# Model and Proof Generation for Heap-Manipulating Programs*

Martin Brain, Cristina David, Daniel Kroening, and Peter Schrammel

University of Oxford
Department of Computer Science
`first.lastname@cs.ox.ac.uk`

**Abstract.** Existing heap analysis techniques lack the ability to supply counterexamples in case of property violations. This hinders diagnosis, prevents test-case generation and is a barrier to the use of these tools among non-experts. We present a verification technique for reasoning about aliasing and reachability in the heap which uses ACDCL (a combination of the well-known CDCL SAT algorithm and abstract interpretation) to perform interleaved proof generation and model construction. Abstraction provides us with a tractable way of reasoning about heaps; ACDCL adds the ability to search for a model in an efficient way. We present a prototype tool and demonstrate a number of examples for which we are able to obtain useful concrete counterexamples.

## 1 Introduction

Heap-manipulating programs are notoriously hard to verify. Although there are successful approaches to proving the safety of such programs, e.g. analyses based on three-valued logic [1] and separation logic [2, 3], these analyses are primarily concerned with proof generation and only very few provide a concrete countermodel when a property is violated [4, 5]. Such countermodels can be used as test cases that lead the program execution to the error, and hence, they are invaluable in debugging and understanding the nature of the defect.

As properties of dynamically allocated data structures involve quantifiers, inductive definitions and transitive closure, the concrete interpretation is impractical, and *abstraction* is used to give an approximate representation of sets of concrete values, providing an effective way of dealing with such specifications. In approaches based on abstract interpretation [6], the behaviour of a program is evaluated over the abstract domain using an abstract transformer, which is iterated until the set of abstract states saturates. The generated abstract fixed point is an over-approximation of the set of reachable states of the original program. Now, the difficulty is that, due to the precision loss involved in this analysis, an *abstract countermodel* might be spurious, i.e. it can reach the error state according to an abstract semantics, but not in the concrete semantics. Our goal

* Supported by the ARTEMIS VeTeSS project, UK EPSRC EP/J012564/1 and ERC project 280053.

is hence to compute a *concrete countermodel*, i.e. a witness for the refutation of a property in the form of a concrete heap configuration that, starting from the initial program state, will reach an error state according to concrete semantics.

**Generation of Concrete Countermodels.** We identify and address two specific issues hindering the generation of concrete countermodels.

The first issue is the loss of precision caused by *join operators* when reasoning about multiple execution paths. Join operations are well-known for losing precision [7, 8]. One way to retain precision in such situations is to use disjunctive (powerset) abstractions that express all the possible behaviours individually, avoiding the need for an overapproximative join operator. However, disjunctive abstractions increase space and time requirements exponentially. Thus, in order to achieve scalability, shape analyses generally relinquish the precision offered by the powerset domain in favour of more practical solutions. Options include partially disjunctive heap abstractions [9], or special join operations for the separation logic domain which abstract information selectively [8]. However, due to the precision loss, generating counterexamples is difficult or even impossible.

We propose an analysis capable of regaining just the right amount of precision lost by the join *without* a powerset domain. This is achieved by exploiting recent results on embedding abstract domains inside the Conflict Driven Clause Learning (CDCL) algorithm used by SAT solvers, a framework known as *Abstract Conflict Driven Clause Learning* (ACDCL). As our main focus is on proving aliasing and reachability properties about the heap, we instantiate ACDCL with a heap domain. Our technique produces an abstract countermodel such that any concrete instantiation of it will always provide a valid concrete countermodel.

The second scenario where concrete countermodels are difficult to generate is in the presence of loops, when a *widening operator* may be required to accelerate convergence by extrapolation. For programs with loops, we propose using a combined approach based on loop unwinding and widening. Unwinding allows us to construct concrete refutation witnesses for property invalidations that appear within a certain number of loop unwindings. As this technique is inconclusive if no such invalidations appear for the specified number of unwindings, widening will be used to prove safety in such situations. Any abstract countermodel generated subsequent to widening may be spurious, i.e., may not correspond to any concrete execution in the original program. The experience from bounded model checking indicates that many bugs are found with a small number of loop unwindings.

**Contributions.** Our contributions are summarised as follows:

- An abstract domain specialised for heap-manipulating programs that is used to express aliasing and reachability facts about the heap.
- A verification technique for heap-manipulating programs that interleaves *model construction and proof generation* by exploiting recent results on embedding abstract domains inside the CDCL algorithm used by SAT solvers. As precision loss caused by joins is recovered through decisions and learning, our technique is capable of path-sensitive reasoning. Crucially, by

generalizing causes of conflict, stronger facts can be learned, avoiding case and path enumeration.
- In contrast to most other heap analysis techniques based on abstract interpretation, our analysis produces abstract countermodels that are under-approximations of concrete countermodels and can hence be used to diagnose the property violation. We provide an algorithm for obtaining concrete instantiations of our abstract countermodels.
- We present a prototype tool and demonstrate that we are able to obtain useful concrete counterexamples for typical list-manipulating programs and for benchmarks from SV-COMP'13 involving various kinds of lists and trees.

## 2 A Running Example

For illustration, let us consider the example in Fig. 1, with the corresponding CFG in Fig. 2. Function *running_example* takes as input two pointers $x$ and $y$ to singly-linked lists, and removes the head of the list $x$ and frees it.

**Property 1.** We instrument the code with the assumption at Line 7, stating that $y$ is non-dangling. At the end of the method we expect $y$ not to be affected by the memory deallocation at Line 12, and to remain non-dangling. Accordingly, in the CFG representation, an error state is reached only if $y$ is dangling at the end of the program. The fact that the error location can be reached is easily discovered by any heap verification technique (if $x$ and $y$ are aliases, the memory location pointed to by $y$ does get deallocated, leaving $y$ dangling at location N5). However, the join operation performed at location N5 will usually

```
 1  typedef struct node {
 2    int val;
 3    struct node *n;
 4  } List;
 5
 6  void running_example(List *x, List *y) {
 7    assume(!Dangling(y));
 8
 9    if(x!=null) {
10      List *aux = x;
11      x = x->n;
12      free(aux);
13    }
14
15    assert(!Dangling(y)); // property 1
16  }
```

**Fig. 1.** Running example (with property 1)

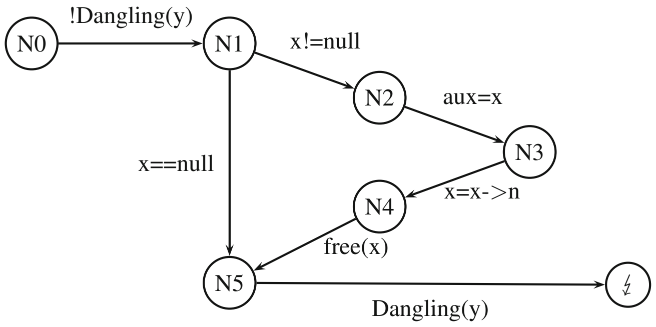

**Fig. 2.** CFG for the running example (with property 1)

```
void counterexample1() {

  List *x, *y;
  x = new(List);
  y = x;


  running_example(x,y);
}
```

**Fig. 3.** A concrete counterexample for property 1 of the running example

```
void counterexample2() {

  List *x, *y;
  y = new(List);
  y->n = null;
  x = y;

  running_example(x,y);
}
```

**Fig. 4.** A concrete counterexample for property 2 of the running example

lose the information about the two independent behaviours corresponding to whether or not $x$ is *null*. This is due to the fact that the most precise property at the join point requires disjunction, and correspondingly a disjunctive domain.

We use a heap domain inside a Conflict-Driven Learning algorithm as used by SAT solvers to construct a countermodel consisting of reachability and aliasing facts without using a powerset domain. The generated abstract countermodel consists of the facts $x \neq null$ and $x = y$. A concrete instantiation of this countermodel as a test case that triggers the property violation is given in Fig. 3. We shall return with details on how the countermodel is generated for this example in Sec. 5, after describing the analysis.

**Property 2.** Now, let us consider a slightly more involved heap property. Assuming the existence of a path from $y$ to *null* via field $n$ at Line 7 in Fig. 1 (by replacing $assume(!Dangling(y))$ with $assume(Path(y, null, n))$), we want to check that this path is preserved at Line 15 ($assert(!Dangling(y))$ is replaced by $assert(Path(y, null, n))$). The abstract countermodel constructed by our technique is such that, in order for the error location to be reached, $x$ must

$$\boxed{\begin{array}{lll}
\mathit{datat} & := & \mathit{struct}\ C\ \{(\mathit{typ}\ v)^*\} \\
e & := & v \mid v{\rightarrow}f \mid \mathrm{new}(C) \mid \texttt{null} \\
S & := & v{:=}e \mid v_1{\rightarrow}f{:=}v_2 \mid \mathrm{free}(v) \mid S_1; S_2 \mid \mathit{if}\ (B)\ S_1\ \mathit{else}\ S_2 \mid \\
 & & \mathit{while}\ (B)\ S \mid \mathrm{assert}(\phi) \mid \mathrm{assume}(\phi) \\
B & := & e_1{=}e_2 \mid e_1{\neq}e_2 \\
A & := & \neg A \mid \mathrm{Path}(v_1, v_2, f) \mid \mathrm{OnPath}(v_1, v_2, v_3, f) \mid \mathrm{Dangling}(v) \\
\phi & := & A \mid B \mid \phi \wedge \phi \mid \phi \vee \phi
\end{array}}$$

**Fig. 5.** Programming Language

be on the path from $y$ to *null*. A concrete countermodel obtained by instantiating the abstract countermodel is shown in Fig. 4.

## 3 Preliminaries

### 3.1 Programming Language

We use the sequential programming language in Fig. 5. It allows heap allocation and mutation, with $v$ denoting a variable and $f$ a pointer field. To simplify the presentation but without loss of expressiveness, we allow only one-level field access, denoted by $v{\rightarrow}f$. Chained dereferences of the form $v{\rightarrow}f_1{\rightarrow}f_2\ldots$ are handled by introducing auxiliary variables. The statement $\mathit{assert}(\phi)$ checks whether the given argument holds for the current program state, whereas $\mathit{assume}(\phi)$ constrains the program state. $\mathit{Path}(v_1, v_2, f)$ captures the fact that the heap location referenced by pointer variable $v_2$ is reachable from the memory location referenced by $v_1$ via the field $f$, whereas $\mathit{OnPath}(v_1, v_2, v_3, f)$ denotes the existence of a memory location referenced by $v_3$ on the path from $v_1$ to $v_2$ via the field $f$. The predicate $\mathit{Dangling}(v)$ indicates that a pointer $v$ points to a non-allocated memory location, i.e. after a call to $\mathit{free}(v)$, pointer $v$ still points to the deallocated memory location.

The predicates *Path*, *OnPath*, and *Dangling* are used to instrument the code in order to prove the absence of memory safety errors such as null pointer dereferences, dangling pointer dereferences and double frees, for example.

### 3.2 Logical Encoding

We translate a program as defined in Sec. 3.1 to an equivalent logical formula. This is done in the spirit of [10] via transforming the program into static single assignment (SSA) form, where each variable is assigned to only once (auxiliary variables are introduced to record intermediate values). This encoding is purely syntactic and can be performed in linear time. It also makes explicit the state dependency of heap accesses and updates. Finally, the formula is translated into conjunctive normal form (CNF).

| | | |
|---|---|---|
| $v_1{:=}v_2{\to}f$ | $\rightsquigarrow_M$ | $v_1{=}\mathit{sel}(M, v_2, f)$ |
| $v_1{\to}f{:=}v_2$ | $\rightsquigarrow_M$ | $M'{=}\mathit{store}(M, v_1, f, v_2)$ |
| $\mathrm{Path}(v_1, v_2, f)$ | $\rightsquigarrow_M$ | $\mathit{Path}(M, v_1, v_2, f)$ |
| $\mathrm{OnPath}(v_1, v_2, v_3, f)$ | $\rightsquigarrow_M$ | $\mathit{OnPath}(M, v_1, v_2, v_3, f)$ |
| $\mathrm{Dangling}(v)$ | $\rightsquigarrow_M$ | $\mathit{Dangling}(M, v)$ |
| $v{:=}\mathrm{new}(C)$ | $\rightsquigarrow_M$ | $M'{=}\mathit{new}(M, v, C)$ |
| $\mathrm{free}(v)$ | $\rightsquigarrow_M$ | $M'{=}\mathit{free}(M, v)$ |
| $\mathrm{assume}(\phi)$ | $\rightsquigarrow_M$ | $\phi$ |
| $\mathrm{assert}(\phi)$ | $\rightsquigarrow_M$ | $\neg\phi$ |

**Fig. 6.** Logical encoding

Most constructs such as sequential statements and *if*-*else* are encoded as usual. *while* loops are unrolled up to a given bound. In Sec. 5 we give an example how iterations beyond this bound can be translated as a fixed point construct in order to prove unbounded safety. Unless otherwise stated, we assume in the sequel that loops have been unwound, and hence, programs are loop-free. For the heap-related statements the encoding rules are given in Fig. 6. We have to capture the effects that an update to a dynamically allocated data structure has on all the pointers referencing that memory location. To this end, we introduce an explicit notion of heap, updated via operators *store*, *new*, *free*. More precisely, if $M$ denotes such a heap, then the encoding rules in Fig. 6 apply, where $M'$ is the updated heap.

Note that for *assert*$(e)$, the negation of the asserted property is added to the formula. Thus, an unsatisfiable formula corresponds to the scenario when the assertion holds, whereas any model of the formula denotes a witness for the invalidation of the asserted property.

*Example 1.* Our example in Fig. 1 is translated to the following formula:

$$\begin{pmatrix} \neg\mathit{Dangling}(M_1, y_1) \wedge \\ \mathit{Dangling}(M_2, y_1) \end{pmatrix} \wedge \left( \begin{pmatrix} x_1{\neq}\mathit{null} \wedge \mathit{aux}_1{=}x_1 \wedge \\ x_2{=}\mathit{sel}(M_1, x_1, n) \wedge \\ M_2{=}\mathit{free}(\mathit{aux}_1) \end{pmatrix} \vee \begin{pmatrix} x_1{=}\mathit{null} \wedge \\ M_1{=}M_2 \end{pmatrix} \right)$$

Transformed into CNF, we have:

$$\begin{array}{ccccc} \neg\mathit{Dangling}(M_1, y_1) & \wedge & (\mathit{aux}_1{=}x_1 \vee x_1{=}\mathit{null}) & \wedge \\ (x_2{=}\mathit{sel}(M_1, x_1, n) \vee x_1{=}\mathit{null}) & \wedge & (M_2{=}\mathit{free}(M_1, \mathit{aux}_1) \vee x_1{=}\mathit{null}) & \wedge \\ (x_1{\neq}\mathit{null} \vee M_1{=}M_2) & \wedge & \mathit{Dangling}(M_2, y_1) \end{array}$$

## 3.3 Concrete Semantics

We define the concrete semantics of a program in terms of the logical formula obtained through the transformation described in the previous section.

Given the set *PVar* of pointer variables and *Fld* of pointer fields, a concrete program state $\rho$ is defined as a triple $(\mathit{Obj}, P, L)$, where $\mathit{Obj}$ denotes the set of all

$$
\begin{array}{ll}
[\![v_1 = v_2]\!]\rho & \equiv L(v_1){=}L(v_2) \\
[\![M'{=}store(M, v_1, f, v_2)]\!]\rho & \equiv \rho \models_c M \Longrightarrow \\
 & \quad (Env, Obj, P/P[(L(v_1), f){\mapsto}L(v_2)], L) \models_c M' \\
[\![M'{=}M]\!]\rho & \equiv \rho \models_c M \Longrightarrow \rho \models_c M' \\
[\![M'{=}free(M, v)]\!]\rho & \equiv \rho \models_c M \Longrightarrow (Env, Obj \backslash L(v), P, L) \models_c M' \\
[\![M'{=}new(M, v, C)]\!]\rho & \equiv \rho \models_c M \Longrightarrow \\
 & \quad \exists o{\notin}Obj.\ (Env, Obj{\cup}\{o\}, P, L/L[v \mapsto o]) \models_c M' \\
[\![sel(M, v, f)]\!]\rho & \equiv P(L(v), f) \\
[\![Path(M, v_1, v_2, f)]\!]\rho & \equiv L(v_1){=}L(v_2)\ \vee \begin{pmatrix} \exists o{\in}Obj, v_3{\notin}PVar.P(L(v_1), f){=}o\ \wedge \\ [\![Path(v_3, v_2, f)]\!](Obj, P, L/L[v_3{\mapsto}o]){\wedge} \\ \neg[\![Path(M, v_2, v_1, f)]\!]\rho \end{pmatrix} \\
[\![OnPath(M, v_1, v_2, v_3, f)]\!]\rho & \equiv \begin{pmatrix} [\![Path(M, v_1, v_3, f)]\!]\rho \wedge [\![Path(M, v_3, v_2, f)]\!]\rho\ \wedge \\ \neg Path(M, v_2, v_3, f)]\!]\rho \end{pmatrix} \\
[\![Dangling(M, v)]\!]\rho & \equiv \nexists o{\in}Obj.\ L(v){=}o
\end{array}
$$

**Fig. 7.** Concrete semantics $ded_c$

the allocated heap nodes plus a distinguished *null* node, $P$ is the set of points-to relations such that $P \subseteq Obj{\times}Fld{\times}Obj$, and $L$ is the pointer variable labelling function, $L : PVar{\rightarrow}Obj$.

Before defining the concrete semantics in Fig. 7, we provide a consistency relation $\rho \models_c M$ between a concrete program state $\rho$ and an explicit heap configuration $M$. The consistency relation states that the same reachability and aliasing facts must hold in both $M$ and $\rho$.

$$
\begin{aligned}
\rho \models_c M \iff & \forall v, v_1, v_2{\in}PVar, f{\in}Fld.[\![Path(M, v_1, v_2, f)]\!]\rho\ \wedge \\
& [\![OnPath(M, v_1, v_2, v, f)]\!]\rho \wedge [\![Dangling(M, v)]\!]\rho
\end{aligned}
$$

Memory allocation $M'{=}new(M, v, C)$ assigns a new heap node to the pointer variable $v$, whereas the deallocation statement $M'{=}free(M, v)$ removes the deallocated heap node from $Obj$. As the mapping of $v$ to the deallocated node is not removed from $L$, the variable points to an invalid memory location, i.e. $v$ is dangling. Note that the definition of *Path* excludes circular paths. We add the predicate *OnPath* for notational convenience; it can be expressed with the help of *Path*.

We refer to concrete program states $(Obj, P, L)$ as structures $\mathcal{S}tructs$, and the concrete domain of structures as $(\mathcal{P}(\mathcal{S}tructs), \subseteq, \cup, \cap)$. A structure $\rho$ is called a model of $\varphi$ if $[\![\varphi]\!]\rho = 1$, and a countermodel if $[\![\varphi]\!]\rho = 0$.

## 4 Our Approach

### 4.1 Abstract Conflict Clause Learning (ACDCL)

Conflict Driven Clause Learning (CDCL) [11] is used by all industrially-relevant propositional SAT solvers. It consists of two complementary phases, iterated until either a model is found or a proof is generated. The first phase, *model search*,

**Algorithm 1.** ACDCL Algorithm

```
while true do
    $S = \top$ ;
    while true do                                              /* PHASE 1: Model Search */
        repeat                                                            /* deduction */
            $S \leftarrow S \sqcap ded(S)$;
        until $S{=}S \sqcap ded(S)$;
        if $S{=}\bot$ then break ;                                          /* conflict */
        if $complete(ded,S)$ then return $(not\ \bot,S)$;         /* return SAT model */
        $S \leftarrow decision(S)$;                                    /* make decision */
    end
    $\mathcal{L} \leftarrow analyse\_conflict(S)$ ;             /* PHASE 2: Conflict Analysis */
    if $\mathcal{L}{=}\top$ then return $(\bot,\mathcal{L})$;                /* return UNSAT */
    $ded \leftarrow ded \sqcap ded_{\mathcal{L}}$;                 /* learn: refine transformer */
end
```

tries to construct a model by using a *partial assignment* of truth values for the propositional variables and extending it using deduction (unit clause propagation) and heuristic guesses. If this constructs a model, then it is returned and the algorithm terminates. If not, a conflicting partial assignment is produced and is passed, along with the reasons for each truth assignment, to the second phase. In contrast to the model-theoretic approach of the first phase, the second phase, *conflict analysis*, takes a proof-theoretic approach. The failed model search is used to guide resolution to produce a clause that allows deduction to avoid the conflict and others like it. If the learnt clause is empty, then the problem is unsatisfiable and the algorithm terminates. The key to the algorithm's effectiveness is the synergy between the two approaches; failed model generation targets the resolution so that high-value clauses are produced and these clauses in turn strengthen deduction and target the heuristics to improve model generation.

An alternative view of CDCL is that the partial assignments are an *over-approximation* of the space of full logical assignments [12]. Thus, the first phase is application of an over-approximate abstract transformer plus extrapolation and the second phase is using an underapproximation and interpolation to generalise the reason for the conflict and increase the precision of the abstract transformer. From this viewpoint it is possible to lift CDCL to give ACDCL [13], which applies the approach but works over a variety of abstract domains (see Alg. 1).

To use ACDCL we first need a concrete domain (a lattice) and a transformer. ACDCL determines whether the transformer projects all elements to $\bot$. In the case of propositional SAT, the concrete domain is the lattice of sets of truth assignments and the transformer projects a set of assignments to the subset of them that are models of the formulae. If this is $\bot$ for all points, then the formulae has no models ($UNSAT$), otherwise it has models ($SAT$). When applied to heaps, the concrete domain is $(\mathcal{P}(\mathcal{S}tructs), \subseteq, \cup, \cap)$ and the transformer projects sets of heaps to the subset of these that are models, thus proving the program safe (if the transformer projects all sets to $\bot$) or finding counterexamples.

The second component needed for ACDCL is an abstract domain and an abstract transformer that over-approximates the concrete transformer. In the case of propositional SAT solvers, the abstract domain is partial assignments and the abstract transformer is unit propagation. In this work, *Heapdom* (see Sec. 4.2) is an abstract domain capable of capturing reachability and aliasing facts in the heap and *ded* (detailed in Sec. 4.3) is the abstract transformer.

The final components required are the completeness test and extrapolation used in the model search phase and the generalisation function used in the conflict analysis phase. Model search applies the abstract transformer until a fixed point is reached (Lines 4 to 6). It then tests for completeness (Line 8). In the case of propositional SAT solvers, this test is checking whether all variables have been assigned. In this work, it is the procedure *complete*, detailed in Sec. 4.4. If the abstract element is not complete (and not $\bot$), then a heuristic guess is needed to add new information (Line 9). Modern propositional SAT solvers use a variant of the VSIDS heuristic, while the procedure *decision* given in Sec. 4.3 is used for heaps. If the model search phase reaches a $\bot$, then the conflict analysis phase is run (Lines 11–13). This uses a generalisation function to underapproximate the reasons for the conflict. In the case of propositional SAT solvers, this is usually First-UIP based learning, while heaps use *analyse_conflict* given in Sec. 4.3.

## 4.2 Abstract Heap Domain

The structure of our abstract heap domain *Heapdom* is given in Fig. 8. An element $S$ of *Heapdom* is a conjunction of predicates *pred* or their negation (literals *Heaplit*), represented as a set. *Heaplit*s are equalities/disequalities representing aliasing information, together with predicates describing reachability/validity facts about heap configurations ($\mathit{Path}^\sharp$, $\mathit{OnPath}^\sharp$ and $\mathit{Dangling}^\sharp$). *Heapdom* forms a lattice $\langle \mathit{Heapdom}, \supseteq, \cup, \cap, \mathit{Heaplit}, \emptyset \rangle$. Note that the meet operation $\sqcap$ is the union of literals $\cup$, the join $\sqcup$ is the intersection $\cap$, and inclusion $\sqsubseteq$ is $\supseteq$. The top element $\top$ is the empty set and bottom $\bot$ is the set of all literals *Heaplit*. For convenience, all inconsistent heap configurations will be projected to bottom, e.g., $\mathit{pred} \wedge \neg \mathit{pred}$ and $\mathit{Path}^\sharp(M, v_1, v_2, f) \wedge \mathit{Dangling}^\sharp(M, v_1)$ are $\bot$.[1] Note that *Heapdom* is finite for programs with a finite number of pointer variables and loops being unwound a finite number of times.

From a *shape analysis* point of view, the reachability predicate $\mathit{Path}^\sharp(M, x, y, n)$ denotes a list segment from $x$ to $y$, whereas $\mathit{Path}^\sharp(M, x, \mathit{null}, n)$ represents a full list. When interested in *heap reachability analysis*, our heap abstract domain is applicable to general heap-allocated data structures that go beyond linked lists.

**Abstract Semantics.** We define now the semantics of programs (as given in Sec. 3) in our abstract heap domain. In Fig. 9, the abstract semantics is given

[1] The intuition behind the latter case is that, in accordance with the semantics in Fig. 7, reachability facts can only involve pointers to allocated heap locations.

$$\begin{array}{lll}
\mathit{Heapdom} & := & \mathcal{P}(\mathit{Heaplit}) \\
\mathit{Heaplit} & := & \mathit{pred} \mid \neg\mathit{pred} \\
\mathit{pred} & := & v{=}e \mid \mathit{Path}^\sharp(M, v_1, v_2, f) \mid \mathit{OnPath}^\sharp(M, v_1, v_2, v_3, f) \mid \mathit{Dangling}^\sharp(v) \\
e & := & \mathit{null} \mid v \mid \mathit{sel}^\sharp(M, v, f) \\
\multicolumn{3}{l}{v \in \mathit{Var}, f \in \mathit{Fld}}
\end{array}$$

**Fig. 8.** Abstract heap domain

in form of an abstract transformer *ded* : *Heapdom* → *Heapdom* that defines the effect $[\![p]\!]^\sharp$ of the predicates $p$ in the logical encoding on abstract heap states $S$.

*Path*, *OnPath*, *Dangling* and *sel* generate the corresponding abstract predicates $\mathit{Path}^\sharp$, $\mathit{OnPath}^\sharp$, $\mathit{Dangling}^\sharp$, and $\mathit{sel}^\sharp$. An equality between heaps $M = M'$ results in duplicating all facts for the new heap.

The transformers for *new*, *free*, and *store* are more complicated because they involve the creation of a new heap $M'$ that is a modified version of the previous heap configuration $M$:

– *new* allocates a new memory location, disjoint from all the other allocated memory locations. Accordingly, the abstract transformer (see Fig. 11) adds a non-null, non-dangling pointer $v$ ($S_2$), generates disequalities between the pointer to the newly allocated location and all the other non-dangling pointers ($S_3$), and copies all known facts to the new heap ($S_1$).
– *free* and *store* have to capture the effects of a *strong update* on the predicates describing heap facts.[2] Corresponding to the effects of a strong update, any pointer variable that is an alias of $v_1$ is pointing to the updated heap object in the new heap, while any pointer variable disjoint from $v_1$ points to the same heap object as it did before the update. Thus, the abstract transformers must capture the truth value of the predicates *Path*, *OnPath* and *Dangling* in the updated heap in a precise manner. For this purpose, there are transformers that define the effects of the memory update on the corresponding positive literal, i.e. whether or not the literal preserves its truth value in the new heap, as well as the effects on the negative literal. The abstract transformer for the *store* operator is the most complex and is shown in Fig. 11. The *free* operation is a simpler version of *store* (omitted here).
The copying of the facts from the previous heap configuration $M$ that are unaffected by the heap update to the new heap $M'$ is performed by the "heap copy" functions, also shown in Fig. 11. The *hcp* functions filter abstract elements present in both $S$ and $S_1$ if the constraint $c$ holds in $S_1$ while also substituting the heap configuration from $M$ to $M'$. The $\hat{\in}$ operator in these definitions is given as $c_1 \wedge c_2 \mathrel{\hat{\in}} S \equiv c_1 \in S \wedge c_2 \in S$, and $c_1 \vee c_2 \mathrel{\hat{\in}} S \equiv c_1 \in S \vee c_2 \in S$.

[2] In shape analysis, a strong update to an abstract memory location overwrites its old content with a new value, whereas a weak update adds new values to the existing set of values associated with that memory location [1, 14].

$$\begin{array}{lll}
[\![v_1 = v_2]\!]^\sharp S & \equiv \{v_1 = v_2\} & \\
[\![v = sel(M, v, f)]\!]^\sharp S & \equiv \{v = sel^\sharp(M, v, f)\} & \\
[\![M'{=}store(M, v_1, f, v_2)]\!]^\sharp S & \equiv ded_{M'=store(M,v_1,f,v_2)} & \text{(see Fig. 11)} \\
[\![M'{=}new(M, v, C)]\!]^\sharp S & \equiv ded_{M'=new(M,v,C)} & \text{(see Fig. 11)} \\
[\![M'{=}free(M, v)]\!]^\sharp S & \equiv ded_{M'=free(M,v)} & \text{(see text)} \\
[\![M'{=}M]\!]^\sharp S & \equiv \{s[M/M'] \mid s \in S\} & \\
[\![Path(M, v_1, v_2, f)]\!]^\sharp S & \equiv \{Path^\sharp(M, v_1, v_2, f)\} & \\
[\![OnPath(M, v_1, v_2, v_3, f)]\!]^\sharp S & \equiv \{OnPath^\sharp(M, v_1, v_2, v_3, f)\} & \\
[\![Dangling(M, v)]\!]^\sharp S & \equiv \{Dangling^\sharp(M, v)\} &
\end{array}$$

**Fig. 9.** Abstract semantics: abstract transformer *ded*

The concretisation function $\gamma$ that relates abstract states $S$ with concrete states $\rho$ (cf. Fig. 7) is given in Fig. 10 with $\gamma(S) = \cap_{s\in S}\gamma_s$.

$$\begin{array}{ll}
\gamma_{v_1=v_2} & \equiv \{\rho \mid [\![v_1 = v_2]\!]\rho\} \\
\gamma_{v=sel^\sharp(M,v,f)} & \equiv \{\rho \mid [\![v = sel^\sharp(M, v, f)]\!]\rho\} \\
\gamma_{Path^\sharp(M,v_1,v_2,f)} & \equiv \{\rho \mid [\![Path(M, v_1, v_2, f)]\!]\rho\} \\
\gamma_{OnPath^\sharp(M,v_1,v_2,v_3,f)} & \equiv \{\rho \mid [\![OnPath(M, v_1, v_2, v_3, f)]\!]\rho\} \\
\gamma_{Dangling^\sharp(M,v)} & \equiv \{\rho \mid [\![Dangling(M, v)]\!]\rho\}
\end{array}$$

**Fig. 10.** Concretisation function $\gamma$

We now state the theorems that establish the soundness of the abstraction:

**Theorem 1.** *The concrete domain $\mathcal{P}(\mathcal{S}tructs)$ and the abstract domain Heapdom form a Galois connection, i.e.* $(\mathcal{P}(\mathcal{S}tructs), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Heapdom, \supseteq)$.

**Theorem 2.** *The abstract semantics is a sound over-approximation of the concrete semantics, i.e.* $(ded_c \circ \gamma)(S) \subseteq (\gamma \circ ded)(S)$.

The proofs establish the inclusion case by case over the structure of $\mathcal{P}(\mathcal{S}tructs)$ and *Heapdom*, respectively $ded_c$ and *ded*.

## 4.3 ACDCL Instantiation

The ACDCL algorithm in Alg. 1 is instantiated using our abstract heap domain as follows:

**Deduction.** The abstract semantics in Sec. 4.2 defines an abstract transformer *ded* : *Heapdom* $\rightarrow$ *Heapdom*, which can also be viewed as a set of deduction rules. During model search ACDCL applies the abstract transformer *ded* to deduce facts until saturation or detection of a conflict (the abstract value $\perp$).

$$[\underline{\textbf{DED}-[\textbf{NEW}]}]$$

$$\frac{\begin{array}{l} S_1 = hcp\left(\left\{\begin{array}{c} v_1{=}sel^\sharp(M,v_2,f), Path^\sharp(M,v_1,v_2,f), \\ OnPath^\sharp(M,v_1,v_2,v_3,f), Dangling^\sharp(M,v)\end{array}\right\}, \texttt{true}, S, M, M'\right) \\ S_2 = \{v{\neq}null, \neg Dangling^\sharp(M',v)\} \\ S_3 = \{v{\neq}v' \mid v'{\in}PVar \wedge v{\neq}v' \wedge \neg Dangling^\sharp(M',v) \in S\}\end{array}}{ded_{M'=new(M,v,C)}(S) = (S_1 \cup S_2 \cup S_3)}$$

$$[\underline{\textbf{DED}-[\textbf{STORE}]}]$$

$$\frac{\begin{array}{lcl}
S_1 & = & hcp\_pos(\{v_3{=}sel^\sharp(M,v_4,f)\}, v_1{\neq}v_3, S, M, M') \\
S_2 & = & hcp\_pos(\{\neg(v_3{=}sel^\sharp(M,v_4,f))\}, v_1{\neq}v_3 \vee v_2{\neq}v_4, S, M, M') \\
S_3 & = & hcp\_neg(\{\neg(v_3{=}sel^\sharp(M,v_4,f))\}, v_1{=}v_3 \wedge v_2{=}v_4, S, M, M') \\
S_4 & = & hcp\_pos\left(\begin{array}{l}\{Path^\sharp(M,v_3,v_4,f)\}, \\ \neg Path^\sharp(M,v_3,v_1,f) \vee \neg OnPath^\sharp(M,v_3,v_4,v_1,f) \vee Path^\sharp(M,v_2,v_4,f), \\ S, M, M'\end{array}\right) \\
S_5 & = & hcp\_neg\left(\begin{array}{l}\{Path^\sharp(M,v_3,v_4,f)\}, \\ Path^\sharp(M,v_3,v_1,f) \wedge OnPath^\sharp(M,v_3,v_4,v_1,f) \wedge \neg Path^\sharp(M,v_2,v_4,f), \\ S, M, M'\end{array}\right) \\
S_6 & = & hcp\_pos(\{\neg Path^\sharp(M,v_3,v_4,f)\}, \neg Path^\sharp(M,v_3,v_1,f) \vee \neg Path^\sharp(M,v_2,v_4,f), S, M, M') \\
S_7 & = & hcp\_neg(\{\neg Path^\sharp(M,v_3,v_4,f)\}, Path^\sharp(M,v_3,v_1,f) \wedge Path^\sharp(M,v_2,v_4,f), S, M, M') \\
S_8 & = & hcp\_pos\left(\begin{array}{l}\{OnPath^\sharp(M,v_3,v_4,v_5,f)\}, \\ Path^\sharp(M,v_3,v_4,f) \wedge \left(\begin{array}{c}\neg Path^\sharp(M,v_3,v_1,f) \vee \\ \neg OnPath^\sharp(M,v_3,v_5,v_1,f) \wedge Path^\sharp(M,v_2,v_5,f)\end{array}\right), \\ S, M, M'\end{array}\right) \\
S_9 & = & hcp\_neg\left(\begin{array}{l}\{OnPath^\sharp(M,v_3,v_4,v_5,f)\}, \\ \neg Path^\sharp(M,v_3,v_4,f) \vee \left(\begin{array}{c}Path^\sharp(M,v_3,v_1,f) \wedge OnPath^\sharp(M,v_3,v_5,v_1,f) \\ \vee \neg Path^\sharp(M,v_2,v_5,f)\end{array}\right), \\ S, M, M'\end{array}\right) \\
S_{10} & = & hcp\_pos\left(\begin{array}{l}\{\neg OnPath^\sharp(M,v_3,v_4,v_5,f)\}, \\ \neg Path^\sharp(M,v_3,v_4,f) \vee \neg Path^\sharp(M,v_3,v_1,f) \vee \neg Path^\sharp(M,v_2,v_5,f), \\ S, M, M'\end{array}\right) \\
S_{11} & = & hcp\_neg\left(\begin{array}{l}\{\neg OnPath^\sharp(M,v_3,v_4,v_5,f)\}, \\ Path^\sharp(M,v_3,v_4,f) \wedge Path^\sharp(M,v_3,v_1,f) \wedge Path^\sharp(M,v_2,v_5,f), \\ S, M, M'\end{array}\right) \\
S_{12} & = & hcp\left(\left\{\begin{array}{c} v_1{=}sel^\sharp(M,v_2,f'), Path^\sharp(M,v_1,v_2,f'), \\ OnPath^\sharp(M,v_1,v_2,v_3,f'), Dangling^\sharp(M,v)\end{array}\right\}, \texttt{true}, S, M, M'\right) \\
S_{13} & = & hcp\_pos(\{Dangling^\sharp(M,v_3)\}, v_1{\neq}v_3, S, M, M') \\
S_{14} & = & \{v_2{=}sel^\sharp(M',v_1,f)\}
\end{array}}{ded_{M'=store(M,v_1,f,v_2)}(S) = \begin{cases}\bot & \text{if } (Dangling^\sharp(M,v_1) \vee v_1{=}null) \mathrel{\hat\in} S \\ \bigcup_{i=1..14} S_i & \text{otherwise}\end{cases}}$$

| Functions for copying heap facts: | | |
|---|---|---|
| $hcp(S,c,S_1,M,M')$ | $=$ | $\{s[M/M'] \mid s \in S \cap S_1 \wedge c \mathrel{\hat\in} S_1\}$ |
| $hcp\_pos(S,c,S_1,M,M')$ | $=$ | $\{s[M/M'] \mid s \in S \cap S_1 \wedge c \mathrel{\hat\in} S_1\}$ |
| $hcp\_neg(S,c,S_1,M,M')$ | $=$ | $\{\neg s[M/M'] \mid s \in S \cap S_1 \wedge c \mathrel{\hat\in} S_1\}$ |

**Fig. 11.** Abstract transformers

In addition to the abstract transformer *ded*, we make use of a *transitive closure* transformer that infers all the possible new heap facts from existent ones, e.g. it infers $Path(M,v_1,v_3,f)$ from $Path(M,v_1,v_2,f)$ and $Path(M,v_2,v_3,f)$.

The transitive closure is also necessary to canonicalize abstract elements. This is in particular important for checking whether an abstract value is equivalent to $\perp$ (which has multiple representations).

We can show that the abstract transformer *ded* we presented in Sec. 4.2 is the best abstract transformer in our heap domain. However, this is not necessary for the completeness and termination of the ACDCL algorithm: less precise transformers can be used, as they will be subsequently refined through decisions and learning. This is frequently a worthwhile trade-off for performance.

**Maintaining the Set of Relevant Decisions.** During this propagation phase, we maintain a set $H$ ("hints") of literals (*Heaplit*) for the benefit of the decision heuristic explained in the next section. The set $H$ consists of those literals that appear in the transformer's hypothesis and are not present in the current abstract model $S$, and hence, they constitute the set of literals that guarantees that a decision actually triggers a deduction. Hints are collected during the application of the transformers based on the $c, S_1$ arguments to the heap copy functions (*hcp*, *hcp_pos*, *hcp_neg*):

$$\mathit{extract\_hints}(c, S_1) = \{h \mid h \in \mathit{literals}(c) \backslash S_1\}$$

where

$$\mathit{literals}(c) = \begin{cases} \mathit{literals}(c_1) \cup \mathit{literals}(c_2) & \text{if } c = (c_1 \wedge c_2) \\ \mathit{literals}(c_1) & \text{if } c = (c_1 \vee c_2) \\ \{s\} & \text{if } c = s \end{cases}$$

returns a set of literals in formula $c$ that is sufficient to trigger a deduction: note that in the case of disjunction in the hypothesis $c$, only one disjunct is added in order to avoid unnecessary decisions. A consequence of this definition is that whenever the completeness test (Line 8 in Alg. 1) fails, there must be at least one hint in $H$.

**Decisions.** Once no new information can be deduced through propagation, the ACDCL algorithm makes a decision by guessing the truth value of a predicate and adding it to the partial abstract model $S$. As explained above, we collect the relevant potential decisions $H$ during deduction in order to restrict the choices for decisions. We may use any decision heuristic *get_a_hint* to return (and remove) an element from $H$. A trivial option is to simply take the first element, but we could also use elaborate ranking heuristics in order to prioritise certain literals. The *decision* function (Line 9 in Alg. 1) adds the obtained literal to the abstract model:

$$\begin{aligned} \mathit{decision}_H &: \mathit{Heapdom} \rightarrow \mathit{Heapdom} \\ \mathit{decision}_H(S) &= S \cup \{\mathit{get\_a\_hint}(H)\} \end{aligned}$$

**Conflict Analysis and Learning.** In the conflict analysis phase, the learning function identifies the cause of the conflict:

$$\begin{aligned} \mathit{analyse\_conflict} &: \mathit{Heapdom} \rightarrow \mathcal{P}(\mathit{Heapdom}) \\ \mathit{analyse\_conflict}(S) &= (\mathit{generalise} \circ \mathit{complement})(\mathit{decisions}(S)) \end{aligned}$$

where *decisions* returns the set of decision literals in the current iteration of main (outer) iteration of the ACDCL algorithm. As a learning heuristic, the conjunction of all the decisions leading to conflict is initially complemented according to the *complement* function:

$$\begin{aligned} &\mathit{complement} && : \mathit{Heapdom} \to \mathcal{P}(\mathit{Heapdom}) \\ &\mathit{complement}(S) && = \{\{\neg s\} \mid s \in S\} \end{aligned}$$

Subsequently, the found cause of conflict is generalised using the *generalise* function:

$$\mathit{generalise} : \mathcal{P}(\mathit{Heapdom}) \to \mathcal{P}(\mathit{Heapdom})$$

Generalisation is important to efficiently prune the search space so as to avoid case enumeration. Generalisation is based on heuristics (e.g. First-UIP in SAT solving). The *generalise* function we have implemented is for example able to perform the following generalisations:

- $x{=}y \Longrightarrow \forall f \in \mathit{Fld}.\mathit{Path}^\sharp(x, y, f)$
- $\neg\mathit{Path}^\sharp(M, x, y, f) \Longrightarrow x{\neq}y$.

The set $\mathcal{L}$ returned by *analyse_conflict* is then used to build the learned transformer $\mathit{ded}_{\mathcal{L}}$ that is used to refine the abstract transformer *ded* (Alg. 1, Line 13):

$$\begin{aligned} &\mathit{ded}_{\mathcal{L}} && : \mathit{Heapdom} \to \mathit{Heapdom} \\ &\mathit{ded}_{\mathcal{L}} && = \textstyle\bigsqcup_{\ell \in \mathcal{L}} \mathit{ded}_{\ell} \end{aligned}$$

meaning $\mathit{ded}_{\mathcal{L}}(S) = \bigcap_{\ell \in \mathcal{L}}(S \cup \ell)$. In our implementation, transformer refinement is realised by conjoining the CNF formula corresponding to $\mathcal{L}$ with the formula.

## 4.4 Soundness and Completeness

Using various properties of the abstract domain, we show that the instantiation of the ACDCL framework given here is a decision procedure (i.e. it is sound, complete and terminating) for loop-free programs. Sketches of the heap-specific parts of the proof are given here, the correctness of the framework is shown in [13]. We recall the definition of $\gamma$-completeness:

**Definition 1.** *A transformer ded is $\gamma$-complete at $S \in \mathit{Heapdom}$ if $\gamma(\mathit{ded}(S)) = \mathit{ded}_c(\gamma(S))$.*

The way we construct the set of relevant possible decisions $H$ enables a simple implementation of *complete*:

$$\mathit{complete}_H(\mathit{ded}, S) \equiv (H = \emptyset)$$

which has the following properties:

**Lemma 1.** *If $\mathit{complete}_H(\mathit{ded}, S)$ is true then ded is $\gamma$-complete at $S$.*
*If $\mathit{complete}_H(\mathit{ded}, S)$ is false then the decision function refines the partial abstract model, $S \subset \mathit{decision}_H(S)$.*

Central to the correctness of the system is the invariant that *ded* is an over-approximation of $ded_c$ and that each iteration of the outer loop strengthens it. This can be proven inductively; Theorem 2 gives the base case and the inductive step is a consequence of the next lemma:

**Lemma 2.** *Given ded, an over-approximation of $ded_c$, the second phase of the algorithm gives a strictly stronger over-approximation of $ded_c$.*

Given $\varphi$, a loop-free program with a finite number of variables, termination, soundness and completeness follow:

**Theorem 3.** *Alg. 1 terminates.*

*Proof sketch. Heapdom* is finite, and thus the application of *ded* will reach a fixed point. Likewise, owing to the second part of Lemma 1, the main loop of phase 1 will either exit as $complete_H(ded, S)$ is true or will eventually reach $\bot$. Finally, as *Heapdom* is finite, there are only a finite number of over-approximations of $ded_c$, so the invariant implies the main loop will terminate.

**Theorem 4.** *If Alg. 1 returns (not $\bot$, S) then* $\forall \rho \in \gamma(S).[\![\varphi]\!]\rho = 1$

*Proof sketch.* The preconditions of the statement that returns *not* $\bot$ include $complete_H(ded, S)$ and $S = ded(S)$. Using Lemma 1, $\gamma(S) = ded_c(\gamma(S))$, thus all elements of the concrete set are models. Note that $\gamma(S)$ can contain an infinite family of models; the next section shows how to produce counterexamples.

**Theorem 5.** *If Alg. 1 returns $\bot$ then* $\forall \rho \in \mathcal{S}tructs.[\![\varphi]\!]\rho = 0$

*Proof sketch.* The only statement that returns $\bot$ occurs when $\mathcal{L}{=}\top$, i.e. $ded \sqcap ded_{\mathcal{L}}$ is the function that maps all abstract elements to $\bot$. Using the invariant this is an overapproximation, thus $ded_c(\top) = \bot$, thus there are no models of $\varphi$.

## 4.5 From Abstract to Concrete Countermodels

In Fig. 2 we provide a high-level overview of the algorithm for the generation of concrete countermodels from abstract ones. Our goal is to compute a concrete countermodel that contains only three types of elements: $v_1{=}v_2$, $v{=}null$ and $v_1{=}sel(M, v_2, f)$. Initially, the abstract model is split into positive reachability-based constraints ($S_1$), and the rest of the abstract model ($S_2$) (Lines 1 and 2, respectively). Subsequently, $S_1$ is used to infer candidate concrete models, which are exhaustively generated in $C$ such that each path constraint in $S_1$ is concretised to a length of at most $l$ (Line 5). The inner loop (Lines 6–13) iteratively attempts to find a valid concrete counterexample. In order to qualify, a candidate must be consistent with the rest of the abstract constraints in $S_2$. This consistency check translates into a satisfiability call to our instantiation of ACDCL (Line 10). If no candidate qualifies, the minimum length of the heap paths is incremented and the process is reiterated with new candidates.

**Algorithm 2.** Concretisation of Abstract Countermodels

```
1  $S_1 \leftarrow \{s | s \in S$ and $(s = Path(M, v_1, v_2, n)$ or $s = OnPath(M, v_1, v_2, v_3, n))\}$;
2  $S_2 \leftarrow S \backslash S_1$ ;
3  $l = 0$;
4  while $true$ do
5      $C \leftarrow \{c \mid c$ is a concrete model of $S_1$ for paths of length $\leq l\}$ ;
6      while $C \neq \emptyset$ do
7          $\pi \leftarrow$ choose a model from $C$;
8          $C \leftarrow C \setminus \pi$;
9          $\forall s_i \in (\pi \cup S_2).\phi \leftarrow \bigwedge_i s_i$ ;
10         if $\phi$ is SAT then
11             return $\pi$;
12         end
13     end
14     $l \leftarrow l + 1$
15 end
```

**Theorem 6.** *Given an abstract counterexample (an abstract element different from $\perp$ at which ded is $\gamma$-complete, then Alg. 2 always terminates with a finite concrete countermodel.*

*Proof sketch.* A partial ordering of the current variables can be computed such that $Path(M, v_1, v_2, n) \Rightarrow v_1 \preceq v_2$ and $OnPath(M, v_1, v_2, v_3, n) \Rightarrow v_1 \preceq v_3 \preceq v_2$. It is always possible to generate a countermodel from this ordering without introducing any auxiliary variables. As $S_1 \cup S_2 \neq \perp$, there must exist one such countermodel that satisfies both $S_1$ and $S_2$.

## 5 Experiments

We have implemented the ACDCL instantiation with the *Heapdom* domain described in Alg. 1 in a prototype solver and connected it to the Model Checker CBMC 4.6. The source code of the prototype tool and the benchmarks are available online.[3] The prototype was subsequently used to verify memory safety and reachability properties for some typical list-manipulating programs for singly-linked lists, e.g. filter, find, bubble sort, and benchmarks from the SV-COMP'13 *list-properties* and *memsafety-ext* sets.

In addition to checking memory safety, i.e. absence of null or dangling pointer dereferences, we have also added reachability assertions, e.g. the reachability predicate Path$(x, y, n)$ denotes a list segment from $x$ to $y$, and Path$(x, \mathit{null}, n)$ represents a full list.

**1. Countermodel Construction.** In order to test the soundness of the tool and its capacity to construct witnesses for property refutation, we applied it

[3] `http://www.cprover.org/svn/cbmc/branches/ESOP2014-heap`

**Table 1.** Experimental results: lines of code (loc), clauses (cls) and analysis time (t, in seconds) for safe and unsafe versions of the benchmarks; timeout 15 minutes (t.o.). All experiments were performed with two loop unwindings.

| | | safe | | unsafe | |
|---|---|---|---|---|---|
| Benchmark | loc | cls | t | cls | t |
| bubble sort* | 40 | 728 | 0.86 | 732 | 10.1 |
| concat* | 24 | 45 | 0.08 | 45 | 0.08 |
| copy* | 40 | 159 | 0.20 | 158 | 1.50 |
| create* | 27 | 100 | 0.15 | 100 | 0.15 |
| filter* | 42 | 259 | 0.68 | 259 | 0.55 |
| find* | 23 | 47 | 0.09 | 35 | 0.08 |
| insert* | 17 | 18 | 0.13 | 16 | 0.15 |
| reverse* | 20 | 81 | 0.07 | 83 | 0.08 |
| traverse* | 15 | 16 | 0.08 | 18 | 0.07 |
| alternating_list | 65 | 278 | 0.21 | 282 | 0.28 |
| list_flag | 62 | 244 | 0.46 | 246 | 0.57 |
| list | 60 | 294 | 1.57 | 296 | 1.14 |
| simple_built_from_end | 34 | 124 | 0.17 | 120 | 0.16 |
| simple | 45 | 157 | 0.18 | 157 | 0.17 |
| splice | 89 | 474 | 0.33 | 478 | 0.55 |
| dll_extends_pointer | 64 | 302 | 0.29 | 308 | 0.79 |
| skiplist_2lvl | 91 | 520 | t.o. | 514 | 4.87 |
| skiplist_3lvl | 105 | 722 | t.o. | 726 | 15.5 |
| tree_cnstr | 85 | 942 | 3.31 | 922 | 2.56 |
| tree_dsw | 117 | 1037 | 2.43 | 984 | 3.24 |
| tree_parent_ptr | 95 | 844 | 0.43 | 811 | 45.9 |
| tree_stack | 93 | 1413 | 0.50 | 1394 | t.o. |

to safe and unsafe, i.e. faulty, versions of our benchmarks (with loops unwound twice), followed by manually inspecting the countermodels generated for the unsafe versions. The results of these experiments are given in Table 1. Both the safe and unsafe versions of each program are instrumented with memory safety assertions. Those marked with a * have additional reachability assertions.

*Example 2.* We describe how countermodel construction proceeds for our running example in Fig. 1. Recall the corresponding logical encoding in Sec. 3.2.

**Model Search (1).** After the *first propagation*, the partial abstract model consists of the elements $\neg Dangling(M_1, y_1)$ and $Dangling(M_2, y_1)$, representing neither a conflict, nor a complete countermodel. Thus, a *decision* constrains $x_1$ to be not *null*, and the model search loop is reiterated. This time, the abstract transformers for $aux_1{=}x_1$, $x_2{=}sel(M_1, x_1, n)$ and $M_2{=}free(M_1, aux_1)$ are applied. As the application of $M_2{=}free(M_1, aux_1)$ is imprecise (no aliasing information for $x_1$ and $y_1$ is available), a second *decision* is made assuming that $y_1$ is not reachable from $x_1$, i.e. $\neg Path(M_1, x_1, y_1, n)$. Consequently, a new application of $M_2{=}free(M_1, aux_1)$ will preserve the non-dangling knowledge about $y_1$ from $M_1$ to $M_2$, resulting in the *conflict* $\neg Dangling(M_2, y_1)$ and $Dangling(M_2, y_1)$.

**Conflict Analysis (2).** The cause of conflict is $x_1 \neq null \wedge \neg Path(M_1, x_1, y_1, n)$. Hence, one possible clause to be learned is $x_1{=}null \vee Path(M_1, x_1, y_1, n)$. As we want to avoid case enumeration, we generalise the cause of conflict. For example, the fact that $x_1$ and $y_1$ are not aliases is more general than $\neg Path(M_1, x_1, y_1, n)$, i.e. $\neg Path(M_1, x_1, y_1, n) \Rightarrow x_1 \neq y_1$.[4] Thus, we learn $x_1{=}null \vee x_1{=}y_1$ and restart the model search phase.

**Model Search (3).** After a decision $x_1 \neq null$, the abstract element $x_1{=}y_1$ is added to the abstract model and $M_2{=}free(M_1, aux_1)$ is now complete. Thus,

[4] A heap path between two pointer variables may be empty (cf. Fig. 7).

the abstract transformer passes the completeness test, and the abstract countermodel $\{x_1 \neq \mathit{null}, x_1{=}y_1\}$ is generated.

**Concrete Countermodel Generation (4).** Fig. 3 shows a test case triggering the property violation obtained from the abstract countermodel using Alg. 2.

**2. Safety Proof Generation.** When failing to construct a concrete refutation witness after a bounded number of unwindings, a safety proof is attempted by applying a fixed point computation. This computation makes use of a widening operator that loses information about individual points-to facts by generalising them to reachability facts, e.g. $y{=}\mathit{sel}(M, x, n)$ is generalised to $\mathit{Path}(M, x, y, n)$.

We do not detail the fixed point computation and the widening operator as they are both rather standard (in particular in the spirit of [15]). In order to investigate feasibility of our approach, we have experimented with the backend solver of our prototype by trying simple list-manipulating programs like filter, concat, copy, and reverse on singly-linked lists, where we computed invariants for each loop.

For instance, for the *concat* example in Fig. 12, we replace the while loop by the invariant $\mathit{Path}(x, \mathit{curr}, n) \wedge \mathit{curr} \to n = \mathit{null}$ resulting from the fixed point computation with widening. The transformer for the store $\mathit{curr} \to n = y$ joins this information yielding $\mathit{Path}(x, y, n)$, thus proving safety.

## 6 Related Work

**ACDCL.** We build on previous results on embedding abstract domains inside the Conflict Driven Clause Learning (CDCL) algorithm used by modern SAT solvers in a framework known as Abstract Conflict Driven Clause Learning (ACDCL) [13]. Other promising instances of this framework include a bit-precise decision procedure for the theory of binary floating-point arithmetic [16].

```
void concat(List *x, List *y) {
  List *curr;
  assume(!Path(x,y));
  if(x==null)  x = y;
  else {
    curr = x;
    while(curr->n != null) curr = curr->n;
    curr->n = y;
  }
  assert(Path(x,y));
}
```

**Fig. 12.** List concatenation

The ACDCL framework enables the design of *property-driven* analyses (analyses that propagate facts starting with states exhibiting a certain property of interest, e.g. backward under-approximation). The model search phase of the ACDCL framework exhibits the property-driven nature of backward analysis, while using transformers that are forward in nature. This differs from most abstract-interpretation-based analyses for heap-manipulating programs [8, 17, 1, 9], which perform exhaustive forward propagation.

**Model vs. Proof Generation.** Among the successful approaches for proving safety of heap-manipulating programs, the most prominent ones are based on three-valued logic [1] and separation logic [2, 3]. Although the majority of these analyses are mainly concerned with proof generation and do not construct witnesses for the refutation of a property [8, 17, 9], there are recent advances in diagnosing failure with the purpose of refining shape abstractions [4, 5]. These works start with failed proofs, and subsequently try to find concrete countermodels from possible spurious abstract ones. Thus, the proof generation phase is *independent* from model construction. The same remark applies to an approach designed to find memory leaks in Android applications [18], which answers reachability queries by refining a points-to analysis through a backwards search for a witness. In contrast, the ACDCL framework, and hence our instantiation, exploits the *interleaving of model construction and proof generation* to mutually support model search and conflict analysis.

**Decidable Logics.** Recently, several decidable logics for reasoning about linked lists have been proposed [19–23]. Piskac et al. provide a reduction of decidable separation logic fragments to a decidable first-order SMT theory framework [20]. A decision procedure for a new logic that is an alternation-free sub-fragment of first-order logic with transitive closure and no alternation between universal and existential quantifiers is described in [19]. While these works design decision procedures for handling quantified constraints, we use an abstract domain enabling us to employ the ACDCL framework. As a direct implication, we do not have a separation between propositional and theory-specific reasoning. Thus, theory-specific facts can be learned during conflict analysis, which may result in better pruning of the search space.

## 7 Conclusions

We have presented a verification technique for reasoning about aliasing and reachability in the heap which uses ACDCL to perform both proof generation and model construction. Proof generation benefits from model construction by learning how to refine the abstract transformer, and in turn, it assists in pruning the search space for a model. The ACDCL framework was instantiated with a newly designed abstract heap domain. From a shape analysis perspective, this domain allows expressing structural properties of list segments, whereas in a more general context of reachability analysis it can denote reachability facts regardless of the underlying data structure.

## References

1. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118 (1999)
2. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)
3. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bulletin of Symbolic Logic 5(2), 215–244 (1999)
4. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.M.: Diagnosing abstraction failure for separation logic-based analyses. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 155–173. Springer, Heidelberg (2012)
5. Beyer, D., Henzinger, T.A., Théoduloz, G., Zufferey, D.: Shape refinement through explicit heap analysis. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 263–277. Springer, Heidelberg (2010)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
7. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 343–358. Springer, Heidelberg (2009)
8. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
9. Manevich, R., Sagiv, M., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
10. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. FMSD 25(2-3), 105–127 (2004)
11. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
12. D'Silva, V., Haller, L., Kroening, D.: Satisfiability solvers are static analysers. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 317–333. Springer, Heidelberg (2012)
13. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: POPL, pp. 143–154 (2013)
14. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
15. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 379–392. Springer, Heidelberg (2007)
16. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding floating-point logic with systematic abstraction. In: FMCAD, pp. 131–140 (2012)
17. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6), 26 (2011)
18. Blackshear, S., Chang, B.Y.E., Sridharan, M.: Thresher: precise refutations for heap reachability. In: PLDI, pp. 275–286 (2013)
19. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 756–772. Springer, Heidelberg (2013)

20. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013)
21. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. J. Log. Alg. Prog. 73(1-2) (2007)
22. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622 (2011)
23. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 167–182. Springer, Heidelberg (2012)