Generalized Abstract Symbolic Summaries*

Suzette Person
University of Nebraska-Lincoln
Lincoln, NE
sperson@cse.unl.edu

Matthew B. Dwyer
University of Nebraska-Lincoln
Lincoln, NE
dwyer@cse.unl.edu

Abstract

Current techniques for validating and verifying program changes often consider the entire program, even for small changes, leading to enormous V&V costs over a program's lifetime. This is due, in large part, to the use of syntactic program differencing techniques which are necessarily imprecise. Building on recent advances in symbolic execution of heap manipulating programs, in this paper, we develop techniques for performing abstract semantic differencing of program behaviors that offer the potential for improved precision.

1 Introduction

Symbolic analysis techniques have been studied extensively since they allow a potentially infinite equivalence class of system behaviors to be reasoned about as a single unit. For example, symbolic simulation has been applied to reason about the equivalence between hardware designs. In this setting, researchers have identified the utility of *uninterpreted functions* as a means of coarsely abstracting common portions of hardware circuits in such a way that equivalence checking is preserved [2]. Uninterpreted functions achieve this by sacrificing the precise encoding of a function's meaning while preserving functional congruence, i.e., $x = y \implies f(x) = f(y)$.

Symbolic analysis for software has a long and rich history. Symbolic execution [9], which was developed in the early 1970s, records the program state as symbolic expressions over free variables that represent program input values, accumulates constraints on those input values that define the conditions under which a particular program path is executed, and produces expressions that characterize output values produced along the path. Symbolic execution has seen a recent resurgence of interest for test input generation [4, 7, 15]. While much of the existing work has focused on symbolic execution of scalar data, researchers have generalized symbolic execution to treat heap manipulating programs. In our work, we build on a technique called *lazy initialization* that materializes just the portion of heap that is referenced along the symbolically executed path [8].

Recent work has investigated the application of uninterpreted functions as an abstraction mechanism in the context of symbolic execution. Currie et al. [3] propose the use of uninterpreted functions to abstract common sequences of assembly instructions to make equivalence checking tractable. Siegel et al. [13, 14] manually map portions of sequential and parallel implementations onto uninterpreted functions to permit equivalence checking as a means of avoiding subtle errors in parallelization. In our recent work [11], we define a framework called *differential symbolic execution* (DSE) that compares the results of symbolic executions of two program versions to perform equivalence checking and *precise program differencing*. DSE uses uninterpreted functions to calculate *abstract summaries* of Java program blocks that have not changed between program versions. While some program changes are intended to retain program functionality, for example, performance optimizations, in many cases, program changes are meant to change the program's meaning, for example, when a bug is fixed. In such cases, one would like to be able to identify the equivalence class of program behaviors for which the programs differ. It is only these behaviors that must be subjected to rigorous testing, or in the case of critical systems, to

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 46-55

*This material is based in part upon work supported by the National Aeronautics and Space Administration under grant number NNX08AV20A and by the National Science Foundation under awards CCF-0541263 and CNS-0720654. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NASA or NSF.

verification or manual inspection. Our goal in this paper is to extend the approach in [11] so that it is applicable to heap manipulating programs.

Generalizing abstract summaries to support dynamically allocated data used in modern programming languages, such as Java, presents numerous challenges. For hardware circuits [2] and assembly language programs [3] defining uninterpreted functions for portions of a hardware design or program is straightforward–the input lines or registers that are read define the function's domain and the output lines or registers written define the function's codomain. In more expressive programming languages, the *types* of values that are read or written and the *location* at which those values are referenced may be much more difficult to determine. While manual inspection and reasoning about the program can make such determinations [14], methods that are both broadly applicable and automated are needed.

Our work builds on generalized symbolic execution [8] which extends symbolic execution to heap manipulating programs. The key insight of their approach is that *on a given program path only the heap cells that are accessed can influence the computational effects*. This permits the analysis to create and initialize only the portion of the heap that is referenced along a path–this is called *lazy initialization*. When abstracting common program blocks, however, DSE intentionally *omits* precise tracking of the heap cells accessed within the block. In this paper, we integrate lazy initialization and abstract summaries to safely approximate the computational effects. The specific contributions of this paper involve (1) safely abstracting the computational-effects of common blocks using an adaptation of side-effects analysis on heap data; (2) instantiation of abstract summaries based on the state of the symbolic execution upon entry to each common block; (3) forcing the lazy *re-initialization* of reference fields that may have been written by the abstracted code block; and (4) adapting lazy initialization of heap fields to account for the situation where a common block may assign heap locations that were not present in the symbolic state on entry to the block.

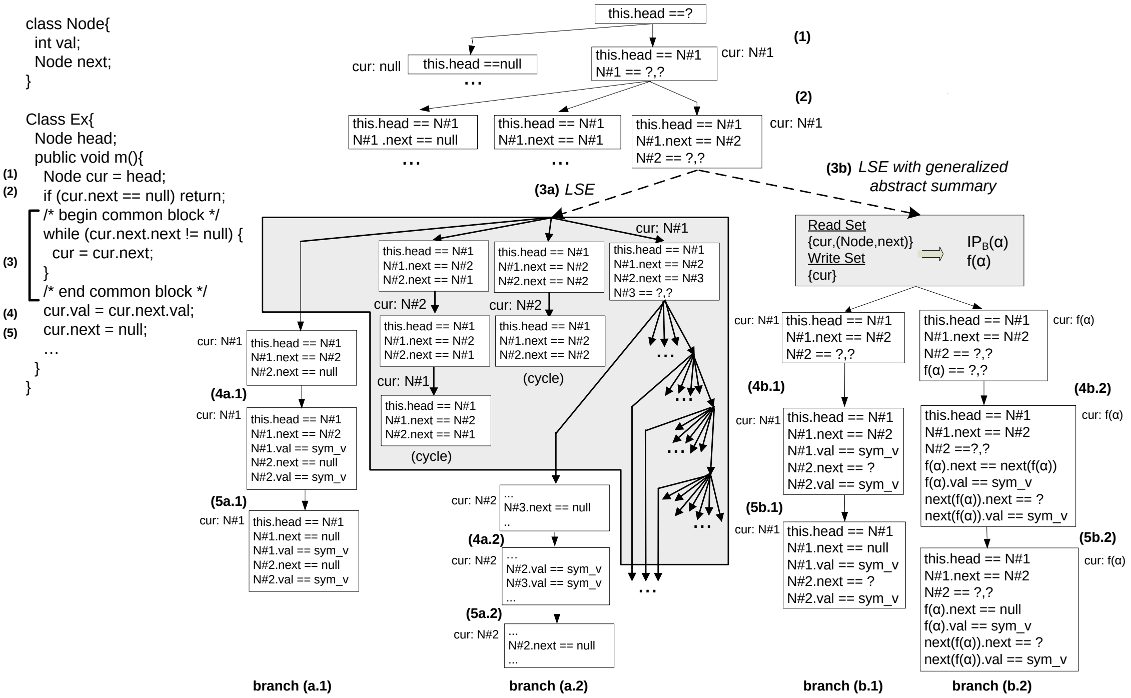
In the next section, we illustrate our approach to computing an abstract summary for a heap-manipulating program by way of example. Section 3 defines the symbolic execution and summarization concepts we build on. Section 4 explains how our technique differs from previous work and details the components of the approach that address the challenges presented by heap-manipulating programs. We conclude with a discussion of the implementation of these techniques in JavaPathfinder [12], our plans for evaluating the cost-effectiveness of the technique, and future work in Section 5.

2 Overview

Imagine having just completed V&V of a large complex system when a fault is detected or a performance optimization is identified. Unless great care is taken to isolate system components from one another, calculating the potential impact of a change (e.g., using data and control dependence analyses) is likely to identify that the majority of system components may behave differently as a result of the change. This, in turn, could lead to significant re-V&V activities even if the changes are actually behavior preserving or only affect a very narrow range of behaviors.

Differential symbolic execution addresses this problem by performing a detailed semantics-based analysis of the behaviors of two program versions. The results of the analysis are rendered as sets of logical formula each of which describes a distinct portion of the program's input space and the corresponding effects of program computation on all inputs in that space. DSE then systematically compares the logical formulae from one version with the formulae from the other. Equivalent pairs of formulae are eliminated since they indicate common behavior across program versions. The remaining formulae are then used to precisely characterize “when”, i.e., for what portion of the program input space, and ”how”, i.e., the computational effects, the program versions differ.

Our technique is an adaptation of lazy symbolic execution (LSE) [8], where LSE is performed on the



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

Class Ex{
  Node head;
  public void m(){
(1)  Node cur = head;
(2)  if (cur.next == null) return;
     /* begin common block */
(3)  while (cur.next.next != null) {
       cur = cur.next;
     }
     /* end common block */
(4)  cur.val = cur.next.val;
(5)  cur.next = null;
     …
  }
}
```

Figure 1: Example with LSE and Abstract Summary-based Symbolic Execution (excerpts)

changed sections of code, and *abstract summaries* are used in place of performing detailed LSE on the common code blocks. Rather than describe the entire DSE method in detail, here we illustrate, by way of a small example, how abstract summaries in DSE can be used to exploit common code blocks, and compare the results of DSE with lazy symbolic execution of the entire method (on both the changed and common code blocks).

2.1 An Example

The left side of Figure 1 shows the source code for classes *Node* and *Ex*. We consider the method $Ex.m()$ in detail and assume that the `while` loop is shared with another version of the method; our technique considers this a *common block*.

The right side of Figure 1 depicts portions of a symbolic execution tree where a node encodes a symbolic representation of the program state and edges correspond to execution of a statement in $Ex.m()$. Figure 1 depicts two trees that share a common top portion. The lower-left side of the tree, beginning with branch (3a), illustrates the results of continuing LSE through the common code block. The lower-right side of the tree, beginning with branch (3b), illustrates how our approach abstracts the common code block and how the abstract summary is used during symbolic execution of the subsequent block of changed code. The shaded areas of the tree highlight the different treatment of the while loop under LSE and using our technique.

Lazy Symbolic Execution operates by maintaining an explicit heap that stores just the objects that have been referenced along a program execution path. Local variables and fields of scalar types are symbolically executed using traditional methods, but the treatment of heap data is closer to the concrete program semantics. LSE accounts for different program behaviors by considering multiple possible values when initializing reference fields at the point when they are first accessed; the term *lazy initialization* refers to the process of waiting to initialize a field until it is first referenced. LSE generates successor states in which the field has a `null` value, points to each type-compatible object that is currently in the heap, and points to a newly allocated object; we refer to this multiplicity of successor states as *branching* in the symbolic execution. LSE systematically explores a program component's behavior in this way until the method returns or some prescribed resource bound is reached.

In Figure 1, LSE of $Ex.m()$ begins with statements (1) and (2). At (1), the uninitialized field $this.head$ is read and assigned to the local variable *cur*. The field reference $this.head$ requires lazy initializion of the field $head$. In this case, because there are no allocated objects of type $Node$, LSE explores just two possibilities: either the field is `null`, or it is set to a new symbolic object of type $Node$. At statement (2), the *next* field of the object referenced by *cur* is read for the first time, so LSE creates a new $Node$ object on the symbolic heap (we detail only this path; other paths include assigning a value of `null`, or the previously created $Node$ object). On this path the new object is not equal to `null`, so symbolic execution continues into the abstracted code block at (3).

At the beginning of the common code block under LSE (3a), a four-way branch is created when *cur.next*$.next$ is read since the fields of *cur.next* have not yet been initialized and there are two existing objects of type $Node$. Execution of statements in the common block are depicted in the shaded area where the value of *cur.next*$.next$ is compared with `null` and *cur* is updated to *cur.next*. As shown in Figure 1, as LSE initializes new $Node$ objects in the $while$ loop, the tree will grow in depth (with new nodes for each loop iteration) and breadth (since iterations may initialize new objects on which LSE will branch). In practice, this expansion will be bounded by fixing the number of loop iterations considered or the maximum depth of the the tree in which case LSE under-approximates the method's behavior. On symbolic execution paths where the loop condition evaluates to false, LSE will continue by updating *cur.val* and *cur.next* at statements (4) and (5).

Abstract Summaries are incorporated into LSE when a common code block is reached as shown in branch (3b). At this point, LSE is suspended and a pre-computed summary of the effects of the block is used to update the symbolic state. Summaries are computed by approximating the set of memory locations that may be read and written by the code block. The shaded area on branch (3b) shows that *cur* may be written and read, and the field *next* of objects of type $Node$ may be read during execution of the block. The read and write sets and the current symbolic state are used to generate and instantiate two uninterpreted functions. $IP_B(\alpha)$ encodes the path condition through the block. $f(\alpha)$ represents the unknown value calculated in the block that is assigned to *cur*, where α parameterizes f with the values of the read set locations on entry to the block.

The symbolic state is updated by applying the functions to appropriate locations. Since *cur* may be written by the block, two successor states are created: one where *cur* is *not* updated (the leftmost branch), and one where *cur* is updated to its new, but unknown value, $f(\alpha)$ (the rightmost branch). Note that since $f(\alpha)$ is a reference value, the symbolic state is extended to hold a new object of type $Node$ whose fields have unknown values.

Considering just the rightmost branch, LSE resumes at (4b.2). Our approach treats object references computed within the block differently; for example, $f(\alpha)$ encodes an address and we use function composition to encode chains of dereferences i.e., $next(f(\alpha))$. At statement (4) *cur.next*$.val$ is read. Since *cur* is bound to a reference value from the abstracted block, whose address is $f(\alpha)$, the reference to its *next* field causes lazy initialization branching. The branch we illustrate creates a new $Node$ object, whose address is $next(f(\alpha))$, and sets the *cur* object's *next* field to that address (4b.2). Dereference of the non-reference field *val* of $next(f(\alpha))$ produces a fresh symbolic value, sym_v, which is then assigned to *cur*'s *val* field, i.e., $f(\alpha).val$. Finally, in statement (5) the field *cur.next* is accessed again, this time to set its value to `null` (5b.2).

2.2 Assessment

Using overapproximating *abstract symbolic summaries*, to represent common code blocks has advantages and disadvantages. There are program structures, such as loops and recursion, that are problematic for symbolic execution. LSE cannot process the common block in our example because there is no information about the length of the *Node* chain. Using abstract summaries avoids this problem, since the block is never executed in detail. Abstract summaries are a safe approximation because all of the behaviors present under LSE are also captured in the summary; branch (b.1) of our approach over-approximates branch (a.1) under LSE, and branch (b.2) of our approach accounts for the all of the remaining branches explored under LSE. Our approach may defer initialization of heap objects as shown on branches (b.1) and (b.2), however, it may also introduce infeasible paths into the analysis. While this is problematic for some applications of symbolic execution, such as test generation, for precise differencing our focus is on identifying only the behaviors on which two versions differ. For most program blocks, we can be assured of deterministic behavior, i.e., the block will perform the same computation when executed from the same state, thus even if abstract summaries introduce infeasible behaviors, many of those behaviors will be equivalent in both versions of the program.

3 Background on Symbolic Execution

Symbolic execution involves the non-standard interpretation of a program using *symbolic values* to represent input data values, and expressions over symbolic values to represent the values of program variables. The state of a symbolic execution is defined as a triple, (pp, pc, v). *pp*, the current *program point*, records the next statement to be executed. *pc*, the *path condition*, is the conjunction of branch conditions encoded as constraints along the current execution path. *v*, the *symbolic value store*, is a map from memory locations to symbolic expressions representing their current values; $v[j]$ is the symbolic value at location j. A store update $v\langle l, s\rangle$ defines a new store v' with the value s at location l.

Computation statements, $x = y$ op z, when executed symbolically in state (pp, pc, v) produce a new state $(pp+1, pc, v\langle x, op(v[y], v[z])\rangle)$ that reflects the update of the value store with a new symbolic value for x that applies op to its operand values. Branching statements, `if` x op y `goto` l, when executed symbolically in state (pp, pc, v) branch the symbolic execution to *two* new states $(l, pc \wedge v[x]$ op $v[y], v)$ and $(pp+1, pc \wedge \neg(v[x]\, op\, v[y]), v)$ corresponding to the “true” and “false” evaluation of the branch condition, respectively.

3.1 Generalized Symbolic Execution

In addition to the symbolic state stored for program variables, e.g., method locals, with generalized symbolic execution a portion of the symbolic state encodes a set of partially-defined heap objects. Lazy initialization is a technique for defining the values of only those object fields that have been referenced during symbolic execution along a path. It operates on statements with dereference expressions, e.g., $x = r.f$, and can update the state in five different ways as shown in Figure 2; where type() denotes the type of the location and Ref the set of reference types.

If the referenced field’s location is in the symbolic store, i.e., $r.f \in \text{dom}(v)$, the successor state is given by (1), where the symbolic value store for x is updated with the value at location $r.f$. If the field’s location is not in the symbolic store and the field is a non-reference type i.e., primitive or scalar type, the successor state is given by (2), where *new* generates a new symbolic variable which is used to update $r.f$ in the symbolic value store ($v\langle r.f, \text{new}\rangle$), and the value at $r.f$ is subsequently used to update the symbolic

$$\begin{array}{rlr}
(pp+1, pc, v\langle x, v[r.f]\rangle) & \text{if } r.f \in \text{dom}(v) & (1) \\
(pp+1, pc, v\langle r.f, \text{new}\rangle\langle x, v[r.f]\rangle) & \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \notin \text{Ref} & (2) \\
(pp+1, pc, v\langle r.f, null\rangle\langle x, v[r.f]\rangle) & \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \in \text{Ref} & (3) \\
(pp+1, pc, v\langle \text{new}, \perp\rangle\langle r.f, \text{new}\rangle\langle x, v[r.f]\rangle) & \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \in \text{Ref} & (4) \\
(pp+1, pc, v\langle r.f, o\rangle\langle x, v[r.f]\rangle) & \text{if } r.f \notin \text{dom}(v) \wedge \text{type}(f) \in \text{Ref} \wedge & \\
 & o \in \text{dom}(v) \wedge \text{type}(v[o]) = \text{type}(f) & (5)
\end{array}$$

Figure 2: Lazy initialization symbolic state updates for $x = r.f$ from (pp, pc, v)

value store for x ($\langle x, v[r.f]\rangle$). If the field is of reference type and it is not in the symbolic store, there are multiple possible successor states and the symbolic execution should explore each state. The field may have a `null` value yielding the successor state given by (3), or the field may store a reference to a new, uninitialized ($\perp$) object yielding the successor state given by (4), where the new object is added to the symbolic store, $r.f$ is updated to reference the new object, and x is updated with the value at $r.f$, and finally, the field may store a reference to a type-compatible object in the current state yielding a successor state given by (5) for each such object.

3.2 Symbolic Summaries

Recent work has proposed compositional symbolic execution to improve the scalability of inter-procedural symbolic execution [7, 1]. This approach works by symbolically executing a method and accumulating information for each path into a summary.

Definition 3.1 (Symbolic Summary [11])**.** *A symbolic summary, for a method m, is a set of pairs* $m_{sum} = \{(i_1, e_1), (i_2, e_2), \ldots, (i_k, e_k)\}$ *where* $\forall 1 \leq j \leq k \forall 1 \leq j' \leq k \wedge j \neq j' : i_j \wedge i_{j'}$ *is unsatisfiable.*

Each constraint i_j, called an input constraint, encodes a path condition. An effect constraint, $e_j = \bigwedge l \in Write(m, i_j) : l = v[l]$, encodes the symbolic state on exit from the method where the effect constraint is a conjunction of equality constraints for locations that are written by the method. $Write(m, \phi)$ denotes the locations written to by a method, m, when called on arguments that satisfy ϕ, and $Write(m) = Write(m, true)$.

Method call, $m(v)$, is executed symbolically in state (pp, pc, v) with summary m_{sum} as follows. For each pair $(i, e) \in m_{sum}$, let $pc' = pc \wedge i \wedge (\bigwedge j \in |v| : a[j] == f[j])$ which conjoins the current path condition with the input partition and with a set of equality constraints that equate symbolic variables for the formal parameters with the symbolic expressions for the actuals. If pc' is satisfiable, the execution continues with on a branch with successor state $(pp+1, pc', v\langle e\rangle)$.

4 Generalized Abstract Symbolic Summaries

In [11] we introduced the concept of an *abstract summary*, m_{abs}, to safely approximate common program behaviors without performing a precise symbolic execution. Our previous work supports symbolic execution of non-reference types and summarization of program methods. In this section, we detail abstract summaries for generalized symbolic execution of heap-manipulating programs and to summarize arbitrary program blocks. We begin by recalling the definition of abstract summary:

Definition 4.1 (Abstract Summary [11])**.** *An abstract summary for a method* $m(\vec{f})$ *with formal parameters* $\vec{f}$ *is a pair,* $m_{abs} = (i(\vec{f}), \bigwedge l \in Write(m) : l == e_l(\vec{f}))$ *where* $i : \vec{f} \rightarrow \{true, false\}$*, and* $e_l : \vec{f} \rightarrow$ type(l) *are uninterpreted functions defined over vectors of formal parameter values.*

An abstract summary is built up from a set of uninterpreted functions: i abstracts the path condition within the method and, for each written location, a function e_l which abstracts the value that the method may write to location l. Symbolically executing a call to method $m(\vec{a})$ in state (pp, pc, v) with m_{abs} involves instantiating the uninterpreted functions with the current symbolic state and effecting the updates to potentially written locations. The resulting state is $(pp+1, pc \wedge i(\vec{a}), v\langle \bigwedge l \in Write(m) : l = e_l(\vec{a})\rangle)$ where binding of formals to actuals is achieved by parameterizing the uninterpreted functions, and the state is updated to be consistent with the summary's effects.

We note that using a *may* analysis to calculate $Write(m)$ leaves open the possibility that an $l \in Write(m)$ may not actually be written by a call to m. Modeling all combinations of possible writes would result in $2^{|Write(m)|}$ possible successor states after instantiation of m_{abs}. Our approach mitigates this for non-reference types by using e_l which represents any possible value of the type of l.

There are three basic elements of this approach: (1) calculating the written (read) locations; (2) defining the set of uninterpreted functions; and (3) instantiating constraints involving those functions at the point of a method call during symbolic execution. Definition 4.1 captures how (1) and (2) are performed when summarizing methods that read and write non-reference types. As we explain in the remainder of this section, for heap manipulating programs element (1) must be generalized significantly, elements (2) and (3) must be performed together since the definition of uninterpreted functions is dependent on the symbolic state at the beginning of the summarized code block, and element (3) must be carried forward during symbolic execution to safely approximate lazy initialization of fields that may be written by an abstracted block.

4.1 Equality Preserving Heap Abstraction

Current approaches to symbolic execution of heap manipulating programs [4, 8, 12], use an explicit heap store where object references maintain their usual semantics. Our approach extends the explicit store to allow object references to be encoded as functions that encode dereference chains or access-paths through the heap; this is refered to as a storeless model of the heap [5].

When overapproximating abstract summaries are used, a symbolic execution may traverse a prefix of an access path, α, then enter an abstracted block where that access path is extended, with β, and then after exiting the block, further dereferencing along the access path may be performed, resulting in an overall access path of $\alpha\beta\gamma$. The explicit store used by LSE cannot faithfully represent an access path where the details of β are not known; it is insufficient to consider only the objects referenced along α since different objects may be referenced within the block.

We make the following observations: (a) in a given program state, an access path, e.g., α, evaluates to a single object, e.g., following α leads to o; and (b) an access path can be represented as the composition of field specific dereference functions applied to an initial object reference. For example, if $\beta = next.next$ where *next* is a field of class $Node$, then the function $Node_{next}(Node_{next}(o))$ would encode the object at the end of chain $\alpha\beta$.

When an abstracted block, b, executes in a given symbolic state, v, and traverses an access path to write to a field, f, of some object type, t, that object field can be uniquely defined by a function $t_f(o)$ where $o \in v$ is the root of the access path. This function stands for an unknown access path within the block. It is sufficient to have t_f be uninterpreted to judge equivalence between object references that are an extension of its access path, e.g., $t_f(o)\gamma$.

4.2 Computing Write (Read) Information

To produce a safely approximating abstract summary of a code block we over-estimate the set of written and read locations in the block. We use a combination of flow-insensitive field-sensitive type-based alias analyses [6] and side-effects analyses [10]. Every side-effects analysis defines a *naming scheme* for heap locations. Ours calculates a set of pairs, $(type, field)$, where a pair indicates a write (read) of $field$ of an object of $type$. Our analysis is initiated, on demand, to target just the common blocks that are encountered during symbolic execution.

The code block, b, is scanned to detect *local* reads and writes (e.g., in JVM bytecodes these include instructions such as `iload` and `istore`) and *heap* reads and writes (e.g., `getfield` and `putfield` bytecodes). The set of locations (e.g., local offsets) subject to local writes (reads) are recorded as $Write_l(b)$ ($Read_l(b)$). The set of $(type, field)$ pairs naming the heap locations that are written (read) in b are recorded as $Write_h(b)$ ($Read_h(b)$).

When a method invocation for $m()$ is encountered during the scan of b, the pre-computed $Write_h(m)$ and $Read_h(m)$ are unioned with the corresponding sets for the caller. If the pre-computed sets for $m()$ are not available, we scan $m()$'s body (and recursively any method it calls) to compute $Write_h(m)$ and $Read_h(m)$, which are cached, and unioned with the caller's sets.

4.3 Realizing Generalized Abstract Summaries

Realizing an abstract summary involves: (1) defining and instantiating the uninterpreted functions, (2) updating the current path condition and symbolic state, and (3) setting the program counter to point to the first instruction following the abstracted code block. When the symbolic program state on entry to a block includes heap locations, Definition 4.1 is inadequate because the abstracted block may read heap locations by dereferencing through parameters. All read locations must be included in the signature of the uninterpreted functions used in the abstract summary–leaving a location out risks judging two instances of a block as having equivalent effects when their behavior may be distinguished by the value at the missing location. This observation forces us to delay the generation of uninterpreted functions until the symbolic state on entry to the block has been computed.

We define sets of written and read locations within a symbolic state as follows:

Definition 4.2 (State-specific Locations)**.** *Given a symbolic state, v, and a code block, b, let*

$$\begin{aligned}
ReadIn(b,v) &= Read_l(b) \cup \{r.f | r.f \in \text{dom}(v) \wedge (\text{type}(r), f) \in Read_h(b)\} \\
WriteIn(b,v) &= \{l | l \in Write_l(b) \wedge \text{type}(l) \notin \text{Ref}\} \cup \\
&\quad\ \{r.f | r.f \in \text{dom}(v) \wedge (\text{type}(r), f) \in Write_h(b) \wedge \text{type}(f) \notin \text{Ref}\} \\
WriteIn_h(b,v) &= \{r.f | r.f \in \text{dom}(v) \wedge (\text{type}(r), f) \in Write_h(b) \wedge \text{type}(f) \in \text{Ref}\} \\
WriteIn_l(b,v) &= \{l | l \in Write_l(b) \wedge \text{type}(l) \in \text{Ref}\}
\end{aligned}$$

be the set of locations read in b, locations written in b of non-reference type, heap locations and locals written in b of reference type, respectively.

The need to make summaries state-specific forces us to extend the notion of an abstract summary from Definition 4.1. In particular, all definitions of uninterpreted functions and constraints on updated locations are deferred to the point where the abstract block is entered, and to subsequent uninitialized field references. Consequently, we modify the definition to simply capture the state-specific location sets that permit that deferred processing.

Definition 4.3 (Generalized Abstract Summary)**.** *An abstract summary for the execution of code block, b, in symbolic state v is the quadruple* $b_{gabs} = (ReadIn(b,v), WriteIn(b,v), WriteIn_h(b,v), WriteIn_l(b,v))$.

Let b be a code block with a single-entry, pp_{entry}, and single-exit, pp_{exit}, point. Instantiation of the generalized abstract summary, b_{gabs}, in state (pp_{entry}, pc, v) results in successor states of the form:

$$\begin{aligned}
(pp+1, pc, v\langle r.f, e_{r.f}(top(w).2)\rangle\langle x, v[r.f]\rangle, w) \qquad & \text{if } r.f \notin \mathrm{dom}(v) \wedge \mathrm{type}(f) \notin \mathrm{Ref} \\
& \wedge(\mathrm{type}(r), f) \in \bigcup w \qquad (6) \\
(pp+1, pc, v\langle e_{r.f}(top(w).2), \perp\rangle\langle r.f, e_{r.f}(top(w).2)\rangle\langle x, v[r.f]\rangle, w) \qquad & \text{if } r.f \notin \mathrm{dom}(v) \wedge \mathrm{type}(f) \in \mathrm{Ref} \\
& \wedge(\mathrm{type}(r), f) \in \bigcup p \in w : p.1 (7)
\end{aligned}$$

Figure 3: Lazy initialization updates for $x = r.f$ from (pp, pc, v, w)

$$\begin{aligned}
(pp_{exit}, pc \wedge i(\mathit{ReadIn}(b,v)), v\ & \langle \textstyle\bigwedge l \in \mathit{WriteIn}(b,v) : l = e_l(\mathit{ReadIn}(b,v))\rangle \\
& \langle \textstyle\bigwedge l \in L_{\mathrm{Ref}} : l = e_l(\mathit{ReadIn}(b,v))\rangle - \mathit{WriteIn}_h(b,v))
\end{aligned}$$

where $L_{\mathrm{Ref}} \subseteq \mathit{WriteIn}_l(b,v)$, and i and e_l are uninterpreted functions with domains corresponding to the types of locations in $\mathit{ReadIn}(b,v)$ and with boolean and type(l) codomains, respectively. Writes within the block are treated in three different ways: (1) non-reference locations, $\mathit{WriteIn}(b,v)$, are bound to an uninterpreted function that represents the unknown value that may have been written to it by the block, (2) each successor state selects a subset of the locals of reference type and updates them with an uninterpreted function defining the reference, and (3) heap reference locations, $\mathit{WriteIn}_h(b,v)$, are eliminated from the symbolic state which makes them uninitialized so that LSE will re-initialize them as explained below. The second case is illustrated by the creation of branches (b.1) and (b.2) in Figure 1.

Initializing Abstracted Objects As LSE proceeds after an abstracted block it may create new locations. If such a location was written in b the summary instantiation shown above will not reflect that update, since that location did not exist in the state on entry to the block. Our solution is to carry $\mathit{Write}_h(b)$ and $\mathit{ReadIn}(b,v)$ forward during symbolic execution and use them to add two additional cases for lazy initialization.

Specifically, the symbolic state is extended to be a quadruple: (pp, pc, v, w). w is a stack of pairs $(\mathit{Write}_h(b), \mathit{ReadIn}(b,v))$ where b is a block in the current execution path and pairs are ordered from most to least recently executed on the path. When b_{gabs} is instantiated in state (pp_{entry}, pc, v, w) the resulting state is

$$\begin{aligned}
(pp_{exit}, pc \wedge i(\mathit{ReadIn}(b,v)),\ & v\langle \textstyle\bigwedge l \in \mathit{WriteIn}(b,v) : l = e_l(\mathit{ReadIn}(b,v))\rangle - \mathit{WriteIn}_h(b,v)), \\
& (\mathit{Write}_h(b), \mathit{ReadIn}(b,v)) : w)
\end{aligned}$$

where $x : y$ denotes that x is pushed on to stack y. Correspondingly, when symbolic execution backtracks across an abstracted block the fourth state component is popped.

We extend lazy initialization to account for uninitialized reference fields that may have been written by the abstracted code block with the adapted update rules in Figure 3: (6) is an adaptation of rule (2) from Figure 2 and (7) is an adaptation of rule (4). We denote the first and second components of pairs with .1 and .2, respectively. These add additional branching to the symbolic execution to set uninitialized reference fields that could have been written in an abstracted block to uninterpreted functions that encode the unknown object reference; rule (7) also creates new objects with an access-path address. Step (4b.2) in Figure 1 illustrates the application of rule (7) to create a new node with an access-path name whose fields are subsequently written at (4b.2) and (5b.2).

5 Conclusions and Future Work

It has long been understood that the cost of re-validating a system after a change is disproportionate to the size of the change. Lightweight techniques, such as *diff*, provide only the most superficial syntactic characterization of program differences. Semantics-based verification and certification techniques require much more precise methods that can accurately describe the portion of a system's input space for which it is known to retain its previous functionality. A logical characterization of that input sub-space can then be leveraged to prune the reasoning required by formal methods tools used to re-verify the changed system.

Symbolic execution coupled with abstract summaries on common program blocks are a powerful foundation for building these kind of differencing analyses. In this paper we have extended the summaries used by our DSE approach to accommodate more general definitions of common program blocks and to safely treat heap-manipulating blocks. We have implemented a prototype version of our technique for computing and realizing generalized abstract symbolic summaries in the Java PathFinder (JPF) symbolic execution framework [12]. At present, our prototype has been applied to small examples like the one in Figure 1. We are planning to conduct a more comprehensive evaluation of our approach and plan to present those results at the workshop. In the longer term, a focus of our future work is to couple DSE with testing, model checking, and deductive methods to realize *incremental* V&V techniques that can be applied cost-effectively across a system's lifetime.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of TACAS*, 2008.

[2] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions Computational Logic*, 2(1):93–134, 2001.

[3] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Intl. Journal of Par. Progr.*, 34(1):61–91, 2006.

[4] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of ASE*, pages 157–166, 2006.

[5] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations using finite representations of right-regular equivalence relations. In *Proceedings of the ICCL*, pages 2–13, 1992.

[6] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of PLDI*, pages 106–117, 1998.

[7] P. Godefroid. Compositional dynamic test generation. In *Proceedings of POPL*, pages 47–54, 2007.

[8] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS*, 2003.

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

[10] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of PLDI*, pages 56–67, 1993.

[11] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of FSE*, pages 226–237, New York, NY, USA, 2008. ACM.

[12] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the ISSTA*, pages 15–25, 2008.

[13] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the ISSTA*, pages 157–168, 2006.

[14] S. F. Siegel and L. F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In *Proceedings of European PVM/MPI Meeting*, volume 5205 of *LNCS*. Springer, 2008.

[15] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of ISSTA*, pages 97–107, 2004.