Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems*

Xianghua Deng
Kansas State University
deng@cis.ksu.edu

Jooyong Lee
BRICS, University of Aarhus
jlee@brics.dk

Robby
Kansas State University
robby@cis.ksu.edu

Abstract

This paper presents Kiasan, a bounded technique to reason about open systems based on a path sensitive, relatively sound and complete symbolic execution instead of the usual compositional reasoning through weakest precondition calculation that summarizes all execution paths. Kiasan is able to check strong heap properties, and it is fully automatic and flexible in terms of its cost and the guarantees it provides. It allows a user-adjustable mixed compositional/non-compositional reasoning and naturally produces error traces as fault evidence. We implemented Kiasan using the Bogor model checking framework and observed that its performance is comparable to ESC/Java on similar scales of problems and behavioral coverage, while providing the ability to check much stronger specifications.

1 Introduction

Best practices in software development techniques nowadays heavily emphasize the development of reusable and modular software, which allows software components to be developed and maintained independently. In addition, many modern programming languages provide language constructs at higher abstraction levels to help manage software complexity. For example, object-oriented programming languages have gained popularity over the recent years as they support the development of software components. While they have improved software development processes employed today, a set of unique challenges have emerged for assuring their quality. One of the main challenges is to ensure software compatibility across independently-developed components. Although design-by-contract [18] offers a promising solution, there is still a need for effective analysis techniques that are able to reason about software contracts and their implementation using modern programming languages in the context of open systems (*i.e.*, where portions of the system are *unavailable* and only their specifications exist.)

We present Kiasan (kē' ah sahn, Indonesian for reasoning with analogy/symbolically), a technique for reasoning about behavioral properties of open (sequential) systems, including strong heap properties that describe heap structures and the data in them. Our approach is driven by a number of design goals that distinguish it in one or more ways from existing work (*e.g.*, [8, 2, 9, 4]):

G1. *Provides fully automated analysis*: To gain wide-spread adoption from software developers, it is crucial for analysis tools to require no manual intervention.

G2. *Handles rich heap-oriented specifications*: Software written with newer programming languages such as Java and C# heavily use dynamically-allocated heap objects. It is imperative for an analysis tool to be able to reason about these objects, their data, and their relationships (*e.g.*, [20].)

G3. *Allows mixed compositional and non-compositional analysis*: While pure compositional reasoning is more scalable, one of its usability problems is that it requires an up-front effort for having comprehensive specifications. In the case where an implementation exists or where it is easier to implement than to specify, we should allow one to easily configure the analysis to use the implementation. This allows one to focus on checking the more important parts of the system without undesirable warning or error messages as is the case when using a pure compositional reasoning approach with incomplete/nonexistent specification.

G4. *Flexibility to adjust analysis cost and coverage*: We believe that an analysis tool should provide enough control over the computational resources an analysis requires, and it should provide quantifiable behavior coverage guarantees. These allow users to increasingly allocate more resources to gain higher levels of confidence from the tool. For example, when assuring correctness of a method which sorts a list, it is of little help to use techniques such as iterative deepening in depth-first state-space exploration (*e.g.*, [11, 22]), since the link between a program and a suitable analysis depth is

*This work was supported in part by a 2005 IBM Eclipse Innovation Grants, by Lockheed Martin, by the Air Force Office of Scientific Research, by the U.S. Army Research Office (DAAD190110564), and by the National Science Foundation (CCR-0306607, CCF-0429149, CCF-04444167).

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

very difficult to see. Rather, one should be able to specify ones willingness to invest the resources necessary to obtain correctness assurance on lists up to a certain size.

G5. *Provides helpful analysis feedback:* In contrast to errors caused by uncaught exceptions such as null-dereference, it is not enough to only point out at a program point for violations of strong contracts. When the analysis finds an error, it should give helpful feedback that explains it, and at the very least, generate an error scenario as evidence.

We believe the key contribution of our approach is a collection of insightful design and engineering decisions that lead to an effective tool that hits a "sweet spot" with respect to the capabilities that software developers would need in practice. Specifically, we believe that the scalability of our technique compares well to frameworks like ESC/Java [8], while providing better support for checking strong heap properties. Our work takes the symbolic execution presented in [11] and adds several important enhancements that allow it to achieve the design goals above: (1) we develop a bounding technique that provides better control appropriate for checking heap properties; we identify and address sources of unsoundness or intractability in [11], (2) we introduce a more efficient lazy initialization algorithm compared to [11], (3) we formalize Kiasan's basic symbolic execution engine and prove its simulation relation to concrete execution and its relative soundness and completeness (available at [26]), (4) we show how Kiasan can be used to reason about strong heap properties of open systems in a mixed compositional/non-compositional way, (5) we present how Kiasan can leverage heap region properties to reduce its analysis cost as well as enabling it to check heap-oriented properties, (6) we implement Kiasan on top of the Bogor software model checking framework [19, 26], and (7) we demonstrate how Bogor/Kiasan can check properties difficult to handle automatically using other methods.

In our paper, we use a specification language with features similar to the Java Modeling Language (JML) [15], but we did not design our approach around a particular specification language. Instead, we require a side-effect free specification that is transformable to an effective executable form. While there is a lot of technical and engineering effort for transforming specification to an executable form, it is beyond the scope of this paper. Existing methods such as `jmlc` for JML runtime monitoring [6] can be employed to transform specification into an executable form. In addition, we also use some relational properties on methods such as transitivity.

The next section presents an example that motivates our approach. Section 3 presents our *k*-bounded symbolic execution approach, and Section 4 outlines how we adapt symbolic execution for reasoning about open systems and how we leverage heap region information to reduce analysis cost. Section 5 discusses our prototype and experiments. Section 6 presents related work, and Section 7 concludes.

```
public class LinkedList<E> {
  //@ inv: isAcyclic();
  @NonNull LinkedNode head = new LinkedNode();

  /*@ pre: isSorted(c)
    @      && other.isSorted(c);
    @ post: isSorted(c); @*/
  void merge(@NonNull LinkedList<E> other,
                      @NonNull Comparator<E> c) {
    LinkedList<E> ll = new LinkedList<E>();
    LinkedNode n1 = this.head.next;
    LinkedNode n2 = other.head.next;
    while (n1 != null && n2 != null)
      if (c.compare(n1.data, n2.data) < 0)
        { ll.addLast(n1.data); n1 = n1.next; }
      else { ll.addLast(n2.data); n2 = n2.next; }
    while (n1 != null)
      { ll.addLast(n1.data); n1 = n1.next; }
    while (n2 != null)
      { ll.addLast(n2.data); n2 = n2.next; }
    head = ll.head;
  }
  class LinkedNode { E data;  LinkedNode next; }
}
```

Figure 1. A Merge Example (excerpts)

2 Motivating Example

Figure 1 presents a sorted list merge example that motivates our approach. Intuitively, the `merge` method's contract indicates that given a non-null and sorted (from the preconditions `@NonNull` and `pre`) acyclic list (from the invariant `inv`) with respect to the specified `Comparator c`, the method merges the content of that list into the receiver list object (given that it is also sorted) and as the result, the receiver object is also a sorted acyclic list (`isAcyclic` and `isSorted` are pure, i.e., they do not modify existing objects). We highlight several non-trivial challenges when reasoning about such programs and specifications that we address in the next sections.

1. The `compare` method is *open-ended*, *i.e.*, in contrast to reasoning about a complete system such as [11] where we know the actual objects and the data being manipulated, we do not know the actual implementation of the method (or even if there is an implementation for the type that will substitute `E`). Thus, the objects used to determine its result are unknown as it may use all data that it can reach. This is also in contrast to reasoning about specific algorithms on data structures whose elements are of scalar types or Java's immutable objects [16, 25, 12, 1], for example, for ordering, where we know the comparison does not use data from other heap objects. Thus, the strongest specification we can have is that the `compare` method is pure, and it returns either a negative integer, zero, or a positive integer. Moreover, `compare` is a total order relation, as specified in the Java 5 Application Programming Interface (API) documentation.

2. In contrast to `compare`, an implementation of the `addLast` method is easily understood, *i.e.*, it *only modifies* the last node's `next` field of the receiver list object by assigning its parameter to it; it is actually easier to use the actual implementation than its specification, *i.e.*, one can focus on checking `merge` first by using an implementation

of `addLast`. Thus, it avoids one of the usability problems when using pure compositional reasoning techniques such as [8] that require comprehensive specifications up-front before being able to check them; otherwise, it generates undesirable error or warning messages.

3. We would like to strengthen `post` to ensure that the resulting list size is the sum of the receiver list size before merging and the size of `other`, and all the elements are from the two lists. In contrast to techniques that mostly concern about heap shapes [16], these kinds of properties make it hard to automatically use heap abstraction techniques that summarize objects because one still has to maintain the elements (whose numbers may be unbounded.)

4. We have to establish that whatever information used for comparison must not be modified by `merge` (or methods called from it) to ensure the comparisons done later in `post` are unaffected. (This is a bit too strong as it is fine to modify any information that will be accessed by `compare` if it does not change `compare`'s result; we only consider the former case in this paper.) Otherwise, there is no guarantee that the receiver object is sorted afterward. For example, suppose that we insert a code just before the end of `merge`. We need to check whether the inserted code invalidates the elements' ordering. This can be detected by using heap region separation. That is, the inserted code cannot invalidate the ordering if it does not modify the element objects. However, establishing this requires a precise heap analysis that is able to leverage, for example, heap region information.

3 k-Bounded Symbolic Execution

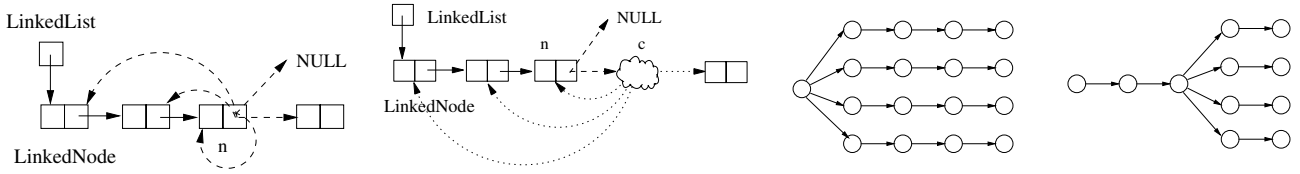
In this section, we describe our basic (non-compositional) and stateless symbolic execution technique that improves [11] by introducing an intuitive bounding technique, as well as techniques to improve its performance and to address its sources of unsoundness or intractability.

Background: *Symbolic execution*[13] is a technique that essentially interprets a program. However, it uses *symbolic values* instead of concrete values (*e.g.*, integers.) While executing a program, it maintains the relationship of the symbolic values as path conditions. For example, consider the following code: `z = x + y; if (z > 0) z++;`
Suppose the values of `x` and `y` in the beginning are α and β, respectively. After executing the assignment, we know that the value of `z` is γ, where $\gamma = \alpha + \beta$, *i.e.*, we add the additional knowledge to the path conditions. When evaluating the branch condition, we do not have enough information to decide which branch to follow, thus, we non-deterministically follow both branches to safely simulate possible real executions. When following the false branch, we know that at the end, $\neg(\gamma > 0)$, and we have finished executing the code. On the other hand, in the true branch, we know that $\gamma > 0$. At the end, `z`'s value is γ', where $\gamma' = \gamma + 1$. Each of these execution paths characterizes (theoretically) an infinite number of real executions.

For objects and arrays, we use the lazy initialization algorithm in [11] that initializes field values on an on-demand basis. If a field is accessed for the first time, then its value is "initialized lazily" as follows. If the field's type is a primitive type, then a new symbolic value is created. Otherwise, the algorithm covers all possible aliases by non-deterministically choosing the NULL value, any existing symbolic object in the heap whose type is compatible with the field's type, or a fresh symbolic object. Arrays present additional challenges, *i.e.*, the length of an array may be unknown. In addition, arrays can be accessed by a symbolic (unknown) integer index. We treat array similar to [1], but with bounding the number of lazy initializations on array elements. We refer the reader to [11, 1] for more detailed discussions on lazy initialization of objects and arrays.

Issues in Symbolic Execution: The symbolic execution described above is quite effective (*e.g.*, it has precise alias information), automatic, and naturally able to produce counter-examples as error evidence, *i.e.*, it is in-line with **G1**, **G2**, and **G5**. However, it can become unsurprisingly expensive. The inherent non-determinism in the lazy initialization algorithm becomes more expensive as the number of symbolic objects grows. Another challenging issue is handling possible non-termination due to loops (and recursions.) Even using a stateful search does not help due to the unboundedness of heap. For example, symbolic execution of a non-terminating loop that adds concrete objects to a linked list does not terminate gracefully. (Abstraction techniques [16, 1] can be employed to summarize some heap structures; while promising, it is still difficult to automatically construct precise abstractions for every strong property.) A common approach to address this (*e.g.*, [11, 9]) is to use a depth-bounded search which provides control over the control flows under consideration, but it provides little control over data structures. That is, it is hard to quantify the behavioral coverage that is guaranteed by such method as mentioned previously in Section 1. Thus, it conflicts with **G4**. Another approach is to exploit loop invariants (*e.g.*, [11]); however, automatically inferring loop invariants is incomplete in general. Thus, it conflicts with **G1**.

***k*-bounding:** To address the above issues, we incorporate a bounding technique to help manage symbolic execution's complexity. That is, we bound the sequence of lazy initializations originating from each initial symbolic object up to k. In other words, we bound the length of reference chains on symbolic objects. (We should only count initializations done inside loops, but this complicates our presentation.) For arrays, we additionally bound the number of lazy initializations on distinct array indices up to k. This user-adjustable bounding provides a fair trade-off between analysis cost and behavioral coverage; we can quantify the amount of coverage on heap objects for a given bound, thus satisfying **G4**. That is, when using a bound k, the analy-



(a) Lazy Initialization (b) Lazier Initialization (c) Lazy Execution Tree (d) Lazier Execution Tree

Figure 2. Lazy and Lazier Initializations

sis can guarantee the correctness of a program on any heap object configuration (satisfying its contract) with reference chains whose lengths are at most k. In the case where the analysis does not exhaust k, a complete behavior coverage is guaranteed. Note that this is different from the bounding techniques used in, for example, Korat [5] where it bounds the number of objects of each type. (Mixed bounding techniques are possible, however, we only use k-bounding to keep our presentation simple.) Due to the nature of lazy initialization, the expanded objects are only the ones that are needed or accessed.

To handle diverging loops, we limit the number of loop iterations *that do not (lazily) initialize any heap object*, *i.e.*, we prefer exhausting the k resource-bound first before resorting to loop bounding to try to guarantee the advertised heap object configuration coverage.

Lazier Initialization: During our initial experiments with *lazy initialization*, we observed a large number of paths generated by its inherent non-determinism that do not really contribute to different errors. The essence of the lazy initialization algorithm is delaying the initialization of a field f of a symbolic object o until $o.f$ is accessed the first time. We observed that it is possible to delay the initialization even further until $o.f$ (for non-primitive field f) is used in field accesses (*e.g.*, $o.f.g$), equality tests (*e.g.*, $o.f = o'$), and as receiver objects for method calls. Hence, we abstract lazy initialization into lazier initialization as follows: a field access $o.f$ only returns a *symbolic reference*; later usages of $o.f$ (*e.g.*, $o.f.g$) will initialize $o.f$ by choosing from compatible existing heap objects or a fresh symbolic object.

Figures 2(a,b) illustrate the difference between lazy and lazier initializations. (Dashed lines represent non-deterministic choices, and dotted lines represent possible future non-deterministic choices; the state before initialization is the state without dashed or dotted lines.) In (a), when `n.next` is lazily initialized, we non-deterministically choose between NULL, the first three nodes, or a fresh symbolic node (it does not choose the list object because of type incompatibility), thus, the symbolic execution branches on five paths. On the other hand, the lazier initialization in (b) creates a symbolic reference (represented as a cloud `c`) and branches two-ways. Only if the actual value of `c` is used (*e.g.*, for field access), then the cloud is dissolved into a non-deterministic choice among existing symbolic objects and a new symbolic object. Thus, it delays the branching until it is actually needed. In (b), we indicate the possible objects for the future non-deterministic choice when the cloud is dissolved; however, this object collection represent a subset of the future non-deterministic choice. The actual set of objects will be computed *at the time when the cloud is dissolved*, which may include more objects. This is crucial for proving the soundness of lazier initialization. Figures 2(c,d) demonstrate the effect of delaying non-deterministic choices in lazier initialization (*i.e.*, delaying branchings in state-space exploration.) The state-space tree in (d) has fewer transitions than (c) because it does its non-deterministic choice a bit later. Thus, it is more efficient to explore the state-space in (d). The lazier initialization algorithm is even more efficient in the case where a symbolic reference is never dissolved.

Formalization: Figure 3 presents the formalization of our k-bounded symbolic execution described above; formal simulation proof as well as relative soundness and completeness proofs of our symbolic execution can be found at [26]. We distinguish two kinds of symbols: primitive symbol (non-object symbols such as symbolic integers), and object (and array) symbols. Each symbol $\alpha_\tau^{m,n}$ has three attributes: the type τ of the symbol, the object field or array element initialization bound m, and the number of array elements bound n. (We will discuss the difference between m and n for arrays near end of this section.) Note that we will often omit attributes when they are unnecessary. $\alpha_\tau^{m,n}$ is a partial function from *Indices* to *Values*: empty mapping for a primitive value and total function (with respect to its actual fields) for a concrete (real/non-symbolic) object. Each symbolic reference (*i.e.*, an element of *SymLocs*), which represents a non-NULL unknown reference for lazier initialization, also has the three attributes. Without loss of generality, we represent a state as a tuple of global variables, a program counter, locals, a stack, a heap, and a conjunctive-set of path conditions (boolean formulas) ϕ. Transition configurations are expressed as the form of $s_1 \vdash_S instr \Rightarrow s_2$. This derives a configuration $s_2 \vdash_S instr' \Rightarrow s_3$, where $instr'$ is the instruction located at the pc of s_2, if it exists. Otherwise, we assume the execution terminates. Reaching an error state stops the execution in the current path and produces an error trace. We also assume the loop bounding discussed earlier is done orthogonally. Moreover, we stop exploring paths whose state's path condition is unsatisfiable.

Arithmetic operations and branch instructions are per-

Semantic Domains

INT	$\in$	$Types_{prim}$	=	primitive types: int,char,float,etc.	v	$\in$	*Values*	=	$Consts \cup Locs \cup Symbols_{prim} \cup SymLocs$
		$Types_{non-prim}$	=	$Types_{record} \uplus Types_{array} \uplus SymTypes$	ι	$\in$	*Indices*	=	$Fields \cup \mathbb{N} \cup Symbols_{\text{INT}}$
τ	$\in$	*Types*	=	$Types_{prim} \uplus Types_{non-prim}$	$\alpha^{m,n}_{\tau}, \beta, \gamma$	$\in$	*Symbols*	=	$\{\alpha^{m,n}_{\tau} \mid \alpha^{m,n}_{\tau} : Indices \rightharpoonup Values\}$
pc	$\in$	*PCs*	=	the set of program counters				=	$Symbols_{prim} \uplus Symbols_{non-prim}$
ϕ	$\in$	Φ	=	$\mathcal{P}$(the set of boolean expressions)	g	$\in$	*Globals*	=	$\{g \mid g : Fields \rightharpoonup Values\}$
i, j	$\in$	*Locs*	=	the set of locations	σ	$\in$	*Stacks*	=	$\{\sigma \mid \sigma$ is a sequence of values$\}$
$\delta^{m,n}_{\tau}$	$\in$	*SymLocs*	=	the set of symbolic locations	l	$\in$	*Locals*	=	$\{l \mid l : \mathbb{N} \rightharpoonup Values\}$
m, n, k	$\in$	$\mathbb{N}$	=	the set of natural numbers	h	$\in$	*Heaps*	=	$\left\{h \mid h : Locs \rightharpoonup Symbols_{non-prim}\right\}$
NULL, c, d	$\in$	*Consts*	=	the set of constants including $\mathbb{N}$	s	$\in$	*States*	=	$Globals \times PCs \times Locals \times Stacks \times Heaps \times \Phi$
LEN, DEF, CONC, f_{τ}	$\in$	*Fields*	=	the set of fields					

Auxiliary Functions ($\downarrow$ = defined, $\uparrow$ = undefined)

default	=	$\lambda\tau.v$, where v is τ's default value	*symbols*	=	$\lambda s.\{\alpha \mid \alpha$ appears in $s\}$		
fields	=	$\lambda\tau.\{f_{\tau'} \mid f_{\tau'}$ is a field in $\tau\}$	*new-prim-sym*	=	$\lambda(\tau, ps).\alpha_{\tau}, \alpha \notin ps$		
$\tau' <: \tau$	=	τ' is a subtype of τ (reflexive)	*new-sym-type*	=	$\lambda ps.\tau$ s.t. $\tau \in SymTypes \wedge \tau$ does not appear in ps		
acc-idx	=	$\lambda\alpha.\{k \in \mathbb{N} \cup Symbols_{\text{INT}} \mid \alpha(k)\downarrow\}$	*array-type*	=	$\lambda\tau.\tau'$, where τ' is a array type of element type τ		
collect	=	$\lambda h.\{i \mid h(i) = \alpha \wedge \alpha(\text{CONC})\uparrow\}$	*subst*	=	$\lambda(s, \delta, i).s'$, s' is the resulting state of substituting δ with i in s.		

new-sym = $\lambda(ps, m, n).\alpha^{m,n}_{\tau}$, s.t. $\alpha \notin ps \wedge \tau = new\text{-}sym\text{-}type(ps) \wedge \forall\iota \in Indices.\alpha(\iota)\uparrow$

new-sarr = $\lambda(ps, m, n).new\text{-}sym(ps \cup \{\alpha\}, m, n)[\text{LEN} \mapsto \alpha]$ where $\alpha = new\text{-}prim\text{-}sym(\text{INT}, ps)$

new-obj = $\lambda(ps, \tau).\alpha^{0,0}_{\tau}$, s.t. $\alpha \notin ps \wedge \forall f_{\tau'} \in fields(\tau).\alpha(f_{\tau'}) = default(\tau')$

new-arr = $\lambda(ps, \tau, v, n).\alpha^{0,n}_{\tau'}, \alpha \notin ps \wedge \tau' = array\text{-}type(\tau) \wedge \text{dom}\,\alpha = \{\text{DEF}, \text{LEN}, \text{CONC}\} \wedge \alpha(\text{DEF}) = default(\tau) \wedge \alpha(\text{LEN}) = v$

init-loc = $\lambda((g, pc, l, \sigma, h, \phi), \delta^{m,n}_{\tau}).\{subst((g, pc, l, \sigma, h', \phi'), \delta, i) \mid i \in collect(h), h' = h, \phi' = \phi \cup \{\tau' <: \tau\}$ where $h(i) = \alpha_{\tau'}$;
$i \notin \text{dom}\,h, h' = h[i \mapsto \gamma^{m,n}_{\tau'}], \phi' = \phi \cup \{\tau' <: \tau\}$ if $\tau \in Types_{record} \wedge m \geq 0$
where $\gamma_{\tau'} = new\text{-}sym(symbols(g, pc, l, \sigma, h, \phi), m, n)$; (the array case is similar)$\}$

Transitions (Non-deterministic, k-Bounded): $s \vdash_{\mathcal{S}} instr \Rightarrow s_1[stmt_1] \mid \ldots \mid s_n[stmt_n] \mid exception, s_0[stmt_0]$

$(g, pc, l, \alpha :: c :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `iadd` $\Rightarrow (g, nxt(pc), l, \beta :: \sigma, h, \phi \cup \{\beta = c + \alpha\})$ where $\beta = new\text{-}prim\text{-}sym(\text{INT}, symbols(g, pc, l, \alpha :: c :: \sigma, h, \phi))$

$(g, pc, l, \alpha :: c :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `if_icmplt` $pc' \Rightarrow (g, nxt(pc), l, \sigma, h, \phi \cup \{c \not< \alpha\}) \mid (g, pc', l, \sigma, h, \phi \cup \{c < \alpha\})$

$(g, pc, l, i :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `getfield` $f_{\tau} \Rightarrow (g, nxt(pc), l, v :: \sigma, h, \phi)$ where $v = \beta^{m,n}(f_{\tau}), \beta^{m,n} = h(i)$ if $\beta^{m,n}(f_{\tau})\downarrow$;

$(g, nxt(pc), l, v :: \sigma, h[i \mapsto \beta^{m,n}[f_{\tau} \mapsto v]], \phi)$ when $\beta^{m,n} = h(i)$,

$v = new\text{-}prim\text{-}sym(symbols(g, pc, l, i :: \sigma, h, \phi), \tau)$ if $\beta^{m,n}(f_{\tau})\uparrow \wedge \tau \in Types_{prim}$;

$v = \text{NULL}$, if $\beta^{m,n}(f_{\tau})\uparrow \wedge \tau \in Types_{non-prim}$;

$v = \delta^{m-1,k}_{\tau}$, where δ is fresh if $\beta^{m,n}(f_{\tau})\uparrow \wedge \tau \in Types_{non-prim}$;

$(g, pc, l, \delta^{m,n}_{\tau} :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `getfield` $f_{\tau} \Rightarrow s'$ where $s' \in init\text{-}loc((g, pc, l, \delta^{m,n}_{\tau} :: \sigma, h, \phi), \delta)$

$(g, pc, l, v :: i :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `putfield` $f_{\tau} \Rightarrow (g, nxt(pc), l, \sigma, h[i \mapsto \gamma[f \mapsto v]], \phi)$ where $\gamma = h(i)$

$(g, pc, l, v :: \delta^{m,n}_{\tau'} :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `putfield` $f_{\tau} \Rightarrow s'$ where $s' \in init\text{-}loc((g, pc, l, v :: \delta^{m,n}_{\tau'} :: \sigma, h, \phi), \delta)$

$(g, pc, l, m :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `anewarray` $\tau \Rightarrow (g, nxt(pc), l, i :: \sigma, h[i \mapsto new\text{-}arr(symbols(g, pc, l, m :: \sigma, h, \phi), \tau, m, m)], \phi)$ where $i \notin \text{dom}\,h$

$(g, pc, l, \alpha :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `anewarray` $\tau \Rightarrow (g, nxt(pc), l, i :: \sigma, h[i \mapsto new\text{-}arr(symbols(g, pc, l, \alpha :: \sigma, h, \phi), \tau, \alpha, k)], \phi \cup \{\alpha \geq 0\})$ where $i \notin \text{dom}\,h$
$\mid$ NegativeArraySizeException, $(g, nxt(pc), l, \sigma, \phi \cup \{\alpha < 0\})$

$(g, pc, l, \alpha :: i :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `iaload` $\Rightarrow$ ArrayIndexOutOfBoundsException, $(g, pc, l, \sigma, h, \phi \cup \{0 < \alpha \vee \alpha \geq h(i)(\text{LEN})\})$
$\mid (g, nxt(pc), l, \beta :: \sigma, h[i \mapsto \gamma^{m,n-1}[\alpha \mapsto \beta]], \phi \cup \{\iota \neq \alpha \mid \iota \in I\} \cup \{0 \leq \alpha, \alpha < \gamma^{m,n}(\text{LEN}), |I| < \gamma^{m,n}(\text{LEN}), n > 0\})$

where $\gamma^{m,n} = h(i), I = acc\text{-}idx(\gamma^{m,n}), \beta = \begin{cases} \gamma^{m,n}(\text{DEF}) & \text{if } \gamma^{m,n}(\text{DEF})\downarrow \\ new\text{-}prim\text{-}sym(\text{INT}, symbols(g, pc, l, \alpha :: i :: \sigma, h, \phi)) & \text{if } \gamma^{m,n}(\text{DEF})\uparrow \end{cases}$

$\mid_{\iota \in acc\text{-}idx(\gamma)} (g, nxt(pc), l, \gamma(\iota) :: \sigma, h, \phi \cup \{\iota = \alpha\})$ where $\gamma = h(i)$

$(g, pc, l, \alpha :: \text{NULL} :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `iaload` $\Rightarrow$ NullPointerException, $(g, pc, l, \sigma, h, \phi)$

$(g, pc, l, j :: i :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `if_acmpeq` $pc' \Rightarrow (g, nxt(pc), l, \sigma, h, \phi)$ if $i \neq j \mid (g, pc', l, \sigma, h, \phi)$ if $i = j$

$(g, pc, l, v :: \delta^{m,n}_{\tau} :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `if_acmpeq` $pc' \Rightarrow s'$ where $s' \in init\text{-}loc((g, pc, l, v :: \delta^{m,n}_{\tau} :: \sigma, h, \phi), \delta)$

$(g, pc, l, \delta^{m,n}_{\tau} :: v :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `if_acmpeq` $pc' \Rightarrow s'$ where $s' \in init\text{-}loc((g, pc, l, \delta^{m,n}_{\tau} :: v :: \sigma, h, \phi), \delta)$

$(g, pc, l, v :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `assume` $\Rightarrow (g, nxt(pc), l, \sigma, h, \phi \cup \{v\})$

$(g, pc, l, v :: \sigma, h, \phi) \vdash_{\mathcal{S}}$ `assert` $\Rightarrow (g, nxt(pc), l, \sigma, h, \phi \cup \{v\}) \mid \text{Error}, (g, pc, l, \sigma, h, \phi \cup \{\neg v\})$

Note: implicit universal quantifications on free variables; mid-bar (|) indicates a non-deterministic choice, and semi-colon (;) indicates different cases

Figure 3. Bytecode-level Symbolic Execution Operational Semantics (excerpts)

formed in the same manner as typical symbolic execution [13]. However, accesses to symbolic objects (*e.g.*, `getfield`) operate according to the *lazier initialization* algorithm described previously. Similar to [11], we limit the choosing range to exclude concrete objects (and arrays) by introducing an additional field, CONC, which is defined for concrete objects, while undefined for symbolic objects as shown in the *collect* function. This eliminates false alarms in the case where concretely created objects are reachable through lazy initialization as this only happens after destructive updates. For example, consider the following:

```
LinkedNode bar(@NonNull LinkedNode a) {
  LinkedNode n = new LinkedNode(); // default constructor
  return a.next; }
```

The return value of `bar` should not include the object *o* pointed by `n`, because *o* did not exist in the calling context, and there is no assignment in `bar` that causes *o* to be reachable via `a`. Thus, we need to distinguish between concretely created objects from symbolic objects, and only include symbolic objects in lazier initialization.

We use bound n on symbol $\alpha^{m,n}$ that limits the number of distinct array elements that can be lazily initialized; each symbolic array allows lazy initializations up to n number of elements. If an array element is accessed through a symbolic index (*e.g.*, `iaload`): (1) the index maybe out of bounds, (2) the index is equal to one of the accessed indices (from the *acc-idx* function), or (3) n is decremented if the above does not hold, the number of distinct indices accessed so far is less than the length of array, and n is greater than zero. Elements of arrays created by `anewarray` should have default values, but we cannot simply assign the default

values because the array length maybe unknown. Instead, we keep a default value for each array on its DEF field and lazily initialize an accessed index's value with it. This simulates ∀-quantifications over uninitialized array elements.

Another difference between Kiasan with the lazy initialization algorithm presented in [11] is the treatment of types for symbolic objects. Since a symbolic object with a type τ really means the object has a type τ' that is a (reflexive) subtype of τ (*i.e.*, $\tau' <: \tau$), a field with type τ can be initialized with any symbolic object of type τ'. [11] can enumerate all possible subtypes of τ, *i.e.*, its non-deterministic choice includes a fresh symbolic object of type τ', for each $\tau' <: \tau$. We believe this is intractable (*e.g.*, consider when τ is `java.lang.Object`.) For practicality, [11] considers $\tau' = \tau$, which is unsound. To address this issue, we use type variables instead of actual types on symbolic objects and encode the constraints over them in path conditions.

4 Contract-based Symbolic Execution

To reason about open systems (**G3**), we employ contract-based reasoning often used in compositional analysis techniques such as ESC/Java [8]. As indicated in Section 1, when analyzing a method M, we require that M's contract is transformable to an executable form similar to [6]. Intuitively, when analyzing M, Kiasan assumes M's effective `pre` at the method entry and asserts M's effective `post` at method exits. To achieve this using symbolic execution, Kiasan creates a wrapper method for M that: (1) `assumes` M's executable `pre`, (2) calls M and stores M's return value (if any) to a temporary variable x, (3) `asserts` M's executable `post` (that uses x's value in place of the return value.) In essence, executing (1) sets up the symbolic state according to M's `pre` (*e.g.*, they initialize the heap appropriately), and states non-conforming to (1) will be ignored. Executing (3) checks whether the resulting states from (2) satisfy M's `post` (if `post` cannot be ensured from the path condition, then an error is raised). Kiasan can check `post` referring to prestate's values (*i.e.*, values at method entry), and JML's `modifies`, `assignable`, and `\fresh` similar to [20].

Instead of directly executing method calls from M, Kiasan uses contracts in place of the actual implementations of open-ended methods (user-configurable). Intuitively, if M calls an open-ended method N, it checks whether N's `pre` is satisfied; an error is raised if that is not the case. If it is satisfied (or if none is specified), Kiasan uses N's `post` to determine the effects of the method call. To do this, for each open-ended method N called by M, it creates a stub for N, and redirects the corresponding method call to N to call to the stub instead; the stub consists of a sequence of statements that: (1) `asserts` N's `pre`, (2) removes values from modified fields stated in N's contract (hence their values become undefined), if the modified fields do not refer to fresh objects created in N (as specified in N's contract using a similar construct such as JML's `\fresh`); otherwise, fresh symbolic objects are created for such fields, (3) non-deterministically pushes a symbolic reference or NULL in M's stack for non-primitive return type and symbolic value for primitive return type, and (4) `assumes` N's `post`. In essence, executing (2) drops information about fields that are modified by N, and executing (4) initializes them with values which satisfy N's `post`. We can summarize method calls similar to [9]; that is, we cache method results and their corresponding context; if subsequent calls use the same context, we use the cached results.

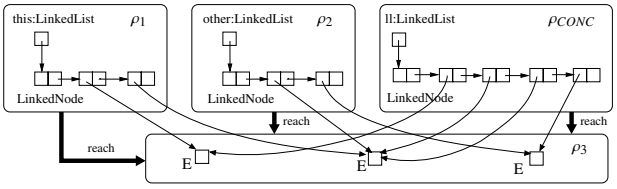


Figure 4. A Region Relation Scenario of `merge`

To close the environment, Kiasan creates a driver for M that starts with a symbolic state where all method parameters and fields that are possibly referenced by M initialized with primitive symbolic values or symbolic references/NULL, according to their types. The analysis proceeds by symbolically executing the driver for M. The subsequent subsections describe two techniques that improve Kiasan's performance as well as enabling us to check strong heap properties in the context of analyzing open systems.

Heap Region Versioning: The last challenge mentioned in Section 2 highlights the need for stronger specification of how objects relate to each other and the possible computations on them. In the example, we need to know that the execution of `merge` does not affect the computation of `compare`; we would like to use this knowledge in the symbolic execution to avoid undesirable error messages that may be generated without such information. One way to do this is to leverage heap region specification. Consider the scenario depicted in Figure 4 where each list object is in its respective region ρ_1 and ρ_2, `this`≠`other`, and the list elements are in a separate region ρ_3. It indicates that objects in ρ_3 cannot reach objects in ρ_1 and ρ_2.

Enhanced with this region specification, the analysis starts with two fresh symbolic references pointed by `this` and `other`, tagged with ρ_1 and ρ_2 as their region descriptors, respectively. Freshly created concrete objects are tagged with a special region ρ_{CONC}, and the lazier initialization of a field f_ρ, that can point to objects from region ρ, can only choose from NULL, a fresh symbolic object or existing symbolic objects in region ρ. This reduces the number of aliasing cases, thus improving the performance of the analysis. Furthermore, we associate a version number to each region that is incremented when any object in the region is updated. This allows us to detect that subsequent method calls such as `compare` whose context is in the same regions and versions, return the same result value. That is, we cache

```
1   public class W {
2     MyInt myInt;
3     void foo(@NonNull W wrapper) {
4       MyInt m = new MyInt();
5       int tmp = wrapper.myInt.f;
6       W w = bar(m);
7       if(w.myInt != m){...}
8     }
9     //@ post: \result.myInt != null;
10    @Fresh W bar(@NonNull MyInt m) {
11      W w = new W(); w.myInt = m; return w;
12    }
13  }
14  class MyInt { int f = 0; }
```

Figure 5. A Context Versioning Example

the method calls, and return previously computed result values if there is no change in any of the objects inside the regions reachable from the context. Therefore, we are able to conclude that `merge`'s `post` holds, because `compare` is pure, and `merge` and `addLast` do not update ρ_3's version.

Context Versioning: Recall that in the symbolic execution presented in Figure 3, we use the CONC field to limit the number of non-deterministic choices. That is, when the lazier initialization algorithm chooses existing objects, it only chooses from heap objects with undefined CONC field. This produces unsoundness in the context of contract-based reasoning described earlier. The example in Figure 5 demonstrates this problem. Suppose we are analyzing the method `foo` where we use `bar`'s specification instead of its implementation. Since `wrapper` is a non-null parameter, the analysis starts with a symbolic object for it. The field access `wrapper.myInt.f` at line 5 will make `wrapper.myInt` initialized and the choosing range does not include the local `MyInt` object created at line 4. This is consistent because `wrapper.myInt` at line 5 can only point to any object from the calling context of `foo`. For the method invocation at line 6, contract-based reasoning creates a symbolic object (because of `@Fresh`) for the return value of `bar` and assigns it to `w`. After the method call at line 6, `w.myInt` points to the object o created at line 4 in real execution. However, according to the algorithm in Figure 3, the choosing range unsoundly excludes o because its CONC field is defined.

To address this, instead of using one bit flag for CONC, we record context version in object by using an integer value for CONC. Each time a method invocation uses the method's specification in place of its implementation, the objects created after the invocation have a higher version number than the objects before that. The generalized algorithm for lazier initialization then chooses from existing heap objects with version number less than or equal to that of the object instead of choosing from existing heap objects with undefined CONC field. For example, the parameter `wrapper` starts with the context version 0, and the version of the object o pointed to by `m` at line 4 is 1. Meanwhile, the returned object from `bar` at line 6 has 2 as its context version. Therefore, the choosing range of `w.myInt` at line 7 includes the object o pointed to by `m` because $1 \leq 2$.

5 Discussion

We have implemented Kiasan using the Bogor framework [19, 26]. The prototype uses a specification processor similar to `jmlc` [6] that translates annotated Java source code and embeds the effective contracts in the code. The resulting code is compiled using a standard Java compiler, which is then translated to Bogor's input language (BIR) extended with Java bytecode instructions modeled as BIR language extensions [26]. Each language extension modeling a Java bytecode is interpreted using the semantics presented in Section 3 and enhanced with the versioning techniques described in Section 4. The Java bytecode to BIR translation virtually generates one atomic transition for each bytecode. During the translation, we close the system as described in Section 4. We use CVC Lite [3] as a decision procedure to determine satisfiability of path conditions.

We have evaluated our prototype on the examples presented in Table 1 as well as some other examples (*e.g.*, examples used to demonstrate ESC/Java.) For all the examples, we use versions that work on primitive Java integer type and object types to demonstrate how Bogor/Kiasan can reason about open systems with and without complete system implementation. The Red-black Tree example is from Java 5's `java.util.TreeMap`; we adapted the primitive integer version from the object type version. The rest of the examples, except for the `merge` example, are from [24].

During our experiments, we found that errors caused by contract violations or runtime exceptions are usually found quickly with small k bound. For example, we tried Bogor/Kiasan on the ESC/Java's `Bag` example and found its errors as quickly as ESC/Java does; on top of that, Bogor/Kiasan produced counter-examples as error evidence. When sorting an array of Comparable objects where all the elements are not guaranteed to be non-null, Bogor/Kiasan signals an exception (as specified in Java 5 API), which are easily found with $k = 2$ in less than one second. Another example is the seeded error on the Red-black Tree example as described in [23], where we also found quickly on both the primitive integer and the object type versions with $k = 2$ in less than two seconds.

To evaluate the performance of Bogor/Kiasan, all examples data in Table 1 are versions without errors. k denotes the resource-bound. For all of the examples, we did not need to use loop bounding. **Pre**, **B-Post**, and **A-Post** denote the number of states at M's entry, before executing M's `post`, and after executing M's post respectively. **States** denotes the number of states executed by Bogor/Kiasan. The **Time** format is in *m*:*s*.*ms*/*m*:*s*.*ms* (rounded, *m*=minutes, *s*=seconds, and *ms*=milliseconds.) The first half shows the overall time required by Bogor/Kiasan (including calls to CVC Lite), and the second half shows the portions of time taken by CVC Lite. The timing data does not include translation time from Java bytecode to BIR. (The translation process

Example	k	Pre	B-Post	A-Post	States	Time
Array	1	1	1	1	181	0:00.6/0:00.2
Binary Heap	2	2	2	2	332	0:01.0/0:00.6
`deleteMin()`	3	3	4	4	628	0:02.3/0:01.7
`(int)`	4	4	7	7	1.1k	0:05.0/0:04.3
Array	1	1	1	1	86	0:00.4/0:00.0
Insertion Sort	2	1	3	3	214	0:00.7/0:00.3
`sort()`	3	1	9	9	760	0:02.1/0:01.6
`(int)`	4	1	33	33	3.4k	0:10.8/0:10.0
Linked-list	1	1	1	1	667	0:00.6/0:00.0
`merge()`	2	4	5	5	3.3k	0:01.0/0:00.0
`(int)`	3	9	19	19	16.1k	0:02.6/0:00.6
	4	16	69	69	78.5k	0:11.2/0:04.9
Binary	1	2	4	4	1.6k	0:00.8/0:00.1
Search Tree	2	5	21	21	12.6k	0:02.7/0:01.1
`insert()`	3	26	236	236	233k	0:55.8/0:39.5
`(int)`	4	-	-	-	-	> 10*min*
Red-black Tree	1	2	5	5	1.4k	0:00.9/0:00.1
`remove()`	2	6	43	43	34.7k	0:07.3/0:04.2
`(int)`	3	31	579	579	1M	5:25.9/4:17.5
	4	-	-	-	-	> 10*min*

Example	k	Pre	B-Post	A-Post	States	Time
Array	1	1	1	1	218	0:00.7/0:00.3
Binary Heap	2	2	2	2	418	0:01.4/0:00.9
`deleteMin()`	3	3	4	4	819	0:04.3/0:03.6
`(Comparable)`	4	4	7	7	1.5k	0:14.5/0:13.6
Array	1	2	2	2	179	0:00.6/0:00.1
Insertion Sort	2	3	4	4	376	0:01.3/0:00.8
`sort()`	3	4	10	10	1.1K	0:06.4/0:05.6
`(Comparable)`	4	5	34	34	4.4k	1:15.1/1:13.9
Linked-list	1	1	1	1	904	0:01.1/0:00.0
`merge()`	2	4	5	5	4.0k	0:01.1/0:00.0
`(Comparator)`	3	9	19	19	18.7k	0:06.2/0:03.9
	4	16	69	69	89.5k	1:06.6/0:58.1
Binary	1	2	4	4	2k	0:00.9/0:00.1
Search Tree	2	5	21	21	15.3k	0:03.1/0:01.1
`insert()`	3	26	236	236	285k	1:09.7/0:49.0
`(Comparator)`	4	-	-	-	-	> 10*min*
Red-black Tree	1	2	5	5	1.9k	0:00.9/0:00.1
`remove()`	2	6	43	43	40k	0:08.3/0:04.7
`(Comparator)`	3	31	579	579	1.3M	6:01.5/4:40.5
	4	-	-	-	-	> 10*min*

Table 1. Experiment Data

can be done transparently and incrementally in a Java Integrated Development Environment.) The experiments were done using a 2.2GHz Opteron workstation using Java 5 32-bit with 64 MB heap size. Space constraints do not permit a thorough discussion, however, we highlight some interesting observations.

Comparable and *Comparator* impose a total order on a set of objects as described in Java 5 API. This is crucial because object ordering relationship is often used in programs manipulating data structures such as the examples in Table 1. The transitivity property of the total order cannot be specified in an executable form because establishing the transitivity property requires the execution history. To facilitate this, we provide a specification pattern for describing transitive closure relation on objects (*e.g.*, via method calls). Empowered with this, Bogor/Kiasan successfully checked the strong properties of `merge` discussed in Section 2.

As can be expected for the array sorting example (on `Comparables`), the number of **A-Post** follows the formula $\Sigma_{i=0}^{k} i!$. This holds because for any array with n elements, there are possibly $n!$ orderings (permutations). In our k-bounded symbolic execution, the state-space for $k = i$ includes cases for $k < i$. In the case for the array integer sorting example, the formula is $\Sigma_{i=1}^{k} i!$ because the case $k = 1$ and $k = 0$ are represented using one symbolic execution path. This happens because in the precondition that we used for the `Comparable` version requires that all elements of the array is non-null, thus, the precondition expands the array elements up to the bound. In the integer case, we do not need to require this, thus, both arrays with zero and one element are not expanded into different execution paths. This also causes their **Pre** numbers to be different.

Lazier initialization significantly reduces Kiasan's performance, for example, Kiasan checked the `merge` method for the strong properties described earlier in about thirty seconds with only lazy initialization for $k = 3$. Lazier initialization reduces this to about three seconds by avoiding non-deterministic choices when calling `compare` with the data of contained in the linked-lists.

From most of the experiment data, we can observe that CVC Lite consumes around half of the running time or more. We plan to have a tighter integration with CVC Lite (as library instead of as a separate process communicating using character strings.)

One weakness that Kiasan shares with all state-enumeration techniques (*e.g.*, Korat [5], Alloy [10]) is that it potentially enumerates too many states in order to check strong specifications precisely (*e.g.*, properties requiring very precise aliasing information.) Note that methods using weakest precondition calculation have a similar problem because in order to be precise, it also has to take possible aliases between objects and variables into account when computing statement weakest precondition. To address this, we plan to investigate using sampling techniques or to develop additional bounding techniques. To improve Kiasan's general performance, we can also adopt complementary techniques such as [12].

6 Related Work

This work is based on [11]; we adapted it to reason about open systems and introduced a quantifiable bounding technique that provides fine-grained controls over heap explorations. In addition, we developed techniques to enhance its performance and alleviate its sources of unsoundess or intractability. Another main difference is the treatment of method preconditions; [11] does not symbolically execute method preconditions early on, instead they maintain necessary mappings to lazily use preconditions during the execution of the method. *For each lazy initialization*, it reconstructs structures to their states before destructive updates, and executes method preconditions; in our case, we only execute the method preconditions once, and we do not require reconstruction of structures. Another work close to ours is XRT [9]. To guarantee termination, [9] bound the maximum

number of transitions per path. Kiasan prefers bounding the length of lazier initialization chains first and only uses loop bounding as the last resort. This provides us with better control (methodologically) on how we increase the coverage on paths that exhaust the k-bound without exhausting the loop bound. Kiasan uses a different bounding technique than the ones used in Alloy [10], TestEra [17], and Korat [5] as described in Section 3. Another difference is that we do not mandate specifications to check methods; Kiasan can still symbolically execute methods without their specifications and check for errors. In contrast to all the work above and [25, 22], Kiasan is designed to check strong heap properties of open systems, where some system parts are unavailable or open ended in addition to unknown input values.

The closest work to ours that address checking properties of open systems is [8, 7, 2]. They use weakest precondition as the basis of its all-paths compositional analysis engine while we depend on a path-sensitive analysis using symbolic execution. Similar to [7], Kiasan's basic symbolic execution is relatively sound and complete. One limitation of Kiasan is that it requires a specification formalism that is transformable to executable form, similar to JML runtime monitoring [6]. An advantage of using a path-sensitive analysis is that it naturally produces a counter-example when an error is found. We believe it is more intuitive and potentially easier to produce an explanation to describe the error. In addition, we only use quantifier-free first-order logic (FOL) in our path condition that is better handled by theorem provers. While this potentially limits the class of specifications that our current approach handles, some quantifications on objects can be unfolded up to the resource-bound of the analysis similar to Alloy [10]. Moreover, we can introduce some specification patterns such as the transitive closure relation on objects to enrich our analysis without fully adopting FOL. This allows us to have tighter controls over the source of incompleteness of our tool (*e.g.*, incompleteness of theorem proving on FOL.) Another difference is Kiasan can check stronger heap properties similar to [20], and it allows a mixed compositional/non-compositional reasoning that addresses one of the usability problems when using pure compositional reasoning tools.

Most work on contract-based software verification manipulates logical formulae. Smallfoot [4] is a symbolic execution tool in which execution means updating (formulae including spatial formulae) rather than operational state transition. TVLA [16] expresses a program and its annotation using FOL formulae with transitive closure and performs data-flow analysis interpreting formulae under three-valued logic structure. Space constraints do not permit in-depth discussion, thus, we only offer a high-level comparison. We trade-off complete soundness (unbounded heap) for fully automatic analysis, while the above approaches either use some fixed algorithms to handle a limited number of heap structures (*i.e.*, limited specification expressiveness) or require some level of user intervention to help the analysis. In the end, we do not limit ourselves to the approach presented here. That is, we can incorporate richer logics and abstractions in the same way we can enhance concrete execution with symbolic execution as they become more mature and automated analysis is possible.

7 Conclusion and Future Work

We have presented Kiasan, an alternative technique to reason about open systems based on symbolic execution that is able to check strong heap properties. We have implemented Kiasan on top of the Bogor framework. Methodologically, we envision our tool being used similar to frameworks like ESC/Java [8]. For example, a user can start checking a method (even compositionally) without annotation and receive error feedback from the tool. One can then inspect the feedback or look at the generated counter-examples to determine whether the errors are really errors or because of a lack of specification. The user can either fix the code or add/modify the specifications and repeat the process. We believe using a small k for interactive programming and checking mode is acceptable, while using larger k increasingly can be employed using, for example, continuous testing [21] or a distributed solution. We now highlight several future research directions to improve Bogor/Kiasan.

1. Currently, Kiasan conjoins class invariants and method preconditions to generate method prestates. Using a two-staged approach, where we first generate states using class invariants, and then later use preconditions when we want to actually check some methods, is better. It allows one to generate the states using class invariants only once to analyze all the methods in the class that should satisfy the invariants.

2. Our prototype uses a rudimentary specification processor that hinders us to conduct systematic case studies. To address this, we plan to collaborate with the JML-oriented Integrated Verification Environment (IVE) effort [14]. By targeting the same specification language, we can reuse a significant amount of previous work, *e.g.*, the Java library models. Moreover, [14] plans to support multiple theorem provers using SMT-Lib [27]. This integration would allow us to cross-pollinate ideas and to conduct systematic case studies comparing ESC/Java2 and Bogor/Kiasan.

3. Section 4 describes how we can leverage heap region and transitivity information for checking strong heap properties. While these reasoning patterns can be used to check, for example, the `merge` method, however, we believe we need various reasoning patterns for different kinds of strong properties (*e.g.*, monotonicity). To address this, we plan to incorporate other common reasoning patterns in Kiasan.

4. Kiasan's stateless analysis is *embarrassingly parallel*; we can fork the analysis when exploring different paths. This can help curbing the analysis time considering processor developments are moving toward multicore architecture.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. *SPIN Workshop on Model Checking of Software*, 2006.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[3] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. *Computer-Aided Verification*, pages 515–518, 2004.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. *Asian Symposium on Programming Languages and Systems*, 2005.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. *International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[6] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java modeling language. *Software Engineering Research and Practice*, 2002.

[7] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/-Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *Programming Language Design and Implementation*, 2002.

[9] W. Grieskamp, N. Tillmann, and W. Schulte. XRT - exploring runtime for .NET - architecture and applications. *Workshop on Software Model Checking*, 2005.

[10] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256 – 290, 2002.

[11] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for Construction and Analysis of Systems*, pages 553–568, 2003.

[12] S. Khurshid and Y. L. Suen. Generalizing symbolic execution to library classes. In *Workshop on Program Analysis for Software Tools and Engineering*, 2005.

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

[14] J. R. Kiniry, P. Chalin, and C. Hurlin. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *Verified Software: Theories, Tools, Experiments*, 2005.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java*, 1998.

[16] T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene-based static analysis. In *International Static Analysis Symposium*, 2000.

[17] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. *Automated Software Engineering*, 2001.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[19] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *9th European Software Engineering Conference/11th Foundations of Software Engineering*, 2003.

[20] Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking JML specifications using an extensible software model checking framework. *International Journal of Software Tools for Technology Transfer*, 2006.

[21] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. *International Symposium on Software Reliability Engineering*, 2003.

[22] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[23] M. Vaziri-Farahani. *Finding Bugs in Software with a Constraint Solver*. PhD thesis, MIT, 2004.

[24] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1998.

[25] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[26] Bogor website. `http://bogor.projects.cis.ksu.edu`.

[27] SMT-LIB: The satisfiability modulo theories library. `http://goedel.cs.uiowa.edu/smtlib`.