Verifying Reference Counting Implementations*

Michael Emmi[1], Ranjit Jhala[2], Eddie Kohler[1], and Rupak Majumdar[1]

[1] University of California, Los Angeles, {mje,kohler,rupak}@cs.ucla.edu
[2] University of California, San Diego, jhala@cs.ucsd.edu

Abstract. Reference counting is a widely-used resource management idiom which maintains a count of references to each resource by incrementing the count upon an acquisition, and decrementing upon a release; resources whose counts fall to zero may be recycled. We present an algorithm to verify the correctness of reference counting with minimal user interaction. Our algorithm performs compositional verification through the combination of symbolic *temporal case splitting* and predicate abstraction-based reachability. Temporal case splitting reduces the verification of an unbounded number of processes and resources to verification of a finite number through the use of *Skolem variables*. The finite state instances are discharged by symbolic model checking, with an auxiliary invariant correlating reference counts with the number of held references. We have implemented our algorithm in Referee, a reference counting analysis tool for C programs, and applied Referee to two real programs: the memory allocator of an OS kernel and the file interface of the Yaffs file system. In both cases our algorithm proves correct the use of reference counts in less than one minute.

1 Introduction

Reference counting is a widely-used resource management idiom where the references to each resource unit (*e.g.,* memory cell, file handle, device structure) are counted. The programmer increments the count when acquiring a resource and decrements it when releasing. A resource may be recycled when its reference count reaches zero.

Despite its ubiquity, reference counting is difficult to implement correctly. Ensuring a resource is not accessed after its count reaches zero requires precisely reasoning about shared heap objects in concurrent programs with a statically unknown number of sharers. In the most benign case, errors in reference counting cause resource leaks: when the last reference to an object is removed but the reference count is not decremented to zero. More dangerous errors can allow unprivileged read or write access to critical regions of memory that have been inappropriately reclaimed and recycled, possibly compromising sensitive information.

We show how predicate-abstraction based software model checking can be extended with compositional reasoning techniques to enable the static verification of the correctness of reference counting implementations (*i.e.,* that accessed objects have positive counts). The problem is difficult as such programs are unbounded in several dimensions; first, an unbounded number of objects may be dynamically allocated, second, each unit may be accessed concurrently by an unbounded number of sharers, and hence, third,

* This research was sponsored in part by the NSF grants CCF-0546170 and CCF-0702743.

the reference count for each individual object may grow without bound. These complications prohibit the direct application of finite-state techniques such as model checking, to verify reference counting. Furthermore, standard program analysis abstractions that summarize an unbounded number of dynamic objects (*e.g.,* clients, resources) are too imprecise since they do not *count* the objects they summarize.

Our approach for verifying reference counting implementations follows the following strategy. As a first step, we perform *compositional reasoning* to reduce the verification problem to a number of finite-state verification problems whose combined validity implies the original program's [22]. One possible verification strategy is to tag each resource with a handle, and ensure that clients only access resources to which they have handles. Correctness then follows separately from the correctness of handling each resource, and each handling process. This first step is called *temporal case splitting*: we check validity for a particular tracked resource and a particular accessee in an environment that abstracts all other resources and accessors.

Temporal case splitting trades the original complex verification problem for an infinite number of separate simpler verification obligations. However, using symmetry, we observe that discharging the proof obligation for an *arbitrary* symbolically identified resource, and an *arbitrary* symbolically identified client, implies discharging each of the infinitely many obligations induced by case splitting. Thus, as a second step, we use *Skolem variables* to name single, but arbitrary, resources and clients. The Skolem variables induce a natural finite abstraction of the system which distinguishes only the fixed resources bound to Skolem variables (and abstracts all other resources). Similarly, instead of tracking every client, the abstraction tracks only the fixed clients bound to Skolem variables, abstracting the effects of other clients. Skolemization enables *strong updates* on the tracked resources: we can follow each increment or decrement to the tracked resource precisely; all updates to the other resources are *weak*: their effects are unknown.

Unfortunately, the strategies given so far are still insufficient; we must also deal with unbounded reference count *values*. On the one hand, abstracting a counter's increments and decrements by untracked clients results in a loss of the precise counter value, and a proof is generally not possible. On the other hand, the abstract domain remains infinite (with a different value for each counter value) if we track each of these writes precisely. To solve this problem we observe that correctness follows from knowing a unit's count is positive if and only if it is referenced by some client. To prove this, as a third step, we introduce a *reference predicate*, specifying the *meaning* of referencing a unit, and automatically insert an *auxiliary variable* whose value, by construction (i.e. instrumentation) equals the number of client mappings satisfying the reference predicate. An *auxiliary invariant* enforces a positive valued auxiliary counter whenever a client satisfies the reference predicate.

With these steps, we reduce proving an object's real reference count positive to checking that (1) the tracked client satisfies the reference predicate, and (2) the real reference count *equals* the auxiliary count. As (1) follows by precisely tracking the truth value of the reference predicate for the tracked client, and (2) follows from reasoning about the equality of program variables, the resulting obligation can be discharged using well-established techniques: either through specially designed *abstract domains* [6] that can prove linear relationships among variables, or, as we implement, through predicate abstraction and counterexample-guided refinement [11, 5, 1, 13] based model checking. The meta-argument used for the auxiliary invariant is manually proved sound outside of the program analysis, and can be *reused* to verify any program that implements reference counting, once the reference predicate is specified. The meta-argument used for the auxiliary invariant is manually proved sound outside of the program analysis, and can be instantiated with different reference predicates to verify any program that implements reference counting. Moreover, the soundness of our approach is independent of the choice of reference predicate: an invalid predicate yields a failed proof, since either (1) or (2) will fail to hold.

In summary, our analysis combines four ingredients:

Temporal case splitting to reduce the (infinite state) verification goal over infinitely many objects to infinitely many (finite state) subgoals over individual objects,

Skolemization to reduce infinitely many verification subgoals over different objects, into a single verification goal over an *arbitrary* object,

Auxiliary state to provide a finite representation of unbounded execution history *i.e.,* the unbounded reference count for a given object, and

Model checking to discharge the finite state verification goals induced by the use of temporal case splitting, skolemization and auxiliary state.

While the general techniques have been known [22], we have not seen them successfully applied in software model checking so far, and believe that our implementation is an interesting application of compositional verification to a relevant systems problem.

We have implemented these ideas in a static analyzer for verifying sound reference counting in C programs. Given a program, and a user-specified set of Skolem variables, our tool instruments the program with auxiliary state and auxiliary invariant instantiation, and performs model checking on the instrumented program abstracted by Skolem- and predicate-abstraction. The model checking engine of our tool is based on the software model checker BLAST [13], and uses an iterative refinement of the abstract transition relation based on counterexample traces [12, 15]. Our technique is not completely automatic, and requires that the user identify reference counted datatypes as well as which variables to perform case-splits on. In our experiments, we have found the identification of reference counted datatypes and case-split variables can be performed with a limited knowledge of the program. Our analysis relieves the programmer of the difficult burden of providing precise inductive assertions at function and loop boundaries, a task which is readily performed by the model checker.

We have applied our tool to two case studies: the virtual memory subsystem of the JOS operating system kernel [17], and the file handle interface routines of the YAFFS file system [23]. In each case, the soundness argument depends on precise reasoning about arbitrarily many clients acquiring and releasing resources. These modules (each of a few hundred lines) encapsulate reference counting; sound counting within them implies sound counting for the entire systems. Each example can be verified by our tool within a minute.

Acknowledgments. We thank the anonymous referees and Alessandro Cimatti for helpful comments.

2 Verification Technique

We now formalize our verification technique and illustrate with an example.

Preliminaries: Programs and Safety. For our formal presentation, we assume an abstract representation of programs by transition systems [21]. A *program* $P = \langle X, L, \ell_0, R\rangle$ consists of a set X of variables, a set L of control locations, an initial location $\ell_0 \in L$, and a transition relation R. Variables in X have values over integers or functions. (Functions are used to model (unbounded) arrays by mapping natural numbers, *i.e.,* the "indices", to values.) A transition $\langle \ell, \rho, \ell'\rangle \in R$ is a move from control location ℓ to location ℓ', satisfying ρ, a constraint over free variables from $X \cup X'$. The variables from X denote values at location ℓ, and the variables from X' denote the values of variables from X at location ℓ'. The sets of locations and transitions naturally define a directed graph, called the program's *control-flow graph* (CFG).

A *data state* of the program P is a valuation of the variables from X; the set of all data states is denoted Σ. We use constraints to represent sets of data states. For a constraint ρ over $X \cup X'$ and a valuation $\langle s, s'\rangle \in \Sigma \times \Sigma'$, we write $\langle s, s'\rangle \models \rho$ if the valuation satisfies the constraint ρ. A *state* $\langle \ell, s\rangle$ consists of a location $\ell \in L$ and a data state s. A *computation* of the program P is a sequence of states $\langle \ell_0, s_0\rangle, \langle \ell_1, s_1\rangle, \ldots, \langle \ell_k, s_k\rangle \in (L \times \Sigma)^*$, where ℓ_0 is the initial location and for each $i \in \{0, \ldots, k-1\}$, there is a transition $\langle \ell_i, \rho, \ell_{i+1}\rangle \in R$ such that $\langle s_i, s_{i+1}\rangle \models \rho$. A data state s is *reachable* at location ℓ if $\langle \ell, s\rangle$ appears in some computation. A state $\langle \ell, s\rangle$ is reachable if the data state s is reachable at location ℓ. Let φ be a set of states. A program P is *safe* w.r.t. φ iff all reachable states of P are contained in φ.

Example. Figure 1 shows an abstraction of a shared memory system in which an arbitrary number of processes (syntactically identified with pid) share an unbounded number of resources, indexed by g, and reference counted by the array count. For readability, we present programs in a C-like syntax instead of as tuples. All reference counts are initially zero. Each process first chooses a resource (line 1), then acquires the resource while incrementing its reference count (line 2), performs some task, then releases the resource while decrementing its reference count (line 5). We assume lines 2 and 5 execute atomically. Although implicit, the system may "recycle" a resource when its reference count reaches 0; to ensure the system does not recycle live resources, we seek to verify the validity of the assertion on line 3. The simple reference count example is "obviously correct". However, consider a modified version where the acquire of the resource and the increment of the reference count are not performed atomically, but in distinct steps. This implementation is buggy: between the resource acquisition and the reference count increment, the resource can be freed, if another process happens to hold the only other reference to the same resource, and calls **decref**. It follows that the **incref** operation can read and write on freed (or worse, reallocated) memory. Similar bugs have been found in Windows device drivers [24].

For this simple example the assertion always holds because the current process at line 3 holds a reference to resource g, and hence count[g]$\geq$1. The assertion is an instance of a safety property, and can be checked by ensuring that no reachable program state violates it. Unfortunately, there are infinitely many reachable states of the system

Initially

```
count[g] = 0 for all g
```

Process pid

```
choose g;
ref[pid] ← g; incref count[g];
assert(count[g] > 0);
do work
ref[pid] ← −1; decref count[g];
```

Fig. 1. Abstract reference counting

```
atomic {
  item ← acquire(0);
  incref(item);
}
repeat {
  choose g;
  atomic {
    new_item ← acquire(g);
    decref(item);
    item ← new_item;
    incref(item);
  }
}
```

Fig. 2. Buggy reference counting

Initially

```
count[g] = 0 for all g
x_Π[g] = 0 for all g
```

Process pid

```
choose g;
ref[pid] ← g; incref count[g];
    update_aux(x_Π[g], ref[pid]=g)
assert(pid=P ⇒ g=G ⇒ count[g]>0);
do work; update_aux(x_Π[g], ref[pid]=g)
ref[pid] ← −1; decref count[g];
    update_aux(x_Π[g], ref[pid]=g)
```

Fig. 3. Abstract reference counting, after Skolemization and Auxiliary Instrumentation. The programmer manually identifies the Skolem variables P and G. The system automatically inserts the auxiliary variables and instrumentation.

as the set of resources, processes, and counter values are all unbounded. Hence we must perform reachability analysis over an *abstraction* of the system.

Here the usual abstraction techniques for arrays [1, 13], such as merging all elements into a single element, are too imprecise; they prohibit the analysis from performing strong updates (*i.e.,* precisely tracking information about a resource), and from distinguishing individual resources. Similarly, to prove the assertion we would require an abstract domain that could distinguish the infinitely many states where count[g] has different values. For example, a predicate abstraction [11] based domain would have to track an unbounded number of predicates of the form count[g]=n for each index g and each integer value n that can be stored in count[g].

Step 1: Temporal Case Splitting

Temporal case splitting [22] is a proof technique that decomposes the proof of a program property into sub-proofs, one for each value in the domain of a particular variable. It is based on the following observation.

Lemma 1. (Case Splitting) *Let x be a variable of program P, and φ a set of states. Then P is safe w.r.t. φ iff for each c in the domain of x, the program P is safe w.r.t. $(x = c) \Rightarrow \varphi$.*

Temporal case splitting can be *nested*: in order to check safety w.r.t. $(x = i) \Rightarrow \varphi$, we can further case split on a second variable, and so on.

Example. For the example of Figure 1, we may split the assertion on line 3 into an infinite number of assertions **assert** (g=0 ⇒ count[g]>0), **assert** (g=1 ⇒ count[g]>0), and so on, one for each resource. By the same reasoning, we can case split further over the process identifier into an infinite number of assertions **assert** (pid=0 ⇒ g=0 ⇒ count[g]>0), **assert** (pid=1 ⇒ g=0 ⇒ count[g]>0), and so on, one for each process and resource pair. Temporal case splitting is sound in that if each subgoal is true, then the original safety property is also true. However, by itself it is not very useful, as it introduces an infinite number of sub-goals.

Step 2: Skolemization

Though case splitting introduces infinitely many sub-goals, the sub-goals are *symmetric* as each process behaves in a manner similar to the others, and the resources are distinct copies of the same entity. Instead of checking each concrete process and resource separately, we can perform a *single* check for an *arbitrary* process and an *arbitrary* resource. If we prove this goal, then the assertion is valid *for all* processes and all resources. To *name* the arbitrary (but fixed) process and resource, require that the programmer identify *Skolem variables*. These are fresh variables, distinct from the original program variables, that are non-deterministically initialized with an arbitrary value from a possibly unbounded range, and not modified subsequently.

Formally, we introduce Skolem variables as follows. Let $P = \langle X, L, \ell_0, R\rangle$ be a program, and let S be a set of *Skolem variables* disjoint from X. We denote by $P[S] = \langle X \cup S, L, \ell_0, R[S]\rangle$ the program P augmented with Skolem variables S, where $\langle \ell, \rho', \ell'\rangle \in R[S]$ iff there is a transition $\langle \ell, \rho, \ell'\rangle \in R$ and $\rho' \equiv \rho \wedge \bigwedge_{s\in S} s' = s$. An extended data state is a valuation to $X \cup S$, an extended state consists of a location ℓ and an extended data state. To distinguish states of P from states of $P[S]$ (which additionally contain valuations to the Skolem variables), we qualify states with the programs by writing P-state, or $P[S]$-state. By definition, the Skolem variables do not alter the program's behavior; they exist solely for the purpose of the proof and need not be maintained at runtime.

Lemma 2. (Skolemization) *Let x be a variable of program P, φ a set of P-states, S a set of Skolem variables, and $s \in S$. P is safe w.r.t. φ iff $P[S]$ is safe w.r.t. $(x = s) \Rightarrow \varphi$.*

Proof. First of all, P is safe w.r.t φ iff $P[S]$ is. By Lemma 1 $P[S]$ is safe w.r.t. φ iff $P[S]$ is safe w.r.t. $(x = c) \Rightarrow \varphi$ for each c in the domain of x. Since the set of states $(x = c) \Rightarrow \varphi$ is equal to $(x = s \wedge s = c) \Rightarrow \varphi$ (s is not assigned to in $P[S]$), and thus equal to $(s = c) \Rightarrow (x = s) \Rightarrow \varphi$, $P[S]$ is safe w.r.t. φ iff $P[S]$ is safe w.r.t. $(s = c) \Rightarrow (x = s) \Rightarrow \varphi$ for each c in the domain of x, and again by Lemma 1, iff $P[S]$ is safe w.r.t. $(x = s) \Rightarrow \varphi$.

Example. For the program of Figure 1, we (manually) identify two Skolem variables corresponding to the unbounded arrays of processes and resources: P corresponds to an arbitrary process, and G corresponds to an arbitrary resource. Since G and P are never assigned to (they do not even exist in the original program), the infinite number of assertions **assert** (pid=i ⇒ g=j ⇒ count[g] > 0), one for each i and j, are equivalent to the single assertion **assert** (pid=P $\Rightarrow$ g=G $\Rightarrow$ count[g]>0), because G=0 $\vee$ G=1 $\vee \cdots$, and P=0 $\vee$ P=1 $\vee \cdots$ are both valid formulæ.

The key benefit of the Skolem variables is that they induce a sound finite abstraction on the state space. Instead of a possibly unbounded number of processes, we (strongly) track the *single* process whose identifier is equal to P, and effectively merge all the other processes (whose identifiers are different from P) into one abstract "summary" process. Similarly, instead of an unbounded number of indices of the count array, we strongly track the resource at index G, and merge the cells whose index is different from G into a single summary cell. For example, using predicate abstraction, we would track the predicate ref [P]=G, rather than ref [p1]=G, ref [p2]=G, ..., effectively dividing these process-specific facts into the fact at P, and those in any other untracked process.

Example. Consider the following C program:

```
1: for (i = 0; i < N; i++) a[i] = 0;
2: for (i = 0; i < N; i++) assert(a[i] == 0);
```

To verify the assert on Line 2, the analysis must infer that the loop on Line 1 initializes *all* the cells with indices between `0` and `N-1` with the value `0`. Instead of reasoning about an unbounded number of cells, suppose the programmer introduces a skolem variable `S`, that represents an arbitrary index into the array. Case splitting w.r.t. `S` replaces the assertion on Line 2 with `assert(i==S => a[i]==0)`. That is, the verification is reduced to an assertion over the single array cell `S` and all others are ignored. Finally, notice that predicate abstraction over predicates `0`$\leq$`i`, `i<N`, `0`$\leq$`S`, `S<N`, `S<i`, `S=i`, `S>i`, and `a[S]=0` suffices to prove the reduced assertion. Using these predicates, the analysis infers that at Line 1, the invariant (a): (`0`$\leq$`S` $\wedge$ `S<N` $\wedge$ `S<i`) $\Rightarrow$ `a[S]=0` holds, using which it it infers that at Line 2, the invariant (b): (`0`$\leq$`S` $\wedge$ `S<N`) $\Rightarrow$ `a[S]=0` holds. Finally, it infers that at the assert, (`0`$\leq$`i` $\wedge$ `i<N`), which with (b) proves the assert. By the choice of predicates, we made the analysis precisely track the cell indexed by `S`, while merging (*i.e.,* ignoring) the values of all other cells.

The choice of Skolems affects the precision but not the soundness of our technique. A poor choice can yield an abstraction that is too coarse for verification. A simple heuristic is to choose a Skolem for each unbounded object (e.g. processes, array indices).

Step 3: Auxiliary Variables and Invariants

We need one more step before applying model checking: strengthening the program transition relation using auxiliary invariants.

Formally, let $P = \langle X, L, \ell_0, R\rangle$ be a program, S a set of Skolem variables for P, Y be a set of *auxiliary variables* disjoint from $X \cup S$, and for each $y \in Y$, an *auxiliary update function* ϕ_y mapping current and next values of $X \cup S$ and current values of Y to a value in the domain of y. A *monitored program* $P[S, Y, \phi] = \langle X \cup S \cup Y, L, \ell_0, R[S, Y]\rangle$ has a transition relation $R[S, Y]$ such that $\langle \ell, \rho, \ell'\rangle \in R[S]$ iff $\langle \ell, \rho', \ell'\rangle \in R[S, Y]$ where $\rho' \Leftrightarrow \rho \wedge \bigwedge_{y \in Y} y' = \phi_y(x, x', s, s', y)$. In other words, the transition relation is extended by updating the auxiliary variables in Y according to both the current and next values of variables in $X \cup S$ and the current values of variables in Y (using the functions ϕ_y). Like Skolem variables, the values stored in auxiliary variables do not alter program behavior: P is safe w.r.t. property φ iff $P[S, Y, \phi]$ is.

Intuitively, the auxiliary variables, also known as *monitors* [21], *ghost variables*, or *spec variables* [22, 8, 2, 18], are additional variables whose values depend on the program state, but do not affect the values of other program variables. The auxiliary state is solely a proof device (*i.e.,* they are not maintained during program execution) and are used to explicate implicit program invariants.

For program $P = \langle X, L, \ell_0, R\rangle$ and predicate ψ over $X \cup X'$, define the *ψ-reduced program* $P_\psi = \langle X, L, \ell_0, R_\psi\rangle$, where $\langle \ell, \rho, \ell'\rangle \in R$ iff $\langle \ell, \rho \wedge \psi, \ell'\rangle \in R_\psi$. An *auxiliary invariant* for $P[S, Y, \phi]$ is a predicate ψ over $X \cup S \cup Y \cup X' \cup S' \cup Y'$ such that the transition relation of $P[S, Y, \phi]$ restricted to the reachable states is a subset of ψ.

Lemma 3. (Auxiliary Invariant) *Let P be a program and φ a set of states of P. For Skolem variables S, auxiliary variables Y, and auxiliary update functions ϕ, if ψ is an auxiliary invariant for $P[S, Y, \phi]$ and $P[S, Y, \phi]_\psi$ is safe w.r.t. φ, then P is safe w.r.t. φ.*

Auxiliary Invariants via Reference Predicates. Even after Skolemization we are left with verification obligations over unbounded state spaces, as the reference counts are unbounded. To solve this problem, we introduce an auxiliary invariant relating a resource's reference count with the number of references to it. A *reference predicate* is a quantifier-free predicate Π over program variables, that is parameterized by two variables, a *source* i and *target* j. A reference predicate Π defines, for each source i, a *reference (target) set*

$$\Pi(i) \doteq \{j \mid \Pi(i, j)\}$$

For each reference predicate, we automatically add auxiliary variables that track the *cardinality* of $\Pi(i)$, by instrumenting the program with an unbounded auxiliary array x_Π that maps the domain of sources to the domain of targets. Assume that: (**A1**) in the initial state, the reference predicate is false for all sources and targets, and, (**A2**) each transition affects only a *finite, named* set of sources and targets. The first assumption is semantic, and depends on the choice of the reference predicate. The second assumption can be syntactically enforced. Under these assumptions, we can automatically instrument the program with auxiliary transitions that (1) initialize $x_\Pi[i]$ with 0, and, (2) increment (resp. decrement) the auxiliary counter $x_\Pi[i]$ whenever for some j, a program transition turns $\Pi(i, j)$ toggles from false to true (resp. from true to false). This auxiliary instrumentation ensures "by construction" the invariant: $x_\Pi[i] = |\Pi(i)|$, *i.e.,* that $x_\Pi[i]$ equals the cardinality of the reference target set $\Pi(i)$. Finally, we instrument the program (*i.e.,* conjoin the set of reachable states) with the auxiliary invariant $\Pi(i, j) \Rightarrow x_\Pi[i] > 0$ for all syntactic sources and targets i and j. This invariant follows from the meta-theorem that if for some i, j, the reference predicate $\Pi(i, j)$ holds, then the reference set $\Pi(i)$ is non-empty, and hence its cardinality $x_\Pi[i]$ is positive.

This strategy addresses the unboundedness of the reference counts as follows. First, it uses a semantic notion of reference (the reference predicate) to instrument the program with a "correct-by-construction" reference counter. Second, in the program reduced w.r.t. the auxiliary invariant, we have replaced the *global* check that the implemented reference count is *positive* with the *local* check that the implemented reference count *equals* the auxiliary reference count. Though the auxiliary counter x_Π is an unbounded array, we can apply case splitting and Skolemization to this array as with the original program variables. Notice, that strategy only assumes the simple, enforceable, requirements **A1, A2** about the reference counts and program. In particular, it does not assume that the program performs correct reference counting.

Example. The reference relationship for the program in Figure 1 is captured by the predicate:

$$\Pi(\texttt{g}, \texttt{pid}) \doteq (\texttt{ref[pid]=g})$$

which states that there is a reference to the source object `g` from a target process `pid` iff `ref[pid]=g`. For this reference predicate, x_Π is the auxiliary correct-by-construction reference count array such that x_Π`[g]` equals the *number* of processes that have a reference to `g`. Our tool automatically instruments the program so that each element of x_Π is initially 0. Further, transitions are added to increment (resp. decrement) an element of x_Π whenever the resource corresponding to the element is acquired (resp. released). This instrumentation is performed by the function `update_aux` in Figure 3, which takes as input the counter x_Π`[g]` and the predicate `ref[pid]=g` and increments (resp. decrements) the counter if the predicate toggles from false to true (resp. true to false) in executing the transition. Figure 3 shows the program instrumented with the case splits induced by the Skolem variables (line 3), the auxiliary variable x_Π, and the auxiliary update function `update_aux`. Finally, our tool automatically strengthens the program with the auxiliary invariant `ref[pid]=g` $\Rightarrow$ x_Π`[g]>0` that follows from the instrumentation and the meta-theorem described above. Thus, our technique uses the manually specified reference predicate to instrument the program with correct-by-construction counters, following which the verification task is reduced to proving, that for each `g`, we have `count[g] =` x_Π`[g]` (which, conjoined with the auxiliary invariant proves the reference count assertion on at Line 3). Finally, note that via Skolemization, the above reduces to proving `count[G] =` x_Π`[G]` for an arbitrary object `G`.

Step 4: Model Checking

Once we have introduced Skolem variables and auxiliary invariants, we can apply a software model checker such as SLAM or BLAST to discharge the assertion. Algorithm 1 shows a worklist based abstract model checking algorithm using an auxiliary invariant. Its soundness is standard. The procedure PredAbs on Line 8 computes an abstraction of the concrete transition relation relative to an abstract domain (in our implementation, predicate abstraction with transition refinement [11, 13, 15]). Notice that the abstraction assumes that the auxiliary invariant holds along each transition. (Techniques to automatically find appropriate predicates [5, 12] are orthogonal, and can be combined with our algorithm.)

Lemma 4. (Soundness) *If Algorithm 1 returns* SAFE *and ψ is an auxiliary invariant of P, then P is safe w.r.t. φ.*

Example. In our example, the model checker runs a program consisting of an unbounded number of processes each executing the code, where each instruction in the code (each line in the example) is considered atomic. Consider the predicates:

Algorithm 1: Symbolic Model Checking

Input: Program $P = \langle X, L, \ell_0, R\rangle$, States φ, Predicates Π, Auxiliary Invariant ψ
Result: SAFE or UNSAFE
Data: Queue worklist, Incremental per-location invariant η

```
worklist ← [⟨ℓ0, true⟩];
η ← λℓ.false;
while worklist is not empty do
    remove ⟨ℓ, ŝ⟩ from worklist;
    if ŝ ⇒ η(ℓ) is not valid then
        η ← η[ℓ ↦ ŝ ∨ η(ℓ)];
        foreach ⟨ℓ, ρ, ℓ'⟩ ∈ R do
            add ⟨ℓ', PredAbs(ŝ ∧ ρ ∧ ψ, Π)⟩ to worklist;
        end
    end
end
if ∀ℓ. η(ℓ) ⇒ φ(ℓ) then return SAFE else return UNSAFE
```

location	abstract states			
scheduler	$true$			
1	a		$\bar{a}$	
2	ab	$a\bar{b}$	$\bar{a}b$	$\bar{a}\bar{b}$
3	$abcd$	$a\bar{b}\bar{d}$	$\bar{a}bc$	$\bar{a}\bar{b}$
4	$abcd$	$a\bar{b}\bar{d}$	$\bar{a}bc$	$\bar{a}\bar{b}$
5	$abcd$	$a\bar{b}\bar{d}$	$\bar{a}bc$	$\bar{a}\bar{b}$
exit	$ab\bar{d}$	$a\bar{b}\bar{d}$	$\bar{a}b$	$\bar{a}\bar{b}$

invariant: ef

Fig. 4. The reachable abstract program states of the program in Figure 3 at each program location w.r.t. the predicates (a)–(f) given above. A string of (possibly barred) predicates indicates a partial valuation where each non-barred predicate is true, each barred predicate is false, and each unmentioned predicate may be either true or false. (Conceptually the partial valuation is a disjunction of (total) valuations.) The predicates (e) and (f) hold universally, and the string ef is implicitly appended to each valuation. The scheduler is responsible for deciding which process `pid` executes.

$$\text{(a) } \texttt{pid=P},\ \text{(b) } \texttt{g=G},\ \text{(c) } \texttt{count[G]>0},$$
$$\text{(d) } \texttt{ref[P]=G},\ \text{(e) } x_\Pi\texttt{[G]=count[G]},\ \text{(f) } \texttt{count[G]}\geq\texttt{0}.$$

Predicates (a) and (b) allow us to *strongly* track facts of the "interesting" array indices. (c) and (d) track whether the (arbitrary) process `P` references a resource with a positive count. Predicate (e) tracks whether the auxiliary and actual counters agree on the reference counts, and (f) is needed to derive (c) when the counts is incremented. These predicates are sufficient for our model checker to synthesize the inductive invariant

$$x_\Pi\texttt{[G]=count[G]} \;\wedge\; \texttt{(ref[P]=G} \Rightarrow \texttt{count[G]>0)}$$

describing an over-approximation of the reachable program states (Figure 4 shows this calculation), which suffices to prove the case-split assertion at line 3 of Figure 3. Though the first two predicates do not appear in the invariant, they are essential for its derivation as they enable strong updates on the Skolemized cells of `count`, `ref`, and x_Π.

Temporal case splitting on the Skolem variables reduces the infinite number of processes and resources to a finite set. Similarly, the auxiliary invariant and counter ensure that we need only to track the relationship between the auxiliary and actual counters, and whether the former is positive, instead of precisely tracking an unbounded counter. It is the combination of these techniques that allows the model checker to prove such a complex property of an unbounded system.

Example: Buggy Reference Counting. Figure 2 shows a reference count implementation that is contains a bug that arises from aliasing. The program is motivated by an actual bug in an implementation of the Python language [28]. Each client works atomically in a loop, acquiring a new resource and releasing the old resource in each iteration. The error occurs when the old resource in `item` is the same as the new resource in `new_item`, and the resource has reference count of `1` (*i.e.,* the client holds the only reference to this resource). The `decref` on line 9 then decreases the reference count to `0`, and frees the resource. However, the client still holds a reference to the same resource in `new_item` (from line 8), so the `incref` at line 11 erroneously writes to freed memory, possibly corrupting it. Our tool does find an error trace for this buggy program. Furthermore, our technique is able to prove safe the correct version of the program, where the reference count is incremented before the decrement on line 9.

Limitations. One limitation of our approach is that the Skolem variables necessary for verification are not mechanically determined; this is left for the user of our analysis tool. In our experience with reference counting we have found the number to be small (one, or two, per structure) and easy to find, but the search for appropriate Skolems can be hard in general. Second, our approach is constrained by the invariants that can be expressed by the abstract domain, and the design of an appropriate domain can be hard for complicated invariants, especially with rich quantifier structures. Our technique only checks that when a resource is accessed, it has a positive reference count. This property by itself does not guarantee the absence of memory leaks, for example, those caused by cyclic structures of references that are not reachable from any program variable.

3 Case Studies

In addition to the simple examples from Section 2, we have applied our tool to two case studies of reference counting in real systems code: a page allocator derived from the JOS kernel [17], and the YAFFS file system [23].

We use a *logical memory model*: memory is represented as an unbounded array `Mem` of elements large enough to hold any structure allocated in the program. Each memory cell is annotated with a `valid` bit, initially each with value `0`. Pointers are modeled as indices to the `Mem` array, and index `0` denotes `null`. Our implementation of `malloc` nondeterministically chooses an index `i` such that `Mem[i].valid = 0`, sets `Mem[i].valid` to `1`, and returns `i`. Our implementation of `free(i)` ensures that `Mem[i].valid = 1`, and resets `Mem[i].valid` to `0`.

We model concurrency by calling the top-level procedures, which are considered atomic, inside of a loop which nondeterministically chooses a process/thread identifier `pid` and a procedure `proc` and executes `proc` as `pid`.

JOS Memory Mapping. In JOS [17], a simple operating system used as an educational aid, memory is organized as an array of physical pages, to which user processes

```c
typedef struct env {
    int env_mypp;
    int env_pgdir[NVPAGES];
    struct env *env_prev;
    struct env *env_next;
} env_t;

int pages[NPPAGES];
int page_protected[NPPAGES];
env_t *envs = NULL;
```

Fig. 5. Environment data structures in JOS.

```c
int page_alloc(env_t *env, int vp) {
    int pp = page_getfree();
    if (pp < 0) return -1;
    if (env->env_pgdir[vp] >= 0)
        pages[ env->env_pgdir[vp] ]--;
    env->env_pgdir[vp] = pp;
    pages[pp]++;
    return 0;
}
int page_unmap(env_t *env, int vp) {
    if (env->env_pgdir[vp] >= 0) {
        pages[ env->env_pgdir[vp] ]--;
        env->env_pgdir[vp] = -1;
    }
}
int page_map(env_t *srcenv, int srcvp,
             env_t *dstenv, int dstvp) {
    if (srcenv->env_pgdir[srcvp] < 0)
        return -1;
    pages[ srcenv->env_pgdir[srcvp] ]++;
    if (dstenv->env_pgdir[dstvp] >= 0)
        pages[ dstenv->env_pgdir[dstvp] ]--;
    dstenv->env_pgdir[dstvp] =
        srcenv->env_pgdir[srcvp];
    return 0;
}
```

Fig. 6. Page directory manipulation in JOS. `page_getfree` returns the index to an unused page, if one is exists, and `-1` otherwise.

```c
env_t *env_alloc(void) {
    env_t *env;
    int i, env_pp = page_getfree();
    if (env_pp < 0) return NULL;
    env = (env_t *) malloc(sizeof(env_t));
    env->env_mypp = env_pp;
    for (i = 0; i < NVPAGES; i++)
        env->env_pgdir[i] = -1;

    /* put on list */
    env->env_next = envs;
    env->env_prev = NULL;
    if (envs) envs->env_prev = env;
    envs = env;
    pages[env_pp]++;
    page_protected[env_pp] = 1;
    return env;
}

void env_free(env_t *env) {
    int i;
    for (i = 0; i < NVPAGES; i++)
        if (env->env_pgdir[i] >= 0)
            pages[ env->env_pgdir[i] ]--;
    page_protected[env->env_mypp] = 0;
    pages[ env->env_mypp ]--;

    /* take off list */
    if (env->env_next)
        env->env_next->env_prev =
            env->env_prev;
    if (env->env_prev)
        env->env_prev->env_next =
            env->env_next;
    else envs = env->env_next;
    free(env);
}
```

Fig. 7. Environment (de)allocation in JOS.

(or *environments*) hold virtual page mappings (see Figures 5–7). The environment structure (`env_t`) stores the index of a protected physical page (`env_mypp`), a virtual page table (`env_pgdir`), and pointers used for the kernel's doubly linked list of environments (`env_prev`, `env_next`). The `pages` array maintains the number of virtual page mappings to each physical page, or `1` for protected pages (*i.e.,* the `env_mypp` of some environment, explicitly marked by the `page_protected` array). The kernel ensures that an `env_pgdir` entry is not protected.

To verify that every live `env_pgdir` entry has a positive reference count we introduce: a single physical page Skolem variable, one auxiliary counter variable for each page, and an auxiliary invariant insisting mapped pages' auxiliary counters are positive. Model checking ensures the auxiliary counters are equal to JOS's reference counters. Given the Skolems and the auxiliary invariant, our tool proves the correct use of ref-

```
int yaffs_open(...) {
    ...
    h = yaffsfs_GetHandlePointer(...);
    obj = yaffsfs_FindObject(...);
    ...
    h->obj = obj; obj->inUse++;
    ...
}
```

```
void yaffs_close(...) {
    ...
    h = yaffsfs_GetHandlePointer(...);
    if (h && h->inUse) {
        h->obj->inUse--;
        if (h->obj->inUse <= 0)
            yaffs_DeleteFile(h->obj);
        h->obj = 0;
    }
}
```

Fig. 8. YAFFS reference counting, simplified.

erence counts for any number of pages and environments. (Memory leak freedom is proved by ensuring the values of the actual and auxiliary counters coincide.) The reachability analysis requires 17 predicates and 29 seconds.

Yaffs File Object Management. The YAFFS log-structured filesystem for flash memory [23] represents files with heap-allocated `yaffs_Object` structures, each containing a reference counting `inUse` field (Figure 8 shows fragments of a simplified version, although we have verified the actual implementation). Users access objects indirectly through the `obj` field of a `yaffs_Handle` pointer. The handles are stored in a fixed-sized array, indexed by an integer file handle descriptor.

File read and write operations access `yaffs_Objects` under the assumption that their reference counts are positive. To verify, we introduce a single file object Skolem variable. As done for JOS, we also introduce auxiliary state to track handle-object (un)mappings, and equate that state with actual object reference counts by symbolic model checking. Assuming that each file operation occurs atomically, our tool is able to prove the sound use of reference counts for any number of handles and objects. The reachability analysis requires 34 predicates and 36 seconds.

4 Related Work

Compositional Verification. Our use of temporal case splitting with Skolem variables is inspired by similar approaches in hardware verification [22], where a hardware design is decomposed into units of work and the finite instantiations verified using a BDD-based model checker. Our work differs from the above in two respects. First, we consider C programs where heap locations are allocated dynamically, and need not have static names. Second, by using predicate abstraction over more expressive theories (*e.g.,* equality, arithmetic, arrays) we may track relationships between variables, which is generally required to prove sound reference counting.

A restricted use of Skolem variables to separate a safety verification problem into sub-problems has been suggested before [30, 4]. However, an analysis with a dataflow analysis back-end [30] merges states at join points, and cannot perform case splits over the abstract domain used in our examples. There the benefits of separation were restricted to syntactically disjoint choices (for example, where there were separate assertions on two arrays, and the abstraction would first prove the assertion for the first array while abstracting the second, and then prove it separately for the second). We, on the other hand, perform case splitting on the temporal behavior of the program, thus correlating the choice of a Skolem at one point in the execution to a subsequent check.

Predicate Abstraction. Skolem variables have been used with predicate abstraction to infer universally quantified invariants over the program state [9, 19, 20]. However, the properties considered thus far have been limited, for the most part, to simple intraprocedural reasoning about arrays. We believe that one reason for this is that fast Cartesian predicate abstraction, implemented as the default in software model checkers such as SLAM or BLAST, is too coarse for reasoning about array and pointer variables. Our work builds on the interpolant-based transition relation refinement [15] that lazily refines the Cartesian abstraction to the required precision, and our reachability engine uses this refinement for scalability. The use of quantified predicates in model checking based on predicate abstraction [27, 7] has, for similar reasons, been limited to small and abstract encodings of complicated procedures (*e.g.,* garbage collectors). In these applications quantifiers are instantiated by matching heuristics implemented in the theorem prover, or manually. In contrast, our use of Skolems, while less powerful than full quantification, is more predictable, and does not rely on matching heuristics. While we concentrate on the technique of using predicate abstraction with Skolems and auxiliary state in reachability analysis, techniques to *infer* quantified predicates [20] are orthogonal, and can be combined with our algorithm to find such predicates. Finally, auxiliary invariants have been used in software model checking, *e.g.,* to approximate the shape of the heap using alias analysis [1, 13], or to infer polyhedral invariants in a prepass before applying predicate abstraction [14].

Shape Analysis. Shape analysis [26] and separation logic [25, 3] are powerful frameworks for reasoning about heap manipulating programs. While our techniques can be simulated inside shape analysis, our advantage is the use of already developed efficient and scalable predicate abstraction and manipulation engines (from BLAST) to reason about heap properties on large programs. Shape analysis has also been used to verify concurrent programs with an unbounded number of threads through the use of manually supplied instrumentation predicates [29]. Skolemization and case splitting are orthogonal—once they are performed, three valued logic based analyses can be used to discharge the reduced model checking tasks.

Work on canonical abstraction of arrays [10, 16] is close to our work: there, an (unbounded) array is abstracted with respect to an iterator into the portion of the array before, at, and after the iterator; these portions are summarized with respect to the predicates that hold on them. In contrast, our technique abstracts an array into the locations indexed by Skolems, and all other locations; additional refinements are introduced with additional Skolems and predicate relationships between values at the Skolem indices. Instead of a specialized dataflow analysis, we perform path-sensitive model checking that can then correlate data at Skolem locations. Our experience is that for properties that depend on an arbitrary element of the array, Skolemization and case splitting provides a more natural (and often, a more succinct) abstraction of the program.

References

1. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

2. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*. 2004.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
4. D. Beyer, A.J. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *SAS*, 2004.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*. 2000.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
7. S. Das and D.L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*, 2002.
8. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.
9. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.
10. D. Gopan, T.W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, 2005.
11. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*. 1997.
12. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL*, 2004.
13. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
14. H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, 2006.
15. R. Jhala and K.L. McMillan. Interpolant-based transition relation approximation. In *CAV*, 2005.
16. R. Jhala and K.L. McMillan. Array abstractions from proofs. In *CAV*. 2007.
17. JOS. Jos: An operating system kernel. http://pdos.csail.mit.edu/6.828/2005/overview.html.
18. V. Kuncak, P. Lam, K. Zee, and M.C. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Trans. Software Eng.*, 32(12):988–1005, 2006.
19. S.K. Lahiri and R.E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, 2004.
20. S.K. Lahiri and R.E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, 2004.
21. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
22. K.L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37:279–309, 2000.
23. Aleph One. Yaffs file system. http://www.yaffs.net/.
24. S. Qadeer and D. Wu. KISS: Keep it simple, sequential. In *PLDI*, 2004.
25. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
26. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.
27. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR*, 2000.
28. G. van Rossum. Debugging reference count problems. http://www.python.org/doc/essays/refcnt/.
29. E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *POPL*, 2001.
30. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *PLDI*, 2004.