

Chapter 1

Algorithmic Details behind the Predator Shape Analyser Based on Symbolic Memory Graphs

Kamil Dudka^{1,2}, Petr Muller^{1,2}, Petr Peringer¹, Veronika Šoková¹, Tomáš Vojnar¹

Abstract This chapter, which is an extended and revised version of the conference paper [19], concentrates on a detailed description of the algorithms behind the Predator shape analyser based on abstract interpretation and symbolic memory graphs. Predator is particularly suited for formal analysis and verification of sequential non-recursive C code that uses low-level pointer operations to manipulate various kinds of linked lists of unbounded size as well as various other kinds of pointer structures of bounded size. The tool supports practically relevant forms of pointer arithmetic, block operations, address alignment, or memory reinterpretation. We present the overall architecture of the tool, along with selected implementation details of the tool as well as its extension into so-called Predator Hunting Party, which utilises multiple concurrently-running Predator analysers with various restrictions on their behaviour. Results of experiments with Predator within the SV-COMP competition as well as on our own benchmarks are provided.

1.1 Introduction

Dealing with *pointers* and *dynamic linked data structures* belongs among the most challenging tasks of formal analysis and verification of software 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.

In this chapter, we present a fully-automated approach to formal verification of list manipulating programs that is behind the *Predator shape analyser* and that is designed to cope with all of the above mentioned low-level memory operations. The approach is based on representing sets of heap graphs using the so-called *symbolic memory graphs* (SMGs). This representation is to some degree inspired by works on separation logic with higher-order list predicates [2], but it is graph-based

Brno University of Technology, Faculty of Information Technology, Czech Republic · Red Hat Czech, Brno, Czech Republic

and uses a much more fine-grained memory model. In particular, SMGs and the algorithms designed to make them applicable in a fully-automated shape analysis based on abstract interpretation allow one to deal with *byte-precise* offsets of fields of objects, offsets of pointer targets, as well as object sizes. As our experiments show, Predator can 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).

Symbolic Memory Graphs

Going into slightly more detail, SMGs are directed graphs 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.

SMGs allow us to handle 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 their middle, 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 use 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 1.3.2 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 [37]), our join and abstraction support so-called *0/1 abstract objects*, i.e., objects that may but need not be present.

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 arriving with different offsets (as common, e.g., in memory allocators).

The Predator Analyser

The approach sketched above has been implemented in the Predator analyser [18]. 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*. Since SMGs provide a rather detailed memory model, Predator produces fewer false alarms than 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 buffer overflows on the stack, i.e., the so-called *stack smashing*, which can alter the execution flow and cause serious security vulnerabilities) as well as nasty bugs resulting from dealing with *overlapping blocks of memory* in operations like `memcpy`.

The Predator analyser is a basic building block of the so-called *Predator Hunting Party* (*PredatorHP*). In PredatorHP, the Predator analyser is called a *verifier* due to its sound over-approximation of program semantics. It is allowed to claim programs correct, but it is not allowed to warn about bugs since it can produce false alarms due to the over-approximation it uses. For detecting errors, PredatorHP contains several *Predator hunters* which use SMG-based algorithms but with no abstraction on pointer structures. The different hunters differ in their search strategy (depth-first or breadth-first) and the limits imposed on their search. They can warn about errors (not producing any false alarms unless caused by abstraction of non-pointer data) but cannot prove programs correct. The only exception is the case when the given program has a finite state space and that is entirely explored. As our experiments show, PredatorHP can indeed avoid many false alarms and it also reduces the wall-clock time of the analysis (while usually increasing the CPU time).

Predator has been successfully validated 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 present results of our experiments with 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. As we show on the experimental results that we obtained, many of our case studies go beyond what other currently-existing fully-automated program analysis and verification tools can handle. We further present an experimental evaluation of Predator and PredatorHP on the benchmarks of the International Software Verification Competition 2019 (SV-COMP'19) where we concentrate mainly on the effects of using PredatorHP and also on some recent extensions of Predator.

Outline of the Chapter

The rest of the chapter is organised as follows. First, in Section 1.2.1, we extend the basic intuition on SMGs provided above by a somewhat more detailed but still intuitive explanation. Section 1.2.2 provides a further illustration of SMGs through two examples of SMGs representing data structures used in practice. Then, we provide formal definitions of SMGs and related notions in Section 1.2.3.

Subsequently, Section 1.3 describes principles of the operations on SMGs that are needed for implementing a shape analyser on top of SMGs. The principles presented in Section 1.3 should suffice for getting a decent understanding of all the operations. However, for those more interested, Appendices 1.8–1.11 provide a detailed description of all the operations in the form of pseudo-code.

At this point, let us stress once more that the main computation loop of Predator is that of *abstract interpretation*. It uses sets of symbolic program configurations based on SMGs, i.e., a disjunctive extension of the basic abstract domain, to abstractly record sets of program configurations reachable at program locations. SMG abstraction is used to implement widening, and SMG join and entailment (based itself on the SMG join) are used to reduce the number of SMGs tracked at particular program locations. Due to this very standard approach being employed, we do not provide a further detailed explanation of the basic computation loop itself. However, in Section 1.3.6, we provide a rather complete illustration of the entire computation loop on an example program.

Section 1.4 provides various extensions of the basic notion of SMGs. Section 1.5 discusses the architecture of Predator along with its various implementation details. Moreover, Section 1.5 also introduces PredatorHP. Section 1.6 provides our experimental results. In Section 1.7, we discuss related works.

Finally, Appendix 1.12 contains a brief tutorial on running and configuring Predator and PredatorHP.

1.2 Symbolic Memory Graphs

In this section, we introduce the notion of *symbolic memory graphs* (SMGs) that are intended—together with a mapping from global (static) and local (stack) variables to their nodes—to encode (possibly infinite) sets of configurations of programs with pointers and unbounded dynamic linked lists. 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. 1.1, which shows how SMGs represent cyclic

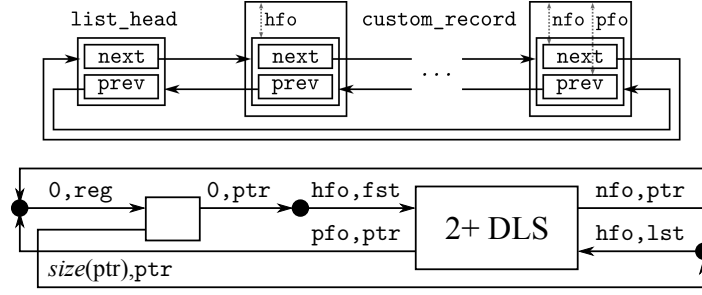


Fig. 1.1: A cyclic Linux-style DLL (top) and its SMG (bottom), with some SMG attributes left out for readability. For the meaning of the acronyms (e.g., hfo stands for the head-field offset) see Section 1.2.3.

Linux-style DLLs. The head node of such lists has no data part (while all other nodes include the head structure as well as custom data), and its next/prev pointers point *inside* list nodes, not at their beginning.

1.2.1 The Intuition behind SMGs

An SMG consists of two kinds of nodes: *objects* and *values* (in Fig. 1.1, 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. 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. 1.1, 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 could be 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. 1.1). Intuitively, the edges express that objects have values and addresses point to objects (non-address values have no outgoing edge). 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

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.

can assign to a given memory area and that we do not want the analyser to recompute 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 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, $(x == \&x \rightarrow \text{next})$ 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 In particular, the notation “2+” used in Fig. 1.1 means that the minimum length of the list segment is 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 backward³. 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. 1.1). 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 in a similar way as heap objects and can thus be manipulated in a similar way (including their

² Later, in Section 1.4, special list segments of length 0 or 1 are mentioned too.

³ The names “next” and “prev” (i.e., previous) are used within our definition of list segments only. The concrete names of these fields in the programs being analysed are irrelevant.

manipulation via pointers, checking for out-of-bounds accesses leading to stack smashing, etc.). Regions representing program variables are tagged by their names and hence distinguishable whenever needed (e.g., when checking for invalid frees of stack/global memory, etc.).

1.2.2 Further Illustration of the Notion of SMGs

We now provide two more illustrative examples of how SMGs represent various data structures common in practice.

The upper part of Fig. 1.2 shows a Linux-style cyclic DLL of cyclic DLLs. All nodes of all nested DLLs point to a shared memory region. The lower part of the figure shows an SMG representing this structure. Note that the top-level DLS as well as the shared region are on level 0 whereas the nested DLSs are on level 1.

The upper part of Fig. 1.3 shows another variant of Linux-style DLLs which is optimised for use in hash tables. The lower part of the figure shows an SMG representing this kind of lists. For lists used in hash tables, the size of list headers determines the amount of memory allocated by an empty hash table. That is why the lists presented in Fig. 1.3 have headers reduced to the size of a single field for the price of having forward and backward links of different types. In particular, forward links are pointers to structures whereas backward links are pointers to pointers to structures. This asymmetry may cause problems to analysers that use a selector-based description of list segments, but it is not a problem for us since our representation is purely offset-based.⁴

1.2.3 Formal Definition of Symbolic Memory Graphs

Let $\mathbb{B}$ be the set of Booleans, $\mathbb{T}$ a set of types, $size(t)$ the size of instances of a type $t \in \mathbb{T}$, $ptr \in \mathbb{T}$ a unique pointer type⁵, $\mathbb{K} = \{\text{reg}, \text{dls}\}$ the set of kinds of objects (distinguishing regions and DLSs), and $\mathbb{S} = \{\text{fst}, \text{lst}, \text{all}, \text{reg}\}$ the set of points-to target specifiers.

Symbolic Memory Graphs

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:

⁴ A need to use a special kind of list segments would arise in SMGs if the head and next offsets were different, but that is unlikely to happen in this special case since it would prevent the list head from having the size of a single pointer only.

⁵ We assume $size(ptr)$ to be a constant, which implies that separate verification runs are needed for verifying a program for target architectures using different address sizes.

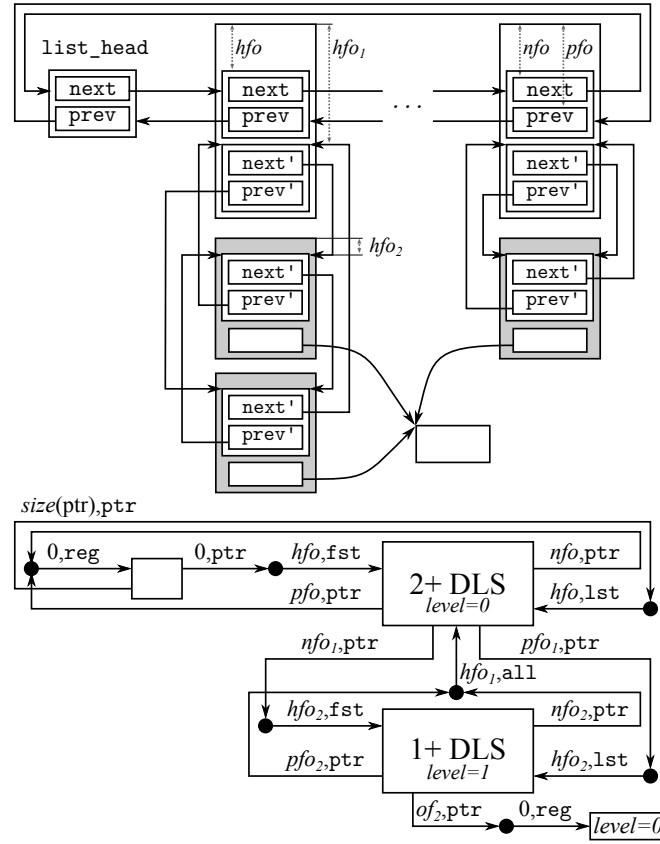


Fig. 1.2: A cyclic Linux-style DLL of DLLs with a shared data element (top) and its SMG (bottom).

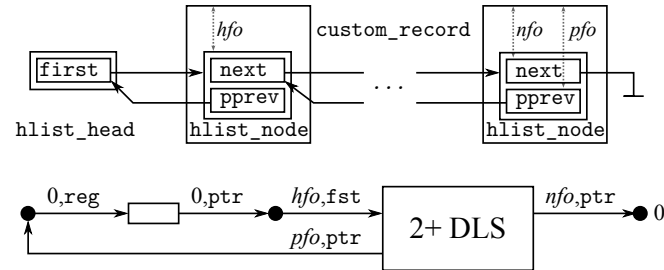


Fig. 1.3: A Linux-style list used in hash tables (top) and its SMG (bottom).

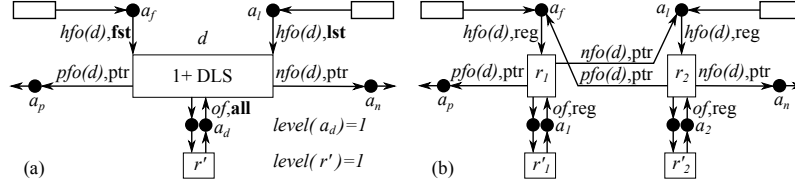


Fig. 1.4: (a) An SMG and (b) its possible concretisation for the case when the DLS d represents exactly two regions (showing key attributes only).

- The kind of objects $kind : O \rightarrow \mathbb{K}$ where $kind(\#) = \text{reg}$, i.e., $\#$ is formally considered a region. We let $R = \{r \in O \mid kind(r) = \text{reg}\}$ be the set of regions and $D = \{d \in O \mid kind(d) = \text{dls}\}$ be the set of DLSs of G .
- The nesting level of objects and values $level : O \cup V \rightarrow \mathbb{N}$.
- The size of objects $size : O \rightarrow \mathbb{N}$.
- The minimum length of DLSs $len : D \rightarrow \mathbb{N}$.
- The validity of objects $valid : O \rightarrow \mathbb{B}$.
- The head, next, and prev field offsets of DLSs $hfo, nfo, pfo : D \rightarrow \mathbb{N}$.
- H is a partial edge function $O \times \mathbb{N} \times \mathbb{T} \rightarrow V$ 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 $V \rightarrow \mathbb{Z} \times \mathbb{S} \times O$ 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 $tg = \text{reg}$ iff $o \in R$. Here, of is an offset wrt the base address of o .⁶ 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. 1.4 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 .

Consistent Symbolic Memory Graphs

In the following, we assume working with so-called consistent SMGs only.⁷ In particular, we call an SMG $G = (O, V, \Lambda, H, P)$ *consistent* iff the following holds:

⁶ Note that the offset can even be negative, which happens, e.g., when traversing a Linux list.

⁷ All the later presented algorithms will be such that they produce a consistent SMG when they are applied on a consistent SMG (or SMGs). Therefore, since our analysis will start from a consistent SMG, there is no need to check the consistency on the fly.

- *Basic consistency of objects.* The null object is invalid, has size and level 0, and its address is 0, i.e., $\text{valid}(\#) = \text{false}$, $\text{size}(\#) = \text{level}(\#) = 0$, and $0 \xrightarrow{0, \text{reg}} \#$. All DLSs are valid, i.e., $\forall d \in D : \text{valid}(d)$. Invalid regions have no outgoing edges.
- *Field consistency.* Fields do not exceed boundaries of objects, i.e., $\forall o \in O \forall of \in \mathbb{N} \forall t \in \mathbb{T} : H(o, of, t) \neq \perp \Rightarrow of + \text{size}(t) \leq \text{size}(o)$.
- *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. $H(d, nfo(d), \text{ptr}) = a_n$ and $H(d, pfo(d), \text{ptr}) = a_p$ (cf. Fig. 1.4). The next pointer is always stored in memory before the prev pointer, i.e., the next and prev offsets are s.t. $\forall d \in D : nfo(d) < pfo(d)$. 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., $\forall a \in A : tg(P(a)) \in \{\text{fst}, \text{lst}\} \Rightarrow of(P(a)) = hfo(d)$ where $d = o(P(a))$.⁸ Finally, in a consistent SMG there is no cyclic path containing 0+ DLSs (and their addresses) only since its semantics would include an address not referring to any object.
- *Nesting consistency.* Each nested object $o \in O$ of level $l = \text{level}(o) > 0$ has precisely one *parent DLS*, denoted $\text{parent}(o)$, that is of level $l - 1$ and there is a path from $\text{parent}(o)$ to o whose inner nodes are of level l and higher (i.e., more nested) only—e.g., in Fig. 1.4, 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. 1.4), i.e., $\forall a \in A : tg(P(a)) \in \{\text{fst}, \text{lst}, \text{reg}\} \Rightarrow \text{level}(a) = \text{level}(o(P(a)))$. On the other hand, addresses with the `all` target go up one level in the nesting hierarchy, i.e., $\forall a \in A : tg(P(a)) = \text{all} \Rightarrow \text{level}(a) = \text{level}(o(P(a))) + 1$ (cf. a_d in Fig. 1.4). 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. 1.4, such an edge leads from the 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 $\text{level}(o) > \text{level}(o')$, $tg(P(a)) = \text{all}$ iff $o' = \text{parent}^k(o)$ for some $k \geq 1$.

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 $\text{len}(d) = n$, 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 offset, target specifier, and object of the triple $P(v)$, respectively. Further, for $o \in O$, we let $H(o) = \{H(o, of, t) \mid of \in \mathbb{N}, t \in \mathbb{T}, H(o, of, t) \neq \perp\}$. We let $A = \{v \in V \mid P(v) \neq \perp\}$ 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 .

Symbolic Program Configurations

Let $GVar$ be a finite set of global variables, $SVar$ a countable set of stack variables such that $GVar \cap SVar = \emptyset$, and let $Var = GVar \cup SVar$. A *symbolic program*

⁸ The last two requirements are not necessary, but they significantly simplify the below presented algorithms (e.g., the DLS materialisation given in Section 1.2.4).

configuration (SPC) is a pair $C = (G, \nu)$ where G is an SMG with a set of regions R , and $\nu : \text{Var} \rightarrow R$ is a finite injective map such that $\forall x \in \text{Var} : \text{level}(\nu(x)) = 0 \wedge \text{valid}(\nu(x))$. Note that ν evaluates to 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 $x \in \text{GVar}$ a *static object*, and each object o such that $\nu(x) = o$ for some $x \in \text{SVar}$ 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.

Special Kinds of SMGs and SPCs

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 $A_F = \{H(o, of, \text{ptr}) \mid of \in F\}$ and nodes that are reachable from o through A_F only. Finally, the sub-SMG of G *nested below* $d \in D$, denoted $\hat{G}_d$, is the smallest sub-SMG of G including d and all objects and values of level higher than $\text{level}(d)$ that are reachable from d via paths that, apart from d , consist exclusively of objects and values of a level higher than $\text{level}(d)$.

1.2.4 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 that 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 the following two transformations any number of times: (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). Moreover, note that the operations of materialisation and removal are used not only to define the semantics, but they will later be used within symbolic execution of C statements over SMGs (and hence as a part of the actual SMG-based analysis) too.

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., $H(d, pfo(d), \text{ptr}) = a_p$ and $H(d, nfo(d), \text{ptr}) = a_n$. The

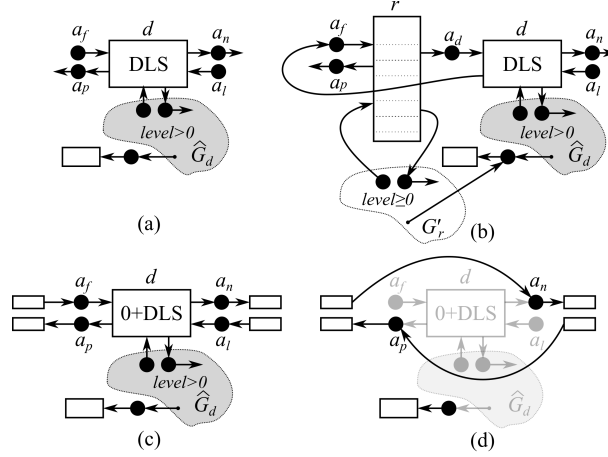


Fig. 1.5: Materialisation of a DLS: (a) input, (b) output (region r got materialised from DLS d). Removal of a DLS: (c) input, (d) output. Sub-SMGs $\widehat{G}_d$ and G'_r are highlighted without their roots.

DLS d can be *materialised* as follows—for an illustration of the operation, see the upper part of Fig. 1.5:

1. *Materialisation of the first region and its nested sub-SMG.* 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 $\text{size}(r) = \text{size}(d)$, $\text{level}(r) = 0$, and $\text{valid}(r) = \text{true}$. The nesting level of each object and value in G'_r (apart from r) is decreased by one.
2. *Interconnection of the materialised region and the rest of the segment.* Let $a_f \in A$ be the address pointing to the beginning of d , i.e., such that $P(a_f) = (\text{hfo}(d), \text{fst}, d)$. 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. $P(a_f) = (\text{hfo}(d), \text{reg}, r)$, $H(r, \text{pfo}(d), \text{ptr}) = a_p$, $H(r, \text{nfo}(d), \text{ptr}) = a_d$, $P(a_d) = (\text{hfo}(d), \text{fst}, d)$, and $H(d, \text{pfo}(d), \text{ptr}) = a_f$.
3. *Interconnection of the materialised sub-heap and non-nested objects.* 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., $\text{level}(H(o, of, t)) = 0$, 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. *Adjusting the minimum length of the rest of the segment.* If $\text{len}(d) > 0$, $\text{len}(d)$ is decreased by one.

Next, let $d \in D$ be a DLS as above with the additional requirement of $\text{len}(d) = 0$ 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. 1.5:

- (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.

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

!!

Concrete Memory Images

The semantics of an MG $G = (R, V, \Lambda, H, P)$ is the set $MI(G)$ of *memory images* $\mu : \mathbb{N} \rightarrow \{0, \dots, 255\}$ mapping *concrete addresses* to *bytes* such that there exists a function $\pi : R \rightarrow \mathbb{N}$, 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., $\forall r_1, r_2 \in R : valid(r_1) \wedge valid(r_2) \Rightarrow \langle \pi(r_1), \pi(r_1) + size(r_1) \rangle \cap \langle \pi(r_2), \pi(r_2) + size(r_2) \rangle = \emptyset$.
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 $r_1 \xrightarrow{of_1, \text{ptr}} a \xrightarrow{of_2, \text{reg}} r_2$ in H and P , resp., $addr(bseq(\mu, \pi(r_1) + of_1, size(\text{ptr}))) = \pi(r_2) + of_2$ where $bseq(\mu, p, size)$ is the sequence of bytes $\mu(p)\mu(p+1)\dots\mu(p+size-1)$ for any $p, size > 0$, and $addr(\sigma)$ 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 $v \neq 0$, $bseq(\mu, \pi(r_1) + of_1, size(t_1)) = bseq(\mu, \pi(r_2) + of_2, size(t_2))$.
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 $0 \leq i < size(t)$.

For an SMG G , we let $MI(G) = \bigcup_{G' \in MG(G)} MI(G')$. Note that it may happen that no concrete values satisfying the needed constraints exist. 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. Finally, note that, for simplicity, we assume that each sequence of bytes of length $size(t)$ corresponds to some instance of the type t , which can be an indeterminate value in the worst case.

1.3 Operations on SMGs

In this section, we propose algorithms for all operations on SMGs that are needed for their application in program verification. In particular, we discuss data reinterpretation (which is used for reading and writing from/to SMGs), join of SMGs (which we use for entailment checking and as a part of the abstraction too), abstraction, inequality checking, and symbolic execution of C programs. More details can be found in Sections 1.8–1.11.

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

1.3.1 Data Reinterpretation

SMGs allow fields of a single object to overlap and to even have the same offset and size, in which case they are 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. [36]) and beyond the scope of this work. 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.⁹

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 pair (G', v) where G' is an SMG with a set of has-value edges H' such that (1) $H'(o, of, t) = v \neq \perp$ and (2) $MI(G) = MI(G')$. The operator thus preserves the semantics of the SMG G but ensures that it contains a has-value edge for the field being read. This edge can lead to a value already present in G but also to a new value derived by the operator from the edges and values existing in G . In the extreme case, a fresh, completely unconstrained value node can be added, representing an unknown value, which can, however, become constrained by further program execution. In other words, read reinterpretation installs a new view on some part of the object o without modifying the semantics of the SMG in any way.

For the particular case of dealing with nullified memory, we use the following concrete read reinterpretation (cf. Section 1.8.1). 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 this is the most precise read reinterpretation that is possible—from

⁹ Apart from the nullified blocks, our implementation also supports tracking of uninitialized blocks of memory and certain manipulations of null-terminated strings (cf. Section 1.5.2).

the point of view of reading nullified memory—with the current support of types and values in SMGs.

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. Section 1.8.2, which includes 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.

1.3.2 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., $MI(G) = MI(G_1) = MI(G_2)$. 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 Section 1.9.

The basic idea of our join algorithm, illustrated in Fig. 1.6, is the following. The algorithm simultaneously traverses a given pair of source SMGs and tries to join each

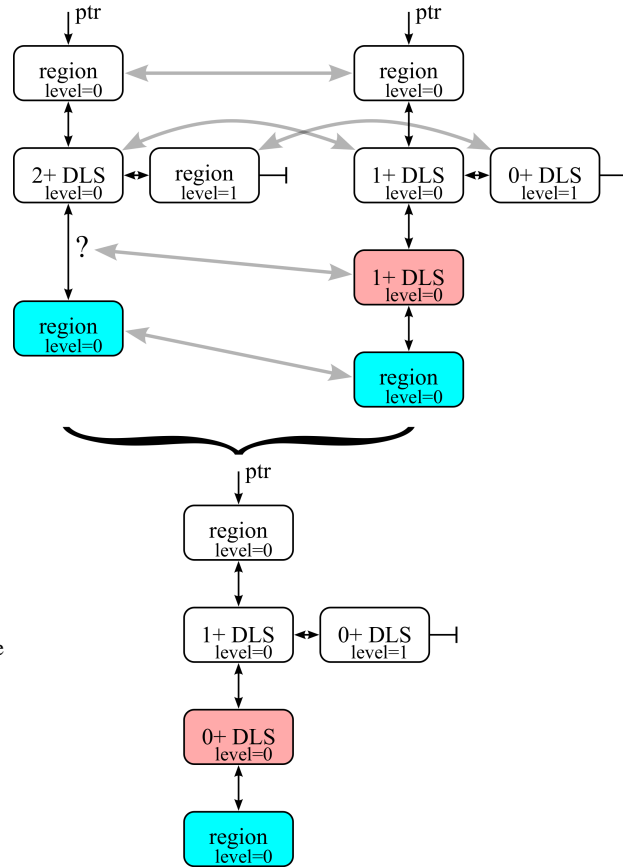


Fig. 1.6 An illustration of the basic principle of the join algorithm. In the figure, a simplified notation for describing SMGs is used, from which value nodes have been left out. The pair of input SMGs is at the top. The gray arrows show the pairs of objects joined by the algorithm during the simultaneous traversal of the input SMGs. The resulting SMG is at the bottom.

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.¹⁰

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. It is also possible to join DLSs with regions that may be approximated as 1+ DLSs for that purpose. 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

¹⁰ Two separately allocated objects are always different, values are only possibly different. Not to restrict the semantics, different objects or (possibly) different values cannot be changed into equal objects or values. Equal values could be changed into possibly different ones, but we currently do not allow this either since this would complicate the algorithm, and we did not see any need for that in our case studies.

a DLS with the 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 since the difference 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 Section 1.9.1, 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.¹¹ 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 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

¹¹ 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.

the successors of the objects being joined). 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 $\approx$. 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.

1.3.3 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 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 that is 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. Section 1.9.8), 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 $\{nfo_c, pfo_c\}$ -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 .

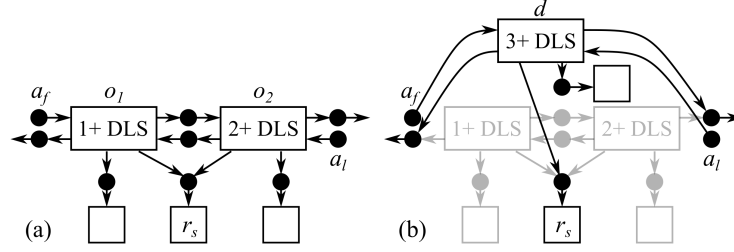


Fig. 1.7: The elementary merge operation: (a) input (b) output.

Candidate DLS Entries

A quadruple $(o_c, hfo_c, nfo_c, pfo_c)$ where $o_c \in O$ and $hfo_c, nfo_c, pfo_c \in \mathbb{N}$ such that $nfo_c < pfo_c$ 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 $H(o_c, nfo_c, \text{ptr}) = a_1$, $P(a_1) = (hfo_c, tg_1, o)$ for $tg_1 \in \{\text{fst}, \text{reg}\}$, $H(o, pfo_c, \text{ptr}) = a_2$, and $P(a_2) = (hfo_c, tg_2, o_c)$ for $tg_2 \in \{\text{lst}, \text{reg}\}$.

Longest Mergeable Sequences

The *longest mergeable sequence* of objects given by a candidate DLS entry $(o_c, hfo_c, nfo_c, pfo_c)$ 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. Section 1.10): (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 $\{nfo_c, pfo_c\}$ -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. 1.7 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 . Further, assume that o_1 is either the first object in the sequence to be merged 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, which is a part of our extended join operator (cf. Section 1.9.8). The sub-SMG nested below d is created using the extended join algorithm too. 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 $P(a_f) = (o_1, hfo_c, tg)$ for some $tg \in \{\text{fst}, \text{reg}\}$, then P is changed such that $P(a_f) = (d, hfo_c, \text{fst})$. Similarly, if there is $a_l \in A$ such that $P(a_l) = (o_2, hfo_c, tg)$ for some $tg \in \{\text{lst}, \text{reg}\}$, then P is changed such that $P(a_l) = (d, hfo_c, \text{lst})$. 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.

The Top-Level Abstraction Algorithm

Assume we are given an SMG G , and a candidate DLS entry $(o_c, hfo_c, nfo_c, pfo_c)$ defining the longest mergeable sequence of objects $\sigma = o_1 o_2 \dots o_n$ in G of length $|\sigma| = n \geq 2$. We define the *cost* of merging a pair of objects o_1, o_2 , denoted $\text{cost}(o_1, o_2)$, as follows. First, $\text{cost}(o_1, o_2) = 0$ iff the $\{nfo_c, pfo_c\}$ -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 , i.e., when not distinguishing whether o_1, o_2 are regions or DLSs as well as ignoring the minimum length constraints on o_1, o_2). This is indicated by the $\approx$ status returned by the modified join algorithm applied on G_1, G_2 . Further, $\text{cost}(o_1, o_2) = 1$ iff G_1 entails G_2 , or vice versa, which is indicated by the status $\sqsupset$ or $\sqsubset$. Finally, $\text{cost}(o_1, o_2) = 2$ iff G_1 and G_2 are incomparable, which is indicated by status $\bowtie$. The cost of merging a sequence of objects $\sigma = o_1 o_2 \dots o_n$, denoted $\text{cost}(\sigma)$, is defined as the maximum of $\text{cost}(o_1, o_2)$, $\text{cost}(o_2, o_3)$, ..., $\text{cost}(o_{n-1}, o_n)$.

Our abstraction is parameterized by associating each cost $c \in \{0, 1, 2\}$ with the *length threshold*, denoted $\text{lenThr}(c)$, defining the minimum length of a sequence of mergeable objects allowed to be merged for the given cost. Intuitively, the higher the cost is, 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 1.5, we, in particular, found as optimal the setting $\text{lenThr}(0) = \text{lenThr}(1) = 2$ and $\text{lenThr}(2) = 3$. 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 $|\sigma| < \text{lenThr}(\text{cost}(\sigma))$ 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 repeats until there is no sequence that can be merged taking its length and cost into account.

1.3.4 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 we allow it to fail in some exceptional cases (e.g., when comparing addresses out of bounds of two distinct objects) in order not to harm its efficiency. The basic idea of the algorithm is as follows (cf. Section 1.11): 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 a sequence of 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.

1.3.5 A Note on Symbolic Execution over SMGs/SPCs

The symbolic execution algorithm based on SPCs is similar to [2]. 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 1.2.4) is performed so that the actual program statements are always executed over concrete objects.¹² 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 1.2.4) 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 1.3.4 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 minimum length in the other (cf. Section 1.11).

To reduce the number of SPCs generated by the symbolic execution, the *join* operator introduced in Section 1.3.2 can be used to join an SPC that was newly generated for some particular program location with some SPC generated for that location sooner (e.g., joining a region with a 2+ DLS into a single 1+ DLS), hence

¹² A DLS can be materialised from its last element too, which is analogous to the materialisation from the first element as described in Section 1.2.4.

reducing the number of SPCs remembered for that location (and explored from that location). Trading speed for precision, the operator can be applied at every control location, at the beginning of every basic block, at every loop head location, or not at all. Apart from the join operator, the *abstraction* mechanism from Section 1.3.3 is to be applied to ensure termination on unbounded list structures. Again, the use of the abstraction may be restricted as in the case of the join operator (it needs not even be used if no unbounded data structures are used). As the terminating criterion, one can use isomorphism or entailment checking between a newly generated SPC and those already known for a given program location. Checking for *isomorphism* can be done using the join operator of Section 1.3.2, making sure that it succeeds and returns the $\simeq$ status. Checking for *entailment* can also be done using the proposed join operator, this time checking for the $\sqsubseteq$ or $\sqsupseteq$ status (allowing one to discard either a sooner generated SPC or the new SPC).

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).

1.3.6 Running Example

We now illustrate some of the main presented concepts on a running example. In particular, we consider the C code shown in Fig. 1.8. For its five designated locations L1–L5, Fig. 1.9 shows the SPCs generated by our analysis. We use a simplified notation for the SPCs similar to that already used in Fig. 1.6. The “bar”-terminated edges denote pointers to NULL. We draw edges corresponding to *prev* pointers to the left of objects and those corresponding to *next* pointers to the right of objects. Bidirectional edges denote a pair of *prev/next* pointers.

The way the different SPCs are obtained is discussed below. However, first, note that all objects of all SPCs are at level 0 (i.e., there are no nested sub-SMGs here). Assuming that the code is compiled for a 64-bit architecture with 8B-long pointers, all objects are of size 16B. The *next* pointers start at offset 0 while the *prev* pointers at offset 8, and they all have the target offset 0.

The first SPC that gets generated at the location L1 is denoted as L1.1. It sets both the *l* and *x* program variables to NULL. From L1.1, the SPC L2.1 is generated at the location L2. As we can see, it consists of a single region with the *prev* field implicitly nullified (as *calloc* was used). L2.1 is propagated as the second SPC for

```

1  #include <stdlib.h>                                // calloc(), free()
2
3  struct list_item {
4      struct list_item *next;
5      struct list_item *prev;
6  };
7
8  int main(void)
9  {
10     struct list_item *l = NULL;                    // pointer to a DLL
11     struct list_item *x = NULL;                    // auxiliary pointer
12
13     while (__VERIFIER_nondet_int() != 0) {
14         L1:
15         x = calloc(sizeof(struct list_item),1);
16         x->next = l;
17         if (l != NULL)
18             l->prev = x;
19         l = x;
20         L2:
21     }
22     L3:
23     while (l != NULL) {
24         x = l;
25         l = l->next;
26         free(x);
27         L4:
28     }
29     L5: return 0;
30 }

```

Fig. 1.8: A running example creating and destroying a DLL.

the location L1, i.e., it becomes L1.2. The join operator presented above cannot join L1.1 and L1.2, and so both of them are kept (for the time being). Through another iteration of the first `while` loop, L2.2 is obtained from L1.2 in a natural way.

Now, several interesting changes happen. First, when closing the second iteration of the first `while` loop, L2.2 gets abstracted to an SPC containing a single 2+ DLS that is NULL-terminated at both of its ends: indeed, L2.2 contains an uninterrupted sequence of two equally-sized regions linked in the fashion of a DLL. Second, the resulting SPC can be joined both with L1.2, yielding an SPC consisting of a 1+ DLS, and then with L1.1, yielding the new SPC L1.3 containing a 0+ DLS, which is kept as the sole SPC at the location L1.

The SPC L1.3 is once more passed through the first `while` loop. Notice that the `if (l != NULL)` statement will split the 0+ DLS present in L1.3 to two cases: the SPC `l=x=NULL` and an SPC containing a 1+ DLS. Only the latter SPC is then subject to the `l->prev = x` statement. For this statement to be symbolically executed, the 1+ DLS is materialised to a sequence of a region, pointed to by `l`, and a 0+ DLS segment linked to it through a pair of `next/prev` pointers. By linking the region newly allocated by `calloc` and pointed to by `x` with the materialised region pointed to by `l`, and by moving the `l` pointer, the L2.3 SPC will eventually arise at the end

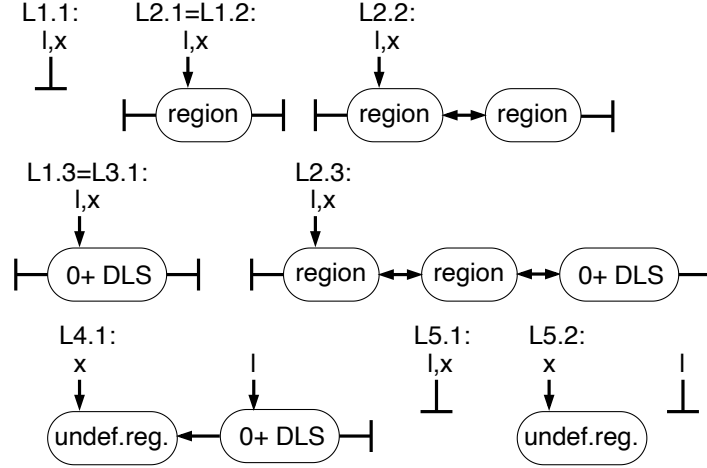


Fig. 1.9: Some of the SPCs generated during analysis of the code from Fig. 1.8.

of the body of the first `while` loop. $L2.3$ entails $L2.2$, and so the latter does not need to be kept at the location $L2$. More importantly, by abstraction applied when going back to the loop header, $L2.3$ will be transformed into an SPC consisting of a $2+$ DLS, which will then be found to be entailed by $L1.3$.

Assuming that the analysis explores the program such that the first `while` loop is first completely explored and only then the analysis continues further on, the only SPC generated for location $L3$ is $L3.1$, which is the same as $L1.3$.

Due to the condition of the second `while` loop, $L3.1$ is split to two SPCs: once again the SPC $l=x=NULL$, which then appears as the SPC $L5.1$ at the end of the program, and an SPC containing a $1+$ DLS. Only the latter enters the first iteration of the second `while` loop. For the $l = l \rightarrow next$ statement to be symbolically executed, the $1+$ DLS is materialised into a sequence of a region, pointed to by both l and x , and a $0+$ DLS. The l pointer is then moved to point to the beginning of the $0+$ DLS. The x -pointed region is subsequently freed, after which all its outgoing edges are removed, and the region itself is marked as undefined, taking us to the $L4.1$ SPC.

The condition of the second `while` loop splits $L4.1$ to two cases again: the case where $l=NULL$, yielding the SPC $L5.2$ at the end of the program, and an SPC where l points to a $1+$ DLS whose `prev` pointer points to the freed region pointed to by x . For the $l = l \rightarrow next$ statement to be symbolically executed, the $1+$ DLS is materialised to a sequence of an l -pointed region and a $0+$ DLS. The `prev`-pointer of the l -pointed region points to the previously freed region (with x already pointing to the same region as l), and its `next` pointer points to the $0+$ DLS. Subsequently, l is moved to point to the beginning of the $0+$ DLS, and the x -pointed region is freed. Due to that, its outgoing edges are removed, and the previously freed region, which is now completely inaccessible, is removed too. Hence, $L4.1$ is again obtained.

1.4 Extensions of SMGs

Next, we point out that the above introduced 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 implemented in the Predator tool.

Explicit Non-equivalence Relations

When several objects have the same concrete value stored in some of their fields, this is expressed by making 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*.¹³ Such a relation can be introduced when the analysis proceeds to some branch of a conditional statement such that the condition of the statement (negated for the false branch) implies the non-equivalence relation. The introduced non-equivalence relations are then to be taken into account in further operations, including symbolic execution of conditional statements and the join operator. In the latter case, when joining two SMGs where some non-equality edge exists in one of them only, it may be dropped for the price of appropriately worsening the resulting join status.

Clearly, SMGs can be quite naturally extended by allowing more predicates on data, which is, however, beyond the scope of this work (up to a small extension by tracking not only the 0 value but also intervals with constant bounds that is mentioned below) and has so far been not implemented in the Predator tool either.

Checking Equivalence of Valid and Invalid Objects

Testing of inequality described in Section 1.3.4 concerns inequality of pointers pointing to different valid objects, null and non-null objects, the same object with different offsets, or to different ends of a doubly-linked list segment with at least two elements. However, there is one more way how inequality can be established, namely, when comparing pointers to a valid region and to an invalid region where the invalid one was allocated later than the valid one (which we can check due to the way the objects are numbered in Predator). Indeed, in such a case, both of the regions must have existed at the same time, the valid object is continuously valid since then, and the two objects must lie on different addresses since the address of a continuously valid object could not have been recycled.

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

¹³ This is similar to the equality and non-equality constraints in separation logic, but only non-equality constraints are kept explicit.

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 pointers¹⁴. 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.

Integer Constants and Intervals

The basic SMG notion allows one to express that two fields have the same value, which is represented by the corresponding has-value edges leading to the same value node, or that their values differ, which is represented 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 integer numbers. These can be respected by the join algorithm up to some given bound and replaced by the unknown value when the bound is exceeded (as done in Predator). Alternatively, they can be abstracted to intervals with bounds being concrete integer constants up to some bound or plus/minus infinity (also supported by Predator), or some other abstract numerical domains may be used (unsupported by Predator).

¹⁴ If a DLL consists of exactly one node, the value of its next pointer is equal to the value of its prev pointer. There is no point in distinguishing them, so we call them both neighbour pointers.

1.5 Implementation

We have implemented the above described algorithms (including most of the mentioned extensions) in the Predator tool.¹⁵ Below, we describe the architecture of the tool—in fact, a *tool suite* centred around the Predator analysis kernel—and various further extensions, optimisations, and implementation details related to it. The description is partly based on the tool paper [22].

1.5.1 Architecture

The architecture of the Predator tool suite is shown in Fig. 1.10. Its *front end* is based on the *Code Listener* (CL) infrastructure [17] that can accept input from both the GCC and Clang/LLVM compilers. CL is connected to both GCC and LLVM as their plug-in (or “pass”).

When used with *GCC*, CL reads in the GIMPLE intermediate representation (IR) from GCC and transforms it into its own *Code Listener IR* (CL IR), based on simplified GIMPLE. The resulting CL IR can be *filtered*—currently there is a filter that replaces `switch` instructions by simple conditions—and stored into the code storage. When used with *Clang/LLVM*, CL reads in the LLVM IR and uses LLVM’s AddressSanitizer for a use-after-scope instrumentation of the lifetime of variables, removes LLVM `switch` instructions, and (optionally) simplifies the IR through a number of other *filters* in the form of LLVM optimisation passes, both LLVM native or newly added. These filters can in-line functions, split composed initialisation of global variables, and/or change memory references to register references (removing unnecessary `alloca` instructions). These transformations can be used independently of Predator to simplify the LLVM IR to have a simpler starting point for developing new analysers. Moreover, CL offers a *listeners architecture* that can be used to further process CL IR. Currently, there are listeners that can print out the CL IR or produce a graphical form of the control flow graphs (CFGs) present in it.

The *code storage* stores the obtained CL IR and makes it available to the Predator verifier kernel through a special API. This API allows one to easily iterate over the types, global variables, and functions defined in the code. For each function, one can then iterate over its parameters, local variables, and its CFG. Of course, other verifier kernels than the one of Predator can be linked to the code storage.

1.5.2 The Kernel of Predator

The kernel of Predator (written in C++ like its front end) implements a form of an abstract interpretation loop over the SMG domain where the widening takes into account the newest computed SMG for a given location only and is based on the abstraction mechanism from Section 1.3.3.

¹⁵ <https://www.fit.vutbr.cz/research/groups/verifit/tools/predator/>

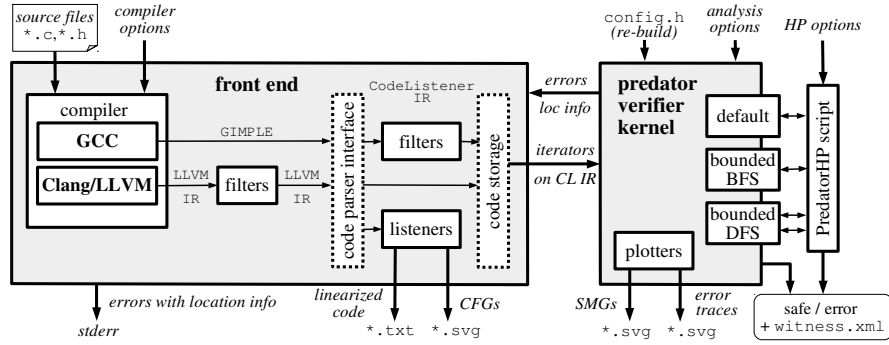


Fig. 1.10: Architecture of the Predator tool suite.

Programs To Be Verified

A program to be verified by Predator must be *closed* in that it must allocate and initialize all the data structures used. 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. From the C standard, the following functions are currently included and properly modelled wrt proving memory safety (there are a few more, GCC-/LLVM-specific ones):

- `malloc`, `alloca`, `calloc`, `free`, and `realloc`;
- `exit` and `abort`;
- `memset`, `memcpy`, and `memmove`;
- `printf` and `puts`; and
- `strlen`, `strncpy`, and `strcmp`.

Models of further functions can be added by the user for the price of recompiling the analyser. Predator also provides several built-in functions that are specific to its verification approach, e.g., functions to dump SMGs or program traces to files.

Interprocedural Features

Predator supports indirect calls via *function pointers*, which is necessary for verification of programs with callbacks (e.g., Linux drivers). Predator does not support *recursive programs*, but it implements an *inter-procedural analysis* based on *function summaries* [33]. The summaries consist of pairs of (sub-)SPCs that appeared at the input/output of a given function during the so-far performed analysis. Depending on the configuration of Predator, input parts of the summaries are created either (1) by taking the entire SPC encountered at a function call or (2) by carving out the part of the SPC that is reachable from function parameters and global variables. The summaries are stored in a call cache. When testing whether a summary for a call with a certain input SPC has already been created (or covered by another SPC), Predator can compare the current input SPC (or its relevant part) with those stored in the call cache either by isomorphism or entailment, depending on its configuration.

Predator monitors how many consecutive cache misses are encountered for each function after the last cache hit, and if that number gets above some configurable threshold, the cache will not be used for the given function.

Regions for *stack variables* are created automatically as needed and destroyed as soon as they become dead according to a *static live variables analysis*¹⁶, performed before running the symbolic execution. When working with *initialised variables*¹⁷, we take advantage of our efficient representation of nullified blocks—we first create a has-value edge $o \xrightarrow{0, \text{char}[\text{size}(o)]} 0$ for each initialised variable represented by a region o , then we execute all explicit initialisers, which themselves automatically trigger the write reinterpretation. The same approach is used for `calloc`-based heap allocation. Thanks to this, we do not need to initialise each structure member explicitly, which would incur a significant overhead.

Various Optimisations

As an optimisation, a *copy-on write* approach is used when creating new SMGs by modifying the already existing ones. Also, the algorithms for abstraction and join implemented in Predator use some further optimisations of the basic algorithms described in Section 1.3. 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 a 0/1 abstract object) are disabled when joining SMGs while enabled when merging nodes during abstraction.

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 equality¹⁸ only, which makes the comparison way faster, especially in case a pair of SMGs cannot be joined. Similarly, the abstraction is by default used at loop boundaries only 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).

Non-Pointer Data

Predator’s support of *non-pointer data* is currently limited. As mentioned already above, Predator can track *integer data* precisely up to a predefined bound (± 10 by

¹⁶ If a program variable is referenced by a pointer, its destruction needs to be postponed.

¹⁷ According to the C99 standard, all static variables are initialised, either explicitly or implicitly.

¹⁸ The join algorithm can be easily restricted to check for equality only—if any action that would imply inequality of the input SMGs is about to be taken, the join operation fails immediately.

default), and once the bound is reached, the values are abstracted out. Optionally, Predator can use intervals with constant bounds (which may be widened to infinity) while also tracking some simple dependences between intervals, such as a shift by a constant and a multiplication by -1 . Reinterpretation is used to handle *unions*. *String* and *float constants* can be assigned, but any operations on these data types conservatively yield an undefined value. *Arrays* are handled as allocated memory blocks with their entries accessible via field offsets much like in the case of structures.

Dealing with Integer Intervals

As can be seen from the previous text, integer intervals may arise in Predator in multiple contexts: (1) One can get pointers with *interval-based target offsets*, e.g., by address alignment or when joining pointers pointing to the same object but with different offsets. (2) Integer variables can get interval-based values, e.g., by joining results of the analysis on different branches or when restricting unknown values by some conditions on integer variables. (3) Finally, there can also arise memory regions of *interval-based size*, e.g., when allocating structures or arrays whose size is given by a variable whose value is given by an interval or when joining SMGs where a pair of corresponding memory regions differs in the size. However, the introduction of intervals can not only lead to some loss of precision (by loosing relations of the individual values in the interval with values of other variables), but Predator currently also imposes many restrictions on how the intervals can subsequently be used. For example, it does not allow one to dereference interval-sized regions, due to which the basic version of Predator behaves poorly when analysing programs with structures or arrays of unknown size (dependent on program input or abstracted away).

To at least partially improve on this situation, Predator has been extended by the following pragmatic heuristic. Namely, whenever it hits a conditional statement that would normally yield an interval value with fixed bounds (e.g., executing the statement `if (n >= 0 && n < 10)` where `n` is unconstrained before the statement), it will split the further run of the analysis into as many branches as the number of values in the interval is, each of them evaluating for a concrete value from the interval. After the split, no interval-based allocations and dereferences (nor any other problematic interval-based operations), which Predator would fail on, happen. Though this solution is rather simple, it works nicely in some cases. Of course, it can lead to a memory explosion when the intervals are large, but then the analysis fails with no answer as it would fail without the heuristic too.

The above modification of Predator concerns dealing with intervals with finite bounds. For the case when one of the bounds is infinite, Predator has been extended to *sample* the interval and perform the further analysis with the sampled values. The sampling is done by simply taking some number of concrete values from the given interval starting/ending with the bound that is fixed (intervals with both bounds infinite correspond to the unknown value). The number of considered samples is currently set to 3. Of course, this strategy cannot be used to soundly verify correctness of programs, and so it is used for detecting bugs only.

Errors Sought and Error Reporting

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. Memory leakage checks are optimized by collecting sets of lost addresses for each operation that can introduce a memory leak (write reinterpretation, `free`, etc.), followed by checking whether reachability of allocated objects from program variables depend on the collected addresses. 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. Predator also looks for violations of *assertions* written in the code.

Predator can also detect *invalid dereferences of objects local to a block* from outside of the block. For that, it tracks usage of the *clobber* instruction of CL IR, which is used to terminate the life time of local variables of code blocks. Whenever the instruction is encountered, the concerned memory region is marked as deallocated, and further dereferences of that region are detected as erroneous.

Predator reports discovered errors together with their location in the code in the standard GCC format, and so they can be displayed in standard editors or IDEs. Predator can also produce error traces in a textual or graphical format or in the XML format of SV-COMP (cf. Section 1.6.2). Predator 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 the SMG is removed, and the symbolic execution then continues.

Options

The kernel supports many *options*. Some of them can be set in the `config.h` file (requiring the kernel to be re-compiled) and some when starting the analysis (cf. the tutorial in Section 1.12). Apart from various debugging options and some options mentioned already above, one can, e.g., decide whether the abstraction and join should be performed after every basic block or at loop points only (abstraction can also be performed when returning from function calls). One can specify the maximum call depth, choose between various search orders, switch on/off the use of function summaries and destruction of dead local variables, control error recovery, and control re-ordering of lists of SMGs kept for program locations (based on their hit ratio) and/or their pruning wrt entailment and their location in CFGs.

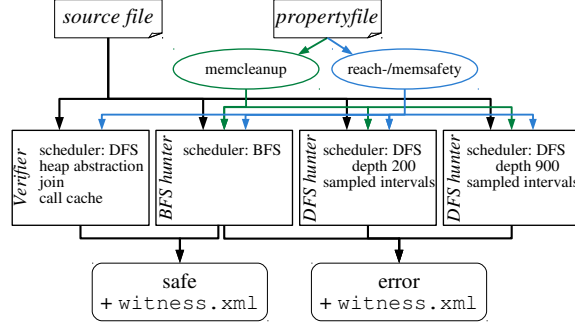


Fig. 1.11: The flow of control of Predator Hunting Party as used in SV-COMP'19.

1.5.3 Predator Hunting Party

The *Predator Hunting Party* (or *PredatorHP* for short) uses the original Predator analyser to prove programs safe while at the same time using several bounded versions of Predator for *bug hunting*. PredatorHP, whose flow of control is shown in Fig. 1.11, is implemented as a Python script that runs several instances of Predator in parallel and composes the results they produce into the final verification verdict.

In particular, PredatorHP starts four Predators: One of them is the original Predator that soundly overapproximates the behaviour of the input program—we denote it as the *Predator verifier* below. Apart from that, three further Predators, modified as follows, are started: Their join operator is reduced to joining SMGs equal up to isomorphism, they use no list abstraction, and they are also prohibited from using the call cache, which is otherwise used for implementing function summaries.¹⁹ Two of them use a bounded depth-first search to traverse the state space, and so we call them *Predator DFS hunters*. They use bounds of 200 and 900 GIMPLE instructions, respectively. The third of them—the *Predator BFS hunter*—uses a breadth-first search to traverse the state space. The DFS hunters are allowed to use the above described heuristic based on sampling intervals when looking for bugs in interval-sized memory regions.

If the Predator verifier claims a program correct, so does PredatorHP, and it kills all other Predators. If the Predator verifier claims a program incorrect, its verdict is ignored since it can be a false alarm (and, moreover, it is highly non-trivial to check whether it is false or not due to the involved use of list abstractions and joins). If one of the Predator DFS hunters finds an error, PredatorHP kills all other Predators and claims the program incorrect, using the trace provided by the DFS hunter who found the error as a violation witness.²⁰ One of the DFS hunters searches quickly for bugs with very short witnesses, and one then searches for longer but still not very long witnesses. If a DFS hunter claims a program correct, its verdict is ignored as it may be unsound. If a BFS hunter manages to find an error within the time budget,

¹⁹ The use of function summaries is prohibited since they may yield over-approximation, in particular, when testing applicability of summaries by entailment.

²⁰ The obtained trace can still be spurious due to abstraction applied on non-pointer data.

PredatorHP claims the program incorrect (without a time limit, the BFS hunter is guaranteed to find every error). If the BFS hunter finishes and does not find an error, the program is claimed correct. Otherwise, the verdict “unknown” is obtained.

The main strength of PredatorHP is that—unlike various bounded model checkers—it treats unbounded heap manipulation in a *sound* way. At the same time, it is also quite *efficient*, and the use of various concurrently running Predator hunters greatly decreases chances of producing *false alarms* (there do not arise any due to heap manipulation, the remaining ones are due to abstraction on other data types).

1.6 Experiments

In this section, we present results of experiments with Predator both outside of SV-COMP as well as within SV-COMP. In the latter case, we concentrate in particular on the 2019 edition of SV-COMP and on the influence of using PredatorHP as well as one of the later introduced optimisations of Predator, namely, that of dealing with intervals of values (Section 1.5.2).

1.6.1 Experiments with Predator Outside of SV-COMP

Already when SMGs and their implementation in Predator were first published in [19], Predator had been successfully tested on a number of case studies. Among them there were 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 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, Predator was also successfully tested on the driver code snippets distributed with SLayer [3] as well as on the *cdrom* driver originally checked by Space Invader [38]. In some of these programs, Predator identified errors not found by the other tools due to their more abstract (not byte-precise) treatment of memory [19].²¹

Further, we also considered two real-life low-level programs: 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 consider a fixed size of the arenas and check safety of repeated allocation and deallocation of blocks from arena pools as well as lists of arena pools. The blocks are of aligned size chosen randomly and ranging up to the arena size.

²¹ Invader did not check memory manipulation via array subscripts, and SLayer did not check size of the blocks allocated on the heap. The case studies of SLayer were later updated, and so they do not contain the identified problems any more.

For this purpose, a support for offset intervals as described above is 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 allows 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 (if compiled with the debug support). 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 we use a simplified test harness where the `lvm2` implementation of hash tables is replaced by the `lvm2` implementation of doubly-linked lists.

The original results of Predator, Invader, and SLAyer on the above described case studies are available in [19]. In Table 1.1, we present results on some of these case studies obtained from a wider selection of tools, complementing the originally considered tools with more tools selected out of those scoring well in heap-related categories of SV-COMP.²² All experiments were run on a computer with an Intel Core i7-3770K processor at 3.5 GHz with 32 GiB RAM. However, due to problems with installing some of the tools and due to some of them not being maintained any more, we had to consider different environments for running the experiments and consider tools made available in different years. Namely, Invader and SLAyer (marked by “*” in the table) were taken in their versions from years 2008 and 2011, respectively. Forester and CPA-kInd (marked by “**” in the table) were taken in their versions from SV-COMP’17 and run in a virtual machine with Ubuntu 16.04 (which restricted the available memory to 17.5 GB) and BenchExec 1.14 [6]. The remaining tools were taken in their versions from SV-COMP’19 and run in a virtual machine with Ubuntu 18.04 (with the available memory again restricted to 17.5 GB) and BenchExec 1.17.

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.

The results show that Predator provides the best results out of the considered tools. Indeed, the other considered tools often even crash, timeout, provide false positives, or even false negatives. Note also that PredatorHP provides worse results than the original Predator analyser on the chosen test cases. This is due to the stress on avoiding false positives (and not allowing the Predator verifier to announce errors) and due to running Predator hunters in parallel with the verifier (causing overall higher time consumption).

²² We have also considered MemCAD 1.0.0 [15], but we were unable to make it work on the chosen programs, and so we do not include it into the results.

Table 1.1: Experimental results on the Invader’s `cdrom` test case and selected Predator’s test cases showing either the verification time or one of the following outcomes: **FP** = false positive, **FN** = false negative, **F** = the expected error not found, another potential error reported (may be spurious: not checked), **T** = time out (900 s), **oom** = out of memory (15 GB), **seg** = segmentation fault, **x** = parsing problems, **xx** = internal error, **U** = inconclusive verification result (some form of “don’t know” explicitly produced by the tool), **?** = unknown failure of the tool (we were unable to closer diagnose the failure).

	cdrom_false-valid-deref.c	five-level-sll-destroyed-bottom-up.c	five-level-sll-destroyed-top-down.c	linux-dll-of-linux-dll.c	list-of-arena-pools-with-alignment.c	lvmcache-add_orphan_vginfo_false-valid-memtrack.c	merge_sort.c	merge_sort_false-unreach-call.c	scope-goto_false-valid-deref.c	cmp-freed_ptr_false-unreach-call.c		
Predator	0.63	1.17	0.12	0.11	0.80	0.80	1.32	0.26	0.26	0.06	0.04	0.04
PredatorHP	T	4.53	0.78	0.73	T	T	x	1.02	1.01	0.32	0.23	0.23
Invader*	FN	FP	FP	T	FP	-	x	FP	-	-	-	-
SLayer*	x	x	x	x	x	-	x	x	-	-	-	-
Forester**	xx	xx	T	T	xx	xx	x	xx	xx	xx	xx	U
CPA-Seq	T	T	T	T	T	T	U	T	T	75.78	T	x
CPA-kInd**	-	-	-	-	-	T	-	-	seg	T	-	x
DepthK	oom	T	T	?	T	T	?	T	T	1.11	?	?
2LS	xx	T	T	xx	xx	xx	xx	T	T	T	x	FN
ESBMC-kind	U	oom	oom	FP	oom	oom	x	T	T	0.68	FN	FN
Map2Check	?	?	?	T	?	?	?	T	?	?	T	?
UAutomizer	xx	T	T	xx	xx	xx	x	T	T	12.99	6.28	5.60
UKojak	xx	T	T	xx	xx	xx	x	T	T	14.18	6.52	5.64
UTaipan	xx	T	T	xx	xx	xx	x	T	T	13.55	6.13	5.72
DIVINE	10.88	T	T	T	U	U	x	FP	T	11.06	seg	FN
DIVINE-explicit	10.42	T	T	T	FP	xx	x	U	U	9.97	U	FN
Symbiotic	F	xx	xx	T	xx	FP	x	T	T	0.40	0.34	FN

Table 1.2: An analysis of results of PredatorHP and its component Predators on SV-COMP’19 benchmarks. The meaning of the columns is as follows: T = correct true, F = correct false, × = FN (false negative) / FP (false positive).

Benchmarks	PredatorHP			Predator verifier			Predator hunters											
	F	T	×	F	T	×	BFS			DFS 200			DFS 900			F	T	×
<i>MemSafety</i>	172	153		134	110	52/0	154	121		124	64	0/1	166	148	0/3			
Arrays	15	0		7	0		7	0		15	29	0/1	15	46	0/1			
Heap	81	67		83	59	16/0	76	51		55	29		76	40	0/2			
LinkedLists	28	66		27	32	36/0	27	49		16	3		28	48				
Other	16	20		17	19		16	21		12	3		15	14				
MemCleanup	32	-		0	-		28	-		26	-		32	-				
<i>ReachSafety</i>																		
Heap	71	129	4/0	70	93	43/0	67	108	4/0	54	56	1/0	71	108	1/0			

1.6.2 Predator and SV-COMP

Predator participated in SV-COMP since its beginning. In the first to third editions (i.e., from SV-COMP’12 to SV-COMP’14), the basic Predator analyser was involved. Since the 4th edition (SV-COMP’15), PredatorHP (see Section 1.5.3) was used. Its usage very significantly reduced the number of false alarms. On the other hand, although PredatorHP decreased the overall wall time of the verification (and some verification tasks were speeded up even in terms of the CPU time), the overall CPU time consumption increased.

In Table 1.2, we present an analysis of the performance of PredatorHP and its components, i.e., Predator hunters and the Predator verifier on benchmarks of several heap- and memory-related categories of SV-COMP’19. To get the data, we ran the experiments on a machine with an Intel Core i7-3770K processor at 3.5 GHz with 32 GiB RAM. As in the previous subsection, they were run in a virtual machine with Ubuntu 18.04 (with the available memory restricted to 17.5 GB) and BenchExec 1.17.

The use of PredatorHP allowed us to avoid 52 false alarms generated by Predator Verifier in the MemSafety category: Under PredatorHP, 40 of these benchmarks are successfully verified, 10 benchmarks end by a timeout, and the expected error is not reported in 2 benchmarks (instead a false alarm about another issue is raised). Likewise, in the ReachSafety category, 39 false alarms are avoided: 34 of the concerned benchmarks are successfully verified, 5 benchmarks end with a timeout (and there still remain 4 false alarms due to imprecise treatment of integers).

Moreover, the BFS hunter managed to verify various benchmarks with small finite state spaces, some of them with arithmetic operations on data fields of nodes of bounded-length lists (e.g., `list-simple/dll2*`, `list-simple/sl12*`, `ldv-memsafety/memleaks_test23_{1,3}_true-valid-memsafety.i`, or also `heap-data/shared_mem*`). In their case, the list abstraction is not needed, and, by not using it, we avoid interval abstraction on data fields, which causes imprecision and makes the Predator verifier to announce a false alarm (which PredatorHP ignores). In those cases where one needs to track data fields of unbounded lists, a timeout is hit since the verifier produces false alarms and hunters run forever (this happens, e.g., in `heap-data/process_queue_true-unreach-call.c`,

`heap-data/min_max_true-unreach-call.c`, `list-ext-properties/list-ext_flag_1_true-valid-memsafety.c`). Another cause of timeouts is then missing abstraction for non-list data structures (such as, e.g., trees in the benchmarks of `memsafety-ext/tree*`), which prevents both the hunters and the verifier from terminating.

As for the time consumption, the original Predator used 25,200 seconds of CPU/wall time (that are equal in this case) to handle all benchmarks considered in Table 1.2 in the MemSafety category. Further, it needed 14,700 seconds in the ReachSafety category. On the other hand, PredatorHP needed 17,100 seconds of wall time and 36,800 seconds of CPU time for the MemSafety category and 9,850 seconds of wall time and 20,000 seconds of CPU time for the ReachSafety category. This shows what we mentioned already at the beginning of the section, i.e., the fact that PredatorHP decreased the wall time but increased the CPU time.

However, even in terms of the CPU time, PredatorHP was faster in 6 correct and 6 erroneous benchmarks of the MemSafety category (with the correct cases and 3 of the erroneous cases handled by the BFS hunter, and with the 3 remaining erroneous cases handled by DFS hunters). In the ReachSafety category, PredatorHP was faster in 1 correct and 1 erroneous benchmark (both handled by the BFS hunter). The reason for that is that the list abstraction introduces some overhead that is not necessary in some cases: in particular, benchmarks on locks (`locks/test_locks_*`), and benchmarks with arrays (`ldv-regression/test23_{true,false}-unreach-call.c`), and benchmarks on lists with bounded-length and data fields (`list-ext-properties/simple-ext_1_true-valid-memsafety.c`, `list-ext2-properties/simple_and-skiplist_2lvl_false-unreach-call.c`).

Next, we briefly discuss influence of the pragmatic heuristics for dealing with intervals of values discussed in Section 1.5.2. The first of them replaces dealing with an interval of a bounded size by performing the verification independently for each element of the interval. This approach resolved unknown results of the basic Predator in the following two cases: `list-ext3-properties/sll_nondet_insert_true-unreach-call_true-valid-memsafety.c`, which inserts a node at a specific index given by an interval into a list of an unknown but finite length (namely, two to five elements), and `ldv-regression/test24_true-unreach-call_true-termination.c`, which indexes an array by an interval. Apart from that, due to the heuristic sampling of unbounded intervals, DFS hunters found errors in 9 test cases (`array-memsafety/*`) when looking for bugs in interval-sized memory regions allocated by `alloca`.

The efficiency of SMGs together with all the optimisations allowed Predator to win 7 gold medals, 5 silver medals, and 1 bronze medal at SV-COMP'12–19. In 2018 and 2019, it did not win any gold medal, which was caused to a large degree by that SV-COMP merged benchmarks targeting at programs with arrays with those focusing on pointers, dynamic memory, and dynamic linked data structures. However, even in SV-COMP'19, Predator was the first in the MemSafety-Heap and MemSafety-LinkedLists subcategory.

For SV-COMP'20, Predator was further improved in several relatively minor ways (e.g., its SMG-based analysis has been extended to support *memory reallocation* on

the heap, Predator’s handling of intervals has been fine-tuned, etc.)—for more details, see [32]. This allowed Predator to once again win a gold medal in the MemSafety category. Moreover, in SV-COMP’20, Predator has been integrated as an auxiliary tool of Symbiotic where it either helps it to prove some programs correct (if it manages to do so quickly enough), or information contained in its bug reports is combined with results of static pointer analysis implemented in Symbiotic to get a more precise (i.e. smaller) set of potentially misbehaving instructions on which Symbiotic subsequently concentrates its further analysis based on symbolic execution [13].

Predator did not officially participate in SV-COMP’21 due to insufficient manpower for keeping it up-to-date with various changes in the competition’s rules, formats, and with various specific features of new verification tasks. However, it participated “hors concours” with absolutely no change wrt the previous year, and the results indicate that it would still be capable of scoring quite favourably in some categories or sub-categories. Moreover, the integration of Predator with Symbiotic was improved, and Predator appeared in SV-COMP’21 as a part of Symbiotic too.

1.7 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 we said already in the introduction of the chapter, SMGs and the shape analysis based on them are inspired by the fully-automated approaches based on *separation logic* with higher-order list predicates implemented in the Space Invader [2, 38] and SLayer [3] tools. Compared with them, however, we use a purely graph-based memory representation.²³ Our heap representation is finer, which—on one hand—complicates its formalization but allows for treating the different peculiarities of low-level memory manipulation on the other hand. Moreover, somewhat surprisingly, although our heap representation is rather detailed, it still allowed us to propose algorithms for all the needed operations such that they are quite efficient.²⁴

Compared with Space Invader and SLayer, Predator is not only faster, but also terminates more often, avoids false positives and, in particular, is able to detect more classes of program errors (as illustrated in the section on experiments). Both Space Invader and SLayer provide some support for pointer arithmetic, but their 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. A support for pointer arithmetic in combination with separation logic appears in [9] too, which is, however, highly specialised for a particular kind of linked lists with variable length entries used in some memory allocators.

²³ In fact, a graph-based representation was used already in the first version of Predator [18]. However, that representation was a rather straightforward graph-based encoding of separation logic formulae, which is not the case anymore for the representation described here.

²⁴ Indeed, the version of Predator based on SMGs as presented here turned out to be much faster than the first one of [18] while at the same time producing fewer false positives.

As for the memory model, probably the closest to the notion of SMGs is [24], 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 [24] and the older work [14], on which [24] is based, use 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 automated 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-automated* 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 integer nesting level, which allows us to represent hierarchically-nested data structures as flattened graphs. Finally, although [24] points out a need to reinterpret the memory contents upon reading/writing, the corresponding operations are not formalized there.

A graph-based abstraction of sets of heap configurations is used in [29] 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 SMGs. Finally, the abstraction and join operations used in [29] are more aggressive and hence less precise than in our case.

The work [23], which is based on an instantiation of the *TVLA framework* [35], 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 [36] provides a detailed treatment of low-level C features such as alignment, byte-order, padding, or type-unsafe casts 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.

Another tool that can handle some features of pointer arithmetic is Forester [20]. It is based on hierarchically-nested *forest automata*, i.e., tuples of interconnected tree automata, and the approach of abstract regular tree model checking [8]. Forester can handle fully-automatically more general classes of dynamic linked data structures than Predator (trees, trees with additional pointers, skip lists²⁵), has a more flexible

²⁵ Adding a support of such non-list dynamic data structures to Predator is non-trivial. For that, new kinds of heap segments would have to be added, together with algorithms for all the needed operations. In fact, an attempt to add *tree segments* was once done. It was realised that one would need tree segments with and without “holes” on the leaf level, possibly even a variable number of them, through which the tree segments would link to the rest of the heap. However, algorithms that would be capable of handling such segments, combine them reasonably with list segments (one needs to handle appropriately questions such as whether a tree degenerated to the left-most branch is a list or a tree), and prevent state space explosion stemming from introducing a number of different kinds of tree segments have never been finished.

abstraction (which can adjust to various non-standard shapes of data structures), and a support for dealing with ordered data in one of its versions [1] as well as a support for finite data together with a counterexample-guided abstraction refinement loop in another version [11]. However, as our experimental results in Section 1.6.1 show, Forester’s support of low-level features is much more limited and it is often less efficient too (though not always as shown in [20]).

Interestingly, there have appeared two tools that attempt to reimplement SMGs in the context of *configurable program analysis* (CPA) [7]: namely, CPAlien [31] and CPAchecker [26]. CPAlien was an experimental tool with a partial implementation of SMGs only: in particular, abstraction was missing. The support of SMGs in CPAchecker is—as far as we know—more complete, but so far it is also used without abstraction for efficiency reasons (at least that was the case up to SV-COMP’20). Without abstraction, however, CPAchecker cannot successfully verify correct programs in case dealing with unbounded lists is needed for the verification.

The above problem of not being able to soundly verify programs whose verification requires dealing with unbounded dynamic data structures is much more common among the tools that participated in heap- and memory-related (sub-)categories of the different editions of SV-COMP up to SV-COMP’20. This problem manifests, e.g., in Symbiotic [12] or Ultimate Automizer [21]. In particular, in order not to sacrifice soundness, the analysis implemented in these tools cannot successfully terminate on such programs (while still other tools perform bounded analysis only and produce in principle unsound answers). A tool that participates in SV-COMP and that can handle verification on unbounded list structures in a sound way is 2LS [27]. This tool is based on a combination of *template-based invariants*, *k*-induction, abstract domains (for representing suitable parameters of the template-based invariants), and SAT solving. 2LS can handle unbounded list structures and can even reason (to some degree) about data stored in them [28]. However, 2LS currently has no support for pointer arithmetic.

Finally, it is worth mentioning that the above mentioned works on separation logic and Space Invader later led to the so-called *bi-abductive analysis* of programs with dynamic linked data structures [10]. This approach was implemented in a tool called Infer that concentrated on (nested) dynamic linked lists—despite that the approach of [10] itself is more general. An advantage of the approach is that it can handle *open code fragments* (i.e., there is no need to model the environment of the code fragment under verification) and it can perform the analysis *modularly*, analysing functions along the call tree, starting from the leaves (which is quite scalable though it may involve some loss of precision). A generalisation of the approach appeared in relation with the S2 tool [25]. This tool can handle programs with very complex data structures (e.g., trees with linked leaves). On the other hand, the approach is rather fragile in that it relies on the program to handle data structures in a way that is well aligned with their inductive definitions and sometimes it fails even on rather simple programs. Moreover, despite Infer contained some support of pointer arithmetic, the support of low-level pointer features in bi-abductive analyses is quite limited and remains an open problem for the future.

Indeed, for the future, it would be very useful to have an abductive analyser supporting truly low-level features of memory manipulation and at the same time

capable of analysing code fragments (since analysis of such fragments is probably the most welcome in real life according to our experience). As for the low-level features, one can, of course, go even further than Predator does: e.g., sometimes even bit-precision is needed—for instance, when some bits of pointers are used to store non-pointer information by bit-masking (which is used sometimes, e.g., to store the colour of nodes in red-black trees).

Acknowledgement.

The work was supported by the Czech Science Foundation project 20-07487S and the FIT BUT project FIT-S-23-8151.

References

1. Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata. *Acta Informatica*, 53(4):357–385. Springer (2016).
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O’Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Proc. of CAV’07, vol. 4590 of LNCS, pp. 178–192. Springer (2007)
3. Berdine, J., Cook, B., Ishtiaq, S.: SLayer: Memory Safety for Systems-level Code. In: Proc. of CAV’11, vol. 6806 of LNCS, pp. 178–183. Springer (2011)
4. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.M.: Diagnosing Abstraction Failure for Separation Logic-Based Analyses. In: Proc. of CAV’12, vol. 7358 of LNCS, pp. 155–173. Springer (2012)
5. Beyer, D., Dangl, M., Wendler, P.: Boosting k-Induction with Continuously-Refined Invariants. In: Proc. of CAV’15, vol. 9206 of LNCS, pp. 622–640. Springer (2015)
6. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1). Springer (2019)
7. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of In: Proc. of CAV’07, vol. 4590 of LNCS, pp. 504–518. Springer (2007)
8. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular (Tree) Model Checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191. Springer (2012)
9. Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In: Proc. of SAS’06, vol. 4134 of LNCS, pp. 182–203. Springer (2006)
10. Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM*, 58(6):26:1–26:66. ACM (2011)
11. Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Vojnar, T.: Counterexample Validation and Interpolation-Based Refinement for Forest Automata. In Proc. of VMCAI’17, vol. 10145 of LNCS, pp. 288–309. Springer (2017)
12. Chalupa, M., Vitovská, M., Jonáš, M., Slabý, J., Strejček, J.: Symbiotic 4: Beyond Reachability. In: Proc. of TACAS’17, vol. 10206 of LNCS, pp. 385–389. Springer (2017)
13. Chalupa, M., Jašek, T., Tomovič, L., Hruška, M., Šoková, V., Ayaziová, P., Strejček, J., Vojnar, T.: Symbiotic 7: Integration of Predator and More (Competition Contribution). In Proc. of TACAS’20, vol. 12079 of LNCS, pp. 413–417. Springer (2020)
14. Chang, B.-Y. E., Rival, X., Nacula, G.C.: Shape analysis with structural invariant checkers. In: Proc. of SAS’07, vol. 4634 of LNCS, pp. 384–401. Springer (2007)

15. Cox, A., Chang, B.-Y. E., Li, H., Rival, X.: Abstract Domains and Solvers for Sets Reasoning. In: Proc. of LPAR'15, vol. 9450 of LNCS, pp. 356–371. Springer (2015)
16. Dudka, K., Holík, L., Peringer, P., Trtík, M., Vojnar, T.: From Low-Level Pointers to High-Level Containers. In: Proc. of VMCAI'16, vol. 9583 of LNCS, pp. 431–452. Springer (2016)
17. Dudka, K., Peringer, P., Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools. In: Proc. of EUROCAST'11, vol. 6927 of LNCS, pp. 527–534. Springer (2012)
18. Dudka, K., Peringer, P., Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In: Proc. of CAV'11, vol. 6806 of LNCS, pp. 372–378. Springer (2011)
19. Dudka, K., Peringer, P., Vojnar, T.: Predator: Byte-Precise Verification of Low-Level List Manipulation. In: Proc. of SAS'13, vol. 7935 of LNCS, pp. 214–237. Springer (2013)
20. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest Automata for Verification of Heap Manipulation. *Formal Methods in System Design*, **41**(1), Springer (2012)
21. Heizmann, M. et al.: Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata. In: Proc. of TACAS'17, vol. 10206 of LNCS, pp. 394–398. Springer (2017)
22. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: Predator Shape Analysis Tool Suite. In: Proc. of HVC'16, vol. 10028 of LNCS, pp. 202–209. Springer (2016)
23. Kreiker, J., Seidl, H., Vojdani, V.: Shape Analysis of Low-Level C with Overlapping Structures. In: Proc. of VMCAI'10, vol. 5944 of LNCS, pp. 214–230. Springer (2010)
24. Laviron, V., Chang, B.Y.E., Rival, X.: Separating Shape Graphs. In: Proc. of ESOP'10, vol. 6012 of LNCS, pp. 387–406. Springer (2010)
25. Le, Q.L., Gherghina, C., Qin, S., Chin, W.-N.: Shape Analysis via Second-Order Bi-Abduction. In: Proc. of CAV'15, vol. 9206 of LNCS, pp. 52–68. Springer (2015)
26. Löwe, S., Mandrykin, M.U., Wendler, P.: CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses (Competition Contribution). In: Proc. of TACAS'14, vol. 8413 of LNCS, pp. 392–394. Springer (2014)
27. Malík, V., Martiček, Š., Schrammel, P., Srivas, M.K., Vojnar, T., Wahlang, J.: 2LS: Memory Safety and Non-termination (Competition Contribution). In: Proc. of TACAS'18, vol. 10806 of LNCS, pp. 417–421. Springer (2018)
28. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.: Template-Based Verification of Heap-Manipulating Programs. In: Proc. of FMCAD'18. IEEE (2018)
29. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models. In: Proc. of CC'08, vol. 4959 of LNCS, pp. 245–259. Springer (2008)
30. Muller, P., Peringer, P., Vojnar, T.: Predator Hunting Party (Competition Contribution). In: Proc. of TACAS'15, vol. 9035 of LNCS, pp. 443–446. Springer (2015)
31. Muller, P., Vojnar, T.: CPAlien: Shape Analyzer for CPAchecker (Competition Contribution). In: Proc. of TACAS'14, vol. 8413 of LNCS, pp. 395–397. Springer (2014)
32. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution). In: Proc. of TACAS'20, vol. 12079 of LNCS, pp. 408–412. Springer (2020)
33. Reps, T., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Proc. of POPL'95, pp. 49–61. ACM Press (1995)
34. Rival, X., et al.: The MemCAD Analyzer. Available at <https://www.di.ens.fr/~rival/memcad.html>. (2016)
35. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **24**(3), ACM (2002)
36. Tuch, H.: Formal Verification of C Systems Code. *Journal of Automated Reasoning*, **42**(2–4), Springer (2009)
37. 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)
38. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Proc. of CAV'08, vol. 5123 of LNCS, pp. 385–398. Springer (2008)

Appendix

1.8 Data Reinterpretation of Nullified Blocks

In this appendix, we present a detailed description of the algorithms for read and write reinterpretation of nullified blocks, which were briefly introduced in Section 1.3.1. Given an SMG $G = (O, V, \Lambda, H, P)$, we define $H_{ov}(o, of, t)$ as the set of all has-value edges leading from o whose fields overlap with the field (of, t) , i.e.:

$$H_{ov}(o, of, t) = \{(o \xrightarrow{of', t'} v) \in H \mid I(of, t) \cap I(of', t') \neq \emptyset\}.$$

Further, we define $H_{zr}(o, of, t)$ as the subset of $H_{ov}(o, of, t)$ containing all its edges leading to 0, i.e.:

$$H_{zr}(o, of, t) = \{(o \xrightarrow{of', t'} 0) \in H_{ov}(o, of, t)\}.$$

1.8.1 Read Reinterpretation of Nullified Blocks

Algorithm 1 gives the algorithm of read reinterpretation instantiated for dealing with nullified blocks of memory as precisely as possible.

Algorithm 1 *readValue*(G, o, of, t)

Input:

- An SMG $G = (O, V, \Lambda, H, P)$.
- An object $o \in O$.
- A field (of, t) within o , i.e., $of + size(t) \leq size(o)$.

Output:

- A tuple (G', v) that is the result of read reinterpretation of G wrt the object o and the field (of, t) such that fields representing nullified memory are read as precisely as the notion of SMGs allows (i.e., the operator recognises that the field to be read is nullified iff the input SMG guarantees that each byte of the field is indeed zero).

Method:

1. Let $v := H(o, of, t)$.
 2. If $v \neq \perp$, return (G, v) .
 3. If the field to be read is covered by nullified blocks, i.e., if $\forall of \leq i < of + size(t) \exists e \in H_{zr}(o, of, t) : i \in I(e)$, let $v := 0$. Otherwise, extend V by a fresh value node v .
 4. Extend H by the has-value edge $o \xrightarrow{of, t} v$ and return (G, v) based on the obtained SMG G .
-

1.8.2 Write Reinterpretation of Nullified Blocks

Algorithm 2 gives the algorithm of write reinterpretation instantiated for dealing with nullified blocks of memory as precisely as possible. An illustration of how the algorithm works can be found in Fig. 1.12.

1.9 The Join Algorithms

This appendix provides a detailed description of the join algorithms introduced in Section 1.3.2. We first describe the *joinSubSMGs* function, which implements the

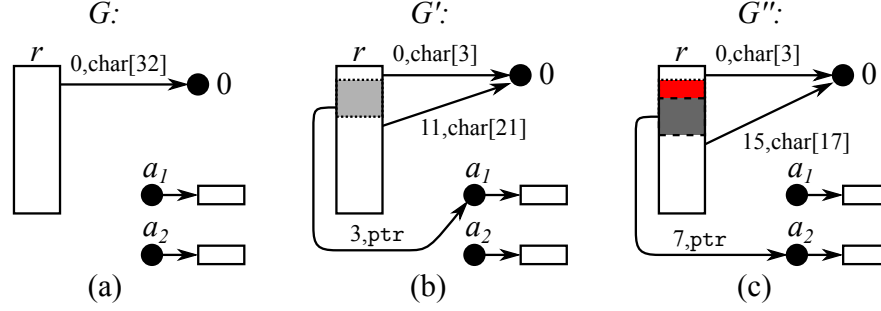


Fig. 1.12: An illustration of write reinterpretation: (a) an initial SMG G , (b) the SMG G' obtained by $\text{writeValue}(G, r, 3, \text{ptr}, a_1)$, and (c) the SMG G'' obtained by $\text{writeValue}(G', r, 7, \text{ptr}, a_2)$. Note that G'' contains an undefined value in a field of size of 4 bytes at offset 3.

Algorithm 2 $\text{writeValue}(G, o, of, t, v)$

Input:

- An SMG $G = (O, V, \Lambda, H, P)$.
- An object $o \in O$.
- A field (of, t) within o , i.e., $of + \text{size}(t) \leq \text{size}(o)$.
- A value v such that $v \notin O$ (needed so that v can be safely added into V).

Output:

- An SMG G' that is the result of write reinterpretation of G wrt the object o , the field (of, t) , and the value v such that as much information on nullified memory as is allowed by the notion of SMGs is preserved (i.e., each byte that is nullified in the input SMG will stay nullified in the output SMG unless it is overwritten by a possibly non-zero value).

Method:

1. If $H(o, of, t) = v$, return G .
 2. Let $V := V \cup \{v\}$.
 3. Remove from H all edges leading from o to non-zero values whose fields overlap with the given field, i.e., the edges in $H_{oV}(o, of, t) \setminus H_{zr}(o, of, t)$.
 4. If $v \neq 0$, then for each edge $(e_z : o \xrightarrow{of_z, t_z} 0) \in H_{zr}(o, of, t)$ do:
 - a. Remove the edge e_z from H .
 - b. Let $of' := of + \text{size}(t)$ and $of'_z := of_z + \text{size}(t_z)$.
 - c. If $of_z < of$, extend H by the edge $o \xrightarrow{of_z, \text{char}[of - of_z]} 0$.
 - d. If $of' < of'_z$, extend H by the edge $o \xrightarrow{of', \text{char}[of'_z - of']} 0$.
 5. Extend H by the has-value edge $o \xrightarrow{of, t} v$ and return the obtained SMG.
-

core functionality on top of which both joining garbage-free SPCs (to reduce the number of SPCs obtained from different paths through the program) as well as merging a pair of neighbouring objects of a doubly-linked list into a single DLS within abstraction are built. Subsequently, we describe the functions joinValues , joinTargetObjects , and $\text{insertLeftDlsAndJoin} / \text{insertRightDlsAndJoin}$ on which joinSubSMGs is based. In fact, joinSubSMGs calls joinValues on pairs of corresponding values that appear below the roots of the sub-SMGs to be joined, joinValues then calls joinTargetObjects on pairs of objects that are the target of value nodes representing addresses, and the joinTargetObjects algorithm recursively calls joinSubSMGs to join the sub-SMGs of

the objects to be joined. The *insertLeft(Right)DlsAndJoin* functions are called from *joinValues* when the given pair of addresses cannot be joined since their target objects are incompatible, and an attempt to save the join from failing is done by trying to compensate a DLS missing in one of the SMGs by inserting it with the minimum length being 0 (which is possible since a 0+ DLS is a possibly empty list segment). Finally, we describe the *joinSPCs* and *mergeSubSMGs* functions implemented on top of the generic *joinSubSMGs* function. The *joinSPCs* function joins garbage-free SPCs into a single SPC that semantically covers both. The *mergeSubSMGs* function merges a pair of objects during abstraction into a single DLS while the non-shared part of the sub-SMGs rooted at them is joined into the nested sub-SMG of the resulting DLS.

As mentioned in Section 1.3.2, the join algorithm computes on the fly the so-called *join status* which compares the semantics of the SMGs being joined (with the semantics being either equal, in an entailment relation, or incomparable). For the purpose of maintaining the join status, the table shown on the right defines the function $updateJoinStatus : \mathbb{J} \times \mathbb{J} \longrightarrow \mathbb{J}$ that combines the current join status s_1 , obtained from joining the so-far explored parts of the SMGs being joined, with a status $s_2 \in \mathbb{J}$ comparing the semantics of the objects/values being currently joined. Note that the function is monotone in that once the status, which is initially $\approx$, becomes $\sqsubset$ or $\sqsubset$, it can never get back to $\approx$, and once the status becomes $\bowtie$, it cannot change any more.

$s_1 \backslash s_2$	$\approx$	$\sqsubset$	$\sqsubset$	$\bowtie$
$\approx$	$\approx$	$\sqsubset$	$\sqsubset$	$\bowtie$
$\sqsubset$	$\sqsubset$	$\sqsubset$	$\bowtie$	$\bowtie$
$\sqsubset$	$\sqsubset$	$\bowtie$	$\bowtie$	$\bowtie$
$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$

In case of the *joinSPCs* function, *joinSubSMGs* needs to be called multiple times for a single pair of SPCs (starting from different program variables), and it is necessary to keep certain state information between the calls. Besides the join status mentioned above, the algorithm maintains a mapping of values and objects between the source SMGs and the destination SMG. This is needed in order to identify potentially conflicting mappings arising when starting the join from different program variables as well as to identify parts of SMGs that have already been processed. The mapping is encoded as a pair of partial injective functions $m_1 : (O_1 \rightarrow O) \cup (V_1 \rightarrow V)$ and $m_2 : (O_2 \rightarrow O) \cup (V_2 \rightarrow V)$. Additionally, we assume that the # object and the 0 address, which have their pre-defined unique roles in all SMGs, never appear in the ranges of m_1, m_2 . In case of the *mergeSubSMGs* function, the mapping of objects and values is used to obtain the sets of nodes recognized as nested data structures.

In the following, we write $kind_1, size_1, level_1, len_1, valid_1, nfo_1, pfo_1$, and hfo_1 to denote $kind, size, level, len, valid, nfo, pfo$, and hfo from Λ_1 , i.e., the labelling function of the first SMG being joined. Likewise for Λ_2 . We further define len' as a wrapper function of len such that $len'(o) = len(o)$ if $kind(o) = \text{dls}$, and $len'(o) = 1$ if $kind(o) = \text{reg}$.

1.9.1 Join Reinterpretation

The read and write reinterpretations described in Section 1.3.1 operate on a single object of a single SMG. However, when joining a pair of SMGs, we need to compare

pairs of their objects, figure out what they have semantically in common, and modify their sets of fields such that they become the same (even if for the price of loosing some information), allowing one to subsequently attempt to join their corresponding sub-SMGs. For that purpose, we introduce *join reinterpretation*.

A join reinterpretation operator inputs a pair of SMGs G_1 and G_2 , whose sets of objects are O_1 and O_2 , respectively, and a pair of objects $o_1 \in O_1$ and $o_2 \in O_2$ such that $size_1(o_1) = size_2(o_2)$. The operator returns a triple (s, G'_1, G'_2) where $s \in \mathbb{J}$ is a join status, and G'_1, G'_2 are two SMGs with sets of has-value edges H'_1, H'_2 , respectively,

such that: (1) The sets of fields of o_1 and o_2 are the same, i.e., $\forall of \in \mathbb{N} \forall t \in \mathbb{T} : H'_1(o_1, of, t) \neq \perp \Leftrightarrow H'_2(o_2, of, t) \neq \perp$. (2) The status s and the semantics of G'_1 and G'_2 are defined according to the table shown on the right.

s	semantics of G'_1	semantics of G'_2
$\approx$	$MI(G_1) = MI(G'_1)$	$MI(G_2) = MI(G'_2)$
$\sqsubset$	$MI(G_1) = MI(G'_1)$	$MI(G_2) \subset MI(G'_2)$
$\sqsubset$	$MI(G_1) \subset MI(G'_1)$	$MI(G_2) = MI(G'_2)$
$\bowtie$	$MI(G_1) \subset MI(G'_1)$	$MI(G_2) \subset MI(G'_2)$

Intuitively, the status $\sqsubset$ means that some aspects of G_1 are less restrictive than those of G_2 , implying a need to lift these restrictions in G_2 to keep chance for its successful join with G_1 , which enlarges the semantics of G_2 while that of G_1 stays the same. Likewise for the other symbols of $\mathbb{J}$.

For the particular case of dealing with nullified memory, we implement the join reinterpretation as follows (cf. Algorithm 3). First, nullified fields are shortened, split, and/or composed in each of the objects with the aim of obtaining the smallest possible number of nullified fields such that either (1) each byte of such fields is nullified in both original SMGs, or (2) the field is nullified in one SMG, and, in the other, it contains a non-null address. The former is motivated by preserving as much information about nullified memory as possible when joining two objects. The latter is motivated by the fact that a null pointer may be interpreted as a special case of a null-terminated 0+ DLS and hence possibly joined with an address in the other SMG if its target is a DLS. Finally, whenever a field (of, t) remains defined in o_1 but not in o_2 after the described transformations, i.e., if $H_1(o_1, of, t) \neq \perp$ and $H_2(o_2, of, t) = \perp$, H_2 is extended such that $H_2(o_2, of, t) = v'$ for some fresh v' (and likewise the other way around). Note that the join status is not updated since it will be updated later on when joining the appropriate values.

1.9.2 Join of Sub-SMGs

The *joinSubSMGs* function (cf. Alg. 4) is responsible for joining a pair of sub-SMGs rooted at a given pair of objects and for constructing the resulting sub-SMG within the given destination SMG. The function inputs a triple of SMGs G_1, G_2, G (two source SMGs and one destination SMG) and a triple of equally-sized objects o_1, o_2, o from the SMGs G_1, G_2, G , respectively. If the *joinSubSMGs* function fails in joining the given sub-SMGs, it returns $\perp$. Otherwise, it returns a triple of SMGs G'_1, G'_2, G' such that:

- $MI(G_1) \subseteq MI(G'_1)$ and $MI(G_2) \subseteq MI(G'_2)$ where G_1 and G_2 can differ from G'_1 and G'_2 , respectively, due to an application of join reinterpretation on some of the pairs of objects being joined only.
- The sub-SMGs G'_1 and G'_2 of G'_1 and G'_2 rooted at o_1 and o_2 , respectively, are joined into the sub-SMG G'' of G' rooted at o , i.e., it is required that $MI(G'_1) \subseteq MI(G'') \supseteq MI(G'_2)$.
- The sub-SMG $G' \setminus G''$ is exactly the sub-SMG of G that consists of objects and values that are not removed in Step 10 of the *joinTargetObjects* function due to using the principle of delayed join of sub-SMGs described in Section 1.9.6 (which will take care of the fact that some sub-SMG may be reachable along some access path in one of the SMGs G_1/G_2 only, due to some optional nested sub-heap missing in the other SMG, in which, however, it may be reachable via some other access path and may thus be discovered and joined later on).

Algorithm 3 *joinFields*(G_1, G_2, o_1, o_2)

Input:

- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$ and $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$ with sets of addresses A_1 and A_2 , respectively.
- Objects $o_1 \in O_1$ and $o_2 \in O_2$, s.t. $size_1(o_1) = size_2(o_2)$.

Output:

- A tuple (s', G'_1, G'_2) consisting of a join status and two SMGs that is the result of join reinterpretation of G_1 and G_2 wrt o_1 and o_2 which reorganizes the nullified fields of o_1 and o_2 with the aim of obtaining the smallest possible number of nullified fields such that either (a) each byte of such fields is nullified in both G_1 and G_2 , or (b) the field is nullified in one of them, and in the other, it contains a non-null address.

Method:

1. Let $H'_1 := H_1$, $H'_2 := H_2$.
 2. Process the set $H_{1,0} = \{o_1 \xrightarrow{of,t} 0 \in H'_1\}$ of edges leading from o_1 to 0 in G_1 as follows:
 - a. Remove the edges that are in $H_{1,0}$ from H'_1 .
 - b. Extend H'_1 by the smallest set of edges $H'_{1,0}$ in which for each $0 \leq i < size_1(o_1)$ there is an edge $o_1 \xrightarrow{of,t'} 0 \in H'_{1,0}$ such that $i \in I(of', t')$ where $t' = \text{char}[n]$ for some $n > 0$ iff $\exists (o_1 \xrightarrow{of_1,t_1} 0) \in H_{1,0} \exists (o_2 \xrightarrow{of_2,t_2} 0) \in H'_2 : i \in I(of_1, t_1) \cap I(of_2, t_2)$.
 - c. For each address $a_2 \in A_2 \setminus \{0\}$ and each edge $(o_2 \xrightarrow{of,ptr} a_2) \in H'_2$ for which there is no $a_1 \in A_1 \setminus \{0\}$ such that $(o_1 \xrightarrow{of,ptr} a_1) \in H'_1$, but $I(of, ptr) \subseteq \bigcup_{e \in H_{1,0}} I(e)$, extend H'_1 by the edge $o_1 \xrightarrow{of,ptr} 0$.
 - Then do the same for o_2 with swapped sets of edges H'_1 and H'_2 , using A_1 instead of A_2 , and $H_{2,0}$ and $H'_{2,0}$ instead of $H_{1,0}$ and $H'_{1,0}$, respectively.
 3. Let $s := \simeq$.
 4. For each $0 \leq i < size_1(o_1)$:
 - If $\exists (e : o_1 \xrightarrow{of,t} 0) \in H_1$ such that $i \in I(e)$ and $\forall (e' : o_1 \xrightarrow{of,t'} 0) \in H'_1 : i \notin I(e')$, let $s := \text{updateJoinStatus}(s, \sqsubset)$.
 - If $\exists (e : o_2 \xrightarrow{of,t} 0) \in H_2$ such that $i \in I(e)$ and $\forall (e' : o_2 \xrightarrow{of,t'} 0) \in H'_2 : i \notin I(e')$, let $s := \text{updateJoinStatus}(s, \sqsupset)$.
 5. For all fields (of, t) such that $H'_1(o_1, of, t) \neq \perp \wedge H'_2(o_2, of, t) = \perp$, extend H'_2 such that $H'_2(o_2, of, t) = v$ for some fresh v added into V_2 . Proceed likewise for non-nullified fields of o_2 not defined in o_1 .
 6. Return (s, G_1, G_2) where $G_1 = (O_1, V_1, \Lambda_1, H'_1, P_1)$ and $G_2 = (O_2, V_2, \Lambda_2, H'_2, P_2)$.
-

The *joinSubSMGs* function first applies the join reinterpretation operator (denoted *joinFields*) on G_1 , G_2 and o_1 , o_2 , which ensures that the sets of fields of o_1 and o_2 are identical. The function then iterates over the set of fields of these objects, and for each field (of, t) does the following:

- finds the pair of values v_1 and v_2 which the has-value edges of o_1 and o_2 labelled by (of, t) lead to,
- calls the *joinValues* function for v_1 and v_2 , which is responsible for recursively joining the remaining parts of the sub-SMGs rooted at them (before the call, the nesting level difference between G_1 and G_2 is possibly updated as described below), and
- extends the set of edges of G by $o \xrightarrow{of, t} v$ where v is the value returned by the *joinValues* function.

Adjusting the Nesting Level Difference

In Section 1.3.2, it is said that the levels of the objects being joined can differ since the objects may sometimes appear below a DLS and sometimes below a region (and while an object that appears below a DLS may be considered nested—provided that each node of the segment has a separate copy of such an object—there is no notion of nesting below regions since for regions which represent concrete objects there is no need to distinguish private and shared sub-SMGs). The functionality of *joinSubSMGs* therefore includes tracking of the difference in levels (denoted l_{diff}) at which objects and values to be joined within some sub-SMG can appear. When objects o_1 and o_2 are being joined, the difference is computed as follows: If o_1 is a DLS and o_2 is a region, the current value of l_{diff} is increased by one. Symmetrically, if o_1 is a region and o_2 is a DLS, the value of l_{diff} is decreased by one. Note that when going to sub-heaps in both G_1 and G_2 , the change of the difference is 0. The new difference is then used when joining the values of the fields of o_1 and o_2 (apart from the next and prev fields of course).

1.9.3 Join of Values

The *joinValues* function (cf. Alg. 5) joins a pair of sub-SMGs rooted at a given pair of values and returns a single value node that represents both the input values in the destination SMG. The function inputs a triple of SMGs G_1 , G_2 , G (two source SMGs and one destination SMG) and a pair of values v_1 and v_2 from G_1 and G_2 , respectively. If the function fails in joining the given values, it returns $\perp$. Otherwise, it returns a triple of SMGs G'_1 , G'_2 , G' , and a value v from G' such that:

- $MI(G_1) \subseteq MI(G'_1)$ and $MI(G_2) \subseteq MI(G'_2)$ where G_1 and G_2 can differ from G'_1 and G'_2 , respectively, due to an application of join reinterpretation on some of the pairs of objects being joined only.
- The sub-SMGs G''_1 and G''_2 of G'_1 and G'_2 rooted at v_1 and v_2 , respectively, are joined into the sub-SMG G'' of G' rooted at v , i.e., it is required that $MI(G''_1) \subseteq MI(G'') \supseteq MI(G''_2)$.

Algorithm 4 $joinSubSMGs(s, G_1, G_2, G, m_1, m_2, o_1, o_2, o, l_{diff})$ **Input:**

- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Section 1.9.
- Objects $o_1 \in O_1$, $o_2 \in O_2$, $o \in O$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:

- $\perp$ in case the sub-SMGs of G_1 and G_2 rooted at o_1 and o_2 cannot be joined.
- Otherwise, a tuple $(s', G'_1, G'_2, G', m'_1, m'_2)$ where:
 - $s' \in \mathbb{J}$ is the resulting join status.
 - G'_1, G'_2, G' are SMGs as defined in Section 1.9.2.
 - m'_1, m'_2 are the resulting injective partial mappings of nodes.

Method:

1. Let $res := joinFields(G_1, G_2, o_1, o_2)$. If $res = \perp$, return $\perp$. Otherwise let $(s', G_1, G_2) := res$ and $s := updateJoinStatus(s, s')$.
2. Collect the set F of all pairs (of, t) occurring in has-value edges leading from o_1 or o_2 .
3. For each field $(of, t) \in F$ do:
 - Let $v_1 = H_1(o_1, of, t)$, $v_2 = H_2(o_2, of, t)$, and $l'_{diff} := l_{diff}$.
 - If $kind_1(o_1) = dls$ and (of, t) is not the next/prev field of o_1 , let $l'_{diff} := l'_{diff} + 1$.
 - If $kind_2(o_2) = dls$ and (of, t) is not the next/prev field of o_2 , let $l'_{diff} := l'_{diff} - 1$.
 - Let $res := joinValues(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l'_{diff})$. If $res = \perp$, return $\perp$. Otherwise, let $(s, G_1, G_2, G, m_1, m_2, v) := res$.
 - Introduce a new has-value edge $o \xrightarrow{of, t} v$ in H .
4. Return $(s, G_1, G_2, G, m_1, m_2)$.

- The sub-SMG $G' \setminus G''$ is exactly the sub-SMG of G that consists of objects and values that are not removed in Step 10 of the $joinTargetObjects$ function due to using the principle of delayed join of sub-SMGs described in Section 1.9.6.

If the input values are identical ($v_1 = v_2$), the resulting value v is the same identical value, say v_1 , which prevents shared nodes from being processed as nested data structures during abstraction. If both values are mapped to the same value node in the destination SMG, i.e., $m_1(v_1) = m_2(v_2)$, the node $m_1(v_1)$ (or, equivalently, $m_2(v_2)$) to which they are mapped is returned since such a pair of values has already been successfully joined before. A pair of non-address values visited for the first time, i.e., a pair of values for which $m_1(v_1) = \perp = m_2(v_2)$, is joined by creating a fresh value node v in G' with the appropriate nesting level²⁶, and the mapping of nodes is extended such that $m_1(v_1) = m_2(v_2) = v$. Non-address values are never joined with addresses—if such a situation occurs, the whole join operation fails.²⁷

²⁶ In case the difference of $level_1(v_1)$ and $level_2(v_2)$ differs from l_{diff} , the join status is appropriately updated. This reflects the fact that, e.g., a single unknown (or interval) abstract value that is used as the value of multiple fields of some region (for instance, an array) that is more nested than the region may concretise to *different* concrete values for each of the fields while it must concretise to the *same* concrete value if the abstract value is on the same level as the given region.

²⁷ Note that the handling of non-address values is kept simple since the basic notion of SMGs does not distinguish any special kinds of non-address values (numbers, intervals, etc.), but there is still room for improvement, especially in conjunction with the extensions described in Section 1.4.

Algorithm 5 $joinValues(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$ **Input:**

- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Section 1.9.
- Values $v_1 \in V_1$ and $v_2 \in V_2$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:

- $\perp$ in case the sub-SMGs of G_1 and G_2 rooted at v_1 and v_2 cannot be joined.
- Otherwise, a tuple $(s', G'_1, G'_2, G', m'_1, m'_2, v')$ where:
 - $s' \in \mathbb{J}$ is the resulting join status.
 - G'_1, G'_2, G' are SMGs as defined in Section 1.9.3.
 - m'_1, m'_2 are the resulting injective partial mappings of nodes.
 - v' is a value in G' satisfying the conditions stated in Section 1.9.3.

Method:

1. If $v_1 = v_2$, return $(s, G_1, G_2, G, m_1, m_2, v_1)$. In this case, the given pair of values matches trivially, which happens whenever a shared value is reached during abstraction.
2. If $m_1(v_1) = m_2(v_2) = v \neq \perp$, return $(s, G_1, G_2, G, m_1, m_2, v)$. In this case, the pair of values is already joined.
3. If both v_1 and v_2 are non-address values, i.e., $P_1(v_1) = \perp$ and $P_2(v_2) = \perp$, then:
 - If $m_1(v_1) \neq \perp$ or $m_2(v_2) \neq \perp$, return $\perp$.
 - Create a new value node $v \in V$ such that $level(v) = \max(level_1(v_1), level_2(v_2))$.
 - Extend the mapping of nodes such that $m_1(v_1) = m_2(v_2) = v$.
 - If $level_1(v_1) - level_2(v_2) < l_{diff}$, let $s := updateJoinStatus(s, \sqsubset)$.
 - If $level_1(v_1) - level_2(v_2) > l_{diff}$, let $s := updateJoinStatus(s, \sqsupset)$.
 - Return $(s, G_1, G_2, G, m_1, m_2, v_1)$.
4. If $P_1(v_1) = \perp$ or $P_2(v_2) = \perp$, return $\perp$.
5. Let $res := joinTargetObjects(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$.
If $res = \perp$, return $\perp$. If $res \neq \hookleftrightarrow$, then return res .
6. Let $o_1 := o(P_1(v_1))$ and $o_2 := o(P_2(v_2))$.
7. If $kind_1(o_1) = dls$, let $res = insertLeftDlsAndJoin(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$.
If $res = \perp$, return $\perp$. If $res \neq \hookleftrightarrow$, then return res .
8. If $kind_2(o_2) = dls$, let $res = insertRightDlsAndJoin(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$.
If $res \in \{\perp, \hookleftrightarrow\}$, return $\perp$. Otherwise, return res .

For addresses seen for the first time, the algorithm tries to join their targets with each other with three possible outcomes:

1. The join succeeds, and $joinValues$ then succeeds too.
2. The join fails in a recoverable way (denoted by the result being $\hookleftrightarrow$). Intuitively, this happens when trying to join addresses that are found to be obviously different, i.e., addresses that are found to be different without going deeper into the sub-SMGs rooted at them—e.g., due to they point to some object with different offsets, different target specifiers, incompatible levels, size, validity, linking fields (for DLSs), or when the objects they point to are already mapped to some other objects. In case of the recoverable failure, if at least one of the target objects is a DLS, the algorithm tries to virtually *insert a DLS* in one of the SMGs, which

allows it to create a 0+ DLS in the destination SMG and continue by joining the appropriate successor values. If this fails too, the whole join operation fails.²⁸

3. The join fails in an irrecoverable way (denoted by the result being $\perp$) in which case *joinValues* fails too.

1.9.4 Join of Target Objects

The *joinTargetObjects* function (cf. Alg. 6) joins a pair of sub-SMGs rooted at the given pair of addresses and returns a single address node that represents both the input addresses in the destination SMG. The function inputs a triple of SMGs G_1 , G_2 , G (two source SMGs and one destination SMG) and a pair of addresses a_1 and a_2 from G_1 and G_2 , respectively. If the function fails in joining the given addresses, it returns $\perp$ in case of an irrecoverable failure and $\leftrightarrow$ in case of a recoverable failure (intuitively, for efficiency reasons, this happens when the offsets, target specifiers, nesting levels, the kinds of the target objects, their sizes, or validity are found incompatible without going deeper in the sub-SMGs rooted at the given addresses). If the function succeeds, it returns a triple of SMGs G'_1 , G'_2 , G' and an address a from G' such that:

- $MI(G_1) \subseteq MI(G'_1)$ and $MI(G_2) \subseteq MI(G'_2)$ where G_1 and G_2 can differ from G'_1 and G'_2 , respectively, due to an application of join reinterpretation on some of the pairs of objects being joined only.
- The sub-SMGs G''_1 and G''_2 of G'_1 and G'_2 rooted at a_1 and a_2 , respectively, are joined as the sub-SMG G'' of G' rooted at a , i.e., it is required that $MI(G''_1) \subseteq MI(G'') \supseteq MI(G''_2)$.
- The sub-SMG $G' \setminus G''$ is exactly the sub-SMG of G that consists of objects and values that are not removed in Step 10 of the function due to using the principle of delayed join of sub-SMGs described in Section 1.9.6.

The algorithm first checks compatibility of the offsets that the points-to edges leading from a_1 and a_2 are labelled with and checks whether the difference in the nesting depth is appropriate. Then, provided that the target objects $o_1 = o(P_1(a_1))$ and $o_2 = o(P_2(a_2))$ are of the same kind, the algorithm checks whether they are reached via the same target specifiers (when one is a DLS and one a region, the specifiers may differ since the target specifier is not important for the region). If these tests do not pass, the algorithm fails in a recoverable way.

Next, if both o_1 and o_2 are null or if they have already been joined with each other (and they are now reached through a different pair of addresses only), the function *mapTargetAddress* (cf. Alg. 7) is used to join the addresses a_1 and a_2 into a fresh address $a \in A$ (so that $m_1(a_1) = m_2(a_2) = a$) and to create a points-to edge from a to the join of o_1 and o_2 , i.e., the object $m_1(o_1) = m_2(o_2)$. If the objects o_1 and o_2 have not been joined so far, the algorithm checks through the *matchObjects* function (Alg. 8 discussed below) whether o_1 and o_2 can be safely joined (based on their labels, labels of their outgoing edges, and the state of their mapping so far). If

²⁸ As mentioned already in Section 1.3.2, the recovery could be tried even when the impossibility of joining two addresses is discovered much later during joining the sub-SMGs rooted at the given addresses. Then, however, back-tracking would be necessary, which we try to avoid for efficiency reasons.

Algorithm 6 *joinTargetObjects*($s, G_1, G_2, G, m_1, m_2, a_1, a_2, l_{diff}$)**Input:**

- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Section 1.9.
- Addresses $a_1 \in V_1$ and $a_2 \in V_2$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:

- $\perp$ in case of an irrecoverable failure.
- $\leftrightarrow$ in case of a recoverable failure.
- Otherwise, a tuple $(s', G'_1, G'_2, G', m'_1, m'_2, a')$ where:
 - $s' \in \mathbb{J}$ is the resulting join status.
 - G'_1, G'_2, G' are SMGs as defined in Section 1.9.4.
 - m'_1, m'_2 are the resulting injective partial mappings of nodes.
 - a' is an address in G' satisfying the conditions stated in Section 1.9.4.

Method:

1. If $of(P_1(a_1)) \neq of(P_2(a_2))$ or $level_1(a_1) - level_2(a_2) \neq l_{diff}$, return $\leftrightarrow$.
2. Let $o_1 := o(P_1(a_1))$ and $o_2 := o(P_2(a_2))$.
3. If $kind_1(o_1) = kind_2(o_2)$ and $tg(P_1(a_1)) \neq tg(P_2(a_2))$, return $\leftrightarrow$.
4. If $o_1 = \# = o_2$ or $m_1(o_1) = m_2(o_2) \neq \perp$,
 let $(G, m_1, m_2, a) := mapTargetAddress(G_1, G_2, G, m_1, m_2, a_1, a_2)$ and return
 $(s, G_1, G_2, G, m_1, m_2, a)$. In this case, the targets are already joined, and we need to create
 a new address for the corresponding object $o \in O$.
5. Let $s := matchObjects(s, G_1, G_2, m_1, m_2, o_1, o_2)$. If $s = \perp$, return $\leftrightarrow$.
6. Create a new object $o \in O$.
7. Initialize the labelling of o to match the labelling of o_1 if $kind_1(o_1) = \text{dls}$, or to match the
 labelling of o_2 if $kind_2(o_2) = \text{dls}$, otherwise take the labelling from any of them since both
 o_1 and o_2 are equally labelled regions.
8. If $kind_1(o_1) = \text{dls}$ or $kind_2(o_2) = \text{dls}$, let $len(o) := \min(len_1(o_1), len_2(o_2))$.
9. Let $level(o) := \max(level_1(o_1), level_2(o_2))$.
10. If $m_1(o_1) \neq \perp$, replace each edge leading to $m_1(o_1)$ by an equally labelled edge leading to o ,
 remove $m_1(o_1)$ together with all nodes and edges of G that are reachable via $m_1(o_1)$ only,
 and remove the items of m_1 whose target nodes were removed. Likewise for m_2 and o_2 . Note
 that this mechanism is called a *delayed join of sub-SMGs* and explained in Section 1.9.6.
11. Extend the mapping of nodes such that $m_1(o_1) = m_2(o_2) = o$.
12. Let $(G, m_1, m_2, a) := mapTargetAddress(G_1, G_2, G, m_1, m_2, a_1, a_2)$.
13. Let $res := joinSubSMGs(s, G_1, G_2, G, m_1, m_2, o_1, o_2, o, l_{diff})$. If $res = \perp$, return $\perp$.
 Otherwise return $(s, G_1, G_2, G, m_1, m_2, a)$.

so, a fresh object o in G is created that is intended to semantically cover both o_1 and o_2 .

We allow DLSs to be joined with regions as well as with DLSs of a different minimal length, which requires the minimal length of the object o to be adjusted so that it covers both cases, i.e., $len(o) := \min(len_1(o_1), len_2(o_2))$. The nesting level must also be properly chosen: the larger value is chosen since a path through more DLSs is more abstract and hence more general (covering the more concrete path through less DLSs).

Line 10 now solves the case when exactly one of the mappings $m_1(o_1)$ or $m_2(o_2)$ is already defined—the situation when both of them are already defined is handled on line 4 and in function *matchObjects*. This case can occur as a consequence of

the DLS insertion algorithm described in Section 1.9.5, and it is further discussed in Section 1.9.6. For now, it is enough to note that line 10 eliminates the impact of the DLS insertion algorithm on o_1 or o_2 (depending on which of them was mapped within the DLS insertion) as well as on their sub-SMGs, so that they can subsequently be merged as though both of them were encountered for the first time.

Next, the mapping of nodes is extended such that $m_1(o_1) = m_2(o_2) = o$, followed by using the function *mapTargetAddress* to map the addresses a_1 and a_2 to a new address $a \in A$ and to create a points-to edge from a to o (as we have already seen above). Finally, the *joinSubSMGs* function is called recursively for the triple o_1, o_2, o .

While we believe that Alg. 7 is self-explaining, we provide some more intuition to Alg. 8 that performs a local check whether objects o_1, o_2 from SMGs G_1 and G_2 , can be joined under the current mappings m_1 and m_2 , respectively, possibly for the price of updating the current joint status s . The algorithm assumes that at least one of the objects is non-null (the case of both them being null is handled in Alg. 6). Under this assumption, if one of the objects is null, the algorithm fails since null can be joined with null only.

The algorithm also fails if both o_1 and o_2 are mapped but not to each other. Next, even if o_2 is not mapped, but o_1 is mapped to some other object $o'_2 \in O_2 \setminus \{o_2\}$, a failure happens (and symmetrically with the roles of o_1 and o_2 swapped). The reason for the failures based on the mappings is that at most two objects can be mapped together. The only case when the join can succeed with one of the objects mapped is when the other is not mapped—this scenario is a consequence of the further discussed DLS insertion and will be handled by cancelling the mapping of the object that has already been mapped and by mapping o_1 and o_2 to each other.

Subsequently, the algorithm checks whether the sizes of o_1 and o_2 are equal and whether they have the same validity status. In case the objects are DLSs, their defining offsets are checked for equality too. Then, the algorithm checks for all fields that appear in both objects and whose values are already mapped whether their values are mapped to each other. If some of the tests does not pass, the algorithm fails.

Finally, the algorithm checks whether the lengths of the objects or their kinds imply a need to update the join status (joining a more concrete object with a more abstract one). If so, the join status is updated accordingly.

1.9.5 DLS Insertion

Assume that a pair of addresses a_1 and a_2 from SMGs G_1 and G_2 is to be joined in order to allow G_1 and G_2 to be joined into an SMG G . Further, assume that the objects that the addresses refer to cannot be joined, but at least one of them is a DLS—in particular, assume that it is the object with the address a_1 , denote it d_1 , and denote the other object o_2 (the other possibility being symmetric). As mentioned already at the beginning of Section 1.9, in such a case, we proceed as though a_2 pointed to a 0+ DLS d_2 preceding o_2 and labelled equally as d_1 up to its length. We then join d_1 and d_2 into a single 0+ DLS d in G and continue by joining the addresses a_{next} and a_2 where a_{next} is the value stored in the next/prev pointer of d_1 (depending on whether we came to d_1 via the *fst* or *lst* target specifier, respectively). We call this mechanism a *DLS insertion* because it can be seen as if the join of objects was

Algorithm 7 *mapTargetAddress*($G_1, G_2, G, m_1, m_2, a_1, a_2$)**Input:**

- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$ with the corresponding sets of addresses A_1, A_2, A .
- Injective partial mappings of nodes m_1, m_2 as defined in Section 1.9.
- Addresses $a_1 \in A_1, a_2 \in A_2$ referring with the same offset $of = of(P_1(a_1)) = of(P_2(a_2))$ to objects $o_1 = o(P_1(a_1))$, $o_2 = o(P_2(a_2))$, respectively, which are already joined into an object $o = m_1(o_1) = m_2(o_2)$ and which are accessible via target specifiers such that $kind_1(o_1) = kind_2(o_2) \Rightarrow tg(P_1(a_1)) = tg(P_2(a_2))$.

Output:

- A tuple (G', m'_1, m'_2, a) where:
 - G' is an SMG obtained from G by extending its set of addresses by an address a representing the join of a_1 and a_2 (unless G already contains this address) together with a points-to edge from a to o representing the join of the points-to edges between a_1, a_2 and o_1, o_2 , respectively.
 - m'_1 and m'_2 are the resulting injective partial mappings of nodes that are either identical to m_1 and m_2 (if a already exists in G) or obtained from m_1 and m_2 by extending them such that $m'_1(a_1) = m'_2(a_2) = a$.

Method:

1. Let $o_1 := o(P_1(a_1))$, $of := of(P_1(a_1))$.
2. If $o_1 = \#$, let $o := \#$. Otherwise, let $o := m_1(o_1)$.
3. If $kind_1(o_1) = \text{dls}$, let $tg := tg(P_1(a_1))$. Otherwise, let $tg := tg(P_2(a_2))$.
4. If there is an address $a \in A$ such that $P(a) = (of, tg, o)$, return (G, m_1, m_2, a) .
5. Extend A by a fresh address a , then extend P by a new points-to edge $a \xrightarrow{of, tg} o$.
6. Extend the mapping of nodes such that $m_1(a_1) = m_2(a_2) = a$.
7. Return (G, m_1, m_2, a) .

preceded by a virtual insertion of a DLS from one of the SMGs into the other SMG. 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.

Algorithm 9 implements the DLS insertion. The algorithm first checks whether the DLS d_1 from G_1 that we would like to virtually insert into G_2 has not been processed by the join algorithm already in the past. If this is the case and there is some object o in G_2 that has been joined with d_1 into d , the join fails since the d_2 segment (possibly represented by a region as its concrete instance) is not missing, but it is not connected to the rest of the SMG in a way compatible with d_1 (at least not for the current way G_1 and G_2 are being joined). If no such object o exists, the DLS d to which d_1 is mapped in G is used as the result of joining d_1 with the virtually added segment d_2 , and unless even the address a_1 has already been processed, the join continues by the addresses a_{next} and a_2 . Intuitively, this case arises when inserting a single missing segment that should be reachable through several paths in the SMG. Note that such a situation is, in fact, quite usual since a DLS can be reached both forward and backward. The algorithm, however, has to insert a single virtual segment d_2 for all such paths.

If d_1 has not yet been processed, the algorithm checks whether there is some hope that the virtual insertion of d_2 could help (or whether it is better to try to proceed with the join in some other way: e.g., try to insert a DLS from G_2 into G_1 in case both of the addresses a_1 and a_2 point to DLSs or fall-back to introducing

Algorithm 8 *matchObjects*($s, G_1, G_2, m_1, m_2, o_1, o_2$)**Input:**

- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$ and $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Section 1.9.
- Objects $o_1 \in O_1$ and $o_2 \in O_2$ such that $o_1 \neq \# \vee o_2 \neq \#$.

Output:

- $\perp$ in case o_1 and o_2 cannot be joined.
- Otherwise, $s \in \mathbb{J}$ reflecting the impact of the labels of o_1 and o_2 on the relation of the semantics of G_1 and G_2 .

Method:

1. If $o_1 = \#$ or $o_2 = \#$, return $\perp$.
2. If $m_1(o_1) \neq \perp \neq m_2(o_2)$ and $m_1(o_1) \neq m_2(o_2)$, return $\perp$.
3. If $m_1(o_1) \neq \perp$ and $\exists o'_2 \in O_2 : m_1(o_1) = m_2(o'_2)$, return $\perp$.
4. If $m_2(o_2) \neq \perp$ and $\exists o'_1 \in O_1 : m_1(o'_1) = m_2(o_2)$, return $\perp$.
5. If $size_1(o_1) \neq size_2(o_2)$ or $valid_1(o_1) \neq valid_2(o_2)$, return $\perp$.
6. If $kind_1(o_1) = kind_2(o_2) = \text{dls}$, then:
 - If $nfo_1(o_1) \neq nfo_2(o_2)$, $pfo_1(o_1) \neq pfo_2(o_2)$, or $hfo_1(o_1) \neq hfo_2(o_2)$, return $\perp$.
7. Collect the set F of all pairs (of, t) occurring in has-value edges leading from o_1 or o_2 .
8. For each field $(of, t) \in F$ do:
 - Let $v_1 = H_1(o_1, of, t)$ and $v_2 = H_2(o_2, of, t)$.
 - If $v_1 \neq \perp \neq v_2$ and $m_1(v_1) \neq \perp \neq m_2(v_2)$ and $m_1(v_1) \neq m_2(v_2)$, return $\perp$.
9. If $len'_1(o_1) < len'_2(o_2)$ or $kind_1(o_1) = \text{dls} \wedge kind_2(o_2) = \text{reg}$,
let $s := \text{updateJoinStatus}(s, \sqsupset)$.
10. If $len'_1(o_1) > len'_2(o_2)$ or $kind_1(o_1) = \text{reg} \wedge kind_2(o_2) = \text{dls}$,
let $s := \text{updateJoinStatus}(s, \sqsubset)$.
11. Return s .

a 0/1 abstract object as mentioned in Section 1.4). However, unlike in the function *joinTargetObjects*, if we do not want to go deeper in the SMGs (which we do not want for efficiency reasons), there is not so many properties to check since we do not have two objects whose labelling we could compare, but the single DLS d_1 whose counterpart we want to virtually insert into G_2 only. So, we at least check that there is no conflict of the successor addresses a_{next} and a_2 according to the current mapping of nodes, i.e., we require $m_1(a_{next}) = \perp \vee m_2(a_2) = \perp \vee m_1(a_{next}) = m_2(a_2)$.

If the above checks pass, the DLS insertion proceeds as follows: Let F be the set of the linking fields of d_1 that are oriented forward wrt the direction of the traversal. In particular, let $F = \{nfo_1(d_1)\}$ if $tg(P_1(a_1)) = \text{fst}$, and $F = \{pfo_1(d_1)\}$ if $tg(P_1(a_1)) = \text{lst}$. First, the DLS d representing the join of d_1 and the virtually inserted d_2 is created in the destination SMG with the same labelling as that of d_1 up to $len(d) = 0$. Together with basically copying the DLS d_1 from G_1 to G , we also copy the F -restricted sub-SMG rooted at it from G_1 into G , excluding the nodes for which m_1 is already defined (these were already reached through some other paths in the past, and the newly copied part of the F -restricted sub-SMG rooted at d_1 is just linked to them). Subsequently, the algorithm extends the mapping m_1 for the nodes newly inserted to G , creates the appropriate address node $a \in A$ as well as the points-to edge leading from a to d , and extends the mapping of addresses such that

Algorithm 9 *insertLeftDlsAndJoin*($s, G_1, G_2, G, m_1, m_2, a_1, a_2, l_{diff}$)**Input:**

- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Section 1.9.
- Values $a_1 \in V_1$ such that $kind_1(o(P_1(a_1))) = \text{dls}$ and $a_2 \in V_2$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:

- $\perp$ in case of an unrecoverable failure.
- $\leftrightarrow$ in case of a recoverable failure.
- Otherwise, a tuple $(s', G'_1, G'_2, G', m'_1, m'_2, a')$ where:
 - $s' \in \mathbb{J}$ is the resulting join status.
 - G'_1, G'_2, G' are SMGs as defined in Section 1.9.4.
 - m'_1, m'_2 are the resulting injective partial mappings of nodes.
 - a' is an address in G' satisfying the conditions stated in Section 1.9.4.

Method:

1. Let $(d_1, of, tg) := P_1(a_1)$.
2. If $tg = \text{fst}$, let $nf := nfo_1(d_1)$; if $tg = \text{lst}$, let $nf := pfo_1(d_1)$; otherwise return $\leftrightarrow$.
3. Let $a_{next} := H_1(d_1, nf, \text{ptr})$.
4. If $m_1(d_1) \neq \perp$, then:
 - Let $d := m_1(d_1)$.
 - If $\exists o \in O : m_2(o) = d$, return $\leftrightarrow$.
 - If $m_1(a_1) = \perp$, create a new value node $a \in V$ and a new edge $a \xrightarrow{of, tg} d$ in P , and extend the mapping of nodes such that $m_1(a_1) = a$. Otherwise let $a := m_1(a_1)$ and return $(s, G_1, G_2, G, m_1, m_2, a)$.
 - Let $res := joinValues(s, G_1, G_2, G, m_1, m_2, a_{next}, a_2, l_{diff})$. If $res = \perp$, return $\perp$. Otherwise, let $(s, G_1, G_2, G, m_1, m_2, a) := res$.
5. If $m_1(a_{next}) \neq \perp$ and $m_2(a_2) \neq \perp$ and $m_1(a_{next}) \neq m_2(a_2)$, return $\leftrightarrow$.
6. Let $s' := (len_1(d_1) = 0) ? \sqcap : \bowtie$. Let $s := updateJoinStatus(s, s')$.
7. Extend G by a fresh copy of the $\{nf\}$ -restricted sub-SMG of G_1 rooted at d_1 , but excluding the nodes that are already mapped in m_1 such that the copy of d_1 is a DLS d . Then extend the mapping m_1 such that the newly created nodes in $O \cup V$ are mapped from the corresponding nodes of $O_1 \cup V_1$.
8. Initialize the labelling of d to match the labelling of d_1 up to the minimum length, which is fixed to zero, i.e., $len(d) = 0$.
9. Let $a \in V$ be the address such that $P(a) = (of, tg, d)$ if such an address exists in G . Otherwise, create a new value node $a \in V$ and a new edge $a \xrightarrow{of, tg} d$ in P , and extend the mapping of nodes such that $m_1(a_1) = a$.
10. Let $res := joinValues(s, G_1, G_2, G, m_1, m_2, a_{next}, a_2, l_{diff})$. If $res = \perp$, return $\perp$. Otherwise let $(s, G_1, G_2, G, m_1, m_2, a') := res$.
11. Introduce a new has-value edge $d \xrightarrow{nf, \text{ptr}} a'$ in H .
12. Return $(s, G_1, G_2, G, m_1, m_2, a)$.

$m_1(a_1) = a$. The algorithm then continues by joining the pair of successor values a_{next} and a_2 .

1.9.6 Delayed Join of Sub-SMGs

The mechanism of DLS insertion increases the chances for two SMGs to be successfully joined, but it brings one complication to be taken care of. Assume that SMGs G_1 and G_2 are being joined into an SMG G . When applying the DLS inser-

tion mechanism on a DLS d_1 from G_1 , the not yet traversed sub-SMG G'_1 of G_1 reachable from d_1 is copied into G too (likewise for the symmetric case). However, some nodes of G'_1 that are inserted into G_2 may in fact exist in G_2 and be reachable through some other address than the address at which the DLS insertion is started. Note that this *may* but *needs not* happen, and at the time when the DLS insertion is run, it is unknown which of the two cases applies. In theory, a backward traversal through the SMGs could be used here, but we chose not to use it since we were afraid of its potential bad impact on the performance. That is why, we always insert a DLS together with the sub-SMG rooted at it, and as soon as we realize that some DLS d_1 from G_1 whose counterpart was inserted into G_2 does have a real counterpart in G_2 , the result of the join of d_1 with the inserted DLS is deleted (together with all the values and objects reachable from that DLS only) and a proper join—which we denote as the so-called *delayed join*—is used instead (cf. Point 10 of the function *joinTargetObjects*). Note that running the delayed join is indeed necessary since the insertion of a DLS is optimistic in that its sub-SMG is either missing too, or if it is not missing, it is the same as the sub-SMG of the inserted DLS. This needs, however, not to be the case.

1.9.7 Join of SPCs

The *joinSPCs* function (cf. Alg. 10) is the top-level of the join algorithm used when reducing the number of SPCs generated for particular basic blocks of the program being analysed. It inputs a pair of garbage-free SPCs $C_1 = (G_1, v_1)$, $C_2 = (G_2, v_2)$ where $\text{range}(v_1) = \text{range}(v_2) = \text{Var}$ is the common set of program variables, and for each $x \in \text{Var}$, the objects $v_1(x)$ and $v_2(x)$ are labelled equally (this condition necessarily holds for SPCs generated for the same program). The algorithm either fails and returns $\perp$, or it returns the resulting join status and an SPC $C = (G, v)$ where $\text{range}(v) = \text{Var}$, and the triple G_1, G_2, G satisfies the assertions about joined SMGs stated in Section 1.3.2.

The function starts by initializing the mappings of nodes m_1 and m_2 to the empty set and the join status s to $\approx$. Then, for each program variable $x \in \text{Var}$, a fresh region r is created in G , labelled equally as r_1 (or r_2), and the mappings are extended such that $v(x) = m_1(r_1) = m_2(r_2) = r$ where $r_1 = v_1(x)$ and $r_2 = v_2(x)$. Next, for each program variable $x \in \text{Var}$, the *joinSubSMGs* function is called with the corresponding triple of objects $v_1(x), v_2(x), v(x)$. The value of m_1 , m_2 , and s gets propagated between each pair of subsequent calls. Subsequently, the *joinSPCs* function checks whether there was not created any cycle consisting of 0+ DLSs only in G , and if so, the algorithm fails since the DLS consistency requirement would be broken this way (cf. Section 1.2.3).

1.9.8 Join of Sub-SMGs within Abstraction

The *joinSubSMGsForAbstraction* function (cf. Alg. 11) implements the core functionality of the elementary merge operation used as a part of our abstraction mechanism (Section 1.3.3). It inputs an SMG $G = (O, V, \Lambda, H, P)$, a pair of objects $o_1, o_2 \in O$, and a triple of binding offsets $hfo, nfo, pfo \in \mathbb{N}$. If it succeeds, it returns an SMG $G' = (O', V', \Lambda', H', P')$ and a fresh DLS $d \in O'$ which represents the merge of o_1 and o_2 in G' and which is the entry point of the sub-SMG representing a

Algorithm 10 *joinSPCs*(C_1, C_2)**Input:**

- Garbage-free SPCs $C_1 = (G_1, \nu_1)$, $C_2 = (G_2, \nu_2)$ with SMGs
 $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$
 where $\text{range}(\nu_1) = \text{range}(\nu_2) = \text{Var}$, and for each $v \in \text{Var}$,
 the labelling of $\nu_1(v)$ is equal to the labelling of $\nu_2(v)$.

Output:

- $\perp$ in case C_1 and C_2 cannot be joined.
- Otherwise, a tuple (s, C) where:
 - $s \in \mathbb{J}$ is the resulting join status.
 - $C = (G, \nu)$ where $\text{range}(\nu) = \text{Var}$ and the SMG G satisfies the condition
 $MI(G_1) \subseteq MI(G) \supseteq MI(G_2)$.

Method:

1. Let G be an empty SMG G , $\nu := m_1 := m_2 := \emptyset$, $s := \approx$.
2. For each program variable $v \in \text{Var}$:
 - Let $r_1 := \nu_1(v)$ and $r_2 := \nu_2(v)$.
 - Create a fresh region $r \in O$, initialize its labelling to match the labelling of r_1 .
 - Extend the mappings such that $m_1(r_1) = m_2(r_2) = \nu(v) = r$.
3. For each program variable $v \in \text{Var}$:
 - Let $r_1 := \nu_1(v)$, $r_2 := \nu_2(v)$, and $r := \nu(v)$.
 - Let $\text{res} := \text{joinSubSMGs}(s, G_1, G_2, G, m_1, m_2, r_1, r_2, r, 0)$.
 - If $\text{res} = \perp$, return $\perp$. Otherwise, let $(s, G_1, G_2, G, m_1, m_2) := \text{res}$.
4. If there is any cycle consisting solely of 0+ DLSs in G , return $\perp$.
5. Return (s, C) where $C = (G, \nu)$.

join of the sub-SMGs rooted at o_1 and o_2 . What remains to be done in the elementary merge operation is the reconnection of the pointers surrounding o_1 and o_2 to d (apart from those related to their nested sub-SMGs), cf. Section 1.3.3. As an auxiliary result (used in the algorithm of searching for longest mergeable sequences), the algorithm returns the join status $s \in \mathbb{J}$ comparing the semantics of the sub-SMGs rooted at o_1 and o_2 as well as the sets $O_1, O_2 \subseteq O'$ and $V_1, V_2 \subseteq V'$ that contain those objects and values whose join produced the sub-SMG nested below d . Note that the join status returned by *mergeSubSMGs* is not affected by the kinds of o_1, o_2 and the values in their next/prev fields since the loss of information due to merging o_1 and o_2 into a single list segment is deliberate.

The function proceeds as follows. Using the offsets *nfo*, *pfo*, the values of the next/prev fields of o_1 and o_2 are remembered and temporarily replaced by 0 (in order for the subsequently started join of sub-SMGs not to go through these fields). Then a fresh DLS d is created in O and labelled by the given offsets *hfo*, *nfo*, *pfo* and the minimum length equal to $\text{len}_1(o_1) + \text{len}_2(o_2)$, other labels are taken from o_1 (or o_2 since the other labels are equal). The mapping of objects is initialized such that $m_1(o_1) = m_2(o_2) = d$ and the nesting level difference is initialized based on $\text{kind}(o_1)$ and $\text{kind}(o_2)$ using the rules stated in Section 1.9.2. The generic algorithm *joinSubSMGs* is then called on the triple o_1, o_2, d . If it fails or the resulting SMG contains any cycle consisting of 0+ DLSs only, the algorithm exits unsuccessfully. If it succeeds, the values of the next/prev fields in o_1, o_2 , which were temporarily replaced by 0, are restored to their original values. If $\text{kind}(o_1) = \text{kind}(o_2) = \text{reg}$,

Algorithm 11 *joinSubSMGsForAbstraction*($G, o_1, o_2, hfo, nfo, pfo$)**Input:**

- SMGs $G = (O, V, \Lambda, H, P)$.
- Objects $o_1, o_2 \in O$ that are the roots of the $\{nfo, pfo\}$ -restricted sub-SMGs G_1 and G_2 of G that are to be joined and that are such that $level(o_1) = level(o_2)$ and $size(o_1) = size(o_2)$.
- Candidate DLS offsets $hfo, nfo, pfo \in \mathbb{N}$.

Output:

- $\perp$ in case G_1 and G_2 cannot be joined.
- Otherwise, a tuple $(s, G', d, O_1, V_1, O_2, V_2)$ where:
 - $s \in \mathbb{J}$ is the resulting join status (determines the cost of joining G_1 and G_2).
 - G' is an SMG obtained from the input SMG G by extending it with a new DLS d below which the join of G_1 and G_2 is nested.
 - $O_i \subseteq O$ and $V_i \subseteq V$ for $i = 1, 2$ are sets of non-shared objects and values of G_1 and G_2 , respectively.

Method:

1. Let $a_p := H(o_1, pfo, ptr)$, $a_n := H(o_2, nfo, ptr)$, $a_1 := H(o_1, nfo, ptr)$, and $a_2 := H(o_2, pfo, ptr)$.
2. Replace each has-value edge of H leading from o_1 or o_2 and labelled by (nfo, ptr) or (pfo, ptr) by a has-value edge leading to 0 and having the same label.
3. Extend O with a fresh valid DLS d and label it with the head, next, and prev offsets hfo, nfo , and pfo , the minimum length $len(o_1) + len(o_2)$, level $level(o_1)$, and the size $size(o_1)$.
4. If $kind(o_1) = kind(o_2)$, let $l_{diff} := 0$. Otherwise, let $l_{diff} := (kind(o_1) = dls)?1 : -1$.
5. Let $res := joinSubSMGs(\simeq, G_1, G_2, G, \{(o_1, d)\}, \{(o_2, d)\}, o_1, o_2, d, l_{diff})$. If $res = \perp$, return $\perp$. Otherwise let $(s, \rightarrow, G, m_1, m_2) := res$.
6. If G contains any cycle consisting of 0+ DLSs only, return $\perp$.
7. Drop the temporarily created has-value edges of H leading from o_1 and o_2 to 0 and labelled by (nfo, ptr) or (pfo, ptr) and restore the original has-value edges $o_1 \xrightarrow{pfo, ptr} a_p$, $o_1 \xrightarrow{nfo, ptr} a_1$, $o_2 \xrightarrow{pfo, ptr} a_2$, and $o_2 \xrightarrow{nfo, ptr} a_n$.
8. If $kind(o_1) = kind(o_2) = \text{reg}$, increase by one the level of each object and value that appears in the image of m_1 or m_2 , and relabel all points-to edges leading to d by the `all` target specifier.
9. Return $(s, G, d, O \cap range(m_1), V \cap range(m_1), O \cap range(m_2), V \cap range(m_2))$.

the level of each node that appears in the image of m_1 (or m_2) is increased by one (since these nodes are now recognized as nested), and all points-to edges leading to d are relabelled by the `all` target specifier.

The resulting sets of nodes are computed as follows: $O_1 := range(m_1) \cap O$, $O_2 := range(m_2) \cap O$, $V_1 := range(m_1) \cap V$, and $V_2 := range(m_2) \cap V$. The resulting join status is the status returned by *joinSubSMGs*.

1.10 Longest Mergeable Sequences

In this appendix, we formalize the notion of longest mergeable sequences informally introduced in Section 1.3.3. Assume 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 . The *longest mergeable sequence* of objects given by a candidate DLS entry $(o_c, hfo_c, nfo_c, pfo_c)$ where $o_c \in O$ is the longest sequence of distinct valid heap objects of G 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 , and pfo_c as their head, next, and prev offsets; and the following holds for any two neighbouring objects o_1 and o_2 in the sequence:

1. The objects o_1 and o_2 are doubly-linked, i.e., there are addresses $a_1, a_2 \in A$ such that $o_1 \xrightarrow{hfo_c, ptr} a_1 \xrightarrow{hfo_c, tg_2} o_2$ for $tg_2 \in \{fst, reg\}$ and $o_2 \xrightarrow{pfo_c, ptr} a_2 \xrightarrow{hfo_c, tg_1} o_1$ for $tg_1 \in \{lst, reg\}$.
2. The $\{hfo_c, pfo_c\}$ -restricted sub-SMGs G_1, G_2 of G rooted at o_1 and o_2 can be joined using the extended join algorithm that yields the sub-SMG $G_{1,2}$ to be nested below the join of o_1 and o_2 as well as the sets O_1, V_1 and O_2, V_2 of non-shared objects and values of G_1 and G_2 , respectively, whose join gives rise to $G_{1,2}$.
3. 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. This is, $\forall a \in A \setminus V_1 \forall o' \in O_1 \setminus \{o_1\} : o(P(a)) \neq o', \forall o' \in O \setminus O_1 \forall v \in V_1 : v \notin H(o')$, and likewise for o_2, V_2 , and O_2 . Moreover, the sets O_1 and O_2 contain heap objects only.
4. The objects o_1 and o_2 are a part of an uninterrupted sequence. Therefore:
 - a. Regions that are not the first nor last in the sequence can be pointed to their head offset from their predecessor, successor, or from their non-shared restricted sub-SMG only. Formally, if $o_1 \in R \setminus \{o_c\}$, then $\neg \exists o \in O \setminus (O_1 \cup \{o_2, o'\}) \exists a \in A \exists of \in \mathbb{N} : o \xrightarrow{of, ptr} a \xrightarrow{hfo_c, reg} o_1$ where the object o' is the predecessor of o_1 , i.e., $o' = o(P(H(o_1, pfo_c, ptr)))$.²⁹
 - b. If o_1 (o_2) is a DLS, the only object that can point to its end (beginning) is o_2 (o_1), resp. Formally, if $o_1 \in D$, then $\neg \exists o \in O \setminus \{o_2\} \exists a \in A \exists of \in \mathbb{N} : o \xrightarrow{of, ptr} a \xrightarrow{hfo_c, lst} o_1$. If $o_2 \in D$, then $\neg \exists o \in O \setminus \{o_1\} \exists a \in A \exists of \in \mathbb{N} : o \xrightarrow{of, ptr} a \xrightarrow{hfo_c, fst} o_2$.
 - c. Finally, only non-shared objects of G_1 and G_2 can point to non-head offsets of o_1 and o_2 , respectively. Formally, $\neg \exists o \in O \setminus O_1 \exists a \in A \exists of, of' \in \mathbb{N} \exists tg \in \mathbb{S} : o \xrightarrow{of, ptr} a \xrightarrow{of', tg} o_1 \wedge of' \neq hfo_c$, and likewise for o_2 and O_2 .

1.11 Symbolic Execution of Conditional Statements

Checking equality of values is trivial in SMGs since it reduces to identity checking. To check non-equality, we propose a sound, efficient, but incomplete approach as mentioned already in Section 1.3.4. This approach is formalized in the function *proveNeq* shown as Alg. 12 that contains a number of comments to make it self-explaining. The algorithm uses the *lookThrough* function (Alg. 13) for traversing chains of 0+ DLSs while looking for objects whose existence is guaranteed and whose unique addresses can serve as a basis for a non-equality proof. Note that the algorithms do not allow for comparing values of fields with incompatible types. Hence, we require all type-casts to be explicitly represented as separate instructions of the intermediate code so that the fields being compared are always of compatible types.

If neither equality nor inequality of a pair of values v_1 and v_2 , which are compared in a conditional statement, can be established, the symbolic execution must follow both branches of the conditional statement. For each of the branches, we attempt to reflect the condition allowing the execution to enter that branch in the SMG G to

²⁹ Note that no special formal treatment is needed for o_2 since it will take the role of o_1 when checking the next neighbouring pair in the sequence. The above also implicitly ensures that pointers to the head offset of the last object are not restricted.

be processed in the branch, effectively reducing the semantics of G . However, for efficiency reasons, we do again not reflect all consequences of the branch conditions that could in theory be reflected, but only the easy to handle ones, which is sound, and it suffices in all the case studies that we have considered. In particular, we restrict SMGs according to the branch conditions as follows (if none of the below described cases applies, the SMGs are not modified):

- If v_1 and/or v_2 are non-address values, one of them is replaced by the other in the $v_1 = v_2$ branch (a non-address value can be replaced by an address but not vice versa).
- If there is a chain of 0+ DLSs connected into a doubly-linked list in the given SMG such that the `fst` address of the first DLS is v_1 and the last DLS contains v_2 in its next field (or vice versa), the chain is removed by calling the DLS removal algorithm repeatedly in the $v_1 = v_2$ branch. In the $v_1 \neq v_2$ branch, the computation is split to as many cases as the number of 0+ DLSs in the chain is, and in each of the cases, the minimum length of one of the DLSs is incremented (reflecting that at least one of them must be non-empty).
- If v_1 points to a DLS d with the `fst` target specifier and v_2 points to d with the `lst` target specifier (or vice versa), it is clear that $\text{len}(d) < 2$ since otherwise *proveNeq* would succeed in proving the inequality between v_1 and v_2 . In this case, the following two modifications of the encountered SMGs can be applied in the different branches of the encountered conditional statement:
 - In the $v_1 = v_2$ branch, if $\text{len}(d) = 1$, the DLS d is replaced by an equally labelled region (excluding the DLS-specific labels) since d must consist of exactly one concrete node in this case.
 - In the $v_1 \neq v_2$ branch, if $\text{len}(d) = 1$ or the value of the next address is equal to the value of the prev address, i.e., $H(d, pfo(d), \text{ptr}) = H(d, nfo(d), \text{ptr})$, then the minimum length of d is increased to 2 since d must consist of at least two concrete nodes in this case.

Besides equality checking, we also allow for comparisons of addresses using the *less than* or *greater than* operators in case both of the addresses point to the same (concrete) object—we simply compare the offsets. This functionality is needed for successful verification of the NSPR-based case studies mentioned in Section 1.6.

Algorithm 12 *proveNeq*(G, v_1, v_2)**Input:**

- An SMG $G = (O, V, \Lambda, H, P)$.
- A pair of values $v_1, v_2 \in V$ such that $level(v_1) = level(v_2) = 0$.

Output:

- *true* if the inequality between the given pair of values was proven, *false* otherwise.

Method:

1. Let $(v_1, O_1) := lookThrough(G, v_1)$.
2. Let $(v_2, O_2) := lookThrough(G, v_2)$.
3. If $v_1 = v_2$ or $O_1 \cap O_2 \neq \emptyset$, return *false*. // possible sharing of values
4. If $v_1 \notin A$ or $v_2 \notin A$, return *false*. // simplified handling of data values
5. Let $o_1 := o(P(v_1))$ and $o_2 := o(P(v_2))$.
6. If $o_1 = o_2$:
 - If $tg(P(v_1)) = tg(P(v_2))$, return *true*. // same object, different offsets
 - If $tg(P(v_1)) = fst$ and $tg(P(v_2)) = lst$, return $len'(o_1) \geq 2$.
 - If $tg(P(v_1)) = lst$ and $tg(P(v_2)) = fst$, return $len'(o_1) \geq 2$.
 - Otherwise return *false*.
7. If $of(P(v_1)) < 0$ or $of(P(v_2)) < 0$, return *false*. // out of bounds
8. If $v_1 \neq 0$ and $size(o_1) \leq of(P(v_1))$, return *false*. // out of bounds
9. If $v_2 \neq 0$ and $size(o_2) \leq of(P(v_2))$, return *false*. // out of bounds
10. If $v_1 = 0$ or $v_2 = 0$, return *true*. // 0 and a valid address of an object
11. Return $valid(o_1) \wedge valid(o_2)$. // addresses of allocated objects

Algorithm 13 *lookThrough*(G, v)**Input:**

- An SMG $G = (O, V, \Lambda, H, P)$.
- A value $v \in V$ such that $level(v) = 0$.

Output:

- A pair $(v', Visited)$ where:
 - $v' \in V$ is the value reached after all 0+ DLSs are traversed.
 - *Visited* is the set of all 0+ DLSs reachable from v in the forward direction without traversing any other object.

Method:

1. Let *Visited* := $\emptyset$.
2. Let $o := o(P(v))$.
3. If $o \notin \{\perp, \#\}$ and $len'(o) = 0$, then:
 - Let *Visited* := *Visited* $\cup \{o\}$.
 - If $tg(P(v)) = fst$, let $v := H(o, nfo(o), ptr)$ and continue with step 2.
 - If $tg(P(v)) = lst$, let $v := H(o, pfo(o), ptr)$ and continue with step 2.
4. Return $(v, Visited)$.

1.12 Usage of Predator

The simplest way of running Predator is to use either the `slgcc` or `sl11vm` scripts as shown below with `SOURCE.c` being the C program that Predator should analyse:

```
/PATH_TO_predator_DIRECTORY/sl/slgcc SOURCE.c
/PATH_TO_predator_DIRECTORY/sl/sl11vm SOURCE.c
```

The run of Predator can be influenced by a number of options summarised in Table 1.3. The options can be passed to Predator via the environment variable

SL_OPTS). Further settings can then be provided via the `config.h` file from the Predator distribution as discussed below.

Predator can also be invoked directly through the chosen compiler. In particular, one can proceed as follows with the GCC compiler:

```
gcc [CFLAGS] -fplugin=LIBSL [SL_OPTS] SOURCE.c
```

Here, LIBSL represents a path to the Predator plug-in, which always ends with the `.so` suffix for GCC. For instance, when starting Predator from its main directory, LIBSL should be replaced by `./sl_build/libsl.so`. When launching Predator this way, one can use compiler options CFLAGS, such as `-m32/-m64`, as well as Predator options SL_OPTS described in Table 1.3.

For the Clang/LLVM compiler, one has to first create the bitcode file and then start the analysis:

```
clang [CFLAGS] -Xclang -fsanitize-address-use-after-scope\
-g -S -emit-llvm SOURCE.c -o SOURCE.bc
opt SOURCE.bc -lower-switch -load LIBSL -sl [SL_OPTS]
```

A further possibility not requiring additional options but available on Linux only is the following:

```
clang -g SOURCE.c -Xclang -load -Xclang LIBSL
```

For Clang/LLVM, the Predator plug-in has the `.so` suffix on Linux and the `.dylib` suffix on Darwin.

1.12.1 Deeper Configuration of Predator

Apart from using the above described options, one can further configure the behaviour of Predator using the `config.h` file from its distribution. After that, Predator must, of course, be recompiled. Moreover, usage some of the above described options can be replaced by changing the `config.h` file.

Via the `config.h` file, one can control various aspects of the *abstractions* used in Predator. For instance, one can say whether the abstraction should be performed at each basic block or at loop points and/or returns from function calls only, whether abstraction to singly-/doubly-linked list segments is allowed, what the various abstraction thresholds are, set an additional cost for introducing list segments by abstraction, or decide whether the abstraction of SMGs should be applied to longest sequences of SMG nodes that are amenable to abstraction only or whether it can be applied to shorter sequences too. One can also decide whether *integer intervals* may be used to represent values and/or offsets and whether they can be widened.

One can enable/disable the *call cache* implementing a table of summaries and set its various parameters (e.g., whether matches in the table are sought using isomorphism or entailment, how the cache should be pruned, etc.). One can choose the *scheduler* choosing basic blocks to be explored (corresponding to a depth-first search, breadth-first search, or a load-driven search choosing blocks with fewest SPCs waiting to be explored).

Table 1.3: Predator plug-in options. For the GCC-plugin, use with the prefix `-fplugin-arg-libs1`.

Options	Description
<code>-help</code>	Help
<code>-verbose=<uint></code>	Turn on verbose mode
<code>-pid-file=<file></code>	Write PID of self to <code>file</code>
<code>-preserve-ec</code>	Do not affect the exit code
<code>-dry-run</code>	Do not run the analysis
<code>-dump-pp[=<file>]</code>	Dump linearised CL code
<code>-dump-types</code>	Dump also type info
<code>-gen-dot[=<file>]</code>	Generate CFGs
<code>-type-dot=<file></code>	Generate type graphs
<code>-args=<peer-args></code>	Arguments given to the analyser (see below)
Peer arguments	
<code>track_uninit</code>	Report usage of uninitialised values
<code>oom</code>	Simulate possible shortage of memory (malloc can fail)
<code>no_error_recovery</code>	No error recovery, stop the analysis as soon as an error is detected
<code>memleak_is_error</code>	Treat memory leaks as an error
<code>exit_leaks</code>	Report memory leaks while executing a no-return function
<code>verifier_error_is_error</code>	Treat reaching of <code>__VERIFIER_error()</code> as an error
<code>error_label:<string></code>	Treat reaching of the given label as an error
<code>int_arithmetic_limit:<uint></code>	The highest integer number Predator can count to
<code>allow_cyclic_trace_graph</code>	Create a node with two parents on entailment
<code>forbid_heap_replace</code>	Do not replace a previously tracked node if entailed by a new one
<code>allow_three_way_join[:<uint>]</code>	Using the general join of possibly incomparable SMGs (so-called three-way join) 0 never 1 only when joining nested sub-heaps 2 also when joining SPCs if considered useful 3 always
<code>join_on_loop_edges_only[:<int>]</code>	-1 never join, never check for entailment, always check for isomorphism 0 join SPCs on each basic block entry 1 join only when traversing a loop-closing edge, entailment otherwise 2 join only when traversing a loop-closing edge, isomorphism otherwise 3 same as 2 but skips the isomorphism check if possible
<code>state_live_ordering[:<uint>]</code>	On the fly ordering of SPCs to be processed 0 do not try to optimise the order of heaps 1 reorder heaps when joining 2 reorder heaps when creating their union (list of SMGs) too
<code>no_plot</code>	Do not generate graphs (ignore all calls of <code>__sl_plot*()</code> and <code>__VERIFIER_plot()</code>)
<code>dump_fixed_point</code>	Dump SPCs of the obtained fixed-point
<code>detect_containers</code>	Detect low-level implementations of high-level list containers and operations over them (such as various initialisers, iterators, etc.) [16].

One can specify various *limits* of the analysis such as the maximum call depth, the maximum integer constant to be tracked, or the maximum integer to be used as the minimum length of a list segment. One can also control whether parts of the generated SPCs should be shared using a *copy-on-write* mechanism. One can control how much Predator should try to *recover* after a bug is found. Further, one can also enable/disable a static pre-analysis trying to detect variables that are *dead* at certain program locations whose results may subsequently be used to prune variables tracked by the main analysis.

The `config.h` file can be used to activate various *debugging outputs* too. Further information about what can be configured and how can be found directly in the `config.h` file, which contains many explanatory comments too.

1.12.2 Installing and Using PredatorHP

In order to be able to use the Predator Hunting Party (PredatorHP), one has to download its binary version or source code from <https://www.fit.vutbr.cz/research/groups/verifit/tools/predatorhp>. When building from sources, one has to make sure that `git`, `python`, and all dependencies for Predator itself are installed. In the directory with PredatorHP, one can use the script `build-all.sh`. To analyse a single program using PredatorHP, one can use the following script (whose options are described in Table 1.4):

```
predatorHP.py --propertyfile=<prpfile> [--witness=<file>]
               [--compiler-options=CFLAGS] SOURCE.c
```

Table 1.4: Options of the Predator Hunting Party.

Options	Description
<code>-h, --help</code>	Help
<code>-v, --version</code>	Show the program's version number
<code>--propertyfile=<prpfile></code>	A <code>prpfile</code> specifying properties to be verified according to SV-COMP rules
<code>--compiler-options=CFLAGS</code>	Specify options given to compiler (e.g. <code>--compiler-options="-m32 -g"</code>)
<code>--witness=<file></code>	Write the witness trace in XML to <code>file</code>