Deductive Evaluation: Formal Code Analysis with Low User Burden

Ben L. Di Vito
NASA Langley Research Center, Hampton, VA 23681, USA
b.divito@nasa.gov

ABSTRACT

We describe a framework for symbolically evaluating iterative C code using a deductive approach that automatically discovers and proves program properties. Although verification is not performed, the method can infer detailed program behavior. Software engineering workflows could be enhanced by this type of analysis. Floyd-Hoare verification principles are applied to synthesize loop invariants, using a library of iteration-specific deductive knowledge. When needed, theorem proving is interleaved with evaluation and performed on the fly. Evaluation results take the form of inferred expressions and type constraints for values of program variables. An implementation using PVS (Prototype Verification System) is presented along with results for sample C functions.

1. INTRODUCTION

Both formal code verification and loop invariant generation are enjoying a resurgence. While new verification tools offer promise, their uptake remains limited. Static analysis tools, however, have seen mainstream success, owing to modest goals and ease of use. Software engineering would benefit from usable tools that combine the precision of code verification with the ease of use of static analyzers. Our analysis approach contributes to this goal by adapting verification methods while forgoing explicit verification outcomes. In essence, software engineers are given rigorous analytical feedback for assessing the fitness of their code.

Recent verification work has focused on SMT solvers and first order logic. "Heavyweight" methods using interactive theorem provers and higher order logics (e.g., PVS [15]) tend to be overlooked. Although tools having rich logics require some manual effort, we present a new analysis concept that leverages their strengths while offering end users automatic tools. Using a two-track approach, we achieve an effective division of labor by pre-computing deductive artifacts, then later applying them automatically.

The first track is an invariant synthesis technique revolving around *iteration schemes*, which are expressed in PVS notation. In contrast with most invariant generation methods, this approach is highly data-driven, depending on an extensive body of codified knowledge created by specialists. The second track makes use of code verification methods along with synthesized invariants and conventional symbolic analysis techniques. Collectively these ideas achieve a *deductive evaluation* of C functions having loops, a task that is conducted automatically on behalf of end users without a need to supply assertions or specifications. The developer is simply presented with a best-effort derivation of the effects computed by his or her code.

We have created early-stage prototype tools, hosted within PVS, to demonstrate basic feasibility. Although C is the language used in this study, the approach could be applied to other imperative languages.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. ANALYSIS CONCEPT

Several techniques underlie the code analysis method:

- **Theorem proving in higher order logic.** Our tool choice is PVS, which includes an expressive language having rich type features as well as a powerful theorem prover. Besides its interactive mode, the prover can be invoked programmatically for fully automated proof.
- **Deductive code verification principles.** Floyd-Hoare principles for proving iterative code are used along with concepts of symbolic evaluation/execution.
- **Data-driven invariant synthesis.** Loop invariants are generated from *iteration schemes*, stylized PVS theories for modeling the effects of iterative code fragments. Execution effects within a loop body are matched against a scheme library to derive invariants.

While the resulting capability does not conduct verification, its analyses can contribute to contract-based verification or serve other purposes such as symbolic debugging.

2.1 Mechanization Using PVS

PVS refers to both a language and a set of deduction tools. The language allows formalization of mathematical and logical concepts, although it lacks explicit models of computation. Classical higher order logic and a flexible type system form the theoretical underpinnings. Hosted within Emacs, the tools perform parsing, typechecking and theorem proving.

Declarations (e.g., types, constants, lemmas) are grouped into *theories*. Key features for our purposes are function-valued expressions and *predicate subtypes* [17]. Subtypes

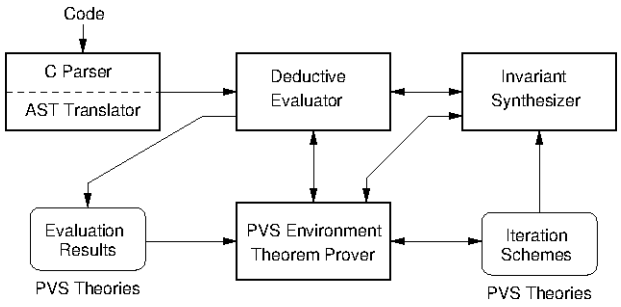

Figure 1: Architecture of the prototype framework.

may be declared as subsets of previously declared or built-in types. The set comprehension notation $\{x : T \mid P(x)\}$ denotes a subtype. Uninterpreted constant declarations allow us to name an arbitrary value of a type. For example,

```
n_1_: {n: int | 0 <= n AND n < q}
```

illustrates this with an integer range subtype. Deductive evaluation exploits this feature to embed derived constraints or assertions in the types of PVS constants.

Fig. 1 depicts the experimental tool framework. C source code is mapped into an abstract syntax tree (AST) using Lisp s-expressions. The deductive evaluator is implemented in Common Lisp and resides in the same process as PVS. It traverses the AST and performs the analysis. A PVS output theory is built incrementally as described in Section 4.

Iteration schemes are collected in a library of PVS theories and must first be "registered" for later use during invariant generation. Registration extracts key details from those theories to build data structures for searching and matching. When the evaluator needs invariants for a fragment of C code, synthesizer functions are invoked as described in Section 3. The synthesizer is implemented in Common Lisp and loaded along with the deductive evaluator. Only a modest library of schemes (around 20) has been developed so far.

2.2 C Features Supported

The current prototype is limited to a subset of C language features. Data types are integers and arrays of integers. Function declarations and basic C statements are supported; other declarations are not. Function calls are supported as well. Expressions must be free of side effects, although `i++` is allowed as a statement. Array arguments must not be aliased (no overlaps). Pointers and dynamic memory features are excluded in this early stage. The evaluation prototype is limited to partial correctness results (no termination proofs). C integers are modeled using mathematical numbers rather than machine numbers. Some of these limitations will be relaxed in future versions.

2.3 Evaluation Example

Fig. 2 shows a C function to multiply integers along with an excerpt of the evaluation output. Deductive evaluation produces a PVS theory for each C function. Each such theory ends with a declaration named `final` that characterizes the result(s) computed by the C function. In this case the evaluator deduced that the result is the product of parameters `m` and `n`. Numeric suffixes are attached to PVS identifiers to disambiguate C variable values at different execution points. Additional declarations and evaluator comments from the generated theory are shown in Fig. 3, which typically would be hidden in end-user displays.

```
int add_mult(unsigned int m, int n) {
  int p = 0;
  unsigned int i = 0;
  while (i < m) {
    p += n;
    i++;
  }
  return p; }
```

```
add_mult_deval [ (IMPORTING iter_schemes@prog_types)
                 m_0_: nat, n_0_: int ] : THEORY
BEGIN
  %% Analysis details appear in Figure 3.
  final: return_values = (# result_ := m_0_ * n_0_ #)
END add_mult_deval
```

Figure 2: C function and its evaluation in PVS.

```
p_0_: int = 0
i_0_: nat = 0
result_0_: int
return_values: TYPE = [# result_: int #]

% Analyzing while loop at depth 1.
% Found dynamic variables: p, i
% Found static variables: m, n
% Found possible index variables: i
% Values at top of loop:
k_1_: nat  % implicit loop index
p_1_: int  % dynamic variable
i_1_: nat  % dynamic variable
% Effects of loop body:
p_2_: int = p_1_ + n_0_
i_2_: nat = i_1_ + 1

% Invariants for loop index i (scheme loop_index_recur):
%   (index_var_expr . i_1_ = k_1_)
%   (iter_k_expr . k_1_ = i_1_)
%   (initial_bound . TRUE)
%   (final_bound . i_1_ < 1 + m_0_)
% Invariants for variable p (scheme arith_series_recur):
%   p_1_ = (k_1_ * n_0_)

% Values of dynamic variables on (normal) loop exit:
k_2_: nat = m_0_
i_3_: nat = m_0_
p_3_: int = m_0_ * n_0_
% End of for/while loop at depth 1.
```

Figure 3: Declarations for theory in Fig. 2.

For modest functions such as that in Fig. 2, deductive evaluation provides significant benefits. If the result expression is the desired specification, then verification has been performed implicitly because that evaluation result is a machine-checked inference that follows from the C function's semantics. If the code contains an error, reviewing the result expression should aid in its discovery and localization. For example, if the while-condition had been `(i <= m)`, the result would have been `m_0_ * n_0_ + n_0_`, greatly helping to identify the off-by-one error. In other cases, the user will receive less specific analytical feedback.

To obtain the results in Figures 2 and 3, a user need

only invoke the evaluator; no annotation or interaction is required. If a particular code fragment is not covered by the iteration scheme library, the evaluator will produce only partially useful results. While the prototype generates output in the PVS language, translation to other notations is possible. The C specification language ACSL [2], for instance, could be used when only first-order features are involved.

3. INVARIANT SYNTHESIS

Deducing the effects of iterative code can be achieved using a Hoare-style proof rule such as the following, where predicate Q serves as loop invariant.

$$\frac{P \Rightarrow Q \qquad \vdash \{B \wedge Q\}\,\mathtt{S}\,\{Q\} \qquad Q \Rightarrow (R \vee B)}{\vdash \{P\}\ \mathtt{while\ B\ do\ S}\ \{R\}} \quad (1)$$

Our approach for automated invariant generation relies on the expressive power of higher order logic to build a library of iteration schemes, which are later instantiated to create specific invariants. Over time, growth of this collection will enable broad coverage for typical iterative code.

3.1 Predicate-Based Recurrence Relations

Recurrence relations have been applied in other work on invariant generation [12]. Their application to loop invariants is quite natural. For example, given the code

```
p = 1; for (i = 0; i < m; i++) p *= 2;
```

we could formulate the recurrence $F(0) = 1; F(n+1) = 2F(n)$, which has the solution $F(n) = 2^n$. From this solution we could infer the invariant $p = 2^i$.

In mathematics, solutions to elementary recurrences are functions. Since many invariants state relational properties, we generalize to accommodate predicates as solutions. For example, the recurrence above could become $I(u,0) \equiv u = 1; R(u,v,n) \equiv v = 2u$, where I constrains u initially and R relates next value v to current value u. $P(u,n) \equiv u = 2^n$ is a solution. Note there are multiple solutions, many of which are trivial (e.g., $P(u,n) \equiv u > 0$). This does not diminish the utility of the approach; scheme authors provide solutions strong enough to serve as effective invariants.

To simplify tool design, we provide a stylized method to express predicate recurrences directly in PVS theories. Each theory includes a lemma and inductive proof that the solution satisfies the recurrence. The proof is constructed by the scheme developer using the PVS interactive prover.

3.2 Iteration Schemes in PVS

Fig. 4 presents the structure of iteration schemes in example form, using a scheme applied in the evaluation of Fig. 2. Most of what is shown in Fig. 4 will appear in every scheme; it is basically a template. Schemes typically capture the behavior of a single, dynamic program variable. Dynamic variables are those that change value during a loop; static "variables" may be unchanging variables as well as constants, functions, or static expressions. Recurrences are grouped into several categories according to form and function.

Declarations `recurrence` and `solution` form the heart of the scheme, details[1] of which appear in Fig. 5. Parameters for these functions have fixed names: `I` for initial values of dynamic variables, `S` for static variables, and `U`, `V` for dynamic variable values before and after each iteration. Any of these could be a tuple, hence the template uses LET expressions to perform de-structuring. Also, included in every scheme is an implicit loop index k, which is part of the modeling framework and separate from any program variables.

[1]PVS notation: `(a, b)` is a tuple, `(: a, b :)` is a list, and `(# a := b, c := d #)` is a record.

```
arith_series_recur : THEORY
BEGIN
  dyn_vars:   TYPE = int
  stat_vars:  TYPE = int
  IMPORTING recur_pred_defn[dyn_vars, stat_vars]
  k:       VAR nat
  I,U,V:   VAR dyn_vars
  S,W:     VAR stat_vars
  recur_type: recurrence_type = var_function

  recurrence(I, S)(U, V, k): recur_cond = . . .
  solution(I, S)(U, k): invar_list = . . .

  recur_satis: LEMMA sat_recur_rel(solution, recurrence)
END arith_series_recur
```

Figure 4: Scheme used in evaluation of Fig. 2.

```
recurrence(I, S)(U, V, k): recur_cond =
    LET s0 = I, d = S, u = U, v = V IN
      (# each := (: (iter_effect, v = u + d) :),
         once := (: :) #)
solution(I, S)(U, k): invar_list =
    LET s0 = I, d = S, u = U IN
      (: (func_val_expr, u = k * d + s0),
         (initial_bound,
          IF d < 0 THEN u <= s0 ELSE u >= s0 ENDIF) :)
```

Figure 5: Recurrence, solution details for Fig. 4.

Recurrence definitions have two kinds of labeled conditions: "`each`" conditions must hold before and after every iteration, and "`once`" conditions hold initially or constrain constant expressions. Labeling conditions helps the evaluator provide more precise information during the matching phase of invariant synthesis. Solution definitions provide labeled invariant expressions, the conjunction of which is a solution predicate. Providing different invariant types helps the evaluator make better use of derived information.

Higher order logic figures prominently in the formulation of schemes, enabling generic expression of both conditions and solutions. Function variables can be restricted to have necessary properties, such as monotonicity:

```
FORALL (p, q: nat): p < q IMPLIES f(p) < f(q))
```

For typical bindings of `f`, such properties can be proved automatically by PVS.

3.3 Instantiating Iteration Schemes

Given a loop body S and dynamic variable x, two PVS constants, e.g., x_1 and x_2, will denote x's value before and after an arbitrary iteration. Evaluation of S will derive an expression e for the value x_2. Subsequently, x_1, x_2 and e become inputs to the scheme-matching process.

At scheme registration time, recurrence conditions are turned into patterns for matching. Variables in a `recurrence` declaration (e.g., `s0,d,u,v` from Fig. 5) become pattern variables. Each can be bound to a program variable or expression of the appropriate type and dynamic status.

```
int add_mult_exp(unsigned int m, int n) {
  int p = 0;
  unsigned int d = m;
  int y = n;
  while (d > 0) {
    if (d % 2 == 1) p += y;
    y += y;
    d /= 2; }
  return p; }

add_mult_exp_deval
  [ (IMPORTING iter_schemes@prog_types)
    m_0_: nat, n_0_: int ] : THEORY
BEGIN
  %% Internal analysis details omitted.
  final: return_values = (# result_ := n_0_ * m_0_ #)
END add_mult_exp_deval
```

Figure 6: A more realistic multiply algorithm.

During evaluation and synthesis, schemes will be searched and matches attempted. The "`each`" conditions are matched in order against program expressions until all pattern variables are bound. Remaining conditions are instantiated with accumulated bindings. Only after all recurrence conditions are met, which requires theorem proving, will a matching scheme be recognized. Included are any conditions needed to constrain the initial state. After a successful match is found, each `solution` expression is instantiated with terms from the pattern variables and emitted as invariants.

An important aspect of this method is that synthesized invariants are valid logical inferences of loop behavior. They are not merely candidates needing further checking because all necessary conditions are proved to hold and each scheme solution is known to satisfy its recurrence. This in turn reduces the theorem proving burden during evaluation. Mathematically deep properties can be placed in schemes without taxing the deduction performed during evaluation.

3.4 Additional Features

One feature that enhances invariant synthesis is the ability to specify that a scheme depends on previously generated invariants. Consider the example of Fig. 6, a multiply algorithm similar to a hardware shift-and-add method. The scheme that `p` satisfies includes the conditions

```
(dep_var_func, d = floor(d0 / 2^k)),
(dep_var_func, y = y0 * 2^k),
```

which are matched by invariants generated for `d` and `y`.

Another helpful feature concerns the additional paths created when loops are exited via `return` and `break` statements. Loop exits can sometimes induce useful invariants. For exit condition $P(e)$, we can often infer cases e' where $\neg P(e')$ holds at the top of every iteration. One sufficient condition is that the loop index is the only dynamic variable P references. This allows us to conclude an invariant such as $\forall\, j < k : \neg P(j)$. Dedicated schemes cover such cases.

As the scheme library grows, features will be needed to cope with increasing scale. Performance considerations will dictate more sophisticated search and more careful library organization. Curtailing the generation of excess invariants when multiple schemes apply will require attention. The need for filtering, though, should not be as great as with the Daikon tool [7], for example. Schemes are hand-crafted, emphasizing causation rather than correlation, and thus less likely to spawn useless invariants.

4. DEDUCTIVE EVALUATION

Given C code transformed to ASTs and rendered as s-expressions, the deductive evaluator attempts to infer effects that would be produced by the code when executed. The evaluator works in units of individual C functions. For C function `F`, a PVS theory named `F_deval` will be built incrementally to record the evaluation results and provide declarations usable when evaluating other functions. Much of the processing draws from established techniques; the novel parts concern loop handling.

4.1 Path Analysis

Evaluation proceeds in a forward direction, similar to symbolic execution or strongest-postcondition analysis. Function parameters and local variables are represented by parameters and constant declarations in PVS theory `F_deval`. The initial value of `v` is named `v_0_`. Values at later execution points are denoted by constants with higher suffixes.

C data types and expressions are mapped into semantic equivalents in PVS, except that unbounded integers are used instead of machine integers. Arrays are represented by functions from $\{0, \ldots, N\}$ into a base type (currently integers). Relational and logical expressions produce numeric results, as per C semantics. State vectors of variable values are maintained and updated as statements are processed. Values are symbolic expressions in PVS notation.

Statements are processed in order along each execution path. Assignments cause allocation of new constants to theory `F_deval` and the updating of state vector(s). Array assignments make use of PVS "function update" expressions. `A[i] = A[j]`, for instance, leads to a declaration such as:

```
A_3_: int_array(A_size_) =
        A_2_ WITH [(i_1_) := A_2_(j_3_)]
```

Conditional statements cause path branching in the usual way, along with the accumulation of new conjuncts for path conditions. Unlike symbolic execution, though, the paths are unified at the close of a conditional statement. Because PVS has conditional expressions, a symbolic value after an if-statement takes the form, `IF a THEN b ELSE c ENDIF`.

Control transfer statements such as `return` and `break` can create extra paths with cloned state vectors. A `return` statement at the end of a function also invokes return-value processing. A call to function `G` makes use of `G`'s previous evaluation saved in theory `G_deval`.

4.2 Loop Processing

Deducing the effects computed by loops requires the application of proof rule (1) as well as the generation of invariants for dynamic variables. During the AST stage, for-loops are translated to while-loops, so we assume each loop has the general form, `while (B) S`, where the body `S` is a statement or block. Processing begins by identifying static and dynamic variables, and noting initial value I of the dynamic variables. There is also an attempt to identify a loop index variable (loop counter) among the dynamic variables.

The evaluator first creates a state vector U to represent values at the start of an arbitrary iteration of the loop. Supplying fresh PVS constants for dynamic program variables serves this purpose. Next, the evaluator is run on condition

B to yield expression B, then run on loop body S, resulting in state vector V. Due to conditional statement handling, all iterating paths are merged into a single path having a single state vector. Values in V include cumulative updates to dynamic variables that result from all statements in S.

At this point, the evaluation process seeks invariants. For each dynamic variable x, the synthesizer is called with B, I, U, V, and supporting information, where the matching process described in Section 3 is carried out. If an invariant Q is returned, it will be saved along with some context information. If no valid invariant can be inferred for a variable, its value will remain unconstrained.

After the generated invariants $\{Q_i\}$ have been gathered, they are used to derive expressions for the final values of dynamic variables upon loop termination, the point where $\neg B$ is assumed to hold. Some schemes include auxiliary facts to help infer final values. When applicable, final values of the loop index variable and implicit index k are deduced. For example, if R is "<," $d = 1$ and B is $i < n$, the following auxiliary fact allows us to deduce $i = n$.

```
(final_index_value,
 R(0, d) AND NOT R(i, n) => i = n + mod(i0 - n, d))
```

These derivations, in turn, help deduce final value expressions for other dynamic variables, which are introduced as new final-value constants. Afterward, the evaluator will have a new state vector W that characterizes variable values immediately after loop termination. W will be used to continue evaluation. If loop exit paths exist due to `break` statements, these paths are merged with the normal exit path. State vector merging requires conditional or disjunctive expressions to describe variable values at the merge point.

4.3 Array Handling and Well-Formedness

Arrays are modeled using values of PVS function types. Computing with arrays, however, modifies only one element at a time. This leads to loop invariants that quantify over array indices to describe work completed. In Fig. 7 is a function that sets the first n elements of array A to v. After iteration k, the invariant $\forall\, q < k : A(q) = v$ holds. When k reaches n, we obtain the final quantified expression shown for `val_A` in the figure. This example shows how arrays, which normally require predicates to describe their values, can be represented by named declarations having predicate subtypes. Traditional assertions are thereby obviated.

To ensure well-formedness, array index expressions must be within bounds. Consider two types of parameter declarations: 1) `int A[N]` and 2) `int A[]`. (1) triggers a check for each index expression i that $i < N$ (well-formedness condition, WFC). (2) is handled by introducing an implicit size parameter S for function F and generating a well-formedness obligation (WFO) that implies $i < S$. Appended to the PVS theory, a WFO needs to be established in the calling environment. Invariants augment what is known about array accesses within loops. If we can infer $i < m$ for all iterations, we can generate the WFO $m \leq S$, as in Fig. 7. Special schemes are provided to help establish these bounds.

Other kinds of well-formedness, e.g., absence of divide-by-zero errors, are not yet incorporated in the analysis. Still under study is the possible role of PVS type correctness conditions (TCCs) to help confirm when code is free of such anomalies. Also under study are alternative methods for discharging *subtype* and *existence* TCCs that are spawned

```
void array_init(int A[], unsigned int n, int v) {
  int i, m;
  for (i=0; i<n; i++) A[i] = v;
}

array_init_deval [ (IMPORTING iter_schemes@prog_types)
                   A_size_: posnat,
                   A_0_: int_array(A_size_),
                   n_0_: nat, v_0_: int ] : THEORY
BEGIN
  %% Internal analysis details omitted.
  val_A: {r_: int_array(A_size_) |
          FORALL (q: below(n_0_)): r_(q) = v_0_}
  final: return_values = (# A := val_A #)
  WFO: boolean = n_0_ <= A_size_
END array_init_deval
```

Figure 7: Initialization of n array elements.

when typechecking theory F_deval. A comprehensive solution will require switching to a multi-pass tool design.

4.4 Array Examples

Fig. 8 collects the evaluation results for three common types of array algorithms. These illustrate further how final array values are characterized using subtypes. With traditional verification tools, these would take the form of user-provided postconditions, although the essential constraints would be the same.

Note how the result constraint for the linear search example mentions all array values tested before finding the target value. This faithfully describes the computation, although it is stronger than one might write in a specification. Note also the nested loops in the bubble sort example. Generally these can be handled by the evaluator without special techniques, provided there are iteration schemes that deduce outer loop behavior from inferred inner loop behavior. Used in these schemes is the PVS function, `permutation_of?`, which appears in NASA Langley's PVS library collection [14].

4.5 Precondition Discovery

When iterative algorithms need restricted parameter values, automatic discovery of preconditions can be achieved in many cases. Consider, for example, a binary search function. An applicable iteration scheme S would have a condition C requiring that the array elements be ordered. Suppose that during the matching process, all of the conditions except C are satisfied. Rather than rejecting the match, we could instead emit the invariant $C \supset P$, where P comes from S's solution expressions.

If this conditional invariant is conjoined with another invariant Q, we will have $(C \supset P) \wedge Q$. Now we can push C outward by introducing the weaker formula $C \supset (P \wedge Q)$, which is implied by the previous one. In the fully general case, $(C_1 \wedge \ldots \wedge C_m) \supset (P_1 \wedge \ldots \wedge P_n)$ would collect the conditions, creating a precondition for the loop. Eventually these rise to the top of the function to create an overall precondition for the function. Checking whether $C_1 \wedge \ldots \wedge C_m$ is satisfiable would be a possible enhancement.

Although this technique cannot infer preconditions for the non-iterative parts of a function, it has promise for dealing with common iterative algorithms. Conventions can be added to the scheme notation so authors can indicate which conditions should be eligible for this treatment.

```
int array_min(const int A[], unsigned int nm1) {
  int i, m;
  m = A[0];
  for (i=1; i < 1+nm1; i++)
    if (A[i] < m) m = A[i];
  return m; }
```

```
val_result_: {r_: int |
   ((FORALL (l: below(1 + nm1_0_)): (r_ <= A_0_(l))) AND
    (EXISTS (j: below(1 + nm1_0_)): A_0_(j) = r_))}
final: return_values = (# result_ := val_result_ #)
WFO: boolean = 1 + nm1_0_ <= A_size_
END array_min_deval
```

```
int linear_search(const int A[],
                  unsigned int n, int v) {
  int i = 0;
  while (i < n) {
    if (A[i] == v) return i;
    i += 1;
  }
  return -1; }
```

```
val_result_: {r_: int |
 (((r_ = -(1)) AND
   (FORALL (j: below(n_0_)): NOT A_0_(j) = v_0_)) OR
  (A_0_(r_) = v_0_ AND (r_ < n_0_) AND (0 <= r_) AND
   (FORALL (j: below(r_)): NOT A_0_(j) = v_0_)))}
final: return_values = (# result_ := val_result_ #)
WFO: boolean = n_0_ <= A_size_
END linear_search_deval
```

```
void bubble_sort(int A[], unsigned int nm1) {
  unsigned int i, j;
  int t;
  for (i=0; i < nm1; i++) {
    for (j = i + 1; j < 1 + nm1; j++) {
      if (A[j] < A[i]) {
        t = A[i]; A[i] = A[j]; A[j] = t; }
      } } }
```

```
val_A: {r_: int_array(A_size_) |
        ((FORALL (p: below(nm1_0_)):
            (r_(p) <= r_(1 + p))) AND
         permutation_of?(r_, A_0_))}

final: return_values = (# A := val_A #)
WFO: boolean = 1 + nm1_0_ <= A_size_
END bubble_sort_deval
```

Figure 8: Evaluation results for three common array algorithms.

Deriving the end-to-end behavior of inverse operations is a promising application of deductive evaluation. It requires a different sort of precondition discovery. Consider the problem of showing that two functions achieve lossless data compression. One might construct a C function similar to that needed for a test case:

```
void data_compression(unsigned int n) {
  int A[1000], B[1000], C[1000];
  unsigned int m;
  m = compress(n, A, B);
  decompress(m, B, C); }
```

We hope to infer that, on exit, $A = C$ (initial n elements).

Assume the evaluator has derived $P(n, A, B)$ as the behavior of `compress`. Doing the same for `decompress` is problematic because it does not process arbitrary values of array B, only those having the data format encoded by `compress`.

Two approaches are feasible. The first would create a modified form of the `decompress` function in which the type of input array B is constrained by $P(n, A, B)$ using a predicate subtype. The second approach would evaluate the function `data_compression` after first performing an inline expansion of `decompress`. This would cause `decompress`'s code to be evaluated under the assumption $P(n, A, B)$.

Given that many operations and services come in complementary pairs, end-to-end evaluation would provide strong assurance of key behaviors. Generalization to other function combinations and properties should be possible as well.

5. RELATED WORK

Recent verification tools for C have exploited the power of modern SMT solvers to refresh the classic notion of program verifiers. VCC [3] and Frama-C [6] both carry out functional verification with high automation, provided that specifications and loop invariants are supplied by users. Seahorn [1] is a newer tool with a modular architecture using model checkers and abstract interpreters. A verification technique hosted within Java PathFinder [16] combined ideas from symbolic execution, model checking and invariant generation. Verification using strongest postconditions was proposed for reverse engineering [10].

ESC/Java [8] performs lighter-weight verification using deductive techniques, but only to verify selected properties, relying on some user annotations and hints.

Abstract interpretation [4] can be used to derive conservative loop properties. These constitute valid constraints, although they are often weaker than human-generated invariants. Nevertheless, leading tools in this category [5] can conduct analyses on a realistic scale.

Early work on loop invariant generation began in the 1970s [19]. The last 10–15 years have seen a wide variety of new investigations into invariant generation.

One category generates plausible invariant candidates using dynamic methods, although these are not guaranteed to be invariants. Daikon [7] is a leading tool of this type.

More relevant is the use of logical and mathematical techniques to derive invariant formulas. One popular group of methods is based on predicate abstraction. SMT-based implementations of invariant generation [18] sometimes couple this idea with templates to seed the search process.

First-order theorem provers such as Vampire provide the substrate for several generation methods [13, 11]. Heuristics for extracting loop properties try to identify key facts, for instance, the aggregate effects of array updates.

A few methods are designed to work backwards from postconditions or other assertions. Heuristics that examine the detailed structure of postconditions [9] can consider the role that variables and expressions play.

6. CONCLUSION

A preliminary framework for deductive evaluation and invariant synthesis has been demonstrated. PVS features have

been leveraged with good effect. Whether function results are "closed form" expressions or relations encoded via subtypes, relevant language and tool support is available. Function calls propagate results and their types upward, offering good prospects for analysis across several levels.

A full implementation of the framework would have several potential uses. These include supplementing or replacing unit testing, analyzing software component libraries, analyzing software for specialized domains, and carrying out symbolic debugging. An attractive cost-benefit tradeoff, similar to that of some static analyzers, is a reasonable goal.

Inherent limitations of the current prototype need to be addressed in future work. Foremost among these is populating the iteration scheme library. We speculate that hundreds of schemes, possibly a few thousand, would be needed for adequate coverage of common C functions. Experience with the NASA PVS library [14] (over 1500 theories) suggests that such a formalization effort is achievable. Moreover, any experienced PVS user can create schemes; tool developers are not needed. Once the core engine is mature, capability grows as long as the library grows.

In practice, a deductive evaluator could be embedded within an IDE (integrated development environment) and designed to generate results usable by developers without specialized training. While such a tool would address only a subset of correctness problems, it could be coupled with testing to increase assurance. Eventually, tools along these lines could become a new niche in the service of software engineering.

7. REFERENCES

[1] A.Gurfinkel, T.Kahsai, A. Komuravelli, and J.A.Navas. The SeaHorn verification framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361, 2015.

[2] P. Baudin, J. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL Specification Language (2013)*. `http://frama-c.com/acsl.html`.

[3] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: a practical system for verifying Concurrent C. In *Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages*, pages 238–353, 1977.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The ASTREE analyzer. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30, 2005.

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *SEFM*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.

[7] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02*, pages 234–245, New York, NY, USA, 2002. ACM.

[9] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In A. Blass, N. Dershowitz, and W. Reisig, editors, *Fields of Logic and Computation*, pages 277–300. Springer, 2010.

[10] G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 188, Washington, DC, 1995. IEEE.

[11] K. Hoder, L. Kovács, and A. Voronkov. Case studies on invariant generation using a saturation theorem prover. In *Proc. of 10th Mexican Intl. Conf. on Advances in Artificial Intelligence - Vol. Part I*, MICAI'11, pages 1–15. Springer, 2011.

[12] L. Kovács and T. Jebelean. Automated generation of loop invariants by recurrence solving in Theorema. In *Proc. 6th Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, pages 451–464, 2004.

[13] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proc. of FASE*, 2009.

[14] NASA Langley Research Center. PVS library collection. Theories and proofs available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/`.

[15] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992.

[16] C. S. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *11th International SPIN Workshop, Barcelona, Spain*, volume 2989 of *LNCS*, pages 164–181, Apr. 2004.

[17] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, Sept. 1998.

[18] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of PLDI*, 2009.

[19] B. Wegbreit. The synthesis of loop predicates. *Comm. ACM*, 17(2):102–112, 1974.