of DLSs, and we omit them from the description in order to simplify it.

used to encode links going to list nodes from the structures nested below them (e.g., in a DLL of DLLs, each node of the top-level list may be pointed from its nested list).

A key advantage of representing values (including addresses) as a separate kind of nodes is that a single value node is then used to represent values which are guaranteed to be equal in all concrete memory configurations encoded by a given SMG. Hence, distinguishing between *equal* values and *possibly different* values reduces to a simple identity check, not requiring a use of any prover. Thanks to identifying fields of objects by offsets (instead of using names of struct/union members), comparing their addresses for equality simplifies to checking identity of the address nodes. For example, `(x == &x->next)` holds iff `next` is the first member of the structure pointed by `x`, in which case both `x` and `&x->next` are guaranteed to be represented by a single address node in SMGs. Finally, the distinction of has-value and points-to edges saves some space since the information present on points-to edges would otherwise have to be copied multiple times for a single target.

Objects and values in SMGs are labelled by several *attributes*. First, each object is labelled by its *kind*, allowing one to distinguish regions and DLSs. Next, each object is labelled by its *size*, i.e., the amount of memory allocated for storing it. For DLSs, the size gives the size of their nodes. All objects and values have the so-called *nesting level* which is an integer specifying at which level of hierarchically nested structures the object or value appears (level 0 being the top level). All objects are further labelled by their *validity* in order to allow for safe pointer arithmetic over freed regions (which are marked invalid, but kept as long as there is some pointer to them).

Next, each DLS is labelled by the *minimum length* of the sequence of regions represented by it.[2] Further, each DLS is associated with the offsets of the *"next"* and *"prev" fields* through which the concrete regions represented by the segment are linked forward and backward. Each DLS is also associated with the so-called *head offset* at which a sub-structure called a *list head* is stored in each list node (cf. Fig. 1). The usage of list heads is common in system software. They are predefined structures, typically containing the next/prev fields used to link list nodes. When a new list is defined, its node structure contains the list head as a nested structure, its nodes are linked by pointers pointing not at their beginning but inside of them (in particular, to the list head), and pointer arithmetic is used to get to the beginning of the actual list nodes.

Global and stack *program variables* are represented by regions like heap objects and can thus be manipulated in a similar way (including their manipulation via pointers, checking for out-of-bounds accesses leading to stack smashing, etc.). Regions corresponding to program variables are tagged by their names and hence distinguishable whenever needed (e.g., when checking for invalid frees of stack/global memory, etc.).

2.2 Symbolic Memory Graphs

Let $\mathbb{B}$ be the set of Booleans, $\mathbb{T}$ a set of types, $\mathit{size}(t)$ the size of instances of a type $t \in \mathbb{T}$, $\mathtt{ptr} \in \mathbb{T}$ a unique pointer type[3], $\mathbb{K} = \{\mathtt{reg}, \mathtt{dls}\}$ the set of kinds of objects

[2] Later, in Section 4, special list segments of length 0 or 1 are mentioned too.

[3] We assume $\mathit{size}(\mathtt{ptr})$ to be a constant, which implies that separate verification runs are needed for verifying a program for target architectures using different address sizes.

(distinguishing regions and DLSs), and $\mathbb{S} = \{\texttt{fst}, \texttt{lst}, \texttt{all}, \texttt{reg}\}$ the set of points-to target specifiers. A *symbolic memory graph* is a tuple $G = (O, V, \Lambda, H, P)$ where:

- O is a finite set of objects including the special null object $\#$.
- V is a finite set of *values* such that $O \cap V = \emptyset$ and $0 \in V$.
- Λ is a tuple of the following labelling functions:
  - The kind of objects *kind* : $O \to \mathbb{K}$ where *kind*$(\#) = \texttt{reg}$, i.e., $\#$ is formally considered a region. We let $R = \{r \in O \mid \textit{kind}(r) = \texttt{reg}\}$ be the set of regions and $D = \{d \in O \mid \textit{kind}(d) = \texttt{dls}\}$ be the set of DLSs of G.
  - The nesting level of objects and values *level* : $O \cup V \to \mathbb{N}$.
  - The size of objects *size* : $O \to \mathbb{N}$.
  - The minimum length of DLSs $len : D \to \mathbb{N}$.
  - The validity of objects *valid* : $O \to \mathbb{B}$.
  - The head, next, and prev field offsets of DLSs *hfo*, *nfo*, *pfo* : $D \to \mathbb{N}$.
- H is a partial edge function $O \times \mathbb{N} \times \mathbb{T} \rightharpoonup V$ which defines *has-value edges* $o \xrightarrow{of,t} v$ where $o \in O$, $v \in V$, $of \in \mathbb{N}$, and $t \in \mathbb{T}$. We call (of, t) a *field* of the object o that stores the value v of the type t at the offset of.
- P is a partial injective edge function $V \rightharpoonup \mathbb{Z} \times \mathbb{S} \times O$ which defines *points-to edges* $v \xrightarrow{of,tg} o$ where $v \in V$, $o \in O$, $of \in \mathbb{Z}$, and $tg \in \mathbb{S}$ such that $tg = \texttt{reg}$ iff $o \in R$. Here, of is an offset wrt. the base address of o.[4] If o is a DLS, tg says whether the edge encodes pointers to the *first*, *last*, or *all* concrete regions represented by o.

We define the first node of a list segment such that the next field of the node points inside the list segment (and the last node such that the prev field of the node points inside the list segment). As already mentioned, the *all* target specifier is used in hierarchically nested list structures where each nested data structure points back to the node of the parent list below which it is nested. Fig. 2 illustrates how the target specifier affects the semantics of points-to edges (and the corresponding addresses): The DLS d is concretized to the two regions r_1 and r_2, and the nested abstract region r' to the two concrete regions r'_1 and r'_2. Note that if r' was not nested, i.e., if it had $level(r') = 0$, it would concretise into a single region pointed by both r_1 and r_2.

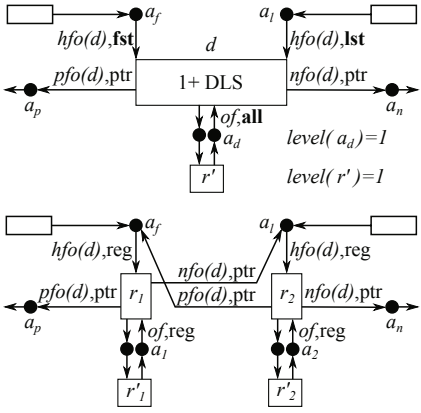

Fig. 2. An SMG and its possible concretisation for the case when the DLS d represents exactly two regions (only important attributes are shown).

Let $G = (O, V, \Lambda, H, P)$ be an SMG with a set of regions R and a set of DLSs D. We denote a DLS $d \in D$ of minimum length n, for which $\textit{len}(d) = n$, as an $n+$ DLS. We use $\perp$ to denote cases where H or P is not defined. For any $v \in V$ for which $P(v) \neq \perp$, we denote by $of(P(v))$, $tg(P(v))$, and $o(P(v))$ the particular items of the triple $P(v)$. Further, for $o \in O$, we let $H(o) = \{H(o, of, t) \mid of \in \mathbb{N},\ t \in$

[4] Note that the offset can even be negative, which happens, e.g., when traversing a Linux list.

$\mathbb{T},\ H(o, of, t) \neq \bot\}$. We let $A = \{v \in V \mid P(v) \neq \bot\}$ be the set of all *addresses* used in G. Next, a *path* in G is a sequence (of length one or more) of values and objects such that there is an edge between every two neighbouring nodes of the path. An object or value $x_2 \in O \cup V$ is *reachable* from an object or value $x_1 \in O \cup V$ iff there is a path from x_1 to x_2.

We call G *consistent* iff the following holds:

- *Basic consistency of objects.* The null object is invalid, has size and level 0, and its address is 0, i.e., $\mathit{valid}(\#) = \mathit{false}$, $\mathit{size}(\#) = \mathit{level}(\#) = 0$, and $0 \xrightarrow{0,\mathtt{reg}} \#$. All DLSs are valid, i.e., $\forall d \in D : \mathit{valid}(d)$. Invalid regions have no outgoing edges.
- *Field consistency.* Fields do not exceed boundaries of objects, i.e., $\forall o \in O\ \forall \mathit{of} \in \mathbb{N}\ \forall t \in \mathbb{T} :\ H(o, \mathit{of}, t) \neq \bot \Rightarrow \mathit{of} + \mathit{size}(t) \leq \mathit{size}(o)$.
- *DLS consistency.* Each DLS $d \in D$ has a next pointer and a prev pointer, i.e., there are addresses $a_n, a_p \in A$ s.t. $H(d, \mathit{nfo}(d), \mathtt{ptr}) = a_n$ and $H(d, \mathit{pfo}(d), \mathtt{ptr}) = a_p$ (cf. Fig. 2). The next pointer is always stored in memory before the prev pointer, i.e., the next and prev offsets are s.t. $\forall d \in D : \mathit{nfo}(d) < \mathit{pfo}(d)$. Points-to edges encoding links to the first and last node of a DLS d are always pointing to these nodes with the appropriate head offset, i.e., $\forall a \in A : \mathit{tg}(P(a)) \in \{\mathtt{fst}, \mathtt{lst}\} \Rightarrow \mathit{of}(P(a)) = \mathit{hfo}(d)$ where $d = o(P(a))$.[5] Finally, there is no cyclic path containing $0+$ DLSs (and their addresses) only in a consistent SMG since its semantics would include an address not referring to any object.
- *Nesting consistency.* Each nested object $o \in O$ of level $l = \mathit{level}(o) > 0$ has precisely one *parent DLS*, denoted $\mathit{parent}(o)$, that is of level $l - 1$ and there is a path from $\mathit{parent}(o)$ to o whose inner nodes are of level l and higher only (e.g., in Fig. 2, d is the parent of r'). Addresses with `fst`, `lst`, and `reg` targets are always of the same level as the object they refer to (as is the case for a_f, a_l, a_1, a_2 in Fig. 2), i.e., $\forall a \in A : \mathit{tg}(P(a)) \in \{\mathtt{fst}, \mathtt{lst}, \mathtt{reg}\} \Rightarrow \mathit{level}(a) = \mathit{level}(o(P(a)))$. On the other hand, addresses with the `all` target go up one level in the nesting hierarchy, i.e., $\forall a \in A : \mathit{tg}(P(a)) = \mathtt{all} \Rightarrow \mathit{level}(a) = \mathit{level}(o(P(a))) + 1$ (cf. a_d in Fig. 2). Finally, edges representing back-pointers to all nodes of a list segment can only lead from objects (transitively) nested below that segment (e.g., in Fig. 2, such an edge leads from region r' back to the DLS d, but it cannot lead from any other regions). Formally, for any $o, o' \in O$, $a \in H(o)$, $o(P(a)) = o'$, and $\mathit{level}(o) > \mathit{level}(o')$, $\mathit{tg}(P(a)) = \mathtt{all}$ iff $o' = \mathit{parent}^k(o)$ for some $k \geq 1$.

From now on, we assume working with consistent SMGs only. Let *GVar* be a finite set of global variables, *SVar* a countable set of stack variables such that $\mathit{GVar} \cap \mathit{SVar} = \emptyset$, and let $\mathit{Var} = \mathit{GVar} \cup \mathit{SVar}$. A symbolic program configuration (SPC) is a pair $C = (G, \nu)$ where G is an SMG with a set of regions R, and $\nu : \mathit{Var} \to R$ is a finite injective map such that $\forall x \in \mathit{Var} :\ \mathit{level}(\nu(x)) = 0 \ \wedge\ \mathit{valid}(\nu(x))$. Note that ν gives the regions in which values of variables are stored, not directly the values themselves. We call each object o such that $\nu(x) = o$ for some $x \in \mathit{GVar}$ a *static object*, and each object o such that $\nu(x) = o$ for some $x \in \mathit{SVar}$ a *stack object*. All other objects are called *heap objects*. An SPC is called *garbage-free* iff all its heap objects are reachable from static or stack objects.

[5] The last two requirements are not necessary, but they significantly simplify the below presented algorithms (e.g., the DLS materialisation given in Section 2.3).

We define the *empty SMG* to consist solely of the null object, its address 0, and the points-to edge between them. The *empty SPC* then consists of the empty SMG and the empty variable mapping. An SMG $G' = (O', V', \Lambda', H', P')$ is a *sub-SMG* of an SMG $G = (O, V, \Lambda, H, P)$ iff (1) $O' \subseteq O$, (2) $V' \subseteq V$, and (3) H', P', and Λ' are restrictions of H, P, and Λ to O' and V', respectively. The sub-SMG of G *rooted at* an object or value $x \in O \cup V$, denoted G_x, is the smallest sub-SMG of G that includes x and all objects and values reachable from x. Given $F \subseteq \mathbb{N}$, the *F-restricted* sub-SMG of G rooted at an object $o \in O$ is the smallest sub-SMG of G that includes o and all objects and values reachable from o apart from the addresses $A_F = \{H(o, of, \texttt{ptr}) \mid of \in F\}$ and nodes that are reachable from o through A_F only. Finally, the sub-SMG of G *nested below* $d \in D$, denoted $\widehat{G}_d$, is the smallest sub-SMG of G including d and all objects and values of level higher than $\mathit{level}(d)$ that are reachable from d via paths that, apart from d, consist exclusively of objects and values of a level higher than $\mathit{level}(d)$.

2.3 The Semantics of SMGs

We define the semantics of SMGs in two steps, namely, by first defining it in terms of the so-called memory graphs whose semantics is subsequently defined in terms of concrete memory images. In particular, a *memory graph* (MG) is defined exactly as an SMG up to it is not allowed to contain any list segments. An SMG then represents the class of MGs that can be obtained (up to isomorphism) by applying any number of times the following two transformations: (1) *materialisation* of fresh regions from DLSs (i.e., intuitively, "pulling out" concrete regions from the beginning or end of segments) and (2) *removal* of $0+$ DLSs (which may have become $0+$ due to the preceding materialisation).

Materialisation and Removal of DLSs. Let $G = (O, V, \Lambda, H, P)$ be an SMG with the sets of regions R, DLSs D, and addresses A. Let $d \in D$ be a DLS of level 0. Further, let $a_n, a_p \in A$ be the next and prev addresses of d, i.e., $H(d, \mathit{pfo}(d), \texttt{ptr}) = a_p$ and $H(d, \mathit{nfo}(d), \texttt{ptr}) = a_n$. The DLS d can be *materialised* as follows—for an illustration of the operation, see the upper part of Fig. 3:

1. G is extended by a fresh copy G'_r of the sub-SMG $\widehat{G}_d$ nested below d. In G'_r, d is replaced by a fresh region r such that $\mathit{size}(r) = \mathit{size}(d)$, $\mathit{level}(r) = 0$, and $\mathit{valid}(r) = \mathit{true}$. The nesting level of each object and value in G'_r (apart from r) is decreased by one.
2. Let $a_f \in A$ be the address pointing to the beginning of d, i.e., such that $P(a_f) = (\mathit{hfo}(d), \texttt{fst}, d)$. If a_f does not exist in G, it is added. Next, A is extended by a fresh address a_d that will point to the beginning of the remaining part of d after the concretisation (while a_f will be the address of r). Finally, H and P are changed s.t. $P(a_f) = (\mathit{hfo}(d), \texttt{reg}, r)$, $H(r, \mathit{pfo}(d), \texttt{ptr}) = a_p$, $H(r, \mathit{nfo}(d), \texttt{ptr}) = a_d$, $P(a_d) = (\mathit{hfo}(d), \texttt{fst}, d)$, and $H(d, \mathit{pfo}(d), \texttt{ptr}) = a_f$.
3. For any object o of $\widehat{G}_d$, let o' be the corresponding copy of o in G'_r (for $o = d$, let $o' = r$). For each field $(of, t) \in (\mathbb{N} \times \mathbb{T})$ of each object o in $\widehat{G}_d$ whose value is of level 0, i.e., $\mathit{level}(H(o, of, t)) = 0$, the corresponding field of o' in G'_r is set to the same value, i.e., the set of edges is extended such that $H(o', of, t) = H(o, of, t)$.
4. If $\mathit{len}(d) > 0$, $\mathit{len}(d)$ is decreased by one.

Next, let $d \in D$ be a DLS as above with the additional requirement of $\mathit{len}(d) = 0$ with the addresses a_n, a_p, a_f, and a_l defined as in the case of materialisation. The DLS d can be *removed* as follows—for an illustration, see the lower part of Fig. 3: (1) Each has-value edge $o \xrightarrow{of,t} a_f$ is replaced by the edge $o \xrightarrow{of,t} a_n$. (2) Each has-value edge $o \xrightarrow{of,t} a_l$ is replaced by the edge $o \xrightarrow{of,t} a_p$. (3) The subgraph $\widehat{G}_d$ is removed together with the addresses a_f, a_l, and the edges adjacent with the removed objects and values.

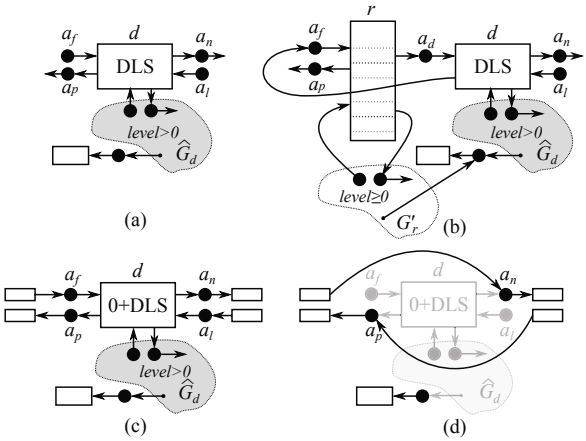

Fig. 3. Materialisation of a DLS: (a) input, (b) output (region r got materialised from DLS d). Removal of a DLS: (c) input, (d) output. Sub-SMGs $\widehat{G}_d$ and G'_r are highlighted without their roots.

Given an SMG $G = (O, V, \Lambda, H, P)$ with a set of DLSs D, we denote by $\mathit{MG}(G)$ the class of all MGs that can be obtained (up to isomorphism) by materializing each DLS $d \in D$ at least $\mathit{len}(d)$ times and by subsequently removing all DLSs.

Concrete Memory Images. The semantics of an MG $G = (R, V, \Lambda, H, P)$ is the set $\mathit{MI}(G)$ of *memory images* $\mu : \mathbb{N} \to \{0, \ldots, 255\}$ mapping *concrete addresses* to *bytes* such that there exists a function $\pi : R \to \mathbb{N}$, called a *region placement*, for which the following holds:

1. Only the null object is placed at address zero, i.e., $\forall r \in R : \ \pi(r) = 0 \Leftrightarrow r = \#$.
2. No two valid regions overlap, i.e., $\forall r_1, r_2 \in R : \ \mathit{valid}(r_1) \wedge \mathit{valid}(r_2) \Rightarrow \langle \pi(r_1), \pi(r_1) + \mathit{size}(r_1)) \cap \langle \pi(r_2), \pi(r_2) + \mathit{size}(r_2)) = \emptyset$.
3. Pointer fields are filled with the concrete addresses of the regions they refer to. Formally, for each pair of has-value and points-to edges $r_1 \xrightarrow{of_1,\mathtt{ptr}} a \xrightarrow{of_2,\mathtt{reg}} r_2$ in H and P, resp., $\mathit{addr}(\mathit{bseq}(\mu, \pi(r_1)+of_1, \mathit{size}(\mathtt{ptr}))) = \pi(r_2)+of_2$ where $\mathit{bseq}(\mu, p, \mathit{size})$ is the sequence of bytes $\mu(p)\mu(p+1)...\mu(p+\mathit{size}-1)$ for any $p, \mathit{size} > 0$, and $\mathit{addr}(\sigma)$ is the concrete address encoded by the byte sequence σ.
4. Fields having the same values are filled with the same concrete values (up to nullified blocks that can differ in their length), i.e., for every two has-value edges $r_1 \xrightarrow{of_1,t_1} v$ and $r_2 \xrightarrow{of_2,t_2} v$ in H, where $v \neq 0$, $\mathit{bseq}(\mu, \pi(r_1) + of_1, \mathit{size}(t_1)) = \mathit{bseq}(\mu, \pi(r_2) + of_2, \mathit{size}(t_2))$.
5. Finally, nullified fields are filled with zeros, i.e., for each has-value edge $r \xrightarrow{of,t} 0$ in H, $\mu(\pi(r) + of + i) = 0$ for all $0 \leq i < \mathit{size}(t)$.

For an SMG G, we let $\mathit{MI}(G) = \bigcup_{G' \in \mathit{MG}(G)} \mathit{MI}(G')$. Note that it may happen that it is not possible to find concrete values satisfying the needed constraints. In such a case, the semantics of an (S)MG is empty. Note also that we restrict ourselves to a flat address space, which is, however, sufficient for most practical cases.

3 Operations on SMGs

In this section, we propose algorithms for all the operations on SMGs that are needed for their application in program verification. In particular, we discuss data reinterpretation, join of SMGs (which we use for entailment checking, too), abstraction, inequality checking, and symbolic execution of C programs. Due to limited space, the description is mostly informal. More details can be found in [8].

Below, we denote by $I(of, t)$ the right-open integer interval $\langle of, of + size(t))$, and for a has-value edge $e : o \xrightarrow{of,t} v$, we write $I(e)$ as the abbreviation of $I(of, t)$.

3.1 Data Reinterpretation

SMGs allow fields of a single object to overlap and even to have the same offset and size, being distinguishable by their types only. In line with this feature of SMGs, we introduce the so-called *read reinterpretation* that can create multiple views (*interpretations*) of a single memory area without actually changing the semantics. On the other hand, if we write to a field that overlaps with other fields, we need to reflect the change of the memory image in the overlapping fields, for which the so-called *write reinterpretation* is used. These two operations form the basis of all operations reading and writing memory represented by SMGs. Apart from them, we also use *join reinterpretation* which is applied when joining two SMGs to preserve as much information shared by the SMGs as possible even when this information is not explicitly represented in the same way in both the input SMGs.

Defining reinterpretation for all possible data types (and all of their possible values) is hard (cf. [15]) and beyond the scope of this paper. Instead of that, we define minimal requirements that must be met by the reinterpretation operators so that our verification approach is sound. This allows different concrete instantiations of these operators to be used in the future. Currently, we instantiate the operators for the particular case of dealing with nullified blocks of memory, which is essential for handling low-level pointer manipulating programs that commonly use functions like `calloc()` or `memset()` to obtain large blocks of nullified memory.[6]

Read Reinterpretation. A read reinterpretation operator takes as input an SMG G with a set of objects O, an object $o \in O$, and a field (of, t) to be read from o such that $of + size(t) \leq size(o)$. The result is a couple (G', v) where G' is an SMG with a set of has-value edges H' such that (1) $H'(o, of, t) = v \neq \bot$ and (2) $MI(G) = MI(G')$. The operator thus preserves the semantics of the SMG but ensures that it contains a has-value edge for the field being read. This edge can lead to a value already present in the SMG but also to a new value derived by the operator from the edges and values existing in the SMG. In the extreme case, a fresh, completely unconstrained value node can be added, representing an unknown value, which can, however, become constrained by the further program execution. In other words, read reinterpretation installs a new view on some part of the object o, but it cannot modify the semantics of the SMG in any way.

[6] Apart from the nullified blocks, our implementation also supports tracking of uninitialized blocks of memory and certain manipulations of null-terminated strings.

For the particular case of dealing with nullified memory, we use the following concrete read reinterpretation (cf. [8]). If G contains an edge $o \xrightarrow{of,t} v$, (G, v) is returned. Otherwise, if each byte of the field (of, t) is nullified by some edge $o \xrightarrow{of',t'} 0$ present in G, $(G', 0)$ is returned where G' is obtained from G by adding the edge $o \xrightarrow{of,t} 0$. Otherwise, (G', v) is returned with G' obtained from G by adding an edge $o \xrightarrow{of,t} v$ leading to a fresh value v (representing an unknown value). It is easy to see that with the current support of types and values in SMGs, this is the most precise read reinterpretation that is possible from the point of view of reading nullified memory.

Write Reinterpretation. The write reinterpretation operator takes as input an SMG G with a set of objects O, an object $o \in O$, a field (of, t) within o, i.e., such that $of + size(t) \leq size(o)$, and a value v that is to be written into the field (of, t) of the object o. The result is an SMG G' with a set of has-value edges H' such that (1) $H'(o, of, t) = v$ and (2) $MI(G) \subseteq MI(G'')$ where G'' is the SMG G' without the edge $e : o \xrightarrow{of,t} v$. In other words, the operator makes sure that the resulting SMG contains the edge e that was to be written while the semantics of G' without e over-approximates the semantics of G. Indeed, one cannot require equality here since the new edge may collide with some other edges, which may have to be dropped in the worst case.

For the case of dealing with nullified memory, we propose the following write reinterpretation (cf. [8], which include an illustration too). If G contains the edge $e : o \xrightarrow{of,t} v$, G is returned. Otherwise, all has-value edges leading from o to a non-zero value whose fields overlap with (of, t) are removed. Subsequently, if $v = 0$, the edge e is added, and the obtained SMG is returned. Otherwise, all remaining has-value edges leading from o to 0 that define fields overlapping with (of, t) are split and/or shortened such that they do not overlap with (of, t), the edge e is added, and the resulting SMG is returned. Again, it is easy to see that this operator is the most precise write reinterpretation from the point of view of preserving information about nullified memory that is possible with the current support of types and values in SMGs.

3.2 Join of SMGs

Join of SMGs is a binary operation that takes two SMGs G_1, G_2 and returns an SMG G that is their common generalisation, i.e., $MI(G_1) \subseteq MI(G) \supseteq MI(G_2)$, and that satisfies the following further requirements intended to minimize the involved information loss: If both input SMGs are semantically equal, i.e., $MI(G_1) = MI(G_2)$, denoted $G_1 \simeq G_2$, we require the resulting SMG to be semantically equal to both the input ones, i.e., $MI(G_1) = MI(G) = MI(G_2)$. If $MI(G_1) \supset MI(G_2)$, denoted $G_1 \sqsupset G_2$, we require that $MI(G) = MI(G_1)$. Symmetrically, if $MI(G_1) \subset MI(G_2)$, denoted $G_1 \sqsubset G_2$, we require that $MI(G) = MI(G_2)$. Finally, if the input SMGs are semantically incomparable, i.e., $MI(G_1) \not\supseteq MI(G_2) \ \wedge \ MI(G_1) \not\subseteq MI(G_2)$, denoted $G_1 \bowtie G_2$, no further requirements are put on the result of the join (besides the inclusion stated above, which is required for the soundness of our analysis). In order to distinguish which of these cases happens when joining two SMGs, we tag the result of our join operator by the so-called *join status* with the domain $\mathbb{J} = \{\simeq, \sqsupset, \sqsubset, \bowtie\}$ referring to the corresponding relations above.

Moreover, we allow the join operation to fail if the incurred information loss becomes too big. Below, we give an informal description of our join operator, for a full description see [8].

The basic idea of our join algorithm is the following. The algorithm simultaneously traverses a given pair of source SMGs and tries to join each pair of nodes (i.e., objects or values) encountered at the same time into a single node in the destination SMG. A single node of one SMG is not allowed to be joined with multiple nodes of the other SMG. This preserves the distinction between different objects as well as between at least possibly different values.

The rules according to which it is decided whether a pair of objects simultaneously encountered in the input SMGs can be joined are the following. First, they must have the same size, validity, and in case of DLSs, the same head, prev, and next offsets. It is possible to join DLSs of different lengths as well as DLSs with regions (approximated as 1+ DLSs). The result is a DLS whose length is the minimum of the lengths of the joined DLSs (hence, e.g., joining a region with a 2+ DLS gives a 1+ DLS). The levels of the joined objects must also be the same up to the following case. When joining a sub-SMG nested below a DLS with a corresponding sub-SMG rooted at a region (restricted by ignoring the next and prev links), objects corresponding to each other appear on different levels: E.g., objects nested right below a DLS of level 0 are on level 1, whereas the corresponding objects directly referenced by a region of level 0 are on level 0 (since for regions, nested and shared sub-SMGs are not distinguished). This difference can, of course, increase when descending deeper in a hierarchically nested data structure as it is essentially given by the different numbers of DLSs passed on the different sides of the join. This difference is tracked by the join algorithm, and only the objects whose levels differ in the appropriate way are allowed to be joined.

When two objects are being joined, a *join reinterpretation* operator is used to ensure that they share the same set of fields and hence have the same number and labels of outgoing edges (which is always possible albeit sometimes for the price of introducing has-value edges leading to unknown values). A formalization of join reinterpretation is available in [8], including a concrete join reinterpretation operator designed to preserve maximum information on nullified blocks in both of the objects being joined. The join reinterpretation allows the fields of the joined objects to be processed in pairs of the same size and type. As for joining values, we do not allow joining addresses with unknown values.[7] Moreover, the zero value cannot be joined with a non-zero value. Further, addresses can be joined only if the points-to edges leading from them are labelled by the same offset, and when they lead to DLSs, they must have the same target specifier. On the other hand, apart from the already above expressed requirement of not joining a single value in one SMG with several values in the other SMG, no further requirements are put on joining non-address values, which is possible since we currently track their equalities only.

To increase chances for successfully joining two SMGs, the basic algorithm from above is extended as follows. When a pair of objects cannot be joined and at least one

[7] Allowing a join of an address and an unknown value could lead to a need to drop a part of the allocated heap in one of the SMGs (in case it was not accessible through some other address too), which we consider to be a too big loss of information.

of them is a DLS (call it d and the other object o), the algorithm proceeds as though o was preceded by a 0+ DLS d' that is up to its length isomorphic with d (including the not yet visited part of the appropriate sub-SMG nested below d). Said differently, the algorithm virtually inserts d' before o, joins d and d' into a single 0+ DLS, and then continues by trying to join o and the successor of d. This extension is possible since the semantics of a 0+ DLS includes the empty list, which can be safely assumed to appear anywhere, compensating a missing object in one of the SMGs.

Note, however, that the virtual insertion of a 0+ DLS implies a need to relax some of the requirements from above. For instance, one needs to allow a join of two different addresses from one SMG with one address in the other (the prev and next addresses of d get both joined with the address preceding o). Moreover, the possibility to insert 0+ DLSs introduces some non-determinism into the algorithm since when attempting to join a pair of incompatible DLSs, a 0+ DLS can be inserted into either of the two input DLSs, and we choose one of them. The choice may be wrong, but for performance reasons, we never backtrack. Moreover, we use the 0+ DLS insertion only when a join of two objects fails locally (i.e., without looking at their successors). When a pair of objects can be locally joined, but then the join fails on their successors, one could consider backtracking and trying to insert a 0+ DLS, which we again do not do for performance reasons (and we did not see a need for that in our cases studies so far).

The described join algorithm is used in two scenarios: (1) When joining garbage-free SPCs to reduce the number of SPCs obtained from different paths through the program, in which case the traversal starts from pairs of identical program variables. (2) As a part of the abstraction algorithm for merging a pair of neighbouring objects (together with the non-shared parts of the sub-SMGs rooted at them) of a doubly-linked list into a single DLS, in which case the algorithm is started from the neighbouring objects to be merged. In the join algorithm, the join status is computed on-the-fly. Initially, the status is set to $\simeq$. Next, whenever performing a step that implies a particular relation between G_1 and G_2 (e.g., joining a $0+$ DLS from G_1 with a $1+$ DLS from G_2 implies that $G_1 \sqsupset G_2$, assuming that the remaining parts of G_1 and G_2 are semantically equal), we appropriately update the join status.

3.3 Abstraction

Our abstraction is based on *merging uninterrupted sequences* of neighbouring objects, together with the $\{nfo, pfo\}$-restricted sub-SMGs rooted at them, into a single DLS. This is done by repeatedly applying a slight extension of the join algorithm on the $\{nfo, pfo\}$-restricted sub-SMGs rooted at the neighbouring objects. The sequences to be merged are identified by the so-called *candidate DLS entries* that consist of an object o_c and next, prev, and head offsets such that o_c has a neighbouring object with which it can be merged into a DLS linked through the given offsets. The abstraction is driven by the *cost* to be paid in terms of the loss of precision caused by merging certain objects and the sub-SMGs rooted at them (in particular, we distinguish joining of equal, entailed, or incomparable sub-SMGs). The higher the loss of precision is, the longer sequence of mergeable objects is required to enable a merge of the sequence.

In the extended join algorithm used in the abstraction (cf. [8]), the two simultaneous searches are started from two neighbouring objects o_1 and o_2 of the same SMG G that are the roots of the $\{\mathit{nfo}_c, \mathit{pfo}_c\}$-restricted sub-SMGs G_1, G_2 to be merged. The extended join algorithm constructs the sub-SMG $G_{1,2}$ that is to be nested below the DLS resulting from the join of o_1 and o_2. The extended join algorithm also returns the sets O_1, V_1 and O_2, V_2 of the objects and values of G_1 and G_2, respectively, whose join gives rise to $G_{1,2}$. Unlike when joining two distinct SMGs, the two simultaneous searches can get to a single node at the same time. Clearly, such a node is shared by G_1 and G_2, and it is therefore *not* included into the sub-SMG $G_{1,2}$ to be nested below the join of o_1 and o_2.

Below, we explain in more detail the particular steps of the abstraction. For the explanation, we fix an SPC $C = (G, \nu)$ where $G = (O, V, \Lambda, H, P)$ is an SMG with the sets of regions R, DLSs D, and addresses A.

Candidate DLS Entries. A quadruple $(o_c, \mathit{hfo}_c, \mathit{nfo}_c, \mathit{pfo}_c)$ where $o_c \in O$ and hfo_c, $\mathit{nfo}_c, \mathit{pfo}_c \in \mathbb{N}$ such that $\mathit{nfo}_c < \mathit{pfo}_c$ is considered a *candidate DLS entry* iff the following holds: (1) o_c is a valid heap object. (2) o_c has a neighbouring object $o \in O$ with which it is doubly-linked through the chosen offsets, i.e., there are $a_1, a_2 \in A$ such that $H(o_c, \mathit{nfo}_c, \texttt{ptr}) = a_1$, $P(a_1) = (\mathit{hfo}_c, \mathit{tg}_1, o)$ for $\mathit{tg}_1 \in \{\texttt{fst}, \texttt{reg}\}$, $H(o, \mathit{pfo}_c, \texttt{ptr}) = a_2$, and $P(a_2) = (\mathit{hfo}_c, \mathit{tg}_2, o_c)$ for $\mathit{tg}_2 \in \{\texttt{lst}, \texttt{reg}\}$.

Longest Mergeable Sequences. The *longest mergeable sequence* of objects given by a candidate DLS entry $(o_c, \mathit{hfo}_c, \mathit{nfo}_c, \mathit{pfo}_c)$ is the longest sequence of distinct valid heap objects whose first object is o_c, all objects in the sequence are of level 0, all DLSs that appear in the sequence have hfo_c, nfo_c, pfo_c as their head, next, prev offsets, and the following holds for any two neighbouring objects o_1 and o_2 in the sequence (for a formal description, cf. [8]): (1) The objects o_1 and o_2 are doubly linked through their nfo_c and pfo_c fields. (2) The objects o_1 and o_2 are a part of a sequence of objects that is not pointed from outside of the detected list structure. (3) The $\{\mathit{nfo}_c, \mathit{pfo}_c\}$-restricted sub-SMGs G_1 and G_2 of G rooted at o_1 and o_2 can be joined using the extended join algorithm into the sub-SMG $G_{1,2}$ to be nested below the join of o_1 and o_2. Let O_1, V_1 and O_2, V_2 be the sets of non-shared objects and values of G_1 and G_2, respectively, whose join gives rise to $G_{1,2}$. (4) The non-shared objects and values of G_1 and G_2 (other than o_1 and o_2 themselves) are reachable via o_1 or o_2, respectively, only. Moreover, the sets O_1 and O_2 contain heap objects only.

Merging Sequences of Objects into DLSs. Sequences of objects are merged into a single DLS *incrementally*, i.e., starting with the first two objects of the sequence, then merging the resulting new DLS with the third object in the sequence, and so on. Each of the *elementary merge operations* is performed as follows (see Fig. 4 for an illustration).

Assume that G is the SMG of the current SPC (i.e., the initial SPC or the SPC obtained from the last merge) with the set of points-to edges P and the set of addresses A, o_1 is either the first object in the sequence or the DLS obtained from the previous elementary merge, o_2 is the next object of the sequence to be processed, and hfo_c, nfo_c, pfo_c are the offsets from the candidate DLS entry defining the sequence to be merged. First, we merge o_1 and o_2 into a DLS d using hfo_c, nfo_c, and pfo_c as its defining offsets

(cf. [8]). The sub-SMG nested below d is created using the above mentioned extended join algorithm. Next, the DLS-linking pointers arriving to o_1 and o_2 are redirected to d. In particular, if there is $a_f \in A$ such that $P(a_f) = (o_1, \mathit{hfo}_c, tg)$ for some $tg \in \{\texttt{fst}, \texttt{reg}\}$, then P is changed such that $P(a_f) = (d, \mathit{hfo}_c, \texttt{fst})$. Similarly, if there is $a_l \in A$ such that $P(a_l) = (o_2, \mathit{hfo}_c, tg)$ for some $tg \in \{\texttt{lst}, \texttt{reg}\}$, then P is changed such that $P(a_l) = (d, \mathit{hfo}_c, \texttt{lst})$. Finally, each heap object and each value (apart from the null address and null object) that are not reachable from any static or stack object of the obtained SPC are removed from its SMG together with all the edges adjacent to them.

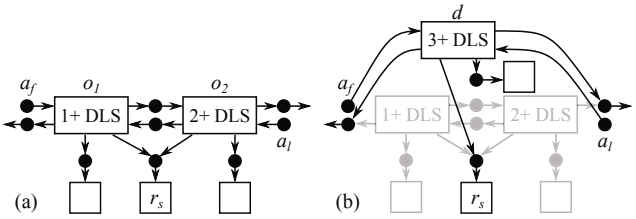

Fig. 4. The elementary merge operation: (a) input (b) output

The Top-level Abstraction Algorithm. Assume we are given an SMG G, and a candidate DLS entry $(o_c, \mathit{hfo}_c, \mathit{nfo}_c, \mathit{pfo}_c)$ defining the longest mergeable sequence of objects $\sigma = o_1 o_2 \ldots o_n$ in G of length $|\sigma| = n \geq 2$. We define the *cost* of merging a pair of objects o_1, o_2, denoted $\mathit{cost}(o_1, o_2)$, as follows. First, $\mathit{cost}(o_1, o_2) = 0$ iff the $\{\mathit{nfo}_c, \mathit{pfo}_c\}$-restricted sub-SMGs G_1 and G_2 rooted at o_1, o_2 are equal (when ignoring the kinds of o_1 and o_2). This is indicated by the $\simeq$ status returned by the modified join algorithm applied on G_1, G_2. Further, $\mathit{cost}(o_1, o_2) = 1$ iff G_1 entails G_2, or vice versa, which is indicated by the status $\sqsupset$ or $\sqsubset$. Finally, $\mathit{cost}(o_1, o_2) = 2$ iff G_1 and G_2 are incomparable, which is indicated by status $\bowtie$. The cost of merging a sequence of objects $\sigma = o_1 o_2 \ldots o_n$, denoted $\mathit{cost}(\sigma)$, is defined as the maximum of $\mathit{cost}(o_1, o_2)$, $\mathit{cost}(o_2, o_3)$, ..., $\mathit{cost}(o_{n-1}, o_n)$.

Our abstraction is parameterized by associating each cost $c \in \{0, 1, 2\}$ with the *length threshold*, denoted $\mathit{lenThr}(c)$, defining the minimum length of a sequence of mergeable objects allowed to be merged for the given cost. Intuitively, the higher is the cost, the bigger loss of precision is incurred by the merge, and hence a bigger number of objects to be merged is required to compensate the cost. In our experiments discussed in Section 5, we, in particular, found as optimal the setting $\mathit{lenThr}(0) = \mathit{lenThr}(1) = 2$ and $\mathit{lenThr}(2) = 3$. Our tool, however, allows the user to tweak these values.

Based on the above introduced notions, the process of *abstracting an SPC* can now be described as follows. First, all candidate DLS entries are identified, and for each of them, the corresponding longest mergeable sequence is computed. Then each longest mergeable sequence σ for which $|\sigma| < \mathit{lenThr}(\mathit{cost}(\sigma))$ is discarded. Out of the remaining ones, we select those that have the lowest cost. From them, we then select those that have the longest length. Finally, out of them, one is selected arbitrarily. The selected sequence is merged, and then the entire abstraction process is repeated till there is a sequence that can be merged taking its length and cost into account.

3.4 Checking Equality and Inequality of Values

Checking equality of values in SMGs amounts to simply checking their identity. For checking inequality, we use an algorithm which is sound and efficient but incomplete. It is designed to succeed in most common cases, but in order not to harm its efficiency, we allow it to fail in some exceptional cases (e.g., when comparing addresses out of bounds of two distinct objects). The basic idea of the algorithm is as follows (cf. [8]): Let v_1 and v_2 be two distinct values of level 0 to be checked for inequality (other levels cannot be directly accessed by program statements). First, if the same value or object can be reached from v_1 and v_2 through 0+ DLSs only (using the next/prev fields when coming through the `fst`/`lst` target specifiers, respectively), then the inequality between v_1 and v_2 is not established. This is due to v_1 and v_2 may become the same value when the possibly empty 0+ DLSs are removed (or they may become addresses of the first and last node of the same 0+ DLS, and hence be equal in case the list contains a single node). Otherwise, v_1 and v_2 are claimed different if the final pair of values reached from them through 0+ DLSs represents different addresses due to pointing (1) to different valid objects (each with its own unique address) with offsets inside their bounds, (2) to the null object and a non-null object (with an in-bound offset), (3) to the same object with different offsets, or (4) to the same DLS with length at least 2 using different target specifiers. Otherwise, the inequality is not established.

3.5 A Brief Note on Symbolic Execution

The symbolic execution algorithm based on SMGs is similar to [1]. It uses the read reinterpretation operator for memory lookup (as well as type-casting) and the write reinterpretation operator for memory mutation. Whenever a DLS is about to be accessed (or its address with a non-head offset is about to be taken), a materialisation (as described in Section 2.3) is performed so that the actual program statements are always executed over concrete objects. If the minimum length of the DLS being materialised is zero, the computation is split into two branches—one for the empty segment and one for the non-empty segment. In the former case, the DLS is removed (as described in Section 2.3) while in the latter case, the minimum length of the DLS is incremented. When executing a conditional statement, the algorithm for checking (in)equality of values from Section 3.4 is used. If neither equality nor inequality are established, the execution is split into two branches, one of them assuming the compared values to be equal, the other assuming them not to be equal. This may again involve removing 0+ DLSs in one of the branches and incrementing their length in the other (cf. [8]).

A Note on Soundness of the Analysis. In the described analysis, program statements are always executed on concrete objects only, closely following the C semantics. The read reinterpretation is defined such that it cannot change the semantics of the input SMG, and the write reinterpretation can only over-approximate the semantics in the worst case. Likewise, our abstraction and join algorithms are allowed to only over-approximate the semantics—indeed, when joining a pair of nodes, the semantics of the

resulting node is always generic enough to cover the semantics of both of the joined nodes (e.g., the join of a $2+$ DLS with a compatible region results in a $1+$ DLS, etc.). Moreover, the entailment check used to terminate the analysis is based on the join operator and consequently conservative. Hence, it is not difficult to see that the proposed analysis is sound (although a full proof of this fact would be rather technical).

4 Extensions of SMGs

In this section, we point out that the notion of SMGs can be easily extended in various directions, and we briefly discuss several such extensions (including further kinds of abstract objects), most of which are already implemented in our tool Predator.

Explicit Non-equivalence Relations. When several objects have the same concrete value stored in their fields, this is expressed by that the appropriate has-value edges lead from these objects to the same value node in the SMG. On the other hand, two different value nodes in an SMG do not necessarily represent different concrete values. To express that two abstract values represent distinct concrete values, SMGs can be extended with a symmetric, irreflexive relation over values, which we call an *explicit non-equivalence relation.* Clearly, SMGs can be quite naturally extended by allowing more predicates on data, which is, however, beyond the scope of this paper (up to a small extension by tracking more concrete values than 0 that is mentioned below).

Singly-linked List Segments (SLSs). Above, we have presented all algorithms on SMGs describing doubly-linked lists only. Nevertheless, the algorithms work equally well with singly-linked lists represented by an additional kind of abstract objects, SLSs, that have no *pfo* offset, and their addresses are allowed to use the `fst` and `all` target specifiers only. The algorithm looking for DLS entry candidates then simply starts looking for SLS entry candidates whenever it does not discover the back-link.

0/1 Abstract Objects. In order to enable summarization of lists whose nodes can *optionally* point to some region or that point to nested lists whose length never reaches 2 or more, we introduce the so-called *0/1 abstract objects*. We distinguish three kinds of them with different numbers of neighbour pointers. The first of them represents 0/1 SLSs with one neighbour pointer, another represents 0/1 DLSs with two neighbour pointers. These objects can be later joined with compatible SLSs or DLSs. The third kind has no neighbour pointer, and its address is assumed to be NULL when the region is not allocated. This kind is needed for optionally allocated regions referred from list nodes but never handled as lists themselves. The 0/1 abstract objects are created by the join algorithm when a region in one SMG cannot be matched with an object from the other SMG and none of the above described join mechanisms applies.

Offset Intervals and Address Alignment. The basic SMG notion labels points-to edges with scalar offsets within the target object. This labelling can be generalized to *intervals of offsets*. The intervals can be allowed to arise by joining objects with

incoming pointers compatible up to their offset. This feature is useful, e.g., to handle lists arising in higher-level memory allocators discussed in the next section where each node points to itself with an offset depending on how much of the node has been used by sub-allocation. Offset intervals also naturally arise when the analysis is allowed to support *address alignment*, which is typically implemented by masking several lowest bits of pointers to zero, resulting in a pointer whose offset is in a certain interval wrt. the base address. Similarly, one can allow the *object size* to be given by an interval, which in turn allows one to abstract lists whose nodes are of a variable size.

Integral Constants and Intervals. The basic SMG notion allows one to express that two fields have the same value (by the corresponding has-value edges leading to the same value node) or that their values differ (using the above mentioned explicit non-equivalence relation). In order to improve the support of dealing with integers, SMGs can be extended by associating value nodes with concrete integral numbers. These can be respected by the join algorithm (at least up to some bound), or they can be abstracted to intervals or some other abstract numerical domains.

5 Implementation

We have implemented the above described algorithms (including the extensions) in a new version of our tool called Predator.[8] Predator is a GCC plug-in, which allows one to experiment with industrial source code without manually preprocessing it first. The verified program must, however, be closed in that it must allocate and initialize all the data structures used. Modular verification of code fragments is planned for the future. By default, Predator disallows calls to external functions in order to exclude any side effect that could potentially break memory safety. The only allowed external functions are those that Predator recognizes as built-in functions and properly models them wrt. proving memory safety. Besides `malloc` and `free`, the set of supported built-in functions includes certain memory manipulating functions defined in the C standard, such as `memset`, `memcpy`, or `memmove`. Predator uses the same style of error and warning messages as GCC itself, and hence it can be used with any IDE that can use GCC. It also supports error recovery to report multiple program errors during one run. For example, if a memory leak is detected, Predator only reports a warning, the unreachable part of SMG is removed, and the symbolic execution then continues.

Predator implements an inter-procedural analysis based on [13]. It does not support recursive programs yet, but it supports indirect calls, which is necessary for verification of programs with callbacks (e.g., Linux drivers). Regions for stack variables are created automatically as needed and destroyed as soon as they become dead according to a static live variables analysis, performed before running the symbolic execution. When working with initialized variables, we take advantage of our efficient representation of nullified blocks—we first create a has-value edge $o \xrightarrow{0,\texttt{char}[size(o)]} 0$ for each initialized variable represented by a region o, then we execute all explicit initializers, which themselves automatically trigger the write reinterpretation. The same approach is used

[8] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`

for `calloc`-based heap allocation. Thanks to this, we do not need to initialize each structure member explicitly, which would not scale for complex structures.

The algorithms for abstraction and join implemented in Predator use some further optimizations of the basic algorithms described in Section 3. While objects in SMGs are type-free, Predator tracks their *estimated type* given by the type of the pointers through which objects are manipulated. The estimated type is used during abstraction to postpone merging a pair of objects with incompatible types. Note, however, that this is really a heuristic only—we have a case study that constructs list nodes using solely `void` pointers, and it can still be successfully verified by Predator. Another heuristic is that certain features of the join algorithm (e.g., insertion of a non-empty DLS or introduction of an 0/1 abstract object) are disabled when joining SMGs while enabled when merging nodes during abstraction. Predator tracks integral values precisely up to a certain bound (± 10 by default) and once the bound is reached, the values are abstracted out. Predator also supports intervals aligned to a power of two as well as tracking of simple dependences between intervals, such as a shift by a constant and a multiplication by -1. All these features are optional and can be easily disabled.

Predator iteratively computes sets of SMGs for each basic block entry of the control-flow graph of the given program, covering all program configurations reachable at these program locations. Termination of the analysis is aided by the abstraction and join algorithms described above. Since the join algorithm is expensive, it is used at loop boundaries only. When updating states of other basic block entries, we compare the SMGs for equality only, which makes the comparison way faster, especially in case a pair of SMGs cannot be joined. Similarly, the abstraction is by default used only at loop boundaries in order not to introduce abstract objects where not necessary (reducing the space for false positives that can arise due to breaking assumptions sometimes used by programmers for code inside loops as witnessed by some of our case studies).

Predator is able to discover or prove absence of various kinds of *memory safety errors*, including various forms of illegal dereferences (null dereferences, dereferences of freed or unallocated memory, out-of-bound dereferences), illegal free operations (double free operations, freeing non-heap objects), as well as memory leakage. Moreover, Predator also uses the fact that SMGs allow for easy checking whether a given pair of memory areas overlap. Indeed, if both of them are inside of two distinct valid regions, they have no overlaps, and if both of them are inside the same region, one can simply check their offset ranges for intersection. Such checks are used for reporting invalid uses of `memcpy` or the C-language assignment, which expose undefined behavior if the destination and source memory areas (partially) overlap with each other.

6 Experiments

The new version of Predator based on the above proposed method was successfully tested on a number of case studies. Among them there are more than 256 case studies (freely available with Predator) illustrating various programming constructs typically used when dealing with linked lists. These case studies include various advanced kinds of lists used in the Linux kernel and their typical manipulation, typical error patterns that appear in code operating with Linux lists, various sorting algorithms (insert sort, bubble

Table 1. Selected experimental results showing either the verification time or one of the following outcomes: FP = false positive, FN = false negative, T = time out (900 s), x = parsing problems

Test Origin	Test	Invader	SLAyer	Predator 2011-10	Predator 2013-02
SLAyer	`append.c`	<0.01 s	10.47 s	<0.01 s	<0.01 s
	`cromdata_add_remove_fs.c`	<0.01 s	FN	<0.01 s	<0.01 s
	`create_kernel.c`	T	FN	<0.01 s	<0.01 s
	`cromdata_add_remove.c`	T	FN	<0.01 s	<0.01 s
	`reverse_seg_cyclic.c`	FP	0.68 s	<0.01 s	<0.01 s
	`is_on_list_via_devext.c`	T	34.43 s	0.20 s	0.02 s
	`callback_remove_entry_list.c`	T	71.46 s	0.14 s	0.10 s
Invader	`cdrom.c`	FN	x	2.44 s	0.66 s
Predator	`five-level-sll-destroyed-top-down.c`	FP	x	FP	0.05 s
	`linux-dll-of-linux-dll.c`	T	x	0.41 s	0.05 s
	`merge-sort.c`	FP	x	1.08 s	0.21 s
	`list-of-arena-pools-with-alignment.c`	FP	x	FP	0.50 s
	`lvmcache_add_orphan_vginfo.c`	x	x	FP	1.07 s
	`five-level-sll-destroyed-bottom-up.c`	FP	x	FP	1.14 s

sort, merge sort), etc. These case studies have up to 300 lines of code, but they consist almost entirely of complex memory manipulation (unlike larger programs whose big portions are often ignored by tools verifying memory safety). Next, we successfully tested Predator on the driver code snippets distributed with SLAyer [2] as well as on the `cdrom` driver originally checked by Space Invader [17]. As discussed below, in some of these examples, we identified errors not found by the other tools due to their more abstract (not byte-precise) treatment of memory.

Further, we also considered two real-life low-level programs (which, to the best of our knowledge, have not yet been targeted by fully automated formal verification tools): a memory allocator from the Netscape portable runtime (NSPR) and a module taken from the `lvm2` logical volume manager. The NSPR allocator allocates memory from the operating system in blocks called *arenas*, grouped into singly-linked lists called *arena pools*, which can in turn be grouped into lists of arena pools (giving lists of lists of arenas). User requests are then satisfied by sub-allocation within a suitable arena of a given arena pool. We have considered a fixed size of the arenas and checked safety of repeated allocation and deallocation of blocks of aligned size randomly chosen up to the arena size from arena pools as well as lists of arena pools. For this purpose, a support for offset intervals as described above was needed. The intervals arise from abstracting lists whose nodes (arenas) point with different offsets to themselves (one byte behind the last sub-allocated block within the arena) and from address alignment, which the NSPR-based allocator is also responsible for. Our approach allowed us to verify that pointers leading from each arena to its so-far free part never point beyond the arena and that arena headers never overlap with their data areas, which are the original assertions checked by NSPR arena pools at run-time. Our `lvm2`-based case studies then exercise various functions of the module implementing the volume metadata cache. As in the case of NSPR arenas, we use the original (unsimplified) code of the module, but (for now) we use a simplified test harness where the `lvm2` implementation of hash tables is replaced by the `lvm2` implementation of doubly-linked lists.

We have compared the capabilities and performance of Invader, SLAyer, and Predator on the above case studies on an Intel® Core™ i7-3770K machine. The memory consumption was below 128 MB in all cases. As we can see in Table 1, Predator successfully verified even the test-cases that were causing problems to Invader or SLAyer. We have also revealed issues of memory safety violation in the examples distributed with Invader and SLAyer because Invader did not check memory manipulation via array subscripts and SLAyer did not check size of the blocks allocated on the heap.[9] All the tools were run in their default configurations. Better results can sometimes be obtained for particular case studies by tweaking certain configuration options (abstraction threshold, call cache size, etc.). However, while such changes may improve the performance in some case studies, they may harm it in others, trigger false positives, or even prevent the analysis from termination.

We have also compared the new version of Predator with its older version that participated in the 1st International Competition on Software Verification (SV-COMP'12). The old Predator produced false positives on many of the more advanced case studies, including NSPR arenas and lvm2, and it was also slower. For example, the merge-sort case study, presented as the most expensive in [7] (Predator 2011-02), now runs approximately $25\times$ faster on the same machine ($5\times$ due to the algorithms presented above and $5\times$ due to an improved live variable analysis). The new Predator participated in the 2nd International Competition on Software Verification (SV-COMP'13) [4], where it won three categories. Moreover, the fact that Predator did not have any false negative over the whole SV-COMP'13 benchmark confirms the soundness of our analysis algorithm.

7 Conclusion and Future Work

We have presented a new approach to fully automated formal verification of list manipulating programs capable of handling various features of low-level memory manipulation. We have experimentally validated the approach on a number of case studies showing its efficiency and capability of handling program behaviour that is beyond what current fully automated shape analysis tools can handle. In the future, a number of extensions of our approach are possible. We are planning a support of (low-level) tree structures, a better support of integer data, a support of arrays and hash tables, as well as a support for modular verification in order to remove the burden of having to write environments for the code to be verified.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Berdine, J., Cook, B., Ishtiaq, S.: SLAYER: Memory Safety for Systems-Level Code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)

[9] We used the latest publicly available version of SLAyer from [2]. The version from [3] was not available, but [3] targets mainly checking of spuriousness of counterexamples.

3. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.M.: Diagnosing Abstraction Failure for Separation Logic-Based Analyses. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 155–173. Springer, Heidelberg (2012)
4. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)
5. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 182–203. Springer, Heidelberg (2006)
6. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape Analysis with Structural Invariant Checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
7. Dudka, K., Peringer, P., Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
8. Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. Technical report FIT-TR-2012-04, FIT BUT (2012), `http://www.fit.vutbr.cz/~idudka/pub/FIT-TR-2012-04.pdf`
9. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest Automata for Verification of Heap Manipulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer, Heidelberg (2011)
10. Kreiker, J., Seidl, H., Vojdani, V.: Shape Analysis of Low-Level C with Overlapping Structures. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 214–230. Springer, Heidelberg (2010)
11. Laviron, V., Chang, B.-Y.E., Rival, X.: Separating Shape Graphs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 387–406. Springer, Heidelberg (2010)
12. Marron, M., Hermenegildo, M.V., Kapur, D., Stefanovic, D.: Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 245–259. Springer, Heidelberg (2008)
13. Reps, T., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Proc. of POPL 1995. ACM Press (1995)
14. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: ACM Transactions on Programming Languages and Systems (TOPLAS), 24(3) (2002)
15. Tuch, H.: Formal Verification of C Systems Code. Journal of Automated Reasoning 42(2-4) (2009)
16. Yang, H., Lee, O., Calcagno, C., Distefano, D., O'Hearn, P.W.: On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London (2007)
17. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)