Deciding Assertions in Programs with References

Shaz Qadeer Sriram K. Rajamani
Microsoft Research
{qadeer,sriram}@microsoft.com

September 2005

Technical Report
MSR-TR-2005-08

Modular analysis of procedures using summaries is a key technique to improve scalability of software model checking. Existing software model checkers do not fully exploit procedural structure for modular analysis. In the SLAM project, modular analysis using procedure summaries is done on a Boolean Program model, which contains only boolean types. We extend Boolean Programs to include reference types, and show that modular analysis using procedure summaries is still possible. As a consequence, we obtain an algorithm for deciding assertions in programs where the lengths of the paths in the heap are bounded, even though the heap size is potentially unbounded. Even in programs with unbounded paths in the heap, the result provides a way to separate reasoning about the finite backbone of the heap from the reasoning about unbounded data structures. We have implemented this algorithm in the ZING model checker, which supports a rich input language with references as well as concurrent threads. Our algorithm improved the performance of the model checker by 30-35% on a concurrent transaction management program with 7000 lines of code, 57 dynamic allocation sites, and several million reachable states and found a subtle concurrency bug. On parameterized examples artificially constructed to demonstrate the benefits of summarization, the algorithm improves performance asymptotically as expected. The implementation is robust —on hundreds of small examples in the SLAM and ZING regression suites, the implementation produces correct results.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Boolean Programs are programs in which all variables are boolean. They have been used successfully as a target for representing automatically extracted models from C programs in the SLAM project [4]. Boolean Programs are infinite-state systems since they can have recursive procedures, and the stack depth is unbounded. Regardless, assertion checking is still decidable for Boolean Programs. A common technique for analyzing such programs is CFL reachability [16, 11] (or equivalently, pushdown model checking [17, 6]), where the key idea is to build procedure summaries. The *summary* of a procedure P contains the state pair (s, s') if in state s, there is an invocation of P that yields the state s' on termination. Summaries enable modular analysis of large programs. Summaries enable reuse—if P is called from two different places with the same state s, the work done in analyzing the first call is reused for the second. This reuse is the key to scalability of interprocedural analyses. Additionally, summarization avoids direct representation of the call stack, and guarantees termination of the analysis even if the program has recursion.

In this paper, we extend Boolean Programs with references and show how we can still retain the benefits of modular analysis using procedure summaries. Analyzing such programs modularly is non-trivial, since we allow unbounded dynamic allocations of objects on the heap, and arbitrary aliasing of objects. A key insight we have is that even with all the extra complexities we can still construct a summary for every invocation of a procedure, that is a pair consisting of (1) the *visible* state that is reachable from the procedure through globals and the formal parameters, and (2) the *effect* that the procedure has on this visible state, which could involve changing some values and allocating new objects and linking them to the visible state. Typically, a procedure does not make use of its entire visible state during its execution. We can generalize the notion of visible state to a *pattern*, which is the subset of the visible state that is actually observed by a procedure during its execution, and both generate fewer summaries, and obtain better re-use of generated summaries.

We define an equivalence relation that relates "similar" visible states. We prove that the index of this equivalence relation is bounded if all paths in the heap are of bounded length, thereby yielding an algorithm to decide assertions in such programs. Note that this result is non-trivial since bounded path length does not mean bounded heap —executions of the program can still have unbounded call stacks and can potentially allocate unbounded memory (see Section 3 for an example). Certain natural syntactic restrictions on the program provide sufficient conditions for bounded path lengths on the heap. For example, if all the reference types in the program are non-recursive, the type structure guarantees that all paths in the heap will be bounded.

Even for programs with recursive data types and unbounded paths in the heap, the result is still useful. For such programs, we can partition the heap into a *backbone* component, which contains paths of bounded length, an an *unbounded* component which contains data structures such as lists or trees. By viewing the unbounded data structures as collections, sets or arrays we can use predicate-abstraction or other finitary abstractions and represent these unbounded components using finitary representations. Thus, we can retain the backbone components "as is", view the unbounded components as collections and abstract them into finitary representations, and use the algorithm in this paper to decide assertions in such programs. Other analyses can be used to prove that the code for the linked lists correctly implements the collection or set abstraction [15]. Since the relevant references on the backbone part are present in the extracted model, all aliasing queries are resolved with full precision on-the-fly during model checking. This feature of our analysis obviates the need for a coarse apriori pointer analysis while doing predicate abstraction as in [2, 7]. A number of iterations in the refinement loop are wasted in discovering extra aliasing predicates to regain the precision lost by the static pointer analysis. These iterations can be avoided making the analysis much more efficient.

In a recent paper [10], we show to use the idea of transactions to build procedure summaries for concurrent programs. However, the work reported in [10] does not deal with reference data-types, and no implementation was presented. By combining the results of this paper with the results from [10], we have implemented a summarization algorithm in ZING, a software model checker being developed in Microsoft Research, for programs that have no restrictions on reference data types or concurrency. Though termination is guaranteed only when the path length in the heap is bounded, base types are finite domain, and recursive procedures are "transactional" as defined by [10], we find that the implementation terminates on several cases and outperforms the model checker without summarization. We obtained a concurrent transaction management program obtained from a product group at Microsoft. The program has recursive data types, but all paths in the heap are bounded. We found that the the model checker with summarization outperforms the model checker without summarization by 30-35%. On parameterized examples artificially constructed to demonstrate the benefits of summarization, the algorithm improves performance asymptotically as expected. The implementation is robust as evidenced by correct results on hundreds of small examples in the SLAM and ZING regression suites.

To summarize, this paper has two contributions:

- We present a new model checking algorithm for deciding assertions in programs with references. Our algorithm terminates and yields precise results even on programs that allocate unbounded amount of memory, as long as the paths in the heap have bounded length.
- We combine the above result, with another result we presented in [10] and implement a general summarization algorithm in the ZING model checker, for programs that that have no restrictions on reference data types or concurrency. Our earlier paper [10] did not deal with references, and did not have an accompanying implementation. We present details and experiments from this implementation.

2 Related work

Interprocedural analyses based on context-free reachability [11] have recently been used in error-detection tools such as SLAM [4] and ESP [5]. SLAM uses an alias-analysis to first conservatively abstract a C program to a Boolean Program (a program without references), and ESP uses value-flow analysis and bit-vectorization to conservatively partition the analysis problem into separate problems, one each per distinct value. Imprecision in alias analysis and value flow analysis can lead to false errors in both approaches. In the case of SLAM some of these false errors can be eliminated using abstraction-refinement, where some extra predicates are added to keep track of specific aliasing relationships more precisely. The treatment of pointers in this paper differs from both these approaches. We show that for models with bounded paths on the heap and finite base types, we can decide assertions interprocedurally

without losing any precision.

In the compiler community, extensive work has been done in the area of pointer analysis(see [8] for an assessment of the state-of-the-art). In particular, prior work on context-sensitive pointer analyses have investigated methods to do interprocedural pointer analysis using partial transfer functions (PTFs) [20], which bear some similarity to the patterns and effects used in this paper. By cloning information at every calling context, and using Binary Decision Diagrams to represent the sharing between various contexts, context-sensitive pointer analyses have been recently made to scale on very large programs [19]. These analyses lose precision to enable scaling, and are mostly flow-insensitive. Though some of our techniques are inspired by such work, our goals and results are qualitatively different. We want to extract a model from a large program, which captures only relevant variables and pointers that are of interest to prove a particular property. Once we construct such a model, we want to decide assertions in this model *without* losing any precision. We do not know of any prior work that precisely decides assertions on possibly recursive programs with bounded path lengths on the heap, and possibly unbounded number of allocations. The algorithm and implementation reported in this paper achieve this result.

In the model checking community, researchers have built model checkers that operate over concurrent heap-manipulating programs written in common programming languages such as Java [18, 9, 14]. None of these model checkers exploit the procedural structure of the program for efficiency in model checking. The model checker BEBOP [3] from the SLAM project exploits summarization in the simpler setting of boolean program models. The algorithm presented in this paper is a generalization of BEBOP to handle models with references. In a prior unpublished paper, Thomas Ball proposed extending Boolean Programs with references and extending BEBOP to handle these extended programs symbolically [1]. He also conjectured that the assertion checking problem is decidable for this extension.

Sagiv et al. [15] have developed an abstract-interpretation framework based on 3-valued logic to reason about heap-manipulating programs. Recent work by Rinetzky et al. [12, 13] has combined the idea of visible states with 3-valued logic to build an interprocedural shape analysis. Their work identified the novel concept of *cut-points*, which are heap cells reachable in the visible state of a called procedure that are reachable from the local variables of the calling procedure. The presence of cutpoints is the main obstacle to performing interprocedural analysis in the presence of a heap. There are technical and algorithmic differences between their work and ours. The technical difference is that they use a storeless semantics whereas we use a store-based semantics for the program. The algorithmic difference is that Rinetzky et al. compute cutpoints eagerly whenever a procedure call happens, whereas we identify cutpoints lazily as we build the summary of the called procedure. In addition, we introduce the novel implementation technique of patterns (See Section 6). Our experimental results show that the use of patterns is crucial for the scalability of the analysis.

3 Overview

In this section, we informally introduce the main ideas of this paper using the example program shown in Figure 1. Inside procedure `M`, at line `L0`, a new object is allocated and assigned to local variable `f`. Then, a nondeterministic choice is made at line `L1`, and in one of the choices, `f.x` is assigned the value of `g1.x`, and then the global `g1` is made to point to the local object created at line `L0`, and pointed-to by `f`. This is followed by a recursive call to `M`. The other choice just terminates `M` and returns. This program can allocate an unbounded amount of memory since there is an execution that always chooses to take the "if" branch of the nondeterministic choice at line `L1` and creates an unbounded stack, allocating an unbounded number of objects each pointed-to by a local variable from a stack frame.

```
class BoolBox {
  bool x;
  //constructor
  BoolBox() {
  x = true;
  }
 }
};
```

```
class Main {
static BoolBox g1, g2;
activate
 static void main() {
 g1 = new BoolBox();
 g2 = g1;
 g1.x = false;
 M();
 assert((g1.x == false)
       && (g2.x == false));
 }
static void M() {
  Foo f;
  L0: f = new BoolBox();
  L1: if(*) {
  L2:  f.x = g1.x;
  L3:  g1 = f;
  L4:  M();
      }
 }
};
```

Figure 1. Example program which can allocate potentially unbounded memory

Visible states and effects. The state of a program contains the globals, the stack and the heap. To do modular analysis of a program, it is useful to consider the notion of visible state of a program with respect to a particular invocation of a procedure (i.e., a stack frame). The visible state of a program consists of the locals, formals, globals in the current stack frame and the subset of the heap that is reachable from the locals, formals and globals. Thus, heap addresses that are only reachable from other stack frames such as the caller, or the caller's caller are not part of the visible state.

Consider the invocation of `M` in procedure `main` from the example. The visible state S_1 of `M` at this invocation consists of `g1`, `g2` and the single heap cell that they both point to, which is of type `BoolBox` and has its `x` field set to false. Let us call the address of the heap cell as `A0`. We will represent visible states by a set of address-value pairs. For example, the visible state described above is represented by: S_1 = {(g1,A0), (g2,A0), (⟨A0,x⟩,false)}.

Two visible states are *equivalent* if they differ in only the actual address of the heap cells, and are indistinguishable otherwise, in terms of aliasing or values of base-types in the state. For example consider the visible state S_2 = {(g1,A1), (g2,A1), (⟨A1,x⟩,false)}. Then, S_1 and S_2 are equivalent. However, the visible state S_3 = {(g1,A0), (g2,A2), (⟨A0,x⟩,false), (⟨A2,x⟩,false)} is not equivalent to S_1 since the aliasing relationship between `g1` and `g2` is different in S_1 and S_3.

Even though the number of heap cells allocated by a program could be unbounded, the number of non-equivalent visible states for a procedure invocation has to be finite if the base types are boolean and the length of the paths in the heap are bounded. This notion is made precise in Sections 4 and 5, and is crucial for our termination theorem (Theorem 3 in Section 5). For example, if we consider all the (unbounded number of) invocation contexts of procedure `M` in

```
class Main {
static int x;
static int y;
static int z;

 activate
  static void main(){
  M0: x = 0; y = 0; z = 0;
  M1: foo();
  M2: y = 1;
  M3: foo();
 }
```

```
static void foo() {
 L1: if(*) {
 L2:  assume(x == 0);
 L3:  z = 1;
      }
      else {
 L4:  assume(y == 1);
 L5:  assert(false);
     }
 }
};
```

Figure 2. Example program to illustrates unsoundness with naive usage of patterns

Figure 1 every visible state is equivalent to either S_1 or S_3— the visible state is equivalent to S_1 for the call made to `M` from `main`, and the visible state is equivalent to S_3 for each of the unbounded number of recursive calls made to `M` at line `L4`.

An *effect* is a function from visible states to visible states. A *summary* of a procedure P is a state pair (S,e), where S is a visible state and e is an effect. Intuitively, $e(S)$ represents a possible visible state at termination of procedure P if the procedure is invoked at visible state S. More concretely, an effect e is represented as a pair (as,m) where as is a set of addresses that represent object allocations, and m is a set of updates. In order to apply an effect $e = (as,m)$ on a state S, one first allocates objects at addresses from as in S and then performs the updates prescribed by m.

For example, if `M` is invoked at visible state S_1 = `{(g1,A0), (g2,A0), (⟨A0,x⟩,false)}`, the procedure `M` can have three different behaviors: (1) it can generate an empty effect e_1 = `({},{})`, which represents the case where the "if" branch is not taken, and the final visible state at the exit of procedure `M` is the same as the visible state on entry, or (2) it can generate a effect e_2 = `( {A1},{(g1,A1),(⟨A1,x⟩,false)} )` , where `A1` is the address of a newly allocated object, and the pair `(g1,A1)` denotes that `g1` is updated to hold the value `A1`, and the pair `(⟨A1,x⟩,false)` denotes the value of the `BoolBox` object at address `A1`, or (3) it can enter an infinite recursion and never return. We do not generate summaries for non-terminating executions since we are checking for safety properties only. Thus, for the visible state S_1 we have two summaries for procedure `M`, namely $\{(S_1,e_1),(S_1,e_2)\}$. The Algorithm in Section 5 shows how these two summaries (and only these two summaries) are computed for `M`.

An invocation to `M` at S_3 = `{(g1,A0), (g2,A2), (⟨A0,x⟩,false), (⟨A2,x⟩,false)}` also can generate the same three behaviors as the ones for S_1. Thus the summaries of `M` are given by the finite set: $\{(S_1,e_1),(S_1,e_2),(S_3,e_1),(S_3,e_2)\}$. Since any invocation to `M` happens at a visible state equivalent to either S_1 or S_3, these summaries can be used to generate all possible visible states at the exit of `M`, without descending into the body of `M`. Applying the effects of these summaries lets us decide that the assertion after the call to `M` in `main` can never get violated.

Patterns. Often, a procedure does not make use of its entire visible state during its execution. In such cases, it is useful to generalize the notion of visible state to a *pattern*, which is the subset of the visible state observed by the procedure `M` during its execution.

A pattern Θ is a set of visible states. It is tempting to generalize a summary to be a pair (Θ,e), where Θ is a pattern and e is an effect. However, this leads to unsoundness in the decision procedure as illustrated by the example in Figure 2. In the example, the procedure `main` calls procedure `foo` twice. The first call is made at line `M1` with visible state SS_1 = `{(x,0), (y,0), (z,0)}`. Inside `foo` we have a non-deterministic branch at line `L1`, which has two targets `L2` and `L3`. The behavior at `L2` passes the assume statement `assume(x == 0)` This results in a summary (Θ_1,ee_1) where the pattern Θ_1 = `{(x,0)}`, an the effect ee_1 = `{(z,1)}`. Thus, the pattern records that the value 0 is read from variable `x` and the effect records that the value 1 is written into variable `z`. The other behavior at `L3` is pruned silently since the assume statement `assume(y == 1)` fails. During the second call to procedure `foo` from line `M3`, the visible state is SS_2 = `{(x,0), (y,1), (z,1)}` . This state matches the pattern Θ_1 = `{(x,0)}`, and if the summary (Θ_1,ee_1) is applied without further analysis inside procedure `foo` we get a single successor state SS_3 = `{(x,0), (y,1), (z,1)}` after returning from `foo`. However, this misses the assertion failure in line `L5` since the second call is made from a state where variable `y` has value 1, and thus can fail the assertion. The reason for the unsoundness is that the pattern Θ_1 does not capture the values of the variables read in *all* the non-deterministic paths inside `foo`.

We therefore generalize a summary to be a pair (Θ,E) where Θ is a pattern over a visible state and E is a set of effects. Intuitively, for such summaries to be sound, the pattern Θ should include all values read over all the non-deterministic behaviors inside the function starting from a given visible state and the set E should include all the resulting effects. Thus, the correct summary for the call to `foo` at line `L1` should be (Θ_1,E) where Θ_1 = `{ (x,0), (y,0) }` , and $E = \{ee_1\}$, is a singleton set with only one effect, since the behavior at line `L3` is pruned. However, even though the behavior is pruned, the fact that value 0 was read from variable `y` is still recorded as part of the pattern. Thus, during the second call to procedure `foo` from line `M3`, with visible state SS_2 = `{ (x,0), (y,1), (z,1) }` , we find that the pattern Θ_2 = `{ (x,0), (y,0) }` does not match the visible state since the pattern requires variable `y` to have the value 0. Thus, procedure `foo` is re-analyzed from this visible state and the assertion failure is detected.

Consider again, the procedure `M` from Figure 1. The value of the global variable `g2` is never observed by procedure `M`. Thus, the portion of the visible state S_1 that is observed by `M` is given by Θ_3 = `{ (g1,A0), (⟨A0,x⟩,false) }`. Similarly, the portion of the visible state S_3 that is observed by `M` is given by Θ_4 = `{ (g1,A0), (⟨A0,x⟩,false) }`. Even though the visible states S_1 and S_3 are not equivalent, the patterns Θ_3 and Θ_4 are equivalent. Thus, the same set of behaviors will be generated by executing `M` from these two visible states. With this generalization, the procedure `M` in Figure 1 has summaries $\{(\Theta_3,\{e_1,e_2\})\}$. By using this generalization, we were able to generate fewer summaries than our earlier representation $\{(S_1,e_1),(S_1,e_2),(S_3,e_1),(S_3,e_2)\}$.

4 Definitions

A state of a program is a 3-tuple (h,l,s), consisting of a heap h, a local store l, and a stack s. The heap h is a collection of cells, each of which has a unique address and contains a finite set of fields. Formally, the heap h is a partial function mapping addresses to a function that maps fields to values. Given address a and field f, the value stored in the field f of cell with address a is denoted by $h(a,f)$. Let $|h|$ be the largest element in *Addr* on which h is defined. The local store l is a valuation to local variables, and the stack s is a sequence of local stores. A field of a particular cell is called a loca-

tion. Each variable or location has a unique type, either boolean or reference. A variable or location of boolean type contains a boolean value and of reference type contains either *null* or the address of a cell.

The behavior of a program is completely specified by the following entities:

1. A control flow graph $C \subseteq PC \times Action \times PC$.
2. The initial program counter $pc_I \in PC$.
3. The initial local store $l_I : LocalVar \to Value$ that assigns *false* to each variable of boolean type and *null* to each field of reference type.
4. The object initialization function $\lambda_I : Field \to Value$ that assigns *false* to each field of boolean type and *null* to each field of reference type.
5. The initial heap $h_I : Addr \to Field \to Value$ that is defined only at the special address *globals* such that $h_I(globals) = \lambda_I$.

Domains

$$
\begin{array}{rcrcl}
b & \in & Bool & = & \{true, false\} \\
a & \in & Addr & = & \{globals\} \cup \{1,2,\ldots\} \\
i,j & \in & PC & & \\
v & \in & Value & = & PC \cup Addr \cup Bool \cup \{null\} \\
f & \in & Field & & \\
x & \in & LocalVar & & \\
\alpha & \in & Expr & = & globals \mid null \mid false \mid x \mid x.f \\
 & & & & \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 = \alpha_2 \\
 & & Action & = & x = new \mid x = \alpha \mid x.f = \alpha \\
 & & & & \mid assume(x) \mid call \mid return \\
h & \in & Heap & = & Addr \to Field \to Value \\
\ell, \langle i,l \rangle & \in & Local & = & PC \times (LocalVar \to Value) \\
s & \in & Stack & = & Local^* \\
\langle h, \ell \rangle & \in & VisibleState & = & Heap \times Local \\
(h,\ell,s) & \in & State & = & Heap \times Local \times Stack \\
as & \in & Addrs & = & Powerset(Addr) \\
loc & \in & Location & = & Addr \times Field \\
r,w & \in & Locations & = & Powerset(Location) \\
m & \in & LocationMap & = & Location \to Value \\
e & \in & Effect & = & Addrs \times LocationMap
\end{array}
$$

We formalize the semantics of a program as a tuple $\langle T, T^+, T^- \rangle$ of three relations:

$T \subseteq VisibleState \times Locations \times Addrs \times Locations \times VisibleState$
$T^+ \subseteq VisibleState \times Local$
$T^- \subseteq VisibleState$

The relation T models steps that do not manipulate the stack. $T(h,\ell,r,as,w,h',\ell')$ holds whenever the program executes an action in the visible state $\langle h,\ell \rangle$, reads locations in r, allocates new heap cells with addresses in as, writes locations in w, and modifies the state to $\langle h',\ell' \rangle$. Note that the set of locations r and w respectively record reads from and writes to fields in the heap but not the local variables. The relation T^+ models a procedure call. $T^+(h,\ell,\ell')$ holds whenever the program executes a call action in the visible state $\langle h,\ell \rangle$ modifying the local store in the current stack frame to ℓ', and pushing a fresh stack frame $\langle pc_I, l_I \rangle$. The relation T^- models a procedure return. $T^-(h,\ell)$ holds whenever the program executes a return action in the visible state $\langle h,\ell \rangle$. This action pops the current stack frame. Both the push and pop actions leave the heap unchanged.

We use the notation $\alpha[h,l]$ to denote the value of the expression α when evaluated over the visible state $\langle h,l \rangle$ and $R(\alpha)[\![h,l]\!]$ to denote the set of locations read during the evaluation of α in $\langle h,l \rangle$. The definition of $R(\alpha)[\![h,l]\!]$ is as follows:

$$
\begin{array}{rcl}
R(null)[\![h,l]\!] & = & \emptyset \\
R(false)[\![h,l]\!] & = & \emptyset \\
R(x)[\![h,l]\!] & = & \emptyset \\
R(x.f)[\![h,l]\!] & = & \{\langle l(x), f \rangle\} \\
R(\neg\alpha)[\![h,l]\!] & = & R(\alpha)[\![h,l]\!] \\
R(\alpha_1 \vee \alpha_2)[\![h,l]\!] & = & R(\alpha_1)[\![h,l]\!] \cup R(\alpha_2)[\![h,l]\!] \\
R(\alpha_1 = \alpha_2)[\![h,l]\!] & = & R(\alpha_1)[\![h,l]\!] \cup R(\alpha_2)[\![h,l]\!]
\end{array}
$$

The formal definitions of T, T^+ and T^- are given below:

Transition relation

$$
\frac{(i, x = new, j) \in C \quad x \in LocalVar \quad h(a) \text{ is undefined} \quad l' = l[l(x) := a]}{T(h,\langle i,l \rangle, \emptyset, \{a\}, \emptyset, h[a := \lambda_I], \langle j,l' \rangle)} \quad \text{(ALLOCATE)}
$$

$$
\frac{(i, x = \alpha, j) \in C \quad x \in LocalVar \quad r = R(\alpha)[\![h,l]\!] \quad v = \alpha[h,l] \quad l' = l[x := v]}{T(h,\langle i,l \rangle, r, \emptyset, \emptyset, h, \langle j,l' \rangle)} \quad \text{(WRITELOCAL)}
$$

$$
\frac{(i, x.f = \alpha, j) \in C \quad x \in LocalVar \quad r = R(\alpha)[\![h,l]\!] \quad v = \alpha[h,l] \quad h' = h[\langle l(x), f \rangle := v]}{T(h,\langle i,l \rangle, r, \emptyset, \{\langle l(x), f \rangle\}, h', \langle j,l \rangle)} \quad \text{(WRITEHEAP)}
$$

$$
\frac{(i, assume(x), j) \in C \quad l(x) = true}{T(h,\langle i,l \rangle, \emptyset, \emptyset, \emptyset, h, \langle j,l \rangle)} \quad \text{(CONDITIONAL)}
$$

$$
\frac{(i, call, j) \in C}{T^+(h,\langle i,l \rangle, \langle j,l \rangle)} \quad \text{(CALL)}
$$

$$
\frac{(i, return, j) \in C}{T^-(h,\langle i,l \rangle)} \quad \text{(RETURN)}
$$

The program starts execution in the state $(h_I, \langle pc_I, l_I \rangle, \varepsilon)$, where h_I is the initial heap, l_I is the initial local store, and ε is the initial empty stack. Let ℓ_I denote the pair $\langle pc_I, l_I \rangle$. When the program makes a transition, its state is updated according to the transition executed in T, T^+ or T^-. The operational semantics $\longrightarrow$ of the program is formally defined as follows:

Operational semantics

$$
\frac{T(\langle h,\ell \rangle, r, as, w, \langle h',\ell' \rangle)}{(h,\ell,s) \longrightarrow (h',\ell',s)} \quad \text{(STEP)}
$$

$$
\frac{T^+(\langle h,\ell \rangle, \ell')}{(h,\ell,s) \longrightarrow (h,\ell_I, s \cdot \ell')} \quad \text{(PUSH)}
$$

$$
\frac{T^-(\langle h,\ell \rangle))}{(h,\ell,s \cdot \ell') \longrightarrow (h,\ell',s)} \quad \text{(POP)}
$$

For each visible state $\langle h,\ell \rangle$, let $Cells(h,\ell)$ be the set of addresses of reachable cells. Formally, $Cells(h,\ell)$ is the least set of addresses satisfying the following conditions: (1) $globals \in Cells(h,\ell)$. (2) if $\ell(x) \in Addr$, then $\ell(x) \in Cells(h,\ell)$. (3) if $f \in Field$, $a \in Cells(h,\ell)$, and $h(a,f) \in Addr$, then $h(a,f) \in Cells(h,\ell)$.

A visible state $\langle h, \ell \rangle$ is called *garbage-free* if $dom(h) = Cells(h, \ell)$. Let $gc(h, \ell)$ denote the garbage-free visible state $\langle h', \ell \rangle$ where h' is h restricted to $Cells(h, \ell)$.

A visible state $\langle h, \ell \rangle$ is *well formed* if h is defined for all addresses in $Cells(h, \ell)$. A state (h, ℓ, s) is *well formed* if the heap h is defined for all address values that are reachable from the or local variables in all the stack frames in s. In the following, we assume that all states and all visible states are well formed.

A partial function $\rho : Value \to Value$ is a *permutation* for a heap h if $\rho(v) = v$ whenever $v \in PC \cup Bool \cup \{null, globals\}$, and for any $a \in Addr$, if $h(a)$ is defined, we have that $\rho(a)$ is defined. If h_1 and h_2 are two heaps and ρ_1 is a permutation for h_1, and ρ_2 is a permutation for h_2, we say that ρ_2 extends ρ_1 if for all a in the domain of h_1, we have that $\rho_1(a) = \rho_2(a)$.

For any permutation ρ, we define

$$\begin{array}{rcl} \rho(\ell) & = & \lambda x \in LocalVar.\ \rho(\ell(x)) \\ \rho(h) & = & \lambda a \in Addr.\, f \in Field.\ \rho(h(\rho^{-1}(a), f)) \end{array}$$

Let $\langle h_1, \ell_1 \rangle$ and $\langle h_2, \ell_2 \rangle$ be garbage-free visible states. Then $\langle h_1, \ell_1 \rangle \equiv \langle h_2, \ell_2 \rangle$, if there exists a permutation ρ such that $\langle \rho(h_1), \rho(\ell_1) \rangle = \langle h_2, \ell_2 \rangle$. Clearly, the relation $\equiv$ is an equivalence relation and partitions the set of garbage-free visible states into a set of equivalence classes. We extend ρ to visible states and define $\rho(\langle h, \ell \rangle) = \langle \rho(h), \rho(\ell) \rangle$. For each state $\langle h, \ell \rangle$, we fix a unique representative which can, for example, be obtained by performing a depth-first search on the heap graph of $\langle h, \ell \rangle$ with l as roots and renaming the index of each heap cell to its depth-first number. Let Λ be the function that maps each visible state $\langle h, \ell \rangle$ to the unique representative of $gc(h, \ell)$. We call Λ the *canonizing function* for the sequential program.

Given a set of addresses as, we define $\rho(as) = \{\rho(a) \mid a \in as\}$. Given a set of locations w, we define $\rho(w) = \{\langle \rho(a), f \rangle \mid \langle a, f \rangle \in w\}$.

We note that transitions are preserved under permutations, which is a key property of programs. To state this formally, we define permutations over stacks as well. If ρ is a permutation and stack s is a sequence $\ell_1, \ell_2, \ldots, \ell_n$ of local variables, then $\rho(s)$ is the sequence $\rho(\ell_1), \rho(\ell_2), \ldots, \rho(\ell_n)$.

LEMMA 1. *for any transition $\langle h, l, s \rangle \longrightarrow \langle h', l', s \rangle$, and for any permutation ρ for h', there exists a permutation ρ' for h', such that ρ' extends ρ, and $\langle \rho(h), \rho(l), \rho(s) \rangle \longrightarrow \langle \rho'(h'), \rho'(l'), \rho'(s) \rangle$.*

5 Algorithm

In this section, we present our algorithm for procedure summarization in the presence of references. Our algorithm uses the relations T, T^+, and T^- to perform a fixpoint computation over the following relations:

$$\begin{array}{rcl} P & \subseteq & VisibleState \times Addrs \times Locations \times VisibleState \\ P^+ & \subseteq & VisibleState \times Addrs \times Locations \times VisibleState \\ \\ Sum & \subseteq & Heap \times Addrs \times LocationMap \\ \\ Q & \subseteq & VisibleState \times VisibleState \\ Q^+ & \subseteq & VisibleState \times VisibleState \\ R & \subseteq & VisibleState \end{array}$$

The relation P is analogous to the set of "path edges" in interprocedural dataflow analyses [11]. $P(h, \ell, as, w, h', \ell')$ holds if there is an execution from $\langle h, \ell \rangle$ to $\langle h', \ell' \rangle$ along which as is the set of allocated addresses that are still reachable from $\langle h', l' \rangle$ and w is the set of locations that were written to, and are reachable either from $\langle h, \ell \rangle$ or from $\langle h', \ell' \rangle$. In this section, we do not aggregate the set r of read locations and consequently the r parameter in T, T^+, and T^- is not used. We will show how to use r in the next section to do a further important optimization.

Algorithm

(INIT)

$$\frac{}{\begin{array}{c} P(h_I, \ell_I, \emptyset, \emptyset, h_I, \ell_I) \\ Q(h_I, \ell_I, \Lambda(h_I, \ell_I)) \end{array}}$$

(STEP)

$$\frac{\begin{array}{c} P(h_1, \ell_1, as, w, h_2, \ell_2) \\ T(h_2, \ell_2, r', as', w', h_3, \ell_3) \\ as'' = (as \cup as') \cap Cells(h_3, l_3) \\ w'' = \{\langle a, f \rangle \in w \cup w' \mid a \in Cells(h_1, l_1) \cup Cells(h_3, l_3)\} \\ \delta = \Gamma(w'') \setminus as'' \\ \neg Q(h_1, \ell_1, \Lambda(h_3, \ell_3 + \delta)) \end{array}}{\begin{array}{c} P(h_1, \ell_1, as'', w'', h_3, \ell_3) \\ Q(h_1, \ell_1, \Lambda(h_3, \ell_3 + \delta)) \end{array}}$$

(PUSH)

$$\frac{\begin{array}{c} P(h_1, \ell_1, as, w, h_2, \ell_2) \\ T^+(h_2, \ell_2, \ell_3) \\ \delta = \Gamma(w) \setminus as \\ \neg Q^+(\Lambda(h_1, \ell_1), \Lambda(h_2, \ell_3 + \delta)) \end{array}}{\begin{array}{c} P^+(h_1, \ell_1, as, w, h_2, \ell_3) \\ Q^+(h_1, \ell_1, \Lambda(h_2, \ell_3 + \delta)) \end{array}}$$

(START SUM)

$$\frac{P^+(h_1, \ell_1, as, w, h_2, \ell_2) \quad \neg R(\Lambda(h_2, \ell_I))}{P(h_2, \ell_I, \emptyset, \emptyset, h_2, \ell_I) \quad R(\Lambda(h_2, \ell_I))}$$

(POP)

$$\frac{\begin{array}{c} P(h_1, \ell_1, as, w, h_2, \ell_2) \\ T^-(h_2, \ell_2) \\ m = Map(w, h_2) \end{array}}{Sum(h_1, as, m)}$$

(USE SUM)

$$\frac{\begin{array}{c} P^+(h_1, \ell_1, as, w, h_2, \ell_2) \quad Sum(h_3, as', m') \\ \rho(gc(h_3, \ell_I)) = gc(h_2, \ell_I) \\ (as'', w'', h_4) = apply(\langle as', m' \rangle, \rho, h_2) \\ as''' = (as \cup as'') \cap Cells(h_4, l_2) \\ w''' = \{\langle a, f \rangle \in w \cup w'' \mid a \in Cells(h_1, l_1) \cup Cells(h_4, l_2)\} \\ \delta = \Gamma(w''') \setminus as''' \\ \neg Q(h_1, \ell_1, \Lambda(h_4, \ell_2 + \delta)) \end{array}}{\begin{array}{c} P(h_1, \ell_1, as''', w''', h_4, \ell_2) \\ Q(h_1, \ell_1, \Lambda(h_4, \ell_2 + \delta)) \end{array}}$$

The relation P^+ denotes those path edges that end in a procedure call. The relation *Sum* is analogous to the set of "summary

edges" [11]. $Sum(h, as, m)$ holds if there is an execution that begins in $\langle h, \ell_I \rangle$, allocates the addresses in *as* and updates the heap according to *m*. A pair $\langle as, m \rangle$ is called an *effect*.

The relations Q and Q^+ contain canonized representations of the edges in P and P^+ respectively. These last two relations are crucial for the termination of the algorithm.

Let Γ be a function that maps a set of locations to a set of heap addresses referred in the set. For example $\Gamma(\{1.f, 2.g\}) = \{1, 2\}$. If l is a set of local variables, and δ is a set of heap addresses, we use $l + \delta$ to denote an augmentation of the local variables with a local data structure containing the set of heap addresses. During canonicalizaiton and garbage collection, if l is a set of local variables, *as* is a set of locally allocated addresses, and w is a set of written locations, we augment the local variables to $l + (\Gamma(w) \setminus as)$ so as to take into account written addresses that have been potentially made garbage due to updates.

Our algorithm is specified as a set of rules for performing a fixpoint computation over the relations mentioned above. To ensure that the fixpoint terminates, we also compute the canonical representative of each new edge generated by the algorithm. Whenever a new edge $(h, \ell, as, w, h', \ell')$ is added to P, its canonical representative $(\Lambda(h, \ell), \Lambda(h', \ell', \Gamma(w) \setminus as))$ is added to Q. Similarly, whenever a new edge $(h, \ell, as, w, h', \ell'$ is added to P^+, its canonical representative $(\Lambda(h, \ell), \Lambda(h', \ell', \Gamma(w) \setminus as))$ is added to Q^+.

Recall that h_I is the initial heap and ℓ_I is the initial local store. The fixpoint computation is kicked off by an application of the first rule (INIT), which adds the edge $(\langle h_I, \ell_I \rangle, \langle \emptyset, \emptyset \rangle, \langle h_I, \ell_I \rangle)$ to P. The rule (STEP) extends an edge in P by exploring a transition. The new edge generated is added to P only if its canonical representative is not already present in Q. The rule (PUSH) is similar to the rule (STEP) and generates an edge in P^+ if the canonical representative of that edge is not present in Q^+. The rule (START SUM) starts off a fresh summary computation in the called procedure.

The rule (POP) creates a procedure summary edge in *Sum*. This edge consists of the heap h_1 and an effect $\langle as, m \rangle$ that describes the updates to the global variables and the heap. The function *Map* takes as arguments a set of locations and a heap. It returns a location map obtained by looking up in the heap the values of the locations in the set.

The rule (USE SUM) is the most complicated rule and deals with the application of a summary edge in *Sum* at a call site. A summary edge is applicable if the heap and global store at its source is isomorphic to the heap and global store at the call site. Suppose ρ is the witness to the equivalence. The function *apply* is used to apply the summary. The operation *apply* takes as input an effect $\langle as, m \rangle$, a permutation ρ, and a heap h. It returns a set of addresses as', a set of locations w', and a heap h' created by performing the following operations in sequence.

1. Let $as' = \{|h|+1, \ldots, |h|+|as|\}$. Extend h to as' so that for all $a \in as'$ and field $f \in Field$, if f has type boolean then $h(a, f) =$ *false* otherwise $h(a, f) =$ *null*.
2. Extend ρ to ρ' so that ρ' maps *as* one-one onto as'.
3. For each address a and field f such that $m(a, f)$ is defined, update $h(\rho'(a), f) = \rho'(m(a, f))$. Let the final heap be h'.
4. Let $w' = \{\langle \rho'(a), f \rangle \mid m(a, f)$ *is defined*$\}$

We now present theorems that establish the correctness of our algorithm.

THEOREM 1 (SOUNDNESS). *If* $(h_I, \ell_I, \varepsilon) \longrightarrow^* (h, \ell, s)$, *then there exist* $\langle h', \ell' \rangle$, *and* δ *such that* $\langle h', \ell' \rangle \equiv \langle h, \ell \rangle$ *and* $Q(h_1, \ell_I, h', \ell' + \delta)$.

THEOREM 2 (COMPLETENESS). *If* $Q(h_I, \ell_I, h', \ell' + \delta)$, *then there exist* $\langle h, \ell \rangle$ *and* s *such that* $\langle h, \ell \rangle \equiv \langle h', \ell' \rangle$ *and* $(h_I, \ell_I, \varepsilon) \longrightarrow^* (h, \ell, s)$.

The proofs of these theorems can be found in the appendix.

We now present the argument for the termination of our algorithm. This argument requires the notion of k-boundedness for some non-negative number k. A visible state $\langle h, \ell \rangle$ is k-bounded if the longest chain of references starting from a global or a local variable has length at most k. Although the set of k-bounded visible states is unbounded, this unbounded set is partitioned into a finite set of equivalence classes by the relation $\equiv$. This observation forms the crux of the argument for the termination of our algorithm.

A sequential program $\langle T, T^+, T^- \rangle$ is k-bounded if whenever $\langle h, \ell \rangle$ is k-bounded and $T(h, \ell, r, as, w, h', \ell')$, then $\langle h', \ell' \rangle$ is k-bounded.

Consider a sequential program all of whose base types have finite domains and all of whose reference types are non-recursive. It is easy to show that such a program is k-bounded for some finite number k that can be determined from the static type structure of the program. We can now state our termination theorem.

THEOREM 3 (TERMINATION). *If the sequential program* $\langle T, T^+, T^- \rangle$ *is k-bounded, then the fixpoint computation specified by the rules described above terminates.*

6 Patterns

In this section, we describe an optimization to the algorithm in Section 5, where we generalize the visible state in summaries to a pattern, which is the subset of the visible state that is actually observed during execution. We generalize summaries to be pairs (Θ, E) where Θ is a pattern over a visible state and E is a set of effects. Recall from Section 3 that for such summaries to be sound, the pattern Θ should include all values read over all the non-deterministic behaviors inside the function starting from a given visible state and the set E should include all the resulting effects.

With this generalization, we find it difficult to present our optimized algorithm in the style of a fixpoint over relations, since some of the relations grow non-monotonically. Hence, we present the optimization as an imperative algorithm. Figure 3 gives the types of variables used in the algorithm. A *Node* is a "shell" around a visible state, with a node type and an effect, which accumulates all alocations and writes along an execution path. There are three node types: *CALL*, *RETURN* and *EXECUTE*. Local searches in each procedure start at roots of type *Root*, where a pair of hash tables *visitedTable* and *returnTable* are used respectively to (1) keep track of visited states and ensure termination of the search, and (2) prevent repeated addition of identical effects. Each root holds a summary, which consists of a pattern and a set of effects.

The main loop of the algorithm is shown in Figure 4. There are 3 global variables, namely (1) *workList*, a list of pending work items, (2) *rootList*, a list of roots where local searches begin inside each procedure (3) *graph*, a set of dependencies between caller and callee procedures. The main loop processes the work list and

```
record Edge {
    num: integer;
    addrMap : Addr → Addr;
}
Graph = (Root × VisibleState) → Root → Edge
AddrMap = Addr → Addr

record Effect {
    alloc: Set (Addr);
    wmap: LocationMap;
}
record Root {
    pattern: LocationMap;
    effects: Array (Effect);
    visitedTable: Set (VisibleState);
    returnTable: Set (VisibleState);
    updated: boolean;
}
enum WorkItemType { STATE, SUMMARY }
record WorkItem {
    root: Root;
    state: VisibleState;
    type: WorkItemType;
}
enum NodeType { CALL, RETURN, EXECUTE }
record Node {
    state: VisibleState;
    effect: Effect;
    type: NodeType;
}
```

Figure 3. Types

```
workList : list(WorkItem);
rootList : list(Root)
graph : Graph

Root r = { visitedTable = ∅; returnTable = ∅; updated = false; };
WorkItem w = { root = r; state = (h0, g0, l0); type = STATE; };
workList.Add(w);
rootList.Add(r);
graph = ∅;

while (workList ≠ ∅) {
    w = workList.Get();
    w.root.updated = false;
    Search(w);
    if (w.root.updated) {
        Root x;
        VisibleState s;
        Root y;
        integer i;
        Graph staleEdges = {(x,s,y,e) ∈ graph | y = w.root};
        graph = graph \ staleEdges;
        foreach ((x,s,y,e) ∈ staleEdges) {
            w = { root = x; state = s; type = SUMMARY};
            workList.Add(w);
        }
    }
}
```

Figure 4. Main loop of optimized algorithm that uses patterns

```
void Search(WorkItem w) {
    Root r = w.root;
    Stack⟨Node⟩ stack = ∅;
    Node node, newNode;
    if (w.type = STATE)
        node = { state = w.state; effect = ∅; type = EXECUTE};
    else
        node = { state = w.state; effect = ∅; type = CALL};
    stack.Push(node);
    r.visitedTable = r.visitedTable ∪ Representative(w.state);
    while (stack ≠ ∅) {
        node = stack.Peek();
        if (node.type = CALL)
            newNode = GetNextSuccessorCall(r, node)
        else
            newNode = GetNextSuccessorExecute(r, node)
        if (newNode = null) {
            stack.Pop(); continue;
        }
        State rep = Representative(newNode.state);
        if (newNode.type = RETURN) {
            if (rep ∉ r.returnTable) {
                r.returnTable = r.returnTable ∪ rep;
                r.effects.Add(newNode.effect);
                r.updated = true;
            }
            continue;
        }
        if (rep ∈ r.visitedTable) continue;
        r.visitedTable = r.visitedTable ∪ rep;
        stack.Push(newNode);
    }
}
```

Figure 5. The procedure *Search*

calls the procedure *Search* on each work item until there are no more work items. A root gets updated when its effects get updated or when it pattern gets updated. In either case, the main loop deletes the edges and reschedules the caller on the worklist so that the dependencies can be computed afresh. Figure 5 shows the *Search* procedure. It implements Depth First Search that calls *GetNextSuccessorCall* for getting the successors of *CALL* nodes and *GetNextSuccessorExecute* for getting the successors of *EXECUTE* nodes. The search terminates at the *RETURN* nodes, and effects are updated at each *RETURN* node.

The lookup of summaries happens inside the implementation of *GetNextSuccessorCall* shown in Figure 6. Procedure *Match* is used to match the current visible state with the pattern of a summary, and function *Apply* is used to apply the effect of a matched summary on a visible state. If a pattern matches with a visible state, the *Match* procedure computes a mapping between the pointers of the visible state and the pattern, and the *Apply* procedure uses this mapping to apply the effect correctly on the visible state. The implementation of *GetNextSuccessorExecute* iterates over the transitions of the visible state associated with an *EXECUTE* node. We assume that the following functions on visible states are available:

1. *boolean ExistsNextSuccessor*(*VisibleState s*)
2. (*Set*(*Location*) × *Set*(*Addr*) × *Set*(*Location*) × *VisibleState* × *NodeType*) *GetNextSuccessor*(*VisibleState s*)
3. *Value Read*(*VisibleState s*, *Location loc*)
4. *void Write*(*VisibleState s*, *Location loc*, *Value v*)
5. *integer HeapSize*(*VisibleState s*)

7 Experiments

We implemented the summarization algorithm from Section 6 in ZING, which is a software model checker being developed in Microsoft Research. Our implementation is more general than the description from Section 6 in two ways: (1) it works on the entire ZING language with both integer and boolean base types, and unrestricted reference types and (2) it also handles concurrent programs in a sound manner using transactions, and the idea of summarizing within a transaction, a technique described in [10]. Termination of the algorithm is guaranteed if base types are boolean, and paths in the heap are bounded, and it is either the case that the program is sequential (due to Theorem 3 of this paper), or it is the case that the program is concurrent, and every recursive function call is transactional (obtained by combining Theorem 3 of this paper with the termination result in [10]).

We describe four sets of experiments that we designed to measure the effectiveness of summarization. The first two experiments were designed to measure the performance gain due to summarization, and the next two were designed to assess the correctness and robustness of the implementation.

Transaction Manager. We obtained a concurrent transaction management program from a product group in Microsoft. It was automatically translated to Zing from C#. It has about 7000 lines of code, several dynamically created objects and two concurrent threads. A grep of the program shows 57 places in the code where new objects are allocated dynamically, and several of these happen in procedures that are called in several call-sites in the program. The ZING model checker discovered a null-pointer dereference bug in this program. We then proposed a fix, and checked that the fix did not have null-pointer dereferences. Both the models have sev-

```
requires node.type = CALL
Node GetNextSuccessorCall(Root r, Node node) {
  Effect newEffect;
  VisibleState newState;
  NodeType newType;
  Node returnNode;
  Root target;
  Edge edge,newEdge;
  WorkItem w;
  AddrMap map;

  if (graph(r, node.state) is undefined) {
    boolean found = false;
    target = null;
    foreach (Root r' ∈ rootList) {
      ⟨found, map⟩ = Match(r'.pattern, state);
      if (found) {target = r'; break;}
    }
    if (target = null) {
      w = { root = target; state = node.state;
            type = STATE;}
      workList.Add(w);
      return null;
    }
    edge = {num = 0; addrMap = map;};
    graph = graph ∪ (r, node.state, target, edge);
  } else
    (target, edge) = graph(r, node.state);

  if (edge.num < target.effects.length) {
    Set(Location)rset = target.pattern \ (LocalVar ∪ {pc});
    foreach (Location loc ∈ rset) {
      if (node.effect.wmap[loc] is undefined){
        r.pattern[loc] = Read(node.state, loc);
        r.updated = true;
      }
    }
    (newEffect, newState) =
      Apply(target.effects[edge.num], addrMap, node.state);
    newEdge = {num = edge.num + 1;
          addrMap = edge.addrMap;};
    graph(r, node.state, target) = newEdge;
    returnNode = { state = newState; effect = newEffect;
          type = EXECUTE;};
  } else
      return null;

  return returnNode;
```

Figure 6. The procedure *GetNextSuccessorCall*

```
⟨boolean, AddrMap⟩ Match(LocationMap pattern, VisibleState state) {
  AddrMap addrMap = ∅;
  foreach (⟨Location loc, Value v⟩ ∈ pattern) {
    Location loc' = case loc of
                      ⟨a, f⟩ → ⟨addrMap[a], f⟩
                      | _    → loc;
    Value v' = Read(state, loc');
    if (v ∈ (Bool ∪ PC ∪ {null})) {
      if (v' ≠ v)
        return ⟨false, ∅⟩;
    } else {
      if ((addrMap[v] is defined ∧ addrMap[v] ≠ v')
          ∨(addrMap^-1[v'] is defined ∧ addrMap^-1[v'] ≠ v))
            return ⟨false, ∅⟩;
      addrMap[v] = v';
    }
  }
  return ⟨true, addrMap⟩;
}
```

Figure 7. The procedure *Match*

```
⟨Effect × VisibleState⟩ Apply(Effect effect, AddrMap addrMap,
                              VisibleState state) {
  Effect newEffect;
  VisibleState newState = state;
  integer count = HeapSize(newState) + 1;
  foreach (Addr a ∈ effect.alloc) {
    newEffect.alloc = newEffect.alloc ∪ {count};
    foreach (f ∈ Field) {
      if (f is boolean)
        Write(newState, ⟨count, f⟩, false);
      else
        Write(newState, ⟨count, f⟩, null);
    }
    addrMap[a] = count;
    count = count + 1;
  }
  foreach (⟨Location loc, Value v⟩ ∈ effect.wmap) {
    Location loc' = case loc of
              ⟨a, f⟩ → ⟨addrMap[a], f⟩
              | _    → loc;
    Value v' = case v of
              Addr → addrMap[v]
              | boolean ∪ PC ∪ {null} → v;
    newEffect.wmap[loc'] = v';
    Write(newState, loc', v');
  }
  return ⟨newEffect, newState⟩;
}
```

Figure 8. The procedure *Apply*

```
requires node.type = EXECUTE
Node GetNextSuccessorExecute(Root r, Node node) {
  Effect newEffect;
  VisibleState newState;
  NodeType newType;
  Node returnNode;

  if (ExistsNextSuccessor(node.state)) {
    Set(Location) rset, wset;
    Set(Addr) as;
    (rset, as, wset, newState, newType) =
      GetNextSuccessor(node.state);
    foreach (Location loc ∈ rset) {
      if (node.effect.wmap[loc] is undefined){
        r.pattern[loc] = Read(node.state, loc);
        r.updated = true;
      }
    }
    newEffect.alloc = node.effect.alloc ∪ as;
    newEffect.wmap = node.effect.wmap;
    foreach (Location loc ∈ wset)
      newEffect.wmap[loc] = Read(newState, loc);
    returnNode = { state = newState; effect = newEffect;
                   type = newType; };
  } else
    return null;

  return returnNode;
```

Figure 9. The procedure *GetNextSuccessorExecute*

Program	**Without summarization** (seconds)	**With summarization** (seconds)
TM with bug	147.750	97.736
TM without bug	227.050	169.292

Table 1. Comparison of model checking times with and without summarization for a transaction management program

eral millions of reachable states. The error happens only in a particular, rarely exercised, interleaving between the two threads, and thus remained undetected in previous testing efforts by the product group. Table 1 shows the total time taken for model checking with and without summarization. The transaction management program has recursive data types, but does not have procedural recursion. However, it only creates bounded chains of objects and our model checker ends up terminating on this example with and without summarization. In both the buggy program, and the bug-fixed program, the model checking time improved by the order of 30%-35% due to summarization.

The use of patterns optimization was crucial. For the bug-fixed program, with the patterns optimization (Section 6) a total of 1125 summaries are computed and looked up 528982 times. Without the patterns optimization (using the visible states as in Section 5), a total of 56881 summaries were computed and looked up 557752 times. Thus, the use of patterns increases the reuse of summaries by over a factor of 50 in this example.

Micro-benchmark. We consider the benchmark from Figure 10. Inside the function `M` makes two recursive calls to `M`, due to the non-

```
class Foo {
bool x;

 Foo Clone() {
  Foo t = new Foo;
  t.x = x;
  return t;
 }
};

class Main {
static Foo g;

activate
 static void main() {
 g = new Foo;
 M(0);
 assert(g.x == false);
 }
```

```
static void M(int i) {
  Foo f;
  bool b;
  if (i < N) {
   f = g.Clone();
   b = choose(bool);
   g.x = b;
   M(i+1);
   assert(g.x == b);
   g.x = f.x;
   }
 }
};
```

Figure 10. Benchmark program for Summarization

N	**Without summarization** (seconds)	**With summarization** (seconds)
5	.942	0.928
10	1.944	0.942
15	44.949	0.988
20	Timeout	1.059
30	Timeout	1.191
40	Timeout	1.268
50	Timeout	1.435
60	Timeout	1.644
70	Timeout	1.848
80	Timeout	2.121
90	Timeout	2.484
100	Timeout	2.848
200	Timeout	7.772

Table 2. Comparison of model checking times with and without summarization for the program in Figure 10. Timeout indicates that the run did not terminate within 10 minutes

deterministic assignment `n = choose(bool)`. Thus, as N varies, a naive model checker analyzing this program needs to make 2^N calls to `M`. However, if we use the summarization algorithm from this paper, only $2N$ summaries for `M` are needed since only values that influence the behavior of `M` are its argument `i`, which can take N different values, and the value of `g.x` which can be either true or false. Thus, a model checker using summarization can scale linearly with N on this program. Note that inside each recursive call, a fresh allocation to local variable `f` is done, and the algorithm is able to handle this case. The empirical results presented in Table 2 show exponential blowup in the model checker without summarization, and linear scaling with summarization.

ZING **Regressions.** We tested all the programs in the ZING regression suite with and without summarization. This suite contains 67 tests. One of the tests is a recursive program called `ParRecursion`. In this example the model checker without summarization enters an infinite loop, but the model checker with summarization terminates. The other tests all run within a few seconds, and the improvements due to summarization are not noticeable. Table 3 shows representative numbers for four of these tests: buggy and fixed versions of a bluetooth device driver, an implementation of Lamport's bakery algorithm, and a model of Dijkstra's dining philosophers. The model checker with the summarization algorithm produces identical results (pass or fail) as the model checker without summarization on all the tests. This gives us confidence that our implementation is working correctly.

Program Program	**Without summarization** (seconds)	**With summarization** (seconds)
ParRecursion	Timeout	0.914
BluetoothBuggy	1.870	1.889
BluetoothFixed	2.118	1.941
BakeryAlgorithm	2.080	1.960
DiningPhilosophers	2.915	2.814

Table 3. Comparison of model checking times with and without summarization for Zing regression tests

SLAM **Regressions.** We have adapted the SLAM toolkit [4] to use ZING as the back-end model checker for boolean programs instead of SLAM's model checker BEBOP. This is a somewhat restricted use of ZING since boolean programs have only boolean variables and do not have any reference types. However, the summarization algorithm presented in the paper should produce identical results to BEBOP's summarization algorithm, when restricted to boolean programs. We were able to check this on 198 of the 204 positive tests in the SLAM regression suite. Both BEBOP and ZING processed each of these tests in a second or less and produced identical results. The 6 remaining tests have several global boolean variables with non-deterministically initialized values. BEBOP is able to handle this initial non-determinism symbolically using Binary Decision Diagrams (BDDs), but ZING is unable to complete analyzing these examples within comparable running times. We are currently augmenting ZING with some symbolic techniques to resolve this problem. This issue is orthogonal to the evaluation of the summarization algorithm.

Summary of Experiments. We find that summarization outperforms the naive model checker if the same procedure is called with the same context a large number of times, as expected. This was demonstrated both using the artificial program in Figure 10 as well as a real-life transaction manager program obtained from a product group at Microsoft, where speedups of 30%-35% were observed. Further, we have done extensive testing of our implementation with almost all the regression tests from ZING and SLAM regression suites. With a few exceptions, the algorithm produces identical results to a naive model checker giving us very high confidence that our implementation is working correctly (we are tracking and fixing the few remaking anomalies at the time of writing this paper).

8 Conclusions

We have presented an algorithm to perform precise interprocedural analysis of programs with references. Our algorithm terminates on programs with finite base types and bounded paths in the heap. Thus, it enables generating models with references as abstractions of large programs during model checking. We have combined this technique with a prior technique we developed to summarize procedures in concurrent programs, and implemented the algorithm for the whole of the ZING modeling language, which has both unrestricted reference types and unrestricted concurrency. We observe that the algorithm improves the speed of the model checker by 30-35% on a model obtained from a distributed program inside Microsoft. We have also done extensive comparisons of our implementation with the BEBOP tool from SLAM for boolean programs.

Acknowledgments. We thank an anonymous reviewer for pointing out a subtle error in an earlier version of the algorithm presented in this paper. We thank Thomas Ball for discussions about extensions to Boolean Programs without losing decidability of model checking. We thank Jakob Rehof for collaborating with us on how to do summarization for concurrent programs. The concurrent transaction manager example was obtained as a result of Mayur Naik and Jakob Rehof's work in the summer of 2004 to build a concurrency testing tool for distributed and concurrent programs. Wolfgang Grieskamp wrote a translator from C# to ZING that enabled us to convert the Transaction Manager example to ZING. We thank Tony Andrews for several discussions on how to adapt the ZING runtime and compiler to implement the summarization algorithm. The idea of instrumenting the ZING runtime so as to record reads and writes, and then generate patterns and effects from such recorded information was proposed by Yichen Xie, when he as an intern with MSR in the summer of 2003. Georg Weissenbacher, Jakob Lichtenberg and Vlad Levin helped build a path from SLAM to ZING, in order to check concurrency properties of drivers during the summer of 2003. This infrastructure was greatly useful in letting us compare the results of BEBOP and ZING on the SLAM regression tests.

9 References

[1] T. Ball. Symbolic reachability of boolean programs with references, unpublished document from personal communication. June 2003.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*. ACM, 2001.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.

[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI: Programming Language Design and Implementation*, pages 57–69. ACM, 2002.

[6] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV 01: Computer Aided Verficiation*, LNCS 2102, pages 324–336. Springer-Verlag, 2001.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.

[8] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE 01: Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. ACM, 2001.

[9] R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN 99: SPIN Workshop*, LNCS 1680, pages 261–276. Springer-Verlag, 1999.

[10] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Principles of Programming Languages*, pages 245–255. ACM, 2004.

[11] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[12] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstraction. In *POPL 05: Principles of Programming Languages*, pages 296–309. ACM, 2005.

[13] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS 05: Static Analysis Symposium*, 2005.

[14] Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.

[16] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[17] B. Steffen and O. Burkart. Composition, decomposition and model checking optimal of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

[18] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Java pathfinder - second generation of a java model checker. In *Proceedings of Post-CAV Workshop on Advances in Verification*, July 2000.

[19] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI 04: Programming Language Design and Implementation*, pages 131–134, 2004.

[20] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, 1995.

Appendix

The proof of the soundness theorem requires the following lemma, which can be proved by induction on the length of execution paths of the form: $(h_I,\ell_I,\varepsilon) \longrightarrow^* (h,\ell,s)$. We first give a few definitions used to state the lemma.

An execution sequence $(h,\ell,s) \longrightarrow^* (h',\ell',s')$ is *balanced* if every push action has a corresponding pop action and every pop action has a corresponding push action. A balanced subsequence σ' of σ is *left-maximal,* if there is no left-extension of σ' that is also balanced. Given an execution sequence $\sigma = (h_I,\ell_I,\varepsilon) \longrightarrow^* (h,\ell,s)$, we define *Entry*$(h,\ell,s)$ to be the first state in the left-maximal sub sequence $\sigma' = (\hat{h},\hat{\ell},s) \longrightarrow^* (h,\ell,s)$ of σ. Intuitively, *Entry*(h,ℓ,s) is the state when the current stack frame on top of the stack was first pushed, and has not been popped since.

LEMMA 2. *Consider any execution sequence* $(h_I,\ell_I,\varepsilon) \longrightarrow^* (h,\ell,s) \longrightarrow (h',\ell',s')$. *Let* $(\hat{h}',\hat{\ell}',s') = Entry(h',\ell',s')$. *Let* σ' *be the subsequence of the execution* $(\hat{h}',\hat{\ell}',s') \longrightarrow^* (h',\ell',s')$. *Let as*$'$ *be the set of addresses allocated in the heap during the execution of* σ' *that are reachable from the visible state* (h',ℓ'). *Let w*$'$ *be the set of heap locations on the heap that were written during the execution of* σ' *and are reachable from either* $(\hat{h}',\hat{\ell}')$ *or* (h',ℓ'). *Let* δ' *be the set of addresses* $\Gamma(w') \setminus as'$.

Then the following two statements hold:

1. *There exists* ρ *such that* $Q(\rho(\hat{h}'),\rho(\hat{\ell}'),\Lambda(\rho(h',\ell'+\delta')))$ *and*

$P(\rho(\hat{h}'),\rho(\hat{\ell}'),\rho(w'),\rho(h'),\rho(\ell'))$.

2. *Supppose the last step in the sequence* $(h,\ell,s) \longrightarrow (h',\ell',s')$ *is a pop action. Let* $(\hat{h},\hat{\ell},s) = Entry(h,\ell,s)$. *Let* σ *be the subsequence of the execution* $(\hat{h},\hat{\ell},s) \longrightarrow^* (h,\ell,s)$. *Let as be the set of addresses allocated in the heap during the execution of* σ *that are reachable from the visible state* (h,ℓ). *Let w be the set of heap locations on the heap that were written during the execution of* σ *and are reachable from either* $(\hat{h},\hat{\ell})$ *or* (h,ℓ). *Let* δ *be the set of addresses* $\Gamma(w) \setminus as$.

   Then, there exist $\rho, h_s, l_s, as_s, m_s, as_r$ *and* w_r *such that* $\rho(gc((h_s,\ell_s))) = gc(\hat{h},\hat{\ell})$ *and* $Sum(\langle h_s,g_s\rangle,\langle as_s,m_s\rangle)$, *and* $apply(\langle as_s,m_s\rangle,\rho,\langle\hat{h},\hat{\ell}\rangle) = \langle h',as_r,w_r\rangle$.

Proof of Lemma 2: The proof is by induction on execution sequences. We sktech the auxiliary lemmas used in the proof. The following lemma states that if there are two paths within the same function that lead to the same term in *Q*, then the two paths agree on the allocated and written addresses and values that are accumulated in *P*. This lemma is used to prove that the STEP rule obeys Lemma 2.

LEMMA 3. *Let* $\sigma_1 = (h,\ell_I,s) \longrightarrow^* (h_1,\ell_1,s)$ *and* $\sigma_2 = (h,\ell_I,s) \longrightarrow^* (h_2,\ell_2,s)$ *be two left-maximal subsequences that start at the same state. Let* as_1 *be the set of addresses allocated in the heap during the execution of* σ_1 *that are reachable from the state* (h_1,ℓ_1). *Let* w_1 *be the set of heap locations on the heap that were written during the execution of* σ_1 *and are reachable from either* (h,ℓ_I) *or* (h_1,ℓ_1). *Let* δ_1 *be the set of addresses* $\Gamma(w_1) \setminus as_1$. *Let* as_2 *be the set of addresses allocated in the heap during the execution of* σ_2 *that are reachable from the state* (h_2,ℓ_2). *Let* w_2 *be the set of heap locations on the heap that were written during the execution of* σ_2 *and are reachable from either* (h,ℓ_I) *or* (h_2,ℓ_2). *Let* δ_2 *be the set of addresses* $\Gamma(w_2) \setminus as_2$.

Then, if $\Lambda(h_1,\ell_1+\delta_1) = \Lambda(h_2,\ell_2+\delta_2)$, *then there exists a bijection* ρ *from* as_1 *to* as_2 *such that* $\rho(as_1) = as_2$ *and* $Map(\rho(w_1),\rho(h_1)) = Map(w_2,h_2)$

The following lemma states that the seqeunce of allocated and writen addresses and values that are accumulated within a summary are sufficient to precisely produce the state after the application of the POP rule.

LEMMA 4. *Consider any execution sequence* $\sigma = (h_I,\ell_I,\varepsilon) \longrightarrow^* (h_1,\ell_1,s) \longrightarrow (h_1,\ell_I,s.\ell_1') \longrightarrow^* (h_2,\ell_2,s.\ell_1') \longrightarrow (h_2,\ell_1',s)$ *such that the subsequence* $\sigma' = (h_1,\ell_I,s.\ell_1') \longrightarrow^* (h_2,\ell_2,s.\ell_1')$ *is left-maximal. Let as be the set of addresses allocated in the heap during the execution of* σ *that are reachable from the state* (h_2,ℓ_2). *Let w be the set of heap locations on the heap that were written during the execution of* σ_1 *and are reachable from either* (h_1,ℓ_I) *or* (h_2,ℓ_2).

Then, if $\langle as',w',h'\rangle = apply(\langle as,Map(w,h_2)\rangle,\rho_I,h_1)$, *where* ρ_I *is the identity permutation, we have that* $h' = h_2$.

The proof of Lemma 2 also requires the following auxililary lemmas showing that the different steps of the algorithm commute under permutations.

LEMMA 5. *Suppose* $T(\langle h,l\rangle,r,as,w,\langle h',l'\rangle)$. *For any permuation* ρ *for h, there exists a permutation* ρ' *for* h', *such that* ρ' *extends* ρ, *and* $T(\langle\rho(h),\rho(l)\rangle,\rho'(r),\rho'(as),\rho'(w),\langle\rho'(h'),\rho'(l')\rangle)$

LEMMA 6. *Suppose the application of the* STEP *rule in the algorithm with* $P(h_1,l_1,as,w,h_2,l_2)$ *as input yields* $P(h_1,l_1,as',w',h_3,l_3)$*as output. Then, for any* ρ *that is a permutation for* h_1 *and* h_2, *there exists a permutation* ρ' *for* h_1 *and* h_3, *such that* ρ' *extends* ρ, *such that the application of the* STEP *rule in the algorithm with* $P(\rho(h_1),\rho(l_1),\rho(as),\rho(w),\rho(h_2),\rho(l_2))$ *as input yields* $P(\rho'(h_1),\rho'(l_1),\rho'(as'),\rho'(w'),\rho'(h_3),\rho'(l_3))$ *as output.*