Symbolic Execution of Programs with Heap Inputs

Pietro Braione[§], Giovanni Denaro[§], Mauro Pezzè[§♯]

[§] University of Milano-Bicocca
Viale Sarca, 336
Milano, Italy 20126
{braione, denaro}@disco.unimib.it

[♯] Università della Svizzera italiana (USI)
Via Giuseppe Buffi, 13
Lugano, Switzerland 6900
pezzem@usi.ch

ABSTRACT

Symbolic analysis is a core component of many automatic test generation and program verification approaches. To verify complex software systems, test and analysis techniques shall deal with the many aspects of the target systems at different granularity levels. In particular, testing software programs that make extensive use of heap data structures at unit and integration levels requires generating suitable input data structures in the heap. This is a main challenge for symbolic testing and analysis techniques that work well when dealing with numeric inputs, but do not satisfactorily cope with heap data structures yet.

In this paper we propose a language HEX to specify invariants of partially initialized data structures, and a decision procedure that supports the incremental evaluation of structural properties in HEX. Used in combination with the symbolic execution of heap manipulating programs, HEX prevents the exploration of invalid states, thus improving the efficiency of program testing and analysis, and avoiding false alarms that negatively impact on verification activities. The experimental data confirm that HEX is an effective and efficient solution to the problem of testing and analyzing heap manipulating programs, and outperforms the alternative approaches that have been proposed so far.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic Execution*

Keywords

Symbolic execution, lazy initialization, data structure invariants

1. INTRODUCTION

Many approaches for automatic test case generation and program verification exploit symbolic execution, which consists of executing a program with *symbolic* values, to compute the execution conditions of program paths (*path conditions*) [26]. Path conditions are used to both generate test cases that execute the specific paths and verify programs against code assertions on those paths.

Symbolic execution handles well programs with numeric input values like integers and reals, but does not cope well with heap data structures, thus limiting the efficacy of program testing and verification approaches. Many popular embodiments of symbolic execution can analyze complete programs with numeric inputs, but experience limitations when testing software programs that make extensive use of heap data structures at the unit, integration and subsystem levels [19, 6, 7, 29, 14, 5, 4]. For example, intra- and inter-class object oriented testing challenges symbolic execution with objects that depend on parameters and state variables of structured data types in the heap.

The few attempts to extend symbolic execution to analyze object oriented programs deal with dynamic heap structures by enriching the path conditions with assumptions on the objects in the initial heap. Such assumptions identify the heap configurations that determine the execution of the different paths [18, 31, 8, 30, 3, 13]. For example, when analyzing a program that accesses the first node of an input list, symbolic execution distinguishes the case of an empty list, which causes an exception, from the case of a non-empty list, which results in accessing the first item of the list. The mainstream approach, referred to as *lazy initialization*, consists in the brute force enumeration of all heap objects that can bind to the structured inputs accessed by the program, for example either the empty or the non-empty list. When symbolic execution accesses for the first time a reference to an input object, it systematically enumerates all the possible input objects that can bind to that reference, and identifies an alternative path condition for each binding.

The brute force enumeration of the input objects may identify many invalid heap configurations that violate properties of the data structures in the heap. Possible properties include data type invariants, class contracts, implicit program assumptions and method preconditions. Symbolic execution approaches that overlook the structural properties can engage in the extensive exploration of invalid program executions that depend on inputs that contradict the properties, thus diverging and raising false alarms.

For example, let us consider the class `NodeTraversal` of the Google Closure Compiler[1] that takes as input a parse tree encoded as an object of type `Node`. Objects of type `Node` refer to a (linked) list of children `Node`s, which can

[1] https://code.google.com/p/closure-compiler

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786842

recursively refer to other generations of `Node`s. The software system maintains the implicit property that the `Node`s of any parse tree form indeed a tree-shaped data structure. Symbolically executing class `NodeTraversal` with the lazy initialization approach enumerates many non-tree shaped parse trees, thus leading to exploring several invalid program traces rooted in the symbolic states that include these invalid parse trees. The enormous amount of invalid traces may exceed by orders of magnitude the number of valid ones, and thus analyzing the invalid traces wastes most of the verification budget. For example symbolically executing (to generate test cases) class `NodeTraversal` with lazy initialization and a time budget of 6 hours led to the exploration of millions of invalid traces, that is, invocations of the class with invalid inputs, without producing any valid test case (we report more details in Section 4). With reference to the buggy version of class `NodeTraversal` reported by Just et al. in [17], the invalid traces led to many false alarms, that is, invalid tests cases that incorrectly reveal the bug. The readers should notice that we refer to unit testing, where the input values are not filtered by the compiler API. System testing exercises the compiler APIs that builds only valid parse trees, but it is infeasible to build a symbolic unit test generator that affords the cost of symbolically analyzing all the compiler software to generate test cases for a single class.

The problem of representing and reasoning about heap data structures with rich structural constraints is widely studied in the context of logic-based automated verification of heap manipulating programs [25, 24, 32, 27, 21, 1, 22, 11]. The different proposals compete on their ability to handle specific kinds of data structures, for instance lists or trees. Conversely, the symbolic execution community has payed little attention to this problem so far. There exist only few preliminary proposals to reason about the properties of heap data structures in the context of symbolic execution [31, 8, 30, 3, 13]. These approaches either re-evaluate the consistency of the whole heap structure after every new assumption, or enumerate in advance the sets of valid structures. The experimental data reported in Section 4 indicate that none of the existing approaches provides a satisfactory solution to the problem.

In this paper, we present a novel approach for symbolically executing programs that take as input both numeric values and heap data structures, and require rich representation constraints over these inputs. We thus provide a new tool to effectively generate test cases and verify software programs by means of symbolic execution. The core of the approach is a language and a corresponding decision procedure, jointly referred to as HEap eXploration Logic (HEX). The language specifies the properties of the data structures as structural constraints in a way that enables to check the constraints against the incremental assumptions on the input objects that emerge during symbolic execution. Symbolic execution augmented with HEX evaluates the constraints incrementally, as the heap is progressively refined by symbolic execution, with advantages on both precision and scalability. This radically differs from previous approaches.

We evaluated HEX in terms of both analysis speedup and effectiveness in supporting the automatic generation of test cases for software programs, and compared the results with competing approaches based on structural properties encoded as executable methods of the data structures [31, 8] and based on bounded/unbounded heap reachability provers [25, 16]. We estimated the relative analysis speedup referring to a set of recursive data structures (lists, trees, etc.) to investigate the different aspects of the application domains in breath. We evaluated the effectiveness of test case generation referring to a set of third-party Java components with structured input data to gain empirical evidence of the applicability of the approach in practical software engineering contexts.

In previous work, we introduced LICS, a framework that allowed us to experiment the preliminary ideas on a case study [3]. In this paper we explicitly introduce and define HEX, and present an extensive evaluation, which indicates that HEX improves on the state-of-the-art approaches. The new results provide evidence that HEX outperforms previous approaches along different dimensions of precision and performance.

The paper is organized as follows. Section 2 overviews lazy initialization to make the paper self-contained. Section 3 presents HEX and the decision procedure based on HEX that we propose in this paper. Section 4 presents the techniques considered in the experiment. Section 5 discusses the results and the empirical comparison of the different approaches. Section 6 surveys the related work, and Section 7 summarizes the main contribution of the paper and indicates future research directions.

2. LAZY INITIALIZATION

Symbolic execution supports test case generation and program analysis by identifying the execution conditions of the paths that we want to cover in testing and verify in analysis. The execution conditions, also known as *path conditions*, are symbolic expressions where symbols represent the input values. When dealing with programs that refer to heap data structures, like the `getList` program in Figure 1 that we use as working example in the paper, symbols may represent structured objects, for instance a list that contains a given number of generic objects.

Current symbolic executors handle accesses to heap data structures by means of a technique commonly referred to as *lazy initialization* [31, 8]. Lazy initialization enumerates all the different structures of the heap objects that can bind to the references accessed while executing the program. For example, Figure 2 shows the symbolic data structures created while symbolically executing the `getList` program in Figure 1. These symbolic data structures correspond to the assumptions that lazy initialization incrementally adds to the path conditions while symbolically executing the program. The figure presents the assumptions graphically, by indicating the objects and the relations among them that are incrementally assumed during the execution.

```
1  List getList(List list, int foo){
2     if(foo < 0) {
3        List first = list;
4        List second = first.next;
5        return second.next;
6     }
7     else {
8        if (foo > 10) return null;
9        if(foo > 5) return new List();
10    }
11 }
```

Figure 1: A program that manipulates heap objects.

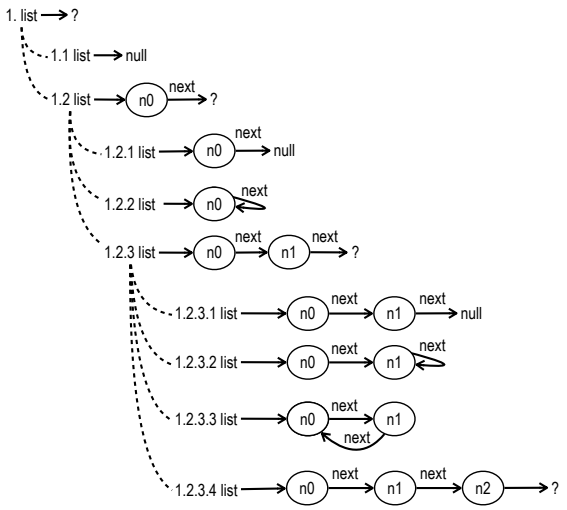


Figure 2: Symbolic data structures analysed during symbolic execution.

When starting symbolically executing the program, the lazy initialization algorithm does not make any assumption on the input data structure `list`. This corresponds to a list that can link to any value, and is concretized in the data structure labeled *1* on the top of Figure 2.

When executed with negative values of the input parameter `foo`, the program `getList` navigates through the list by accessing the field `next` of the nodes of the list. To execute the statement that accesses the first node of the list (line 3), the symbolic executor must distinguish whether the parameter `list` has value `null` or points to a non empty list. Lazy initialization analyses these two cases separately, by branching the execution in two distinct traces that correspond to the assumptions *1.1* and *1.2* in the figure, respectively. These two assumptions represent the different behavior of the program that leads to a runtime exception when executed with data structures that correspond to the assumption *1.1*, and proceeds with a normal execution with data structures that correspond to the assumption *1.2*.

The execution of line 4 with the assumption *1.2* navigates field `next` to access the second node of the list, and the symbolic execution branches into three assumptions labeled *1.2.1* (`next` is `null`), *1.2.2* (`next` points to a compatible heap object – we have only one so far) or *1.2.3* (`next` points to a heap object that has not been met in the analysis yet). Similarly when executing line 5 from state *1.2.3*, the symbolic execution branches into the assumptions *1.2.3.1*, *1.2.3.2*, *1.2.3.3* and *1.2.3.4*. The readers should notice that in this case there are two heap objects compatible with `next` that are treated separately (assumptions *1.2.3.2* and *1.2.3.3*). The analysis of the statements at lines 8 and 9 does not depend on new assumptions on the data structure, and thus proceeds with the initial assumption *1* without distinguishing further subcases.

As illustrated in the example, lazy initialization exhaustively enumerates all possible assumptions on objects and references when needed to symbolically execute a statement. Enumerating all possible assumptions without considering the program invariants that characterize the semantics of the data structures may produce many data structures infeasible in the specific analysis context. For example if `getList` required a circular list, the assumptions *1.2.1, 1.2.3.1* and *1.2.3.2* would correspond to invalid inputs that can generate false alarms and reduce the efficiency of symbolic execution.

The scalability and precision problems of plain lazy initialization have been addressed by either enumerating the valid symbolic data structures before starting symbolic execution or by interleaving symbolic execution with checking engines that evaluate structural properties. The current approaches that we discuss in more details in Section 4 address either scalability or precision, but not both aspects. In the next section, we introduce the HEap eXploration Logic (HEX), a property specification language that provides a scalable and precise solution to the lazy initialization problem by supporting the incremental evaluation of structural properties on partial heaps.

3. HEap eXploration Logic (HEX)

We address the problems that derive from the explicit enumeration of all possible assumptions on the heap data structures by defining a new language HEX that specifies structural constraints over heap data structures in a way particularly suited to handle input objects during symbolic execution. HEX specifies the structural properties of the heap objects as constraints on the incremental refinements that derive from the lazy inizialization process. The semantics of HEX is given with a decision procedure that is particularly efficient to evaluate the satisfiability of a HEX specification over partial heap models, as the ones considered during symbolic execution.

3.1 Partial Heaps

HEX is a language to specify invariants to be verified on partially initialized data structures that we refer to as *partial heaps*. Partial heaps are heap structures where some objects and references may be missing. They represent infinitely many complete heaps, that is, the heaps that include all the objects and references in the partial heap, and may differ in other objects and references.

DEFINITION 3.1. *A partial heap $H \equiv \langle O, root, R\rangle$ is a structure where O is a finite set of typed heap objects, root models the external environment that holds the initial references to heap objects, and $R \subseteq (O \cup \{root\}) \times (O \cup \{null\})$ is a set of (named) object references, where null models the null value. All objects in O must be reachable from root through some sequence of references in R.* □

Partial heaps are well suited to model the symbolic data structures that we described in Section 2. For instance, we can model the symbolic state *1.2* of Figure 2 with the partial heap $\langle\{n_0\}, root, \texttt{list} = \{root \rightarrow n_0\}\rangle$ that assumes the existence of the object n_0 and the reference `list` from *root* to n_0.

The initial state of symbolic execution in general, and in particular the symbolic state *1* in Figure 2, can be modeled as the most abstract partial heap $H_\top = \langle\emptyset, root, \emptyset\rangle$. $H_\top$ represents a partial heap that does not assume any object and reference. The incrementally refined symbolic data structures along a program path, as for instance the sequence of symbolic states $\langle 1, 1.2, 1.2.3, 1.2.3.4, ...\rangle$, result in an ordered set of incrementally refined partial heaps.

The readers should notice that the most abstract heap $H_\top$ is the top element of a partial order $\leq$ on partial heaps,

where $H_a \leq H_b$ if the sets of objects and references in H_b are subsets of the ones in H_a, that is, H_a is more concrete (more refined) than H_b. Complete heaps are the minimal elements of the partial order. The order is partial because not all heaps are comparable.

Partial heaps can be interpreted as connected directed graphs of objects linked by references rooted in *root*. Referring to this graph interpretation, a partial heap H can be covered by a finite number of directed spanning trees rooted in *root*, which we refer to as the *backbones* of H. Informally, a backbone is the structure induced by a visit of the partial heap starting from *root*. From hence we will write B_H for the sets of the possible backbones of H.

3.2 HEX Language

A HEX specification describes structural properties of heap data structures by introducing constraints on the possible visits of the partial heaps. A HEX specification is interpreted by a verifier that either accepts of refuses a partial heap by visiting the heap starting from *root*. At each step of the visit, the verifier visits a (not yet visited) reference, $r \in R$, that may point to *null* (*null*-reference), to a previously visited object (*alias*-reference) or to a not yet visited object (*expand*-reference), and infers the acceptability of the structures visited so far. The expand-references of a visit define a backbone on the heap. HEX sentences specify on which paths the verifier should expect to find null-, alias- or expand-references.

We now define HEX in three steps. We first define the HEX *formulas*, i.e., the atomic propositional formulas of HEX predicating over sets of heap paths, and their semantics for a generic partial heap H relative to a specific visiting order (Definition 3.2). We then extend HEX with propositional connectives, and specify the language for path expressions that identify sets of heap paths (Definition 3.3). We finally generalize the semantics of HEX sentences for all possible visiting orders of H (Definition 3.4).

DEFINITION 3.2. *Let H be a partial heap, $B \in B_H$ be a backbone of H, $\hat{B}$ be the set of all the rooted paths of B, $\hat{H} \stackrel{def}{=} \{\hat{p} \cdot (o \to o') \mid \hat{p} \in \hat{B}, o \to o' \in R, \text{target}(\hat{p}) = o\}$ be the set of the backbone paths leading to all the references, π and π' be path expressions denoting two sets of paths $\hat{\pi}, \hat{\pi}'$ respectively, and type be a data type. Then HEX formulas and their respective semantics are:*

HEX formula $\phi =$	$\langle H, B\rangle \models \phi$ *iff*
π `not null`	$null \notin \text{target}(\hat{\pi} \cap \hat{B})$
π `expands to nothing`	$\text{target}(\hat{\pi} \cap \hat{B}) \setminus \{null\} = \emptyset$
π `expands to` *type*	$\text{target}(\hat{\pi} \cap \hat{B}) \setminus \{null\} \subseteq [\![type]\!]$
π `aliases nothing`	$\hat{\pi} \cap (\hat{H} \setminus \hat{B}) = \emptyset$
π `aliases some` *type*	$\text{target}(\hat{\pi} \cap (\hat{H} \setminus \hat{B})) \subseteq [\![type]\!]$
π `aliases` π'	$\text{target}(\hat{\pi} \cap (\hat{H} \setminus \hat{B})) \subseteq \text{target}(\hat{\pi}' \cap \hat{B})$
π `aliases max` π'	$\text{target}(\hat{\pi} \cap (\hat{H} \setminus \hat{B})) \subseteq \text{target}(\max(\hat{\pi}' \cap \hat{B}))$

where $\text{target}(\hat{\pi})$ *is a function that returns the objects reached by paths $\hat{\pi}$;* $\max(\hat{\pi})$ *is a function that selects the maximal paths in $\hat{\pi}$, i.e., the path that cannot be extended by any suffix to other paths in $\hat{\pi}$; $[\![-]\!]$ is the extension of a data type, defined as the set of the heap objects of that type.* □

Informally, the `not null` HEX formula accepts a partial heap where the references visited through the paths $\hat{\pi}$ do not point to *null*. Similarly, the `expands to nothing` formula forbids the paths $\hat{\pi}$ to point to newly visited objects via expand-references. The `expands to` formula accepts the partial heaps only if the expand-references visited through the paths $\hat{\pi}$ point to objects of a given type. The first two `aliases` formulas accept the partial heaps only if the references visited through the paths $\hat{\pi}$ are never alias-references (first formula) or, if they are, they point to objects of a given type (second formula), respectively. The last two `aliases` formulas accept the partial heaps only if the alias-references visited through the paths $\hat{\pi}$ point to objects visited through either paths $\hat{\pi}'$ or $\max(\hat{\pi}')$, respectively.

DEFINITION 3.3. *HEX is the propositional logic language whose path language is the language of regular expressions over sequences of reference names.* □

DEFINITION 3.4. *Let H be a partial heap and ϕ be a HEX sentence. Then ϕ is* true for H *iff* $H \models \phi$ iff $\forall B \in B_H : \langle H, B\rangle \models \phi$. □

For example, the HEX sentence

$$root.\texttt{list}(.\texttt{next})^+ \texttt{ not null} \ \wedge \tag{1}$$

$$root.\texttt{list}(.\texttt{next})^+ \texttt{ aliases } root.\texttt{list} \tag{2}$$

specifies that the list in Figure 2 is circular by requiring a verifier that scans the chain of `next` fields (identified with the regular expression $root.\texttt{list}(.\texttt{next})^+$) to reject the structure should it meet a null-reference (1) or an alias-reference that does not point to the first node of the list $root.\texttt{list}$ (2). This leaves as acceptable only expand-references to unseen list nodes, or a backlink to the first node, and leads to rejecting the symbolic states *1.2.1*, *1.2.3.1* and *1.2.3.2* in Figure 2. As a final remark note that (1) does not predicate anything on expand- and alias-references, nor (2) predicates on null- and expand-references—in particular (2) does *not* force $root.\texttt{list}(.\texttt{next})^+$ references to be alias-references.

3.3 HEX Incremental Checking

HEX supports the incremental evaluation of structural properties on partial heaps, thus improving symbolic execution. We now present an inductive decision procedure to evaluate HEX formulas on incrementally refined partial heaps, and we discuss the use of this decision procedure during symbolic execution to efficiently check the validity of the symbolic states.

The HEX decision procedure determines whether a partial heap H satisfies a HEX formula ϕ with respect to a backbone $B \in B_H$. We denote this decision problem as $\langle H, B\rangle \models^? \phi$. From the HEX definition (Definition 3.2), we can demonstrate that we can solve the decision problem $\langle H, B\rangle \models^? \phi$ by proving the formula ϕ separately against each reference of the partial heap H. We formalize this result in Definition 3.5 and Theorem 3.6. Definition 3.5 indicates how the decision procedure evaluates a HEX formula with respect to a single reference of a partial heap. Theorem 3.6 demonstrates that we can solve a decision problem by proving the formula for all references in the partial heap.

DEFINITION 3.5. *Let $\langle H, B\rangle \models^? \phi$ be a HEX decision problem; $\hat{B}$, $\hat{H}$, $\hat{\pi}$ and $\hat{\pi}'$ the sets of rooted paths as defined in Definition 3.2; r be a reference of the partial heap H; $p_r \in \hat{H}$ be the path that leads to the visit of the reference r starting from the backbone; $p'_r \in \hat{B}$ be the backbone*

path that reaches the object $\text{target}(r)$. *Then, the function* $\bar{\phi}(H, B, r)$ *evaluates the HEX formula* ϕ *with respect to the single reference* r *of the partial heap* H *as:*

HEX formula ϕ	$\bar{\phi}(H, B, r) \stackrel{def}{=}$
π `not null`	$p_r \notin \hat{\pi} \vee r \notin B \vee null \notin \text{target}(p_r)$
π `expands to nothing`	$p_r \notin \hat{\pi} \vee r \notin B \vee null \in \text{target}(p_r)$
π `expands to` *type*	$p_r \notin \hat{\pi} \vee r \notin B \vee null \in \text{target}(p_r)$ $\vee\ \text{target}(p_r) \subseteq [\![type]\!]$
π `aliases nothing`	$p_r \notin \hat{\pi} \vee r \in B$
π `aliases some` *type*	$p_r \notin \hat{\pi} \vee r \in B \vee \text{target}(p_r) \subseteq [\![type]\!]$
π `aliases` π'	$p_r \notin \hat{\pi} \vee r \in B \vee p'_r \in \hat{\pi}'$
π `aliases max` π'	$p_r \notin \hat{\pi} \vee r \in B$ $\vee\ \text{target}(p_r) \in \text{target}(\max(\hat{\pi}' \cap \hat{B}))$

THEOREM 3.6. *Let* $\langle H, B\rangle \models^? \phi$ *be a decision problem. Then:* $\langle H, B\rangle \models \phi$ *iff* $\forall r \in H : \bar{\phi}(H, B, r)$ □

Informally, the theorem is a consequence of the fact that the definition of $\bar{\phi}$ is a direct interpretation of the semantics of HEX (Definition 3.2) against the heap path that reaches a given reference in one step from the backbone. Thus evaluating $\bar{\phi}$ for all the references of a partial heap is equivalent to evaluating the formula for all paths $\hat{H}$ and $\hat{B}$ considered in Definition 3.2. The readers interested in the proof of the theorem can find the details in our online report on HEX.[2]

Theorem 3.6 allows us to solve the decision problem incrementally when dealing with incrementally refined partial heaps. For a partial heap H_b that is more refined (less abstract) than a heap H_a, we decompose the decision problem $\langle H_b, B_b\rangle \models^? \phi$ in two parts: The first part corresponds to solving the decision problem $\langle H_a, B_a\rangle \models^? \phi$ for some backbone B_a that is a prefix of the backbone B_b. The second part corresponds to proving the formula on the references of H_b that are not in H_a.

We instantiate the above decision procedure inductively to efficiently evaluate the structural properties against heap structures that are incrementally assumed during symbolic execution, thus preventing the exploration of invalid symbolic states.

As discussed in the previous section, the symbolic execution of a program both produces an ordered sequence of incrementally refined partial heaps and identifies incremental backbones of the partial heaps in the sequence. Thus, our inductive decision procedure evaluates the validity of the partial heaps produced during symbolic execution starting from the empty partial heap $H_\top$ that represents the input objects at the beginning of the execution. For the partial heap H_n that represents the assumptions on the input data structures at a given symbolic state S_n, the decision procedure deals with the decision problem $\langle H_n, B_n\rangle \models^? \phi$, where B_n is the backbone of H_n identified on the symbolic trace that led to S_n. To solve this problem, the decision procedure assumes that the problem $\langle H_{n-1}, B_{n-1}\rangle \models^? \phi$ has already been solved at the immediate predecessor symbolic state S_{n-1}, and evaluates the formula under concern against the single reference assumed in H_n after the last execution step.

Working incrementally, our decision procedure radically differs from previous approaches that either re-evaluate the consistency of the whole input structures upon any single new assumption on the heap structures, or enumerate the sets of valid assumptions in advance. In the experiments reported in Section 4, we present empirical evidence that HEX outperforms state-of-the-art other approaches that can be used to prevent the exploration of invalid data structures in symbolic execution.

The inductive decision procedure may produce some false positives due to backbone generalization and `aliases max` propositions. The decision procedure works on the backbones identified during symbolic execution, and the satisfiability verdicts cannot be always generalized to all possible backbones, as required in Definition 3.4. The incremental embodiment of the decision procedure over-approximates the decisions involved in `aliases max` propositions. In fact, the satisfiability verdicts computed with respect to the maximum paths of a backbone could be invalidated after extending the backbone with new paths in some future symbolic state. As discussed in Section 4, none of these sources of unsoundness manifested in our experiments, indicating that these issues are rare in many practical situations. Conversely, our empirical data indicate that the imprecision due to the other approaches can be high.

3.4 HEX Prototype Implementation

We implemented the HEX incremental decision procedure in our JBSE[3] symbolic executor [3]. The implementation of the HEX decision procedure augments the symbolic states with assumptions on the consistency between numeric inputs and the data structures in the partial heaps. For example, if the data type `List` of Figure 2 would maintain an integer field `size` to cache the size of the list, it might be necessary to enforce the assumption that field `size` is always at most/exactly equal to the number of list nodes in the partial/complete data structure assumed in the partial heaps. JBSE accepts an extension of HEX where an atomic predicate can be associated with a *companion method*: When the incremental decision procedure computes the satisfiability of a refined partial heap, JBSE executes the companion methods associated with the atomic predicates whose path expressions were satisfied by the last reference assumed in the heap. The users can exploit this mechanism to encode the assumptions on the numeric fields as assume statements in the companion methods. Our experiment replication package[2] includes several examples of data structure properties specified in HEX.

4. DESIGN OF THE EXPERIMENT

We experimentally evaluate the precision and efficiency of HEX with respect to both the plain lazy initialization algorithm that we refer to as baseline, and the alternative state-of-the-art approaches. The precision of the different approaches is the ability to identify executions that do not violate the constraints on the input data structures. We quantify the *precision* of an approach in terms of valid versus invalid symbolic states identified by the corresponding decision procedure, sizing the set of invalid states that result in false alarms. We measure the efficiency of the approaches comparing the time overhead on symbolic execution.

We executed the experiments on a benchmark that includes third party implementations of classic data structures and classes of open source programs. In the next subsections, we discuss the approaches that we considered in the experiment, the programs used as inputs to the symbolic

[2]http://www.lta.disco.unimib.it/tools/hex/

[3]http://pietrobraione.github.io/jbse/

executor and the metrics that we collected to measure effectiveness and efficiency.

4.1 Techniques

We identify the main classes of alternative approaches as enumerating the valid (non-partial) symbolic data structures before starting symbolic execution [8], and interleaving symbolic execution with checking engines that evaluate structural properties. The latter approaches include executing checking programs that encode the structural properties operationally [31] and querying some established satisfiability prover that evaluates property specifications in some logic [16, 25, 21, 27, 24, 32, 1, 22, 11]. The provers may address either bounded (for instances specified in Alloy [16]) or unbounded (for instance specified in PALE [25]) satisfiability problems. In detail, our experiments consider the following set (T) of techniques:

`Lazy`: The plain lazy initialization algorithm, outlined in Section 2, without techniques for dealing with constraints. This is the baseline to evaluate the different approaches.

`HEX`: The HEX approach, that is, the technique that instantiates the decision procedure presented in the previous section to incrementally check structural properties specified in HEX.

`PrEP`: The classic approach of pre-evaluating executable properties that consists of enumerating the valid symbolic data structures in advance by symbolically executing their constraints encoded as *executable properties* [8]. Encoding the representation invariants of an abstract data type in the form of executable methods, referred to as `repOk` methods, is a common practice in object oriented programming [20]. As an example, the code on white background in Figure 3 illustrates a possible `repOk` method for the class `LinkedList` partially reported in the figure: `repOk` returns `false` if the `header` is `null` (line 7), the `next` and `previous` references do not match between the data nodes (line 15) or the value of the field `size` differs from the count of data nodes (line 19). Technically, given a program `P` that takes as input the data structures $\texttt{in}_i$ with invariants encoded as `repOk` methods, the technique `PrEP` consists of symbolically executing the program $\texttt{if}(\bigwedge_i \texttt{in}_i.\texttt{repOk}())\,\texttt{P}(\texttt{in}_1, \ldots, \texttt{in}_n)$. As a drawback, this technique can over-constrain the symbolic inputs, fostering the analysis of multiple distinct assumptions on objects and fields that are not accessed by the program.

`ConsP`: The extension of the `repOk` approach proposed by Visser et al. that consists of evaluating the executable properties during (rather than before) symbolic execution [31]. The approach relies on the concept of *conservative* `repOk` methods, an extended version of the executable properties that can be evaluated against partially initialized data structures. As an example, Figure 3 reports the conservative extensions of the `repOk` method on grey background. The conservative `repOk` method returns `true` either if the field `header` is not initialized (line 6) or if field `next` of the traversed node is not initialized (line 11). The conservative `repOk` skips the next-previous check if the field "`previous`" of the next node is not initialized (line 14). The effectiveness of this technique depends on the quality of the conservative properties, since a direct extension of the original `repOk` may miss some invalid partial models, for instance, the implementation in Figure 3 misses the invalid partial models in

```
1  class LinkedList{...
2     int size = 0;
3     Entry header = new Entry();
4     class Entry{Entry next, previous; ...}
5     boolean repOk(){
6        if(σ(header)) return true;
7        if(header == null) return false;
8        Entry e = header;
9        int pos = 0;
10       do{
11          if(σ(e.next)) return true;
12          if(e.next == null) return false;
13          if(
14             !σ(e.previous.next) &&
15             e.previous.next != e) return false;
16          e = e.next;
17          if(e != header) pos++;
18       } while (e != header);
19       return pos == size;
20    }
21 }
```

$\sigma(\cdot)$ denotes a query to the symbolic executor on whether a reference holds a symbolic (not yet initialized) value in the current state.

Figure 3: Class `LinkedList` with a `repOk` method and its conservative version.

which nodes are linked through field `previous` while some fields `next` are not initialized yet. A further limitation is that the conservative `repOk` evaluates single data structure instances and cannot enforce constraints over multiple structures, and this may impact on the final results, as discussed in Section 4.

`Alloy`: The approach of using a bounded satisfiability prover to verify the validity of the data structures explored during symbolic execution. We instantiate this approach with Alloy, a relational logic language to describe the properties of object structures [16]. We rely on the Alloy analyzer to verify the satisfiability of the properties in conjunction with the structural assumptions that emerge during symbolic execution. The Alloy analyzer computes the satisfiability of the formulas provided an upper bound to the number of objects that can be assumed in the initial heap. As all bounded verifiers, Alloy can decide the satisfiability of a problem if it identifies a solution, otherwise the result is inconclusive, since a solution may always exist in a larger scope. `Alloy` requires the user to specify the bound limit, and accepts only the symbolic states whose satisfiability can be proved within the given bound limit. The choice of the upper bound impacts on the precision and performance of symbolic execution: small bounds increase the likelihood of rejecting valid symbolic states, big bounds slow down the process.

`PALE`: The approach of expressing the structural properties in some logic that comes with a decision procedure for unbounded problems. We instantiate the technique with the Pointer Assertion Logic Engine (PALE [25]). PALE builds on monadic second order logic to express properties for *tree-shaped graphs* of objects. The decision procedure is implemented on top of MONA, an established tool to address decision problems over a restriction of monadic second order logic [15]. PALE can yield conclusive results both against satisfiable and unsatisfiable problems, while the possible in-

Table 1: Size of the subject programs

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	All
LOC	361	514	322	211	607	7,972	5,972	15,959

LOC = Non-comment lines of code in the class under analysis and in the classes invoked by the class under analysis.

conclusive outcomes depend on the incompleteness of the decision procedure. For example, PALE does not handle numeric constraints and cyclic graphs of objects. The technique `PALE` rejects the symbolic states with conclusive unsatisfiability results, and accepts all others as either satisfiable or potentially satisfiable.

For the experiment reported in this paper, we extended the symbolic executor JBSE [3] to support the techniques `HEX`, `PrEP`, `ConsP`, `Alloy` and `PALE`. We would like to emphasize that only `Lazy`, `PrEP` and `ConsP` have been already proposed in the context of the symbolic execution of heap data structures. `HEX` is proposed in this paper, and `Alloy` and `PALE` are experimented in the context of symbolic execution for the first time in this paper.

4.2 Subject Programs

In the experiments, we challenge symbolic execution to analyze third party Java programs. The subject programs include both classes that incapsulate classic recursive data structures and open source Java components with structured input data. We consider the following subject programs, whose size is reported in Table 1:

P_1: Class `LinkedList` taken from the SIR repository [10] that implements doubly linked lists.

P_2: Class `TreeMap` taken from the SIR repository that implements red-black trees, a type of balanced binary trees.

P_3: An implementation of AVL trees taken from the experimental benchmark used by Galeotti et al. [12].

P_4: A caching circular double linked list that implements the interface `List` from the Apache Commons project, taken from the experimental benchmark used by Galeotti et al. [12].

P_5: Class `TrajectorySynthetizer` of the TSAFE prototype described in [9] that takes as input the position coordinates of a flight and the flight plan represented as a linked list of flight points, and computes the current trajectory of the flight.

P_6: Class `NodeTraversal` of the Google Closure-compiler for JavaScript (Git hub commit `509fec1`) that takes as input a parse tree encoded as an object of type `Node` and refers to a linked list of function control flow graphs, and implements a traversal of the nodes in the parse tree.

P_7: Class `RenameLabels` of the Google closure-compiler for JavaScript (Git hub commit `393fceb`) that takes as input a parse tree encoded as an object of type `Node` and some external `Node`s, and processes the nodes of type *LABEL*.

The experiments on the methods of the recursive data structures (P_1—P_4) study the relative strengths of the different approaches in breath, since these methods take objects of the type of the considered data structures as input and make several different computations that rely on the implicit assumption that the input objects satisfy the structural properties of the data structures. For these subject programs, we symbolically executed all single methods and pairs of method calls for a total of 274 analysis targets.

The experiments on the open source components (P_5—P_7) provide preliminary evidence of the effectiveness of the different approaches when using symbolic execution for testing and verification in practical software engineering contexts. For these programs, we symbolically executed methods with either correctness properties or known faults, referring to the correctness properties of P_5 described in [3] and to the faults of P_6 and P_7 mined by Just et al. and identified as `bug37` and `bug72`, respectively [17].

P_1 and P_2 include the `repOk` implementation of the representation invariant of the data structures, P_3 and P_4 include JML representation invariants. P_5 and P_6 refer to a linked list and share the same invariant of P_1. P_6 and P_7 refer to a parse tree, and thus we implemented the invariant that encodes the tree properties.

For all programs, we rephrased the structural properties as required by the considered technique: We implemented the logic specifications of P_3 and P_4 as executable predicates; We adapted all the executable predicates to a conservative version, by relaxing the constraints of the fields not yet initialized; We implemented the HEX and Alloy versions of all the representation invariants, and the PALE version of the representation invariants of P_2 and P_3. The invariants of the data types P_1, P_4, P_5, P_6 and P_7 cannot be modelled in PALE that does not handle cyclic graphs of objects, thus we do not experiment technique `PALE` on these programs. All property specifications are available as part of the experiment replication package.[2]

We executed the experiments with the JBSE symbolic executor ([3]) suitably extended to deal with the different techniques. Each experiment corresponds to an execution of JBSE instantiated with a technique in T on a method or a pair of methods of a subject program. We allocated a maximum time of 2 hours for each experiment with the recursive data structures P_1—P_4 that include many small size analysis targets, and of 6 hours for the experiments with the open source components P_5—P_7 that represent medium size analysis targets.

Some subject programs include loops and recursive calls, and thus the symbolic execution may not terminate. To enforce termination, we bound both the number of objects in the explored partial heaps, and the maximum depth of the execution traces. In the experiments, we bounded the size of the input list or tree of P_1—P_7 to 5, 3, 4, 4, 3, 4 and 4, respectively. The bounds take into consideration the complexity of the data structure that impacts on the duration of the symbolic execution.

The experiments with Alloy required us to specify a verification bound for the Alloy analyzer. This bound expresses the maximum number of objects that can be used to show that there exists a valid concrete heap that satisfies a partial heap assumed during symbolic execution. To avoid false negatives, the Alloy bound must be set higher than the heap bound of symbolic execution that constrains the size of the partial heaps. In all experiments with Alloy, we carefully optimized the `Alloy` bound as the minimal value to achieve the maximum precision, that is, to exclude all spurious traces without incurring false negatives.

4.3 Metrics

We quantify the *precision* of the symbolic execution in terms of the avoidance of invalid symbolic states and the side-effects induced by the techniques T on the valid symbolic states. We quantify the *efficiency* in terms of the time elapsed to complete the symbolic execution of the valid states.

We measure the avoidance of invalid symbolic states as the number of end-of-trace symbolic states that depend on some invalid initialization of the input objects. If there are no invalid end-of-trace symbolic states the precision is optimal, that is, the symbolic execution explores only valid symbolic states. Increasing amounts of invalid end-of-trace symbolic states correspond to increasingly less precise approaches.

We measure the side-effects induced by a given technique as the ratio R between the number of valid end-of-trace symbolic states counted when using the technique and using no technique (`Lazy`), respectively. The ratio R indicates if the technique $\bar{T} \in T$ interferes with the ability of the symbolic executor to discriminate the valid inputs. The interferences may result in either under-approximating or under-generalising the results of the symbolic execution. The technique $\bar{T}$ under-approximates the results if the symbolic executor explores less valid symbolic states when using the technique $\bar{T}$ than when using no techniques. The technique under-generalizes the results when some symbolic states computed when using $\bar{T}$ are subsumed by smaller sets of symbolic states when using no technique, meaning that the technique reduces the ability of generalising the symbolic execution. Under-approximation corresponds to $R < 1$, no effect to $R = 1$ and under-generalization to $R > 1$. The readers should notice that, when under-approximation and under-generalization effects interfere, values of the ratio R that strongly differ from 1 capture the dominance of either type of effect.

We measure the efficiency as the time required to complete the symbolic execution of each program, aiming to quantify the relative impact of the specific technique on the time cost of the symbolic execution.

We designed a measurement infrastructure that evaluates the structural properties of the final states. The experiments were executed on a linux server equipped with 4 Intel Xeon 8-core CPUs and 64 GB of RAM memory, using a makefile to parallelize the symbolic execution of the benchmark programs.

5. DATA ANALYSIS

This section presents the results of the experiments. Each of the 6 techniques in T is experimented against both the 274 analysis targets of P_1—P_4 and the methods related to correctness properties or faults in P_5—P_7. Some experiments ran out of memory, and are not included in the results. The final dataset includes data from 1,352 experiments.

5.1 Invalid Symbolic States

Figure 4 plots the distribution of invalid end-of-trace symbolic states that we measured across the experiments, and Table 2 reports the details of the experiments with the subject programs P_5 — P_7 that represent practical software engineering cases. The experimental data confirm that the lazy initialization algorithm produces a huge amount of spurious symbolic traces in the presence of structural properties of the symbolic data structures. The technique `Lazy` yielded invalid symbolic traces in 184 experiments with a median value of 2,114 invalid traces and up to a maximum of over 2,419,860 traces (in P_6 where, due to the time limit, it yielded invalid traces only). The techniques `ConsP` and `PALE` yielded invalid symbolic traces in 110 and 53 experiments, with third quartile values of 301 and 186 invalid traces, and maximum of 48,267 and 2,328, respectively, thus performing better than pure lazy initialization but still incurring precision problems in several cases. The techniques `PrEP` does not yield spurious traces for the recursive data structures (P_1 — P_4) but computes several invalid traces for the open source programs P_5 — P_7, as shown in Table 2. The techniques `HEX` and `Alloy` do not produce any false positive across all subject programs, that is, they always succeed in discarding the invalid symbolic structures produced by the lazy initialization algorithm.

Table 2: Experiments on the open source subjects

		Elapsed minutes	#Traces	#Invalid traces	#Alarms	#False alarms
P_5	Lazy	timeout	23,854	23,675	12,434	12,434
	HEX	**2**	**290**	**0**	**0**	**0**
	PrEP	9	2,250	1,690	**0**	**0**
	ConsP	52	1,130	840	**0**	**0**
	Alloy	56	**290**	**0**	**0**	**0**
P_6	Lazy	timeout	2,419,860	2,419,860	62,260	62,260
	HEX	**7**	**2,856**	**0**	**128**	**0**
	PrEP	154	96,256	90,240	4,096	3,840
	ConsP	41	6,426	3,570	288	160
	Alloy	306	**2,856**	**0**	**128**	**0**
P_7	Lazy	timeout	41,723	41,717	14,494	14,494
	HEX	**2**	**925**	**0**	**30**	**0**
	PrEP	12	26,960	25,850	2,130	2,100
	ConsP	251	25,045	24,120	2,100	2,070
	Alloy	17	**925**	**0**	**30**	**0**

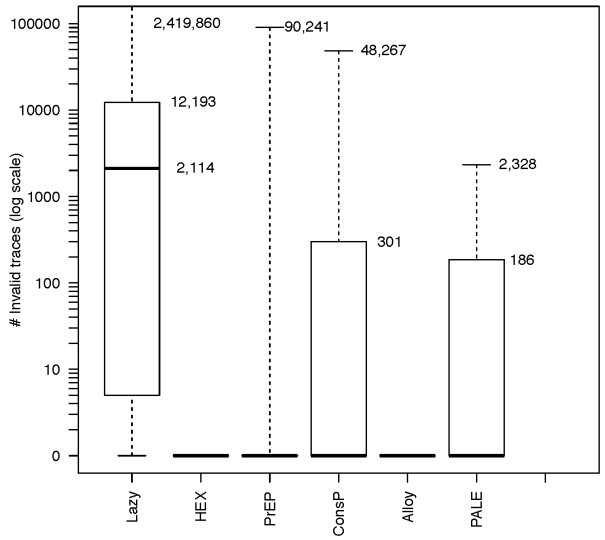
With reference to the faults and the verification goals involved with the analysis of the programs P_5 — P_7, the last two columns of Table 2 indicate that the spurious traces computed during the symbolic execution can result in large amounts of annoying false alarms that may impact on the software engineering practice.

The `PrEP` technique fails in discarding all the invalid traces because of the inability of the `repOk` methods to deal with alias relationships between distinct input structures. For example, the conservative `repOk` of the two parse tree parameters of class `RenameLabels` fails to identify illegal aliases between the input trees. This limit of the `repOk` approach with inter object constraints affects the `ConsP` technique too. Furthermore, the `ConsP` technique suffers from the incompleteness of the considered conservative predicates derived as straight adaptation of the `repOk` methods. For example, we already commented that the conservative adaptation of the `repOk` of class `LinkedList` of Figure 3 iterates only forward on the list, and cannot enforce the invariant on the partial data structures computed when analysing target code that accesses the list in the opposite direction. Thus our experiments provide empirical evidence that relying on `repOk` methods is not sufficient to discard all the invalid traces produced by the lazy initialization algorithm.

The invalid traces in `PALE` depend on the limits of PALE in accounting for the numeric constraints in the invariants.

5.2 Under-Generalization Effects

Figure 5 plots the distribution of the portion of valid end-of-trace symbolic states measured when using a given tech-



The box-plots report the minimum, the first quartile, the median (in bold), the third quartile and the maximum number of invalid traces.

Figure 4: Distribution of the invalid end-of-trace states yielded by symbolic execution when combined with the considered techniques.

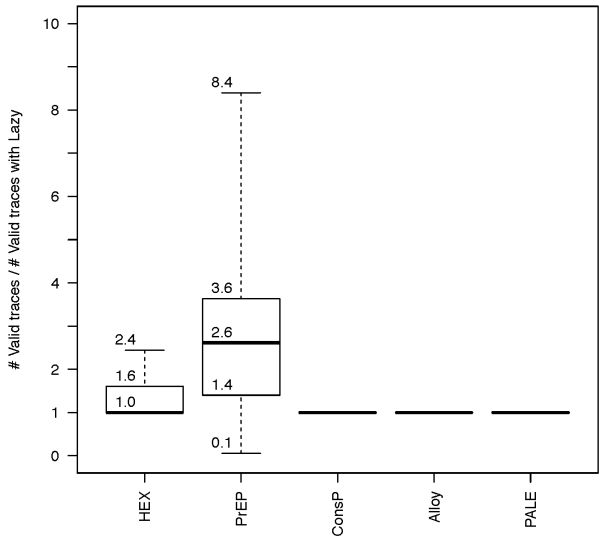


Figure 5: Distribution of the ratio between the valid end-of-trace states yielded by symbolic execution when combined with the considered techniques with respect to plain lazy initialization.

nique with respect to the baseline case of relaying on lazy initialization only. Only `PrEP` incurs in under approximation effects, meaning that it causes symbolic execution to explore less traces than when using `Lazy` (values less than 1 in Figure 5). The reason is that enumerating all objects in the input data structures beforehand results in over-constraining the scalar fields that depend on such objects, for instance, the field `size` of class `LinkedList`.

Both `PrEP` and `HEX` incur some under generalization effect, meaning that they cause symbolic execution to explore more traces than when using `Lazy` (values greater than 1 in Figure 5), `PrEP` to a large and `HEX` only to a partial extent. The relevant under generalization effects in `PrEP` depend on the early enumeration of the possible input structures that competes with the goal of lazy initialization of reducing the number of traces to be analyzed by delaying the resolution of the symbolic references.

The under generalization of the `HEX` approach manifests only for the red-black tree case study (P_2), and depends on the tight relation between the valid shapes of the data structure and the scalar values that represent the red/black status of the internal data nodes: Evaluating the valid initializations of the data nodes requires HEX companion methods that foster the symbolic executor to enumerate the possible configurations of red and black nodes for the incrementally assumed partial heaps. Table 2 confirms that `HEX` does not incur any under generalization on the considered open source programs.

5.3 Efficiency

Figure 6 presents the time for completing the symbolic execution of the subject programs when using each considered technique with respect to `HEX`. We observe that only the performance of `PALE` is comparable with the average execution time of `HEX`, while the median execution times of `PrEP`, `ConsP` and `Alloy` are 2.4, 3.4 and 4.3 times the execution time of `HEX`, respectively.

Table 2 emphasizes the performance advantages of `HEX` with respect to `Alloy`, the only competing technique in line with `HEX` in terms of precision: For the analysis of the subjects P_5, P_6 and P_7, `HEX` is faster than `Alloy` of a factor 28, 44 and 8, respectively. The data confirm the well-known problem that the performance of a bounded verifier quickly falls down with the increase of the bound value, and indicate that `Alloy` can result in unacceptable performance even with medium bound values, for instance the optimal `Alloy` bound was 13 for P_5 and P_6 and 9 for P_7, that are likely to be minimal requirements in the case of realistic applications.

5.4 Summary

The experimental data confirm that the main merit of HEX is the scalability of the approach without giving up precision. The HEX approach outperforms `PrEP`, `ConsP` and `PALE` in terms of precision, in that it does not incur any false positive or false negative, and does not result in under generalization effects for 6 out of the 7 subject programs. Alloy shares similar precision figures as HEX, but the performance of the analysis is significantly better with HEX than with Alloy. The good performance of HEX is a result of the incrementality of its decision procedure that represents a major improvement of HEX on the state of the art approaches.

5.5 Threats to Validity

A main threat to the validity of the results springs from the dependence of the `ConsP` approach on the implementation of the conservative properties. We mitigate this threat in two ways: In the experiments with P_1—P_4 we use the transformation of the properties specified by the developers into conservative properties according to the approach of the `ConsP` proposers [31]; In the experiments P_5—P_7 we have implement optimal versions of the conservative properties based on our experience with the partial models generated during the symbolic execution of the target programs. The results of the experiments confirm that different implementation choices do not affect the results of the experiments.

The quality of the measurements depends on the relia-

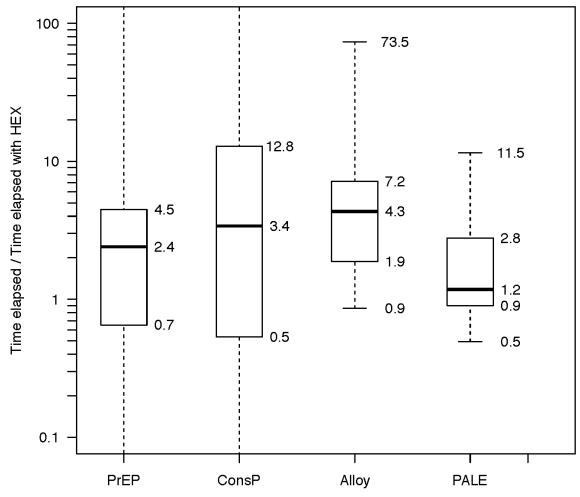


Figure 6: Distribution of the relative time spent to complete the symbolic execution when using either a technique $\bar{T}$ or `HEX` for the experiments where both $\bar{T}$ and `HEX` complete within the time budget.

bility of both the symbolic executor and the measurement infrastructure used in the experiments. We have been developing and testing JBSE for several years. The implementation of the techniques `HEX`, `ConsP`, `Alloy` and `PALE` in JBSE is new, but the consistent number of valid traces that JBSE computes when these techniques are activated and deactivated, supports the trustability of the current implementation. We have crosschecked the results from the different techniques and manually verified samples of the traces identified as spurious to increase our confidence in the reliability of the measurements.

We are aware that the result of few experiments cannot be directly generalized. Nonetheless, the consistency of the measured data across the subject programs is a promising indication that our results may generalize. Our future plans include extending the experiment to other data structures and beyond the experimental samples of this paper.

6. RELATED WORK

In this section we briefly survey the main approaches to deal with the symbolic representation of dynamically allocated structured data. The problem of limiting the assumptions on heap structures by considering their structural constraints has been considered in the context of both symbolic execution and logic-based automated verification of heap manipulating programs.

In the context of symbolic execution, the problem has been initially investigated by Visser et al., who proposed the lazy initialization approach [18, 31] that has been further extended by Deng et al. [8]. We discussed the comparison of HEX with Visser's `PrEP` and Deng's `ConsP` approaches in Section 5. Boyapati et al., Galeotti et al. and Majumdar et al. proposed approaches that enumerate the valid data structures through a bounded search of the input space [2, 12, 23]. Recently, Geldenhuys et al. instantiated the approach of Galeotti et al. to pre-compute the sets of *possibly valid* inter-object edges for the objects of the input structures, and exploited this information within the lazy initialization algorithm to evaluate the validity of the incrementally assumed references [13]. Being based on a bounded verifier, the precision of this approach depends on the choice of the bound value, with similar implications to the ones that we discussed in Section 4.1 with reference to `Alloy`. Furthermore, the set of pre-computed inter-object edges over approximates the edges that are valid for any path condition, and thus the approach of Geldenhuys et al. cannot guarantee the precision figures observed with Hex and Alloy in our experiments.

In the context of logic-based automatic verification, several authors proposed logics and decision procedures to address heap manipulating programs that must maintain reach invariants for the data structures in the heap [16, 25, 21, 27, 24, 32, 1, 22, 11]. Bounded checkers, like Alloy, compute the satisfiability of structural properties within bounded scopes of objects in the solution, translating the decision problems in boolean logic and using a SAT-solver to find a satisfying assignment for the boolean formula [16]. The need to accept a bounded solution is the main limit of this approach. Logics like PAL (the Pointer Assertion Logic) and STRAND (the STRucture ANd Data logic) build on monadic second order logic to prove unbounded properties of graphs of objects [25, 21]. STRAND combines the ability to reason on graph structures, with the ability to reason on the numeric data encapsulated in the objects. Other authors restrict their logics to selected types of reachability properties to enable the implementation of the corresponding decision procedures within SMT solvers [27, 24, 32]. Yet other researchers investigate the use of *separation logic*, aiming to take advantage of the local and compositional reasoning provided by the separation logic [28, 1, 22, 11]. These approaches have not been exploited in the context of symbolic execution yet. In this paper we have experimented with the techniques `Alloy` and `PALE` as representatives of this class of approaches.

7. CONCLUSIONS

Symbolic execution is the core component of many approaches to automatically generate test cases and verify system properties. While symbolic execution has been successfully exploited to generate test cases for complete programs with numeric inputs, generating unit, integration and subsystem test cases for programs that make extensive use of heap data structures is still an open challenge. The mainstream approach to deal with heap data structures, referred to as lazy initialization, does not take into account satisfactorily the structural constraints of data structures, and results in a waste of effort and in many false positives that limit its effectiveness.

In this paper we propose a new approach that prunes input data structures that violate structural constraints incrementally while symbolically executing the target program. In this way symbolic execution focuses on valid data structures, avoiding wasting effort and resources with invalid data structures, and preventing many consequent false alarms.

The experimental results reported in the paper confirm that the HEX approach improves over lazy initialization and over the most relevant state-of-the-art approaches.

We believe that the different approaches can be further strengthened by exploring their complementarities, and we are currently working on combining the different approaches in light of the results obtained so far.

8. REFERENCES

[1] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis*, LNCS, pages 167–182. Springer, 2012.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *International symposium on Software testing and analysis*. ACM, 2002.

[3] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 411–421, New York, NY, USA, 2013. ACM.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*. ACM, 2006.

[6] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. on Soft. Eng.*, 2(3):215–222, Sept. 1976.

[7] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.

[8] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *International Conference on Automated Software Engineering*, 2006.

[9] G. D. Dennis. TSAFE: Building a trusted computing base for air traffic control software. Master's thesis, Massachusetts Institute of Technology, 2003.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[11] C. Enea, V. Saveluc, and M. Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *Proceedings of the European Conference on Programming Languages and Systems*, ESOP'13, pages 129–148. Springer-Verlag, 2013.

[12] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 25–36, New York, NY, USA, 2010. ACM.

[13] J. Geldenhuys, N. Aguirre, M. Frias, and W. Visser. Bounded lazy initialization. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, LNCS 7871. Springer Berlin Heidelberg, 2013.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM Conference on Programming language design and implementation*. ACM, 2005.

[15] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. of the Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAS. Springer-Verlag, 1995.

[16] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[17] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014.

[18] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2619. Springer, 2003.

[19] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[20] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[21] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using strand. In *Proc. of the 18th International Conference on Static Analysis*, SAS'11, pages 43–59. Springer-Verlag, 2011.

[22] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–136, New York, NY, USA, 2012. ACM.

[23] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proc. of the Int. Conference on Automated Software Engineering*, ASE '07, New York, NY, USA, 2007. ACM.

[24] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Proc. of the 17th International Conference on Computer Aided Verification*, CAV'05. Springer-Verlag, 2005.

[25] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 221–231, New York, NY, USA, 2001. ACM.

[26] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transf.*, 11(4):339–353, Oct. 2009.

[27] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an SMT framework. In *Proc. of the 5th International Conference on Automated Technology for Verification and Analysis*, ATVA'07, pages 237–252. Springer-Verlag, 2007.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer*

Science, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and ACM Symposium on Foundations of Software Engineering*, 2005.

[30] J. H. Siddiqui and S. Khurshid. Staged symbolic execution. In *ACM Symposium on Applied Computing*, 2012.

[31] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 97–107. ACM, 2004.

[32] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Proc. of the 9th European Joint Conference on Foundations of Software Science and Computation Structures*, FOSSACS'06. Springer-Verlag, 2006.