# Handling Heap Data Structures in Backward Symbolic Execution

Robert Husák, Jan Kofroň, and Filip Zavoral

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{husak, zavoral}@ksi.mff.cuni.cz, jan.kofron@d3s.mff.cuni.cz

**Abstract.** Backward symbolic execution (BSE), also known as weakest precondition computation, is a useful technique to determine validity of assertions in program code by transforming its semantics into boolean conditions for an SMT solver. Regrettably, the literature does not cover various challenges which arise during its implementation, especially when we want to reason about heap objects using the theory of arrays and to use the SMT solver efficiently. In this paper, we present our achievements in this area. Our contribution is threefold. First, we summarize the two most popular state-of-the-art approaches used for BSE, denoting them as *disjunct propagation* and *conjunct combination*. Second, we present a novel method for modelling heap operations in BSE using the theory of arrays, optimized for incremental checking during the analysis and handling the input heap. Third, we compare both approaches with our heap handling implementation on a set of program examples, presenting their strengths and weaknesses. The evaluation shows that conjunct combination is the most efficient variant, exceeding the straightforward implementation of disjunct propagation in an order of magnitude.

**Keywords:** backward symbolic execution, weakest precondition, heap data structures, input heap, theory of arrays

## 1 Introduction

*Symbolic execution* is an established technique to explore semantics of programs, create tests with high code coverage and discover bugs [2]. To achieve that, it systematically explores the state space of the program reachable from the entry point, transforming the possible execution paths into boolean constraints. These constraints are usually passed to an SMT solver to determine the reachability of the corresponding paths. To reason about objects on the heap, several of the practically-usable tools [6, 18] use the theory of arrays [10], which can be handled by the most of the state-of-the-art SMT solvers [8].

If we are not interested in the exploration of the whole program and we want to inspect only one particular problematic place instead, we can use the *backward* variant of symbolic execution, sometimes referred to also as the *weakest precondition* analysis [7, 9]. As its name suggests, backward symbolic execution starts at the assertion of our interest and traverses the execution direction backwards. If it manages to reach the entry point and find an assignment satisfying the path constraints, it can provide us with a valuable test case. Otherwise, if no under-approximation is used and the assertion violation is proved to be unreachable, it is validated.

As we can see, each run of backward symbolic execution can be very expensive in terms of resources. However, the information it provides is potentially very detailed and useful for detecting the causes of errors. Therefore, it is important to use it in an appropriate context. There is a plethora of techniques which use some kind of abstraction, enabling them to efficiently analyse large programs for the cost of introducing false positives [17, 15, 11]. Backward symbolic execution can be then used only at the places where these techniques found potential errors in order to examine them further. For example, the authors of Snugglebug were able to verify 29 of 38 feasible *null* dereference exceptions found by FindBugs [11] in a Java codebase of 750 kLOC [7]. Another usage of backward symbolic execution is to run it in an interactive fashion, enabling programmers to gather as much information about a specific program error as possible [12].

Although backward symbolic execution can be indeed very useful, it is not as popular in the literature as the forward variant. Therefore, many important design considerations and potential complications have to be rediscovered during each implementation. For example, when calling an SMT solver multiple times, it is often efficient for the subsequently analysed conjunctions to share a common prefix. As we illustrate in Section 2, that complicates the way to handle the constraints, because we then should not alter the existing ones, only add new ones. We tackle this problem and other issues in our contributions:

1. We summarize the existing algorithms commonly used for backward symbolic execution in Section 3.
2. We present a novel way to transform heap operations into boolean constraints in Section 4. These transformations fit into the mentioned algorithms and utilize performance enhancements of the state-of-the-art SMT solvers.
3. We compare the performance of all the presented approaches on a set of code examples in Section 5.

In Section 6, we compare our approach to the most important papers and tools related to our work, while Section 7 concludes.

## 2 Problem

All the issues related to implementing a backward symbolic execution tool stem from its very nature. Forward symbolic execution starts with a fixed set of symbolic input variables and all the gradually constructed constraints can be essentially build from them. With backward symbolic execution, the situation is different, as the set of input variables constantly changes according to the variables encountered along the way. Every time a variable is read, it is added to this set; every time it is assigned to, it is removed from it.

We will illustrate the approaches on a simple method `ScalarExample` in Listing 1.1. The forward variant starts with a symbolic variable $a$ assigned to `a`, at lines 2 and 3 it then assigns 1 to `b` and 2 to `c`. When it comes to the assertion `a != b` at line 4, it interprets it using the known values and negates the expression to discover any error inputs, resulting in a simple condition $a = 1$. The backward variant starts directly at the

Listing 1.1: Sample C# code to demonstrate symbolic execution

```
1  void ScalarExample(int a) {
2      int b = 1;
3      int c = 2;
4      Debug.Assert(a != b);
5  }
6
7  void HeapExample1(Node a) {
8      a.next = new Node();
9      Node b = a.next;
10     Debug.Assert(a != b);
11 }
12
13 void HeapExample2(Node a) {
14     Node b = a.next;
15     Node c = new Node();
16     Debug.Assert(b != c);
17 }
```

assertion, creating a condition $a = b$, where a and b are the input variables corresponding to their symbolic variables $a$ and $b$, respectively. As the condition is not dependent on c in any way, it can safely skip its assignment at line 3. Next, the assignment of 1 into b at line 2 must effectively remove it from the set of input variables and replace it by 1 in the condition, resulting again in $a = 1$. Although this simple example does not demonstrate any significant differences, things become more complicated when heap operations and various efficiency optimizations are involved:

*Constantly changing input heap:* The rule with the constantly changing set of input variables applies to the input heap as well. Moreover, its analysis gets more complicated, because the objects from the input heap might get intertwined with the ones created during the analysed program run. Consider the assertion `a != b` on the line 10 in Listing 1.1. At first, the objects in the input heap might be possibly referenced by both a and b. The field read at line 9 causes b to be loaded from the current input heap. However, we cannot assume that the loaded reference is from the input heap as well, because it can always be assigned an explicitly allocated object, as at line 8. Therefore, we need to provide a way to correctly distinguish between the input heap and the explicitly allocated objects and to enable their various interactions.

*Incremental solving:* There are many usage scenarios of SMT solvers where we need to call them successively on similar formulas. E.g., in the case of symbolic execution, we might want to explore two independent code branches sharing the same prefix. Therefore, a modern SMT solver can be usually used incrementally, with the possibility to cache certain knowledge between subsequent calls. To add and remove assertions, they offer two useful mechanisms: an *assertion stack* and *assumptions*. The former enables us to use a stack-based system of scopes containing the particular assertions, with the ability to destroy all the data of the topmost scope while retaining the remaining ones. The latter works by adding every assertion $a$ in the form of $l \implies a$, where $l$ is a literal specified later during each call of the solver. Because we want to utilize these features to optimize our SMT calls, it is important that we construct the formulas in a proper way, possibly combining the strengths of both techniques.

## 3 Backward Symbolic Execution

### 3.1 Notation

Let us clarify the terminology and semantics of various formulas and symbols used in this paper. If we speak about a *function* or *mapping* $g : A \to B$, it is understood as a partial function, hence defined on the subset of its domain $A$. If $g(a)$ for $a \in A$ is not defined, we denote it as $g(a) = undef$. The function $g[a \to b]$ is defined to be the same as $g$, except for it maps $a$ to $b$. This notation can be generalized for a set: $g[\{a_1, ..., a_n\} \to b] = g[a_1 \to b]...[a_n \to b]$.

As to the formalism used for SMT queries, many-sorted first-order logic is used. Because the meaning of *sort* in logic corresponds to the meaning of *type* in computer science, we will use these two names interchangeably, according to the context. The *signature* $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ comprises a set of *sorts* $\mathcal{S}$, a set of *function symbols* $\mathcal{F}$ and a set of *predicate symbols* $\mathcal{P}$. *Symbolic variables* $\Sigma_v$, *terms* $\Sigma_t$, *atoms* $\Sigma_a$, and *formulas* $\Sigma_f$ are derived from the signature, using the standard recursive way. A formula $\varphi[a/b]$ is constructed by replacing all the occurrences of $a$ in $\varphi$ by $b$. The function $\textsc{Fresh}_{\Sigma_v}$ retrieves a symbolic variable not yet used in any context. In general, for any domain $A$, $\textsc{Fresh}_A$ retrieves a variable $a \in A$, which is not yet used in the analysis.

Let $C$ be a set of classes contained in the analysed program. For each class $c \in C$, there is a corresponding set $F_c$ containing all its fields. All the fields in the program are contained in $F = \bigcup_{c \in C} F_c$. To enable working with reference symbolic variables, we introduce a set of reference sorts $R = \{\sigma_c \mid c \in C\} \subset \mathcal{S}$. As an instrument to reason about types of fields and variables, we use function $t : V \cup F \to \mathcal{S}$. $F_{t(v)}$ as an abbreviation for $F_c$ where $\sigma_c = t(v)$. Reference fields $F_R$ and value fields $F_V$ are defined as follows:

$$F_R = \{f \in F \mid t(f) \in R\} \qquad F_V = F \setminus F_R$$

Analogicaly, reference and value variables:

$$V_R = \{v \in \Sigma_v \mid t(v) \in R\} \qquad V_V = \Sigma_v \setminus V_R$$

Note that the sorts $R$ representing reference variables are used only to ease formal description of heap operations. They are effectively replaced by functions on arrays and integers, as we describe in Section 4. All the reference variables are expected to point to objects on the heap, there is no notion of low-level pointers and of accessing stack variables by references.

To reason about a certain program, we expect it to be given as a control flow graph (CFG). Each node $n$ contains at most one operation $n$.op and each edge $e = (n_1, \psi_e, n_2)$ is marked with a condition $\psi_e$. The possible operations follow: scalar assignment of term

$v_t \leftarrow_s t$, reference assignment $v_t \leftarrow_r v_r$, reference comparison assignment[1] $v_t \leftarrow (v_r^1 = v_r^2)$, new object creation $v_t \leftarrow_r$ `new` $T$, field read $v_t \leftarrow v_r.f$ and field write $v_r.f \leftarrow v_v$. Note that the last two operations can occur both for reference and variable fields, in some cases we denote it by $\leftarrow_r$ and $\leftarrow_s$, respectively. Assertions are modelled as edges to special nodes.

To keep the scope limited, this paper does not directly address handling loops, interprocedural analysis or recursion. In order to evaluate our approach on programs of smaller size, we use a simple preprocessor for CFGs, which unwinds the loops for a given number of iterations. To handle interprocedural calls, we plan to extend it to handle inlining of the procedures up to a certain level of recursive calls. We are aware that this approach is underapproximate and does not scale well on larger programs. In order to mitigate this issue, we will inspire from the existing tools which were able to efficiently extend backward symbolic execution into an interprocedural analysis. Snugglebug uses directed call graph construction and tabulation, enabling it to explore the call graph lazily and reuse certain summaries obtained for each procedure [7]. Alter combines backward and forward symbolic execution to combine method summaries, utilizing interpolant computation to learn from infeasible paths [16].

## 3.2 Algorithm

Whereas in forward symbolic execution we usually want to reasonably spread our analysis among the state space to achieve high code coverage [2], backward symbolic execution often works by gathering summaries towards the entry point [7, 1]. At least in the intraprocedural case it is a natural approach, as we are interested in finding a feasible path between the entry node and the target node.

```
BSE($cfg$, $n_{trg}$)
1: var $states$: $node \rightarrow state$
2: $states[n_{trg}] \leftarrow$ state representing $true$
3: for all node $n$ in $cfg$ sorted by reverse dependency on $n_{trg}$ do
4:     var $deps \leftarrow \{(\psi_e, n_{dep}, states[n_{dep}]) \mid$ edge $(n, \psi_e, n_{dep})$ in $cfg\}$
5:     $states[n] \leftarrow$ MERGE($n$, $deps$)
6:     if DOSOLVE() $\wedge$ SOLVE(GETCONDITION($states$, $n$)) = $UNSAT$ then
7:         $states[n] \leftarrow$ state representing $false$
8: return $states$
```

Fig. 1: Backward symbolic execution algorithm

An overall algorithm structure is shown in Fig. 1. Given a target node $n_{trg}$, it traverses $cfg$ backwards towards the entry node and gathers useful information along the way. The information is stored in the *states* associative array. In the beginning, because we expect $cfg$ to be acyclic, we can sort its nodes according to their topological order in

[1] We did not put reference comparison directly in the edge conditions so that we can describe its processing later in the unified manner with the other heap operations, see Section 4.

the reversed *cfg*, skipping those not reachable from $n_{trg}$. This way, when processing a node, we are sure that the dependent nodes were already processed. For each node, we gather the states of the directly adjacent nodes and their corresponding edge conditions into *deps*. MERGE is a core function responsible for inferring the state of a given node according to its dependencies. DOSOLVE is a heuristic returning *true* for the entry node and possibly also during the exploration so that certain infeasible parts get pruned. GETCONDITION is used to gather the condition corresponding to a given state, returns false if $n_{trg}$ is unreachable from that node. Eventually the algorithm retrieves all the computed states. The caller can then extract interesting pieces of information from it, such as a possible input driving the execution towards $n_{trg}$.

```
state: formula in DNF
MERGEDISJ(n, deps)
 1: var merged ← disjunction of {$\psi_e \wedge d \mid d$ disjunct in $\varphi$, $(\psi_e, n_{dep}, \varphi) \in deps$}
 2: return PROCESSOPERATIONDISJ(SIMPLIFY(merged), n.op)
PROCESSOPERATIONDISJ(state, $v_t \leftarrow_s t$)
 1: return state[$v_t$ / $t$]
GETCONDITIONDISJ(states, n)
 1: return states[n]
```

Fig. 2: Backward symbolic execution implementation using disjunct propagation

In the literature, we have identified two main possible implementations of this algorithm. The first, listed in Fig. 2, is based on formulas in DNF and their propagation in the form of disjuncts [7]. MERGEDISJ merges the disjunctions in all the dependent nodes and enhances them by their corresponding edge conditions, simplifying the resulting formula by SIMPLIFY and passing it to PROCESSOPERATIONDISJ. SIMPLIFY applies various techniques of reducing a disjunction size while maintaining its semantics. PROCESSOPERATIONDISJ handles an assignment $v_t \leftarrow_s t$ by replacing the target variable $v_t$ by the term $t$ representing its value, GETCONDITIONDISJ simply returns the disjunction for the given node. Heap operations and the implementation of GETCONDITION for heaps is described in Section 4.

In Fig. 3, the other implementation is listed [1]. Instead of propagating a set of disjuncts to the entry node, it associates each node $n$ with a condition $\psi_n$ describing its semantics and control flow. As seen in GETCONDITIONCONJ, to reason about the whole path, we can pass a conjunction of these conditions to an SMT solver, which enables an efficient incremental usage. Since we can reason about mutable variables, our state contains also a map *vers* containing a version number for each encountered program variable. Unlike the previous case, we need to store certain information about a symbolic heap in each state; the details will be provided in Section 4.

MERGECONJ works as follows. Each node $n$ is associated with a propositional variable $c_n$ to express that the control flow reached it. The condition $\psi_n$ is an implication with $c_n$ on the left side. The right side consists of two parts: a join condition $\psi_{join}$ and an operation condition $\psi_{op}$. The purpose of $\psi_{join}$ is to model the branching of the control

```
$state$: (node condition $\psi_n$, $vers$: $V_V \to \mathbb{N}$, $heap$)
MergeConj($n$, $deps$)
1: var $mergedVers \leftarrow$ merge $vers$ in $deps$ to get the highest of each entry
2: (var $mergedHeap$, var $heapJoinConds$) $\leftarrow$ MergeHeaps($heaps$ in $deps$)
3: var $\psi_{join} \leftarrow$ disjunction of
   {versioned $\psi_e \wedge c_{n_{dep}} \wedge$ JoinVers($vers$, $mergedVers$) $\wedge$ $heapJoinCond$ for $heap$
   $\mid (\psi_e, n_{dep}, \psi_{n_{dep}}, vers, heap) \in deps$}
4: (var $\psi_{op}$, var $finalVers$, var $finalHeap$) $\leftarrow$ ProcessOperationConj($mergedVers$, $mergedHeap$,
   $n.op$)
5: return ($c_n \implies \psi_{join} \wedge \psi_{op}$, $mergedVers$, $mergedHeap$)
ProcessOperationConj($vers$, $heap$, $v_t \leftarrow_s t$)
1: if $vers[v_t] = undef$ then
2:     return ($true$, $vers$, $heap$)
3: else
4:     var $oldVer \leftarrow vers[v_t]$
5:     var $newVers \leftarrow vers[v_t \to oldVer + 1$, {unknown variables in $t$} $\to 0]$
6:     return ($v_t^{oldVer} = t$ versioned by $newVers$, $newVers$, $heap$)
GetConditionConj($states$, $n$)
1: return $c_n \wedge$ conjunction of $\psi_{n'}$ where $n'$ is reachable from $n$
```

Fig. 3: Backward symbolic execution implementation using conjunct combination

flow by creating a disjunction on the edge conditions where each disjunct redirects the flow to the corresponding $c_{n_{dep}}$ and possibly synchronizes the variable versions of the dependent nodes using JoinVers. An operation condition, created by ProcessOperationConj, handles an assignment by making the given variable under its current version equal to the given term and associating the variable with a new version. Notice that if $v_t$ has not been encountered so far, we can safely ignore the operation. Heap operation handling is described in Section 4, including the merging of heaps.

As we can see, each implementation is connected with certain advantages and disadvantages. The disjunct propagation approach is based on maintaining sets of disjuncts and simplifying them, while the operations are handled as term substitutions. As a result, the final condition can be potentially much simpler than in the other case, because it does not contain any helper variables representing various versions and Simplify can help to get rid of various repetitive patterns. On the other hand, if the simplification is not successful enough, the size of the resulting formula can be exponential with respect to the number of calls to Merge. Furthermore, it cannot fully utilize incremental SMT solvers, as they work by adding immutable conjuncts to an assertion stack. The conjunction combination case is able to use them efficiently and the generated condition size is usually linear with respect to the number of the analysed nodes, which is redeemed by the presence of helper variables.

Although in this work, the implementations are handled as two separate techniques, we plan to pursue a way to efficiently combine them, using the best features of both. Creating simple procedure summaries might be crucial for developing an efficient interprocedural algorithm, whereas utilizing an incremental SMT solver might help with exploring large program state.

## 4 Modelling Heap Using Array Theory

### 4.1 Main Idea

The array theory enables SMT solvers to reason about heap memory in forward symbolic execution and concolic execution [6, 18]. Its axioms, in addition to those of theory of uninterpreted functions, follow [4]:

$$\forall a, i, j\ (i = j \Rightarrow read(write(a, i, v), j) = v)$$

$$\forall a, i, j\ (i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j))$$

$$\forall a, b\ ((\forall i(a[i] = b[i]) \Leftrightarrow a = b)$$

As we can see, array theory generalises the operations of the array data structure, with the only difference being the immutability of the array variables. In the forward variant of symbolic execution, a common approach is to associate an array with each defined field and represent all the references by integers [18]. Reading a value from an instance can be then naturally modelled by using the *read* operation on the corresponding reference and array. Writing a value is similarly performed by using the *write* operation to produce a new version of the particular array. To ensure that different allocations of new objects do not reference the same object, we can use an internal counter and increment it every time an allocation is performed (allocation site counting) [3]. To denote *null* references, 0 is used.

All these principles can be directly adopted for backward symbolic execution as well [7]. However, to our knowledge there is a serious problem not sufficiently tackled in the literature. If we do not analyse a program from its very start, we expect that there are existing objects on the heap, prior to the entry point, where the analysis begins from, being called the input heap. Therefore, each reference can point either to an object located in the input heap, to *null*, or to an object allocated explicitly during the analysis. The problem is that if we do not constrain the references from the input heap to be distinct from the explicitly allocated objects, the SMT solver might produce a model where the references from those two distinct groups are equal.

Consider the method `HeapExample1` in Listing 1.1. Apart from the instance created at line 8, there is also an instance passed as the parameter `a`. Because this instance was created before the method call, we must assert that it is distinct from the former. Otherwise, an SMT solver might create an invalid model where $a = b$, so the input heap contains a reference to the explicitly created instance before it even exists.

Furthermore, all the references from `a` in the beginning of the method must point either to *null* or to other input heap instances. In method `HeapExample2`, we can see the reason. If we do not constrain the reference loaded from `a.next` in any way, the SMT solver can create a model where $b = c$.

A natural approach used in our solution is to restrict all the input heap objects to be represented as negative integers. In the case of forward symbolic execution, we can remember the first version of the variable representing each field and then constrain it whenever we access it from any reference. In `HeapExample1`, we start with an input reference $a \leq 0$ and an array variable $next^0$ representing the field `next` in the beginning. At line 8 we assert $next^1 = write(next^0, a, 1)$, making $next^1$ the current version of the field. Nevertheless, when we access the field at line 9, we can retrospectively add a constraint $read(next^0, a) \leq 0$, making `a.next` from the input heap either to be *null* or to reference another object from the input heap. As we only add constraints and never alter the existing ones, this approach is naturally efficient for incremental solving.

When trying to using this approach in backward symbolic execution, we encounter a major problem. Because the view of the input heap continues to change as the analysis proceeds backwards, we cannot use any single version of the array variable representing the given field. For example, if we decide to set the input heap constraint at line 9 as $read(next^0, a) \leq 0$, we prevent `a.next` to be assigned any explicitly created instance, which exactly happens at line 8.

As we explain below, we tackle this problem by creating a helper "input" array variable for each field and firmly asserting its equality with the current field variable version only when explicitly checking the condition. A similar solution is created also for the reference variables, as they face the same issue.

## 4.2 Operation Definitions

The implementation of heap operation handling for the disjunct propagation algorithm from Fig. 2 is shown in Fig. 4. To mark symbolic variables corresponding to reference variables and fields, we use the $s$ superscript. For a reference variable $v$, $v^s$ represents a symbolic integer variable; for a field $f$, $f^s$ represents a symbolic array variable indexed by integers. The value sort of $f^s$ is $t(f)$ if $f \in F_V$, integer otherwise. The semantics of a reference variable $v$ is as follows. If $v^s = 0$, $v$ is *null*; therefore, $null^s = 0$. If $v^s > 0$, $v$ references an object explicitly created during the analysed part of the program. Otherwise, if $v^s < 0$, $v$ references an object in the input heap, i.e., it is created in the not yet analysed code.

We can see that assignments, comparisons and new object creations are implemented as simple replacements of the corresponding target variables in the existing formula. $\textsc{Fresh}_{\mathbb{N}_+}$ ensures that each created object is represented by a distinct number. A field write replaces all the occurrences of the given field array variable $f^s$ by an expression that writes the given value $v_v$ to $f^s$ on the index given by the instance $v_r$. Because $v_v$ can be either a reference variable or a scalar value (term), we use a helper function $\textsc{Symb}$ which optionally adds the $s$ superscript if $v_v \in V_R$. Because the operation would not have been executed if $v_r$ was *null*, we also add the condition $v_r^s \neq 0$.

When reading a value from a field, we distinguish between the scalar case $\leftarrow_s$ and the reference case $\leftarrow_r$. In the scalar case, we just replace the read variable by the formula representing array read and assert that $v_r$ is not *null*. In the reference case, we also need to handle the aforementioned problems with input heap. Therefore, for each field $f$, we create also a helper symbolic array variable $f^{in}$, which is never rewritten during any operation. By adding $read(f^{in}, v_r^s) \leq 0$ we ensure that any read from the input heap using $v_r$ will always either be *null* or reference an input heap object. These variables are then used in $\textsc{GetConditionDisjHeap}$, where we associate all the constraints gathered for

```
ProcessOperationDisjHeap(state, op)
 1: switch op do
 2:     case $v_t \leftarrow_r v_v$
 3:         return $state[v_t^s \;/\; v_r^s]$
 4:     case $v_t \leftarrow (v_1 = v_2)$
 5:         return $state[v_t \;/\; (v_1^s = v_2^s)]$
 6:     case $v_t \leftarrow_r$ new $T$
 7:         return $state[v_t^s \;/\; \textsc{Fresh}_{\mathbb{N}^+}()]$
 8:     case $v_r.f \leftarrow v_v$
 9:         return $state[f^s \;/\; write(f^s, v_r^s, \textsc{Symb}(v_v))] \land v_r^s \neq 0$
10:     case $v_t \leftarrow_s v_r.f$
11:         return $state[v_t \;/\; read(f^s, v_r^s)] \land v_r^s \neq 0$
12:     case $v_t \leftarrow_r v_r.f$
13:         return $state[v_t^s \;/\; read(f^s, v_r^s)] \land v_r^s \neq 0 \land read(f^{in}, v_r^s) \leq 0$

GetConditionDisjHeap(states, n)
 1: var inputRefs ← gather reference symbolic variables in states[n]
 2: return $states[n] \land \bigwedge_{f \in F_R} f^s = f^{in} \land \bigwedge_{v \in inputRefs} v \leq 0$
```

Fig. 4: Heap operation modelling in the disjunct propagation approach from Fig. 2

them with their corresponding fields. We also identify all the input heap references and constrain them to be $\leq 0$ as well.

As we can see from the algorithms in Fig. 2 and Fig. 4, the disjunct propagation approach is straightforward to implement and the condition transformations directly correspond to the operations. However, its efficiency heavily depends on the implementation of formula handling, especially their substitution and simplification. The best results are supposed to be obtained by a custom implementation which reflects all the requirements of the particular project [7]. It is also possible to reuse existing solutions, for example the efficient algorithms for terms in Z3 using its API [8].

Nevertheless, even with the best implementation possible, the conditions in certain programs can grow beyond a reasonable complexity, where every term substitution or simplification consumes too many resources. Therefore, we will now focus on the implementation of heap operations in Fig. 5 for the conjunct combination based algorithm shown in Fig. 3. Although the semantics regarding fields as array variables and references as integer variables remains the same, there are several differences, making the operations more complex. Because each condition is associated with the semantics of a single node and we cannot manipulate conditions for the already processed nodes, we are not allowed to use term substitution. Instead, we utilize a version-based mechanism similar to the implementation of assignment in ProcessOperationConj, where the version of the given variable is incremented and its equality with the particular term is added to the condition.

As a result, each node is also associated with a symbolic heap $(\eta, \alpha)$. The environment $\eta$ contains all the current input heap reference variables and maps each of them either to 0 or to an integer symbolic variable. In the beginning of the analysis, $\eta$ contains only the mapping from *null* to 0. The field version map $\alpha$ associates each field $f \in F$ with a non-negative integer representing the current version of its array symbolic variable. If $\alpha[f] = i$, the variable is denoted $f^i$. Initially, all fields have the version 0.

```
heap: (environment $\eta : V_R \to \{0\} \cup \Sigma_v$, field versions $\alpha : F \to \mathbb{N}^0$}
ProcessOperationConjHeap($vers$, ($\eta$, $\alpha$), $op$)
 1: var $\varphi \leftarrow true, vers' \leftarrow vers, \eta' \leftarrow \eta, \alpha' \leftarrow \alpha$
 2: switch $op$ do
 3:     case $v_t \leftarrow_r v_v$
 4:         if $\eta[v_t] \neq undef$ then
 5:             if $\eta[v_v] = undef$ then
 6:                 $\eta' \leftarrow \eta[v_v \to \eta[v_t], v_t \to undef]$
 7:             else
 8:                 $\eta' \leftarrow \eta[v_t \to undef]$
 9:                 $\varphi \leftarrow \eta[v_t] = \eta[v_v]$
10:     case $v_t \leftarrow (v_1 = v_2)$
11:         $\eta' \leftarrow$ Init($\eta, v_1, v_2$)
12:         $vers' \leftarrow vers[v_t \to vers[v_t] + 1]$
13:         $\varphi \leftarrow v_t^{vers[v_t]} = (\eta'[v_1] = \eta'[v_2])$
14:     case $v_t \leftarrow_r$ new $T$
15:         if $\eta[v_t] \neq undef$ then
16:             $\eta' \leftarrow \eta[v_t \to undef]$
17:             $\varphi \leftarrow \eta[v_t] = $ Fresh$_{\mathbb{N}^+}$()
18:     case $v_r.f \leftarrow v_v$
19:         $\eta' \leftarrow$ Init($\eta, v_r, v_v$)
20:         $\alpha' \leftarrow \alpha[f \to \alpha[f] + 1]$
21:         $\varphi \leftarrow (f^{\alpha[f]} = write(f^{\alpha'[f]}, \eta'[v_r], $ Symb$(\eta', v_v))) \wedge (\eta'[v_r] \neq 0)$
22:     case $v_t \leftarrow_s v_r.f$
23:         $\eta' \leftarrow$ Init($\eta, v_r$)
24:         $vers' \leftarrow vers[v_t \to vers[v_t] + 1]$
25:         $\varphi \leftarrow (v_t^{vers[v_t]} = read(f^{\alpha[f]}, \eta'[v_r]) \wedge \eta'[v_r] \neq 0)$
26:     case $v_t \leftarrow_r v_r.f$
27:         $\eta' \leftarrow$ Init($\eta, v_r$)
28:         $\varphi \leftarrow (\eta'[v_r] \neq 0)$
29:         if $\eta[v_t] \neq undef$ then
30:             if $v_t = v_r$ then
31:                 $\eta' \leftarrow \eta'[v_r \to$ Fresh$_{\Sigma_v}$()]
32:             else
33:                 $\eta' \leftarrow \eta'[v_t \to undef]$
34:             $\varphi \leftarrow \varphi \wedge (\eta[v_t] = read(f^{\alpha[f]}, \eta'[v_r]) \wedge read(f^{in}, \eta'[v_r]) \leq 0)$
35: return $(\varphi, vers', (\eta', \alpha'))$
GetConditionConjHeap($states$, $n$)
 1: var $inputRefs \leftarrow$ gather symbolic variables in $\eta$ of the heap in $states[n]$
 2: return GetConditionConj($states$, $n$) $\wedge \bigwedge_{f \in F_R} f^{\alpha[f]} = f^{in} \wedge \bigwedge_{v \in inputRefs} v \leq 0$
```

Fig. 5: Heap operation modelling in the conjunct combination approach from Fig. 3

Let us proceed to the semantics of ProcessOperationConjHeap. The reference assignment $v_t \leftarrow_r v_v$ distinguishes three cases. If we have not yet encountered $v_t$, it is not contained in $\eta$ and we are not interested in any value assigned to it. Otherwise, if we do not know $v_v$, we associate it with variable[2] $\eta[v_t]$. If both $v_t$ and $v_v$ are known, we must assert the equality of their symbolic variables. Eventually, in any case, we must remove $v_t$ from $\eta$, because by being assigned to it was effectively removed from the set of input heap references. When comparing two references $v_1$ and $v_2$, we use helper function Init, which associates them in $\eta$ with fresh symbolic integer variables, if they are not already present there. Then, the scalar assignment of boolean term $\eta[v_t] = \eta[v_v]$ to $v_t$ is performed, updating the version of $v_t$ in *vers* accordingly. A new object creation is again modelled only if we have encountered the target reference variable $v_t$ before. Its symbolic integer variable $\eta[v_t]$ is asserted to be equal with a fresh positive number and $v_t$ is removed from $\eta$. A field write $v_r.f \leftarrow v_v$ needs to manipulate $\alpha$ by incrementing the version of $f$ and using its two distinct versions to express the write. Note that due to the backward approach of our analysis, the version being written to is the current one.

Again, a field read operation is the most complicated one to model. In both scalar and reference cases, we use Init to ensure that there is a symbolic integer variable corresponding to $v_r$, constrain it not to be equal to *null* by $\eta'[v_r] \neq 0$ and use *read* to model the read of the field from the heap. In the scalar case $\leftarrow_s$, we must also handle the assignment into $v_t$ by increasing its version in *vers*. In the reference case $\leftarrow_r$, when we are interested in the reference stored in $v_t$, we also use the helper $f^{in}$ array variable enabling us to constrain the input heap later in GetConditionConjHeap. Note that we also explicitly handle the situation when $v_t = v_r$ in order not to accidentally remove $v_r$ from the environment. MergeHeaps uses the same version map merging as MergeConj utilizing JoinVers. To merge environments with two or more distinct values corresponding to one reference variable, it is suitable to randomly pick one of them and constrain all the others to point to it. In the algorithm, we must avoid introducing unintentional aliases in the resulting environment.

## 4.3 Example

To demonstrate the operations on a real-life example, let us examine the assertion in Fig. 6, which corresponds to inspecting the reachability of the node $n_{13}$ in the CFG. Notice that the heap operations from the code were decomposed into the atomic ones, producing helper variables such as `tv`, `tn` or `rnv`.

The solution using the disjunct propagation approach is depicted in Table 1. Each row captures the current state of the condition computed for it, starting from $n_{13}$ and going backwards to $n_0$. The table is divided into four blocks according to the shape of the CFG. To simplify the notation, we do not use the $s$ superscripts to denote symbolic variables, as all the variables in the condition are symbolic. Instead, they are differentiated by their font, as the program variables from the CFG use a monospaced one.

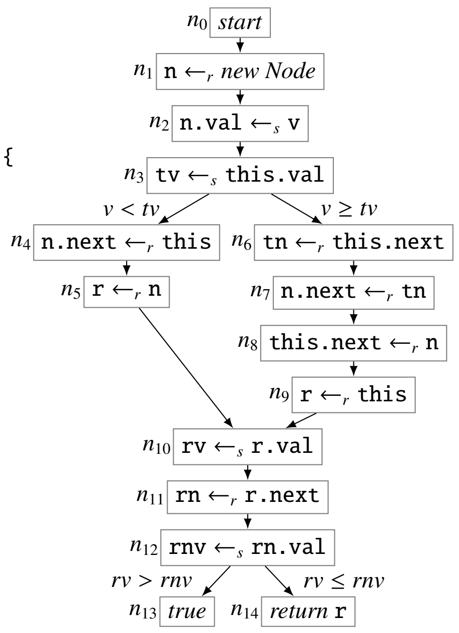
Since the reachability from $n_{13}$ to $n_{13}$ is trivial, the condition starts with *true*. Next, to reach it from $n_{12}$, the condition $rv > rnv$ is added and the field read is performed, replacing $rnv$ with $read(val, rn)$ and ensuring that $rn$ is not *null*. The next read into

[2] We expect that *null* cannot be on the left side of the assignment.

```
class Node {
  public int val;
  public Node next;

  public Node AddSmaller(int v) {
    Node n = new Node();
    n.val = v;
    Node r;
    if (v < this.val) {
      n.next = this;
      r = n;
    } else {
      n.next = this.next;
      this.next = n;
      r = this;
    }
    Assert(r.val <= r.next.val);
    return r;
  }
}
```

(a) C# code



$n_0$ *start*
$n_1$ n $\leftarrow_r$ *new Node*
$n_2$ n.val $\leftarrow_s$ v
$n_3$ tv $\leftarrow_s$ this.val
$v < tv$
$v \geq tv$
$n_4$ n.next $\leftarrow_r$ this
$n_5$ r $\leftarrow_r$ n
$n_6$ tn $\leftarrow_r$ this.next
$n_7$ n.next $\leftarrow_r$ tn
$n_8$ this.next $\leftarrow_r$ n
$n_9$ r $\leftarrow_r$ this
$n_{10}$ rv $\leftarrow_s$ r.val
$n_{11}$ rn $\leftarrow_r$ r.next
$n_{12}$ rnv $\leftarrow_s$ rn.val
$rv > rnv$
$rv \leq rnv$
$n_{13}$ *true*
$n_{14}$ *return* r

(b) CFG

Fig. 6: Sample C# code with heap objects and the corresponding CFG

$rn$ is a reference one; therefore, $read(next^{in}, r) \leq 0$ is added. The helper variable $rv$ is replaced by its semantics in $n_{10}$. Notice that if we called GetConditionDisjHeap at this point, the condition $next = next^{in} \land r \leq 0$ would be temporarily added, ensuring that the input heap consisting of $r$ is separated from the objects potentially created during the analysis.

In $n_9$, the last node of the `else` branch, the assignment `r` $\leftarrow_r$ `this` causes the replacement of $r$ by $this$. After the field write in $n_8$, $read(write(next, this, n), this)$ is simplified to $n$. Notice that now $next$ is not a part of the formula and $this$ and $n$ are already constrained not to be $null$, so the operations in $n_7$ and $n_6$ do not have any effects. The semantics of the positive `if` branch is similar, as it replaces $r$ by $n$ and then reduces both occurrences of $read(next, n)$ to $this$.

Node $n_3$ merges the disjuncts from nodes $n_6$ and $n_4$, adds their respective conditions and performs the replacement of $tv$ by $read(val, this)$. By the assignment `n.val` $\leftarrow_s$ `v` in $n_2$, we reduce $read(val, n)$ to $v$. The creation of new object in $n_1$ replaces $n$ by 1 in both disjuncts, simplifying away the conditions $n \neq 0$. Finally, the condition for $n_0$ enhanced with input heap handling is passed to the SMT solver, proving the assertion by returning $UNSAT$.

Table 1: The verification of the assertion in Fig. 6 using disjunct propagation

| | |
|---|---|
| $n_{13}$ | $true$ |
| $n_{12}$ | $rv > read(val, rn) \land rn \neq 0$ |
| $n_{11}$ | $rv > read(val, read(next, r)) \land read(next, r) \neq 0 \land read(next^{in}, r) \leq 0 \land r \neq 0$ |
| $n_{10}$ | $read(val, r) > read(val, read(next, r)) \land read(next, r) \neq 0 \land read(next^{in}, r) \leq 0 \land r \neq 0$ |
| $n_9$ | $read(val, this) > read(val, read(next, this)) \land read(next, this) \neq 0 \land read(next^{in}, this) \leq 0 \land this \neq 0$ |
| $n_8, n_7, n_6$ | $read(val, this) > read(val, n) \land n \neq 0 \land read(next^{in}, this) \leq 0 \land this \neq 0$ |
| $n_5$ | $read(val, n) > read(val, read(next, n)) \land read(next, n) \neq 0 \land read(next^{in}, n) \leq 0 \land n \neq 0$ |
| $n_4$ | $read(val, n) > read(val, this) \land this \neq 0 \land read(next^{in}, n) \leq 0 \land n \neq 0$ |
| $n_3$ | $(v \geq read(val, this) \land read(val, this) > read(val, n) \land n \neq 0 \land read(next^{in}, this) \leq 0 \land this \neq 0) \lor (v < read(val, this) \land read(val, n) > read(val, this) \land this \neq 0 \land read(next^{in}, n) \leq 0 \land n \neq 0)$ |
| $n_2$ | $(v \geq read(write(val, n, v), this) \land read(write(val, n, v), this) > v \land n \neq 0 \land read(next^{in}, this) \leq 0 \land this \neq 0) \lor (v < read(write(val, n, v), this) \land v > read(write(val, n, v), this) \land this \neq 0 \land read(next^{in}, n) \leq 0 \land n \neq 0)$ |
| $n_1, n_0$ | $(v \geq read(write(val, 1, v), this) \land read(write(val, 1, v), this) > v \land read(next^{in}, this) \leq 0 \land this \neq 0) \lor (v < read(write(val, 1, v), this) \land v > read(write(val, 1, v), this) \land this \neq 0 \land read(next^{in}, 1) \leq 0)$ |

Table 2 shows how the conjunct combination variant works. As its name suggests, the assertions created for all the relevant nodes are combined using conjunction. In order to determine the reachability from $n_1$, we must combine all the conditions in the table. Notice that for each node $n_i$, there exist an environment $\eta_i$, a field version map $\alpha_i$ and a helper $c_i$ to express that the control flow reached it.

The semantics of the operations is the same as in the former case, but the construction is different. In general, $\eta_i$ and $\alpha_i$ keep track of the symbolic variables which represent the current versions of references and fields, respectively. As we can see in $n_8$, $n_7$, $n_4$ and $n_2$, every field read causes the corresponding $\alpha_i$ to create another version of its corresponding array symbolic variable. Whenever we read an unknown reference, we create a symbolic integer variable for it, such as in the case of $\eta_{12}$. As soon as that reference is being assigned to, we forget it, e.g. in $\eta_{11}$.

Let us have a look on the assignments in $n_9$ and $n_5$. The former one causes all the usages of `this` in the `else` branch to be represented by $r$, whereas the latter one does the same in the positive `if` branch for `n`. Their versions are properly united after being merged in $n_3$.

Table 2: The verification of the assertion in Fig. 6 using conjunct combination

| | | |
|---|---|---|
| $n_{13}$ | $c_{13} \implies true$ | $\eta_{13} = \{(null, 0)\}$ <br> $\alpha_{13} = \{(\texttt{next}, 0), (\texttt{val}, 0)\}$ |
| $n_{12}$ | $c_{12} \implies c_{13} \wedge rv > rnv \wedge rnv = read(val^0, rn) \wedge rn \neq 0$ | $\eta_{12} = \eta_{13}[\texttt{rn} \to rn]$ |
| $n_{11}$ | $c_{11} \implies$ <br> $c_{12} \wedge rn = read(next^0, r) \wedge r \neq 0 \wedge read(next^{in}, r) \leq 0$ | $\eta_{11} = \eta_{12}[\texttt{rn} \to undef]$ |
| $n_{10}$ | $c_{10} \implies c_{11} \wedge rv = read(val^0, r) \wedge r \neq 0$ | $\eta_{10} = \eta_{11}[\texttt{r} \to r]$ |
| $n_9$ | $c_9 \implies c_{10}$ | $\eta_9 = \eta_{10}[\texttt{r} \to undef, \texttt{this} \to r]$ |
| $n_8$ | $c_8 \implies c_9 \wedge next^0 = write(next^1, r, n) \wedge r \neq 0$ | $\eta_8 = \eta_9[\texttt{n} \to n]$ <br> $\alpha_8 = \alpha_{13}[\texttt{next} \to 1]$ |
| $n_7$ | $c_7 \implies c_8 \wedge next^1 = write(next^2, n, tn) \wedge n \neq 0$ | $\eta_7 = \eta_8[\texttt{tn} \to tn]$ <br> $\alpha_7 = \alpha_8[\texttt{next} \to 2]$ |
| $n_6$ | $c_6 \implies$ <br> $c_7 \wedge tn = read(next^2, r) \wedge r \neq 0 \wedge read(next^{in}, r) \leq 0$ | $\eta_6 = \eta_7[\texttt{tn} \to undef]$ |
| $n_5$ | $c_5 \implies c_{10}$ | $\eta_5 = \eta_{10}[\texttt{r} \to undef, \texttt{n} \to r]$ |
| $n_4$ | $c_4 \implies c_5 \wedge next^0 = write(next^1, r, this) \wedge r \neq 0$ | $\eta_4 = \eta_5[\texttt{this} \to this]$ <br> $\alpha_4 = \alpha_{13}[\texttt{next} \to 1]$ |
| $n_3$ | $c_3 \implies$ <br> $((c_4 \wedge v < tv \wedge next^1 = next^2 \wedge r = n)$ <br> $\vee (c_6 \wedge v \geq tv \wedge r = this))$ <br> $\wedge\ tv = read(val^0, this) \wedge this \neq 0$ | $\eta_3 = \{(null, 0), (\texttt{this}, this), (\texttt{n}, n)\}$ <br> $\alpha_3 = \{(\texttt{next}, 2), (\texttt{val}, 0)\}$ |
| $n_2$ | $c_2 \implies c_3 \wedge val^0 = write(val^1, n, v) \wedge n \neq 0$ | $\alpha_2 = \alpha_3[\texttt{val} \to 1]$ |
| $n_1$ | $c_1 \implies c_2 \wedge n = 1$ | $\eta_1 = \eta_3[\texttt{n} \to undef]$ |
| $n_0$ | $c_0 \implies c_1$ | |

We can see that in our simple example, the formula resulting from disjunct propagation is much shorter than the one from conjunct combination. However, in case of larger programs with more branches, the number of disjuncts can grow in an exponential manner if we do not simplify them efficiently.

## 5 Evaluation

We implemented the techniques into a development version of AskTheCode, an open-source tool for backward symbolic execution of C# code, which uses Z3 as the SMT solver. In order to compare the efficiency of the aforementioned approaches, we prepared a simple program which can be parametrized so that its complexity and validity of the assertions can vary. *Degree counting*$(a, b)$ is an algorithm receiving a linked list as the input. Each of its nodes contains an additional reference to another node and the algorithm calculates for each node its in-degree: the number of nodes referencing it. The assertion fails if it encounters a node whose in-degree is greater than its zero-based index in the list and also greater than a given number $a$. The second parameter $b$ specifies the number of loop unwindings, i.e., the number of nodes inspected from the start of the list. As a result, the assertion is refutable if and only if $a + 2 \leq b$. Increasing $b$ produces a larger CFG with also potentially more complicated conditions, but the counterexample might be easier to find due to a larger number of paths corresponding to it.

The execution time of analysis of each input variant is shown in Table 3[3]. Notice that there are multiple approaches both to disjunct propagation *Disj* and to conjunct combination *Conj*. Because we considered creating a custom implementation of term simplification and efficient representation too complex, we decided to use the well-optimized terms available in the API of Z3. $Disj_{Set}$ uses a set of Z3 terms to represent the disjuncts in each state. Their uniqueness is ensured by the hash consing implemented in Z3. The simplification is performed for each term separately. On the other hand, $Disj_{Z3}$ represents each state using a Z3 term; merging is performed by creating a disjunction of all the terms in the dependent nodes. $Disj_{Comb}$ is a combination of the two approaches. A state is represented as a Z3 term set, but the merging is performed by creating a disjunction term and putting it as a single item of the set. In $Conj_{Never}$, DoSolve always returns *false*, so no intermediate calls of the SMT solver are performed. An opposite extreme is $Conj_{Always}$, where DoSolve always returns *true*. In $Conj_{Loops}$, *true* is returned only for entry nodes of loops. The underlying solver is used incrementally, which enables it to reuse the information gained during the previous checks.

The results show that for our problem, conjunct combination was more efficient than disjunct propagation. The best times were obtained for $Conj_{Never}$, where the SMT solver was called only once at the very end of the analysis. However, in case of more complicated examples where an early check may prevent the analysis from inspecting large regions of code, the incremental usage of the SMT solver might be useful. The results of $Conj_{Always}$ show that it is unnecessary and inefficient to call it on every operation, as it causes an overhead of more than 250% on average. Instead, when we carefully select the nodes where to perform these additional checks like we did in $Conj_{Loops}$, the overhead is less than 25% on average.

[3] We conducted the experiments on a desktop with an Intel Core i7 CPU and 6GB RAM.

Table 3: Performance evaluation, the times are in milliseconds

| Test Case | $Disj_{Set}$ | $Disj_{Z3}$ | $Disj_{Comb}$ | $Conj_{Never}$ | $Conj_{Always}$ | $Conj_{Loops}$ |
|---|---|---|---|---|---|---|
| *Degree counting (0, 3)* | 298 | 775 | 668 | 18 | 55 | 20 |
| *Degree counting (1, 3)* | 302 | 773 | 688 | 21 | 60 | 26 |
| *Degree counting (2, 3)* | 284 | 752 | 675 | 15 | 59 | 20 |
| *Degree counting (1, 4)* | 2062 | 1225 | 791 | 31 | 119 | 46 |
| *Degree counting (2, 4)* | 2075 | 1152 | 822 | 43 | 121 | 53 |
| *Degree counting (3, 4)* | 1949 | 874 | 754 | 25 | 115 | 32 |
| *Degree counting (2, 5)* | 13334 | 1856 | 1287 | 91 | 242 | 102 |
| *Degree counting (3, 5)* | 13381 | 1947 | 1360 | 85 | 232 | 99 |
| *Degree counting (4, 5)* | 13226 | 1764 | 1125 | 40 | 246 | 50 |
| *Degree counting (3, 6)* | 81282 | 4728 | 4052 | 200 | 469 | 219 |
| *Degree counting (4, 6)* | 80853 | 4566 | 4214 | 161 | 427 | 178 |
| *Degree counting (5, 6)* | 80915 | 3116 | 2364 | 62 | 390 | 78 |

We believe that implementing a custom well-optimized simplifier will lead a substantial performance improvement of disjunct propagation, as achieved in the case of Snugglebug [7]. However, writing such a simplifier might be a challenging feat, whereas the utilization of incremental solving can efficiently move the problem to a well-optimized SMT solver.

## 6 Related Work

The disjunct propagation approach originates from Snugglebug [1], a tool using weakest preconditions to assess the validity of assertions in Java code. Snugglebug uses the algorithm for intraprocedural analysis, utilizing a custom-made simplifier over the propagated disjuncts. For interprocedural analysis, various other methods are used, such as directed call graph construction or tabulation. The SMT solver is utilized only at the entry point, as many infeasible paths are rejected using the simplifier. The conjunct combination approach is used in UFO [1] as the under-approximation subroutine. UFO, however, does not handle heap objects.

Microsoft Pex [18] is a tool generating unit tests for .NET programs using dynamic symbolic execution. It executes the program with concrete inputs and observes its behaviour, using the Z3 SMT solver to generate new inputs steering the execution to uncovered parts of the code. It also uses the array theory to model heap operations, but the way it works with the input heap is different from our pure symbolic approach.

KLEE [6] is a symbolic virtual machine utilizing the LLVM [13] infrastructure, used mainly for C and C++ projects. It uses array theory not only to reason about heap operations, but also about pointers, low-level memory accesses, etc. This differs from our approach, because we target only higher level languages with reference semantics, without the usage of pointers. Furthermore, KLEE does not support running symbolic execution backwards.

Symbolic execution tools JBSE [5] and Java StarFinder (JSF) [14] both employ lazy initialization to reason about heap objects, which lazily enumerates all the possible shapes of the heap. They differ by the languages used for specification of the heap objects' invariants. Whereas JBSE uses custom-made HEX, JSF utilizes separation logic. Although we use a different approach for the core of the heap operations, taking heap invariants into account might help us to prune infeasible paths and save resources.

## 7 Conclusion

In this paper, we focused on the task of demand-driven program analysis by studying methods of efficiently implement backward symbolic execution. We identified two main approaches used for the core algorithm, namely disjunct propagation and conjunction combination. The former one has the benefit of easier implementation and creating potentially simpler conditions passed to an SMT solver, while the latter one is more predictable in terms of the resulting condition size and can better utilize incremental SMT solvers. To handle heap operations in both approaches, we use the theory of arrays, paying attention to properly handle the notion of an input heap throughout the analysis. The evaluation on our code examples shows that the effort put into the implementation of the conjunct combination approach is reasonable, because its results exceeded the straightforward implementation of disjunct propagation in an order of magnitude.

Due to the narrow focus of this work, the application of our technique is currently limited mainly by the inability to soundly handle loops, interprocedural calls and recursion. Our future work will mainly focus on removing these limitations by exploring the possibilities of computing and reusing procedure summaries, possibly learning from infeasible paths using interpolants. We will build on our knowledge of how disjunct propagation and conjunct combination perform in different circumstances, combining them to reach a valuable synergy.

## Acknowledgements

This work was supported by the project PROGRESS Q48, the Czech Science Foundation project 17-12465S the grant SVV-2017-260451.

## References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS (2012). https://doi.org/10.1007/978-3-642-28756-5_12
2. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 50 (2018)
3. Bjørner, N.: Engineering theories with z3. In: Yang, H. (ed.) Programming Languages and Systems. pp. 4–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag, Berlin, Heidelberg (2007)
5. Braione, P., Denaro, G., Pezzè, M.: Jbse: A symbolic executor for java programs with complex heap inputs. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1018–1022. ACM (2016)

6. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
7. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: A powerful approach to weakest preconditions. SIGPLAN Not. **44**(6), 363–374 (Jun 2009), http://doi.acm.org/10.1145/1543135.1542517
8. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
9. Dinges, P., Agha, G.: Targeted test input generation using symbolic-concrete backward execution. In: 29th IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM, Västerås, Sweden (September 15-19 2014)
10. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. pp. 12–17. SMT '08/BPR '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1512464.1512468
11. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Notices **39**, 92–106 (12 2004). https://doi.org/10.1145/1028664.1028717
12. Husák, R., Kofroň, J., Zavoral, F.: AskTheCode: Interactive call graph exploration for error fixing and prevention. Electronic Communications of the EASST **77** (2019). https://doi.org/10.14279/tuj.eceasst.77.1109, InterAVT 2019
13. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673
14. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Testing heap-based programs with java starfinder. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 268–269. ACM (2018)
15. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (May 2002). https://doi.org/10.1145/514188.514190, http://doi.acm.org/10.1145/514188.514190
16. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Alternate and learn: Finding witnesses without looking all over. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 599–615. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_42, http://dx.doi.org/10.1007/978-3-642-31424-7_42
17. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Aliasing in object-oriented programming. chap. Alias Analysis for Object-oriented Programs, pp. 196–232. Springer-Verlag, Berlin, Heidelberg (2013), http://dl.acm.org/citation.cfm?id=2554511.2554523
18. Tillmann, N., De Halleux, J.: Pex: White box test generation for .net. In: Proceedings of the 2Nd International Conference on Tests and Proofs. pp. 134–153. TAP'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792786.1792798