Deductive Evaluation: Implicit Code Verification 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 C code using a deductive approach that discovers and proves program properties. The framework applies Floyd-Hoare verification principles in its treatment of loops, with a library of iteration schemes serving to derive loop invariants. During evaluation, theorem proving is performed on-the-fly, obviating the generation of verification conditions normally needed to establish loop properties. A PVS-based prototype is presented along with results for sample C functions.

1 Introduction

Both formal code verification and techniques for loop invariant generation are enjoying a moderate resurgence. While much recent work is focused on SMT solvers and first order logic, "heavyweight" methods using interactive theorem provers and higher order logics (e.g., PVS [21]) are often considered less practical. Although tools having rich logics require some manual effort, we present a new analysis concept that leverages their strengths and offers promising results. Using a two-track approach, we can 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 and rely on PVS tools for deduction support. In contrast with most invariant generation methods, our approach emphasizes a body of codified knowledge. The second track makes use of Floyd-Hoare verification principles [8, 14] 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 without user-supplied assertions or specifications. The user 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. Examples are provided throughout the paper. C is the language used in this study, but the approach could be applied to other imperative languages. Deductive evaluation offers a powerful, automated analysis capability that requires little effort to use, potentially serving in a variety of software development/verification roles.

2 Analysis Concept

Our concept for analyzing source code in imperative languages is based on several tools and techniques:

- *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 the interactive interface, the prover can be invoked programmatically to conduct fully automated proving.
- *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*, which are stylized PVS theories for modeling the effects of iterative algorithms. Execution effects within a loop body are matched against a library of iteration schemes to derive invariants.

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

2.1 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. Expressions must be free of side effects (`i++` is allowed as a statement, however). 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. Only a modest library of iteration schemes (around 20) has been developed so far. If a C function assumes its inputs satisfy some constraints, there is not yet a direct way to declare or infer pre-conditions.

Some of these limitations will be relaxed in future versions.

2.2 Mechanization Using PVS

PVS (Prototype Verification System) [21] refers to both a language and a set of deduction tools. The language allows formalization of mathematical and logical concepts, although it lacks any explicit models of computation. Classical higher order logic and a flexible type system form the theoretical underpinnings. Hosted within Emacs, the tools perform parsing, type checking and theorem proving.

PVS declarations (e.g., types, constants, lemmas) are grouped into *theories*. Important features for our purposes are function-valued expressions and predicate subtypes [23]. Subtypes may be declared as subsets of previously declared or built-in types. The set comprehension notation $\{x : T \,|\, P(x)\}$ is the basic way to express predicate subtypes. 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 a predicate subtype. Deductive evaluation exploits this feature to embed derived constraints or assertions in the types of PVS constants.

2.3 Prototype Tool Architecture

Fig. 1 depicts the experimental tool framework for deductive evaluation and invariant synthesis. C source code is first mapped into an abstract syntax tree (AST) using Lisp s-expressions. The deductive evaluator, which is implemented in Common Lisp and resides in the same process as PVS, traverses the AST and carries out analysis steps. A PVS output theory is built incrementally during this process. Section 4 describes the operation in more detail.

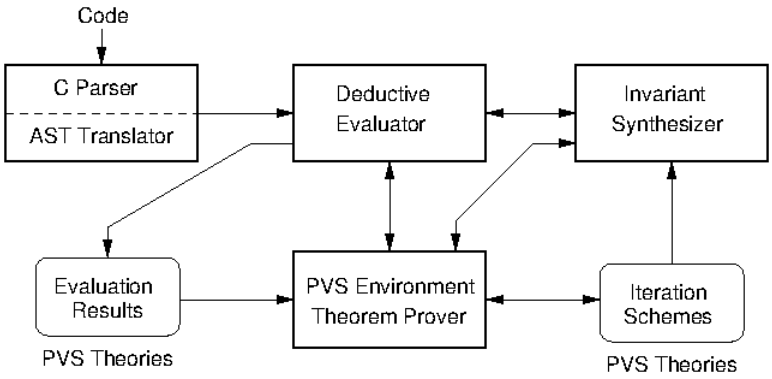

Fig. 1. Architecture of the prototype tool framework.

Iteration schemes are collected in a library, each represented as a PVS theory, 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 to carry out the generation. Operation of the invariant synthesizer is discussed in Section 3. It is implemented in Common Lisp and loaded along with the deductive evaluator.

2.4 Evaluation Example

To illustrate the concepts in some detail, we use a simple iterative algorithm. Fig. 2 shows a C function to multiply integers alongside an excerpt of the deductive 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) returned by the C function. (The declaration `WFO` is a

```
int add_mult(                    add_mult_deval
        unsigned int m,            [ (IMPORTING iter_schemes@prog_types)
                 int n) {            m_0_: nat, n_0_: int ] : THEORY
  int p = 0;                     BEGIN
  unsigned int i = 0;              %% Analysis details in next figure.
  while (i < m) {
    p += n;                        final: return_values =
    i++;                             (# result_ := m_0_ * n_0_ #)
  }
  return p;                        WFO: boolean = TRUE
}                                END add_mult_deval
```

Fig. 2. Simple C function and its evaluation result in PVS (excerpted).

well-formedness obligation, explained in Section 4.3.) 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.

```
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
% (from scheme loop_index_recur):
%   (index_var_expr . i_1_ = k_1_)
%   (iter_k_expr . k_1_ = (i_1_ / 1))
%   (initial_bound . TRUE)
%   (final_bound . i_1_ < 1 + m_0_)

% Invariants for variable p
% (from 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.
```

Fig. 3. Additional declarations for evaluator output of 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 with which to assess the code's fitness.

Note that a user need only invoke the evaluator to obtain the results in Fig. 2. Postconditions are not provided, and intermediate assertions such as loop invariants are generated as needed. If a particular code fragment is not covered by the iteration scheme libraries, 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 [1], 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.

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

Invariants cannot, in general, be found algorithmically; they must be derived using heuristic methods. 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 [17]. 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 standard mathematics, solutions to elementary recurrences are functions. For our purposes this would be overly limiting since many invariants state relational properties. Thus, we choose to generalize the problem form 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 R relates next value v to current value u. $P(u, n) \equiv u = 2^n$ is a solution to this recurrence.

Note there are multiple solutions, many of which are trivial. This does not diminish the utility of the approach because authors of iteration schemes provide the solutions, which we expect to be strong enough to serve as effective invariants. Moreover, predicate-based recurrences support a range of data types by allowing solutions to be quantified expressions and other Boolean expressions.

To simplify tool design, we provide a stylized method to express predicate recurrences directly in PVS theories. Each theory includes a lemma and proof that the solution satisfies the recurrence. The proof is constructed by the developer of the iteration scheme using the PVS interactive prover. This setup allows schemes to be developed "offline" and saved in a library. Afterward, they will be available for use by the evaluator, where they will be tried automatically.

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, i.e., it is basically a template. Schemes typically capture the behavior of a single program variable. In this case, variable `p` of Fig. 2 is a matching dynamic variable of type `int`. 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.

```
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
```

Fig. 4. Excerpt from iteration scheme used in evaluation of Fig. 2.

The heart of the scheme is the pair of declarations `recurrence` and `solution`, 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 use of 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 similar program variables.

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

```
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) :)
```

Fig. 5. Simple example of recurrence and solution (details for Fig. 4).

Recurrence definitions have the type of record of lists of labeled conditions. Two categories of conditions are used: "`each`" conditions must hold before and after every iteration, and "`once`" conditions hold initially or constrain constant expressions. Each list element is a pair (C, Q), giving a condition type C and Boolean expression Q. Use of condition types helps the evaluator provide more precise information during the search and matching phase of invariant synthesis.

Solution definitions have the type of list of labeled invariant expressions, the conjunction of which is a solution predicate. Providing different invariant types helps the evaluator make better use of derived information. For example, `func_val_expr` means an invariant has the form $u = e$. Definitions from a meta-model, expressed in a separate PVS theory, denote what it means for a solution predicate to satisfy its recurrence relation.

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

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

For typical bindings of `f`, such properties can be proved automatically by PVS. Other uses for function variables in schemes include accommodating ascending and descending iteration, or both directions of ordering relations, as well as representing mappings between implicit index k and loop counters in the code.

3.3 Instantiating Iteration Schemes

Given a loop body S and dynamic variable x, two 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.

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

During evalution, applicable schemes will be searched and matches attempted. Only after all recurrence conditions are met, which requires theorem proving, will

a matching scheme be recognized. Included are any conditions needed to ensure the invariants hold in the initial state. After a successful match is found, each `solution` expression will be instantiated with terms from the pattern variables and emitted as invariants.

Matching and condition checking play a role similar to unification in the application of lemmas by a theorem prover. In fact, schemes can be thought of as lemmas in a meta-theory of iteration for a model of computation.

In the running example of Fig. 2, the loop body has an effect on variable `p` that is recorded as `p_2_ = p_1_ + n_0_`, with the initial condition `p_0_ = 0` (see Fig. 3). This would satisfy the recurrence condition of Fig. 5, leading to the invariant `p_1_ = k_1_ * n_0_ + 0`.

An important feature 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 shown to hold and each scheme solution is proved 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 Special Features

Several features increase the range of invariant synthesis. First is the ability to specify that a scheme depends on other facts that would be generated as separate invariants. Consider the example of Fig. 6, which is a more realistic multiply algorithm, similar to a hardware shift-and-add algorithm. Three dynamic variables are used, where `y` is doubled each time, `d` is halved, and `p` is the partial product. 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`. Fig. 7 shows the relevant evaluation details.

A second feature favors writing schemes with explicit function application instead of simple expressions. For instance, the following recurrence conditions

```
(dep_index,     i = g(k)),
(dyn_expr_func, j = f(i)),
```

include function-valued variables `f` and `g`. The evaluator applies lambda abstraction to convert an expression such as `k+1` into `(LAMBDA (i: nat): i+1)(k)` so that `f` will match `(LAMBDA (i: nat): i+1)`. This binding can then be used to instantiate a solution predicate such as

```
all_in_bounds(f o g, k, n) IMPLIES
    FORALL (q: below(k)): A(f(g(q))) = h(g(q))
```

Despite increasing the difficulty of creating schemes, this technique makes them less sensitive to the form of expressions needing to be matched.

```
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_ #)

  WFO: boolean = TRUE
END add_mult_exp_deval
```

Fig. 6. A more realistic multiply algorithm and its evaluation.

```
% Invariants for variable d (from scheme div2_exp2_recur):
%   d_1_ = floor((m_0_ / (2 ^ k_1_)))

% Invariants for variable y (from scheme double_exp2_recur):
%   y_1_ = (n_0_ * (2 ^ k_1_))

% Invariants for variable p (from scheme exp2_mult_recur):
%   p_1_ = m_0_ * n_0_ - floor((m_0_ / (2 ^ k_1_))) * (2 ^ k_1_) * n_0_

% Values of dynamic variables on (normal) loop exit:
k_2_: nat
d_3_: nat = 0
y_3_: int = n_0_ * (2 ^ k_2_)
p_4_: int = n_0_ * m_0_
```

Fig. 7. Invariants derived for example of Fig. 6.

A third feature adds a category of auxiliary facts to the solution part of iteration schemes. This is intended for facts that follow directly from recurrence solutions, so they can be proved without the usual inductive reasoning. A typical use of such facts (shown below) is to deduce final variable values when the loop exit condition is reached.

```
(final_func_expr, d = 0 IMPLIES p = y0 * d0)
```

The last feature concerns the additional exit paths created when loops are exited via `return` and `break` statements. In some contexts, loop exits can 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 $\forall\, j < k : \neg P(j)$. An iteration scheme covers this case.

4 Deductive Evaluation

Given C code transformed to ASTs and rendered as s-expressions, the deductive evaluator attempts to infer effects produced by the code when executed. The evaluator works in units of individual C functions. Each will have its effects modeled symbolically via a PVS theory. Much of the processing draws from established techniques; the novel parts concern loop handling.

4.1 Path Analysis

A C function is analyzed in stages: 1) formal parameters, 2) local variable declarations, 3) statement block, and 4) returned result and call-by-reference parameters. For a 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.

Evaluation proceeds in a forward direction, in the same manner as 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 represented by new 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 the allocation of new constants to theory `F_deval` and the updating of state vector(s). Array assignments make use of "function update" expressions in PVS. Assignment `A[i] = A[j]` 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, however, the paths are unified at the close of a conditional statement. Because PVS provides conditional expressions, a symbolic value after an if-statement can have 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`.

Throughout the evaluation process, small deduction steps are carried out to check conditions or simplify expressions. The PVS theorem prover, invoked programmatically, is used for this purpose.

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 the way conditional statements are handled, 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 is ready to find invariants. For each dynamic variable x, the invariant synthesizer is called with B, I, U, V, and supporting information, where the searching and matching process described in Section 3 is carried out. If it returns an invariant Q, it will be saved along with some context information. If a candidate loop index variable exists, it is handled first as a special case using a dedicated category of iteration schemes. Other variables will be matched against a subset of invariant types as appropriate. 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 iteration 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) IMPLIES i = n + mod(i0 - n, d)),
```

These derivations, in turn, are used to help deduce final value expressions for other dynamic variables. New final-value constants are generated, then substituted into each final expression.

Afterwards, the evaluator will have a new state vector W that characterizes variable values immediately after termination of the loop. W will be used to continue evaluation, should there be more statements on the path that contains the loop. 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

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. 8 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 with this technique.

```
void array_init(
       int A[],
       unsigned int n,
       int v) {
  int i;
  int 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
```

Fig. 8. Initialization of n array elements.

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 help constrain what is known about array accesses within loops. If we can infer $i < m$ for all iterations, we can generate the WFO $m \leq S$. Special schemes are provided to help establish these bounds. Fig. 8 illustrates this technique.

4.4 Array Examples

Fig. 9 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 postconditions, although the essential constraints would be the same.

Note that the bubble sort example has nested loops. Generally these can be handled by the evaluator without special techniques, provided there are iteration schemes that deduce outer loop behavior from inner loop effects. Note also the appearance of the PVS function, `permutation_of?`, which is found in NASA

```
int array_min(
         const int A[],
         unsigned int nm1) {
  int i;
  int 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_)))}
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 = 0;
  unsigned int j = 0;
  int t;
  while (i < nm1) {
    j = i + 1;
    while (j < 1 + nm1) {
      if (A[j] < A[i]) {
        t = A[i]; A[i] = A[j];
        A[j] = t; }
      j++; }
    i++; } }
```

```
  . . .
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 =
       (nm1_0_ <= A_size_) AND
       (1 + nm1_0_ <= A_size_)
END bubble_sort_deval
```

Fig. 9. Evaluation results for common array algorithms.

Langley's PVS library collection [20]. Relying on the extensive formalizations in such libraries enhances the feasibility of the iteration scheme approach.

4.5 End-to-End Verification

Under investigation is a means to extend the evaluation machinery so end-to-end properties of inverse operations can be established automatically. Consider the

problem of showing that two functions achieve lossless data compression. We could construct a C function to represent their combined effect, in a manner similar to how one might construct a test case:

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

We would like to infer that $A = C$ when execution of this function ends. Furthermore, it is desirable to prove the overall property without the user ever having to construct specifications for `compress` and `decompress`, or to reason explicitly about their interaction. If this can be done in a fully automatic way, it would constitute a form of "don't-care verification," where only the high-level result is of interest and all the lower-level formalization and proof is left as an exercise for the tool.

Although this work is still in progress, the outlook is promising. Assume the evaluator has derived $P(A, B)$, the behavior of `compress`. Doing the same for `decompress` is problematic because the function does not process arbitrary values of array B. Analyzing `decompress` requires that we restrict B's values according to the data format computed by `compress`. Conventional verification tools use preconditions for this purpose, although they generally must be provided by users. What we hope instead is to work from $P(A, B)$.

Two approaches are under study. The first would create a modified form of the `decompress` function in which the type of input array B is constrained by $P(A, B)$ 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 $P(A, B)$. Experiments with these two approaches are currently underway for a simple data compression algorithm.

If end-to-end evaluation can be achieved, a valuable form of automated analysis would result. Given that many low-level operations and services come in complementary pairs, there would be ample opportunities to apply this technique. Although it would not establish full functional correctness, it would provide strong assurances nevertheless. Generalization to other function combinations and properties should be feasible as well, effectively mimicking the concept of algebraic specification directly in C code.

5 Related Work

Early work on invariant generation [25] began shortly after the seminal papers on verification by Floyd and Hoare. The last 10–15 years have seen a wide variety of investigations into loop invariant generation. Two broad categories of techniques have been pursued.

The first category seeks to generate plausible invariant candidates using dynamic methods. Often these are not guaranteed to be invariants; they might fail

to hold in some cases. Daikon [6] is a leading tool of this type. InvGen [12] uses dynamic techniques to improve the performance of its static techniques.

The second category uses logical and mathematical techniques to generate invariant formulas. One popular group of methods is based on the underlying idea of predicate abstraction [11]. SMT-based implementations of invariant generation [24] sometimes couple this idea with templates to seed the search process. An approach aimed at Frama-C likewise uses predicate abstraction [16] with predicates supplied as hints.

First-order theorem provers such as Vampire provide the substrate for several generation methods [18, 15]. 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. Other ideas exploit refinement and iterative invariant strengthening [22].

Recent verification tools for C have exploited the power of modern SMT solvers to update the classic notion of program verifiers. VCC [2] and Frama-C [5] both carry out functional verification with high automation, provided that specifications and loop invariants are provided by users. A similar tool performs the same tasks for the custom language Dafny [19].

Other tools have addressed lighter-weight forms of verification. ESC/Java [7] uses deductive techniques, but only to verify selected properties. It relies on some user annotations and hints. Verification using strongest postconditions was proposed as a means to achieve reverse engineering of existing software [10].

A verification technique hosted within Java PathFinder [22] combines ideas from symbolic execution, model checking and invariant generation. It was demonstrated on several Java methods that compute using arrays or pointers.

A tool called Valigator [13] was developed as an automated program verifier using SMT solvers for back-end deduction. It generates polynomial invariants for loops that compute with numbers.

Abstract interpretation [3] can be used to derive conservative loop properties. These constitute valid constraints, although they might not be as strong as human-generated invariants. Nevertheless, leading tools in this category [4] can conduct analyses on a realistic scale.

6 Conclusions

A preliminary framework for deductive evaluation and invariant synthesis has been demonstrated. Improvements to both theoretical and implementation aspects are anticipated. The basic concepts should be portable to other theorem provers having features comparable to PVS. An expanded tool configuration, such as integration with SMT solvers or computer algebra systems, would likely bring greater capabilities. Specialized invariant generation heuristics also could be integrated to supplement the generic approach.

The framework is still in an early stage of development. A more complete implementation 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. Using the framework would yield results of varying utility, but given the high level of automation, users should find the effort worthwhile. A favorable cost-benefit tradeoff is likely, similar to that of some static analyzers.

Inherent limitations of the current prototype suggest several areas needing improvement. These include expanding the language features supported, canonicalizing expressions to compensate for syntactic condition matching, populating the iteration scheme library, improving the searching and matching efficiency, identifying principles for orderly library development to minimize problems such as overlapping schemes, and strengthening the theoretical justification for the overall framework. As currently formulated, deductive evaluation would reach a complexity ceiling as larger units of code are attempted. Methods to increase modularity would therefore be valuable.

We speculate that hundreds of iteration schemes, possibly a few thousand, would be needed for adequate coverage of common C functions. Our experience with the NASA PVS library [20] (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 can grow as long as deduction-library developers remain active.

A more extensive version of the framework could be embedded within an IDE and could generate results usable by developers without specialized training. In particular, the evaluator output presented to a user could take forms different from that shown in the paper, likely using customization to make feedback more familiar and directly relevant to a user's needs. While a deductive evaluator would solve only a subset of verification problems, it could be coupled with testing-based methods to achieve high levels of assurance. Alternatively, the invariant synthesizer and evaluator could possibly function as external components for verification tools such as VCC [2] and Frama-C [5].

References

1. Baudin, P., Fillitre, J., Hubert, T., March, C., Monate, B., Moy, Y., Prevosto, V.: ACSL Specification Language (2013), `http://frama-c.com/acsl.html`
2. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying Concurrent C. In: Theorem Proving in Higher Order Logics, TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer (2009)
3. Cousot, P., Cousot, R.: 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. pp. 238–353 (1977)
4. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., Rival, X.: The ASTREE analyzer. In: European Symposium on Programming (ESOP'05). LNCS, vol. 3444, pp. 21–30 (2005)

5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: SEFM. LNCS, vol. 7504, pp. 233–247. Springer (2012)
6. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27(2), 99–123 (2001)
7. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI '02. pp. 234–245. ACM, New York, NY, USA (2002)
8. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. Symp. in Applied Math., vol. 19, pp. 19–32. Providence, RI (1967)
9. Furia, C.A., Meyer, B.: Inferring loop invariants using postconditions. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) Fields of Logic and Computation, pp. 277–300. Springer (2010)
10. Gannod, G.C., Cheng, B.H.C.: Strongest postcondition semantics as the formal basis for reverse engineering. In: WCRE 95: Proceedings of the Second Working Conference on Reverse Engineering. p. 188. IEEE, Washington, DC (1995)
11. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: 9th CAV. LNCS, vol. 1254, pp. 72–83. Springer (1997)
12. Gupta, A., Rybaldchenko, A.: InvGen: An efficient invariant generator. In: CAV. LNCS, vol. 5643, pp. 634–640 (2009)
13. Henzinger, T.A., Hottelier, T., Kovács, L.: Valigator: A verification tool with bound and invariant generation. In: Proc. of 15th Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 333–342. LPAR '08, Springer (2008)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (Oct 1969)
15. Hoder, K., Kovács, L., Voronkov, A.: 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. pp. 1–15. MICAI'11, Springer (2011)
16. Kalyanasundaram, K., Marché, C.: Automated generation of loop invariants using predicate abstraction. Tech. Rep. 7714, INRIA (Jul 2011)
17. Kovacs, L., Jebelean, T.: Automated generation of loop invariants by recurrence solving in Theorema. In: Proc. 6th Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04). pp. 451–464 (2004)
18. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: In Proc. of FASE (2009)
19. Leino, K.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16. LNCS, vol. 6355, pp. 348–370. Springer (2010)
20. NASA Langley Research Center: PVS library collection, theories and proofs available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`
21. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Saratoga, NY (Jun 1992)
22. Pasareanu, C.S., Visser, W.: Verification of Java programs using symbolic execution and invariant generation. In: 11th International SPIN Workshop, Barcelona, Spain. LNCS, vol. 2989, pp. 164–181 (Apr 2004)
23. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Transactions on Software Engineering 24(9), 709–720 (Jun 1998)
24. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Proceedings of PLDI (2009)
25. Wegbreit, B.: The synthesis of loop predicates. Comm. ACM 17(2), 102–112 (1974)

A Additional Material

This appendix contains optional material for discretionary use by the reviewers and is not intended for the final paper. It shows two PVS theories: the iteration schemes used for the inner and outer loops in the bubble sort example of Fig. 9.

```
exchange_order_recur [n: nat] : THEORY
  BEGIN

  IMPORTING prog_types
  IMPORTING structures@permutations[n, int]

  real_rel:     TYPE = [real, real -> bool]

  dyn_vars:     TYPE = [int_array(n), nat]
  stat_vars:    TYPE = [nat,  [nat -> nat], real_rel, real_rel]

  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_relation

  recurrence(I, S)(U, V, k): recur_cond =
    LET (A0, j0_) = I, (i, f, R, Req) = S, (A, j) = U, (vA, vj_) = V IN
      (# each := (: (iter_effect,
                     (in_bounds(i, n) AND in_bounds(j, n)) AND
                     vA = IF R(A(j), A(i))
                             THEN A WITH [(i) := A(j), (j) := A(i)]
                             ELSE A
                          ENDIF),
                    (dep_index, j = f(k)),
                    (dyn_cond,  i < j) :),
         once := (: (stat_cond, R = reals.< OR R = reals.>),
                    (stat_cond,
                     FORALL (p, q: nat): p < q IMPLIES f(p) < f(q)),
                    (let_var,
                     Req = IF R = reals.< THEN reals.<= ELSE reals.>= ENDIF) :)
       #)

  solution(I, S)(U, k): invar_list =
    LET (A0, j0_) = I, (i, f, R, Req) = S, (A, j) = U IN
      (: (rel_val_expr,
          in_bounds(i, n) AND all_in_bounds(f, k, n) IMPLIES
              (FORALL (p: below(k)): Req(A(i), A(f(p))))
              AND permutation_of?(A, A0)
```

```
            AND FORALL (p: below(n)):
                  (p < i OR f(k) - 1 < p) IMPLIES A(p) = A0(p))
     :)

  swap_is_perm: LEMMA FORALL (A: int_array(n)), (i,j: below[n]):
                       permutation_of?(A WITH [(i) := A(j), (j) := A(i)], A)

  recur_satis: LEMMA sat_recur_rel(solution, recurrence)

  END exchange_order_recur



sorted_array_recur [n: posnat] : THEORY
  BEGIN

  IMPORTING prog_types
  IMPORTING structures@permutations[n, int]

  real_rel:     TYPE = [real, real -> bool]

  dyn_vars:     TYPE = [int_array(n), nat, nat]
  stat_vars:    TYPE = [nat, real_rel]

  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_predicate

  recurrence(I, S)(U, V, k): recur_cond =
    LET (A0, i0_, j0_) = I, (m, Req) = S, (A, i, j) = U, (vA, vi_, vj_) = V IN
      (# each := (: (iter_effect,
                     (in_bounds(i, n) AND m < n) AND
                     (FORALL (p: below(m-i)): Req(vA(i), vA(1+p+i))) AND
                     permutation_of?(vA, A) AND
                     (FORALL (p: below(n)):
                         (p < i OR m < p) IMPLIES vA(p) = A(p))),
                    (dyn_cond,  i = k) :),
         once := (: (stat_cond, Req = reals.<= OR Req = reals.>=) :)
       #)

  solution(I, S)(U, k): invar_list =
    LET (A0, i0_, j0_) = I, (m, Req) = S, (A, i, j) = U IN
      (: (rel_val_expr,
          k <= m AND m < n IMPLIES
              (FORALL (p: below(k)): Req(A(p), A(p+1))) AND
              permutation_of?(A, A0)),
```

```
        (supporting_fact,
         k <= m AND m < n IMPLIES
             (FORALL (p: below(k)):
                 FORALL (q: subrange(k, m)): Req(A(p), A(q))))
      :)


  subarray_perm((A, B: int_array(n)), (i, j: below(n))): bool =
      EXISTS (f: [subrange(i, j) -> subrange(i, j)]):
          bijective?(f) AND FORALL (k: subrange(i, j)): A(k) = B(f(k))

  perm_subarray_perm: LEMMA FORALL (A, B: int_array(n)):
      n > 0 IMPLIES (permutation_of?(A, B) IFF subarray_perm(A, B, 0, n-1))

  same_perm_subarray:
      LEMMA FORALL (A, B: int_array(n)), (p: below(n)):
                n > 0 AND subarray_perm(A, B, 0, n-1) AND
                (FORALL (k: below(p)): A(k) = B(k))
              IMPLIES subarray_perm(A, B, p, n-1)

  same_perm_subarray_both:
      LEMMA FORALL (A, B: int_array(n)), (p, q: below(n)), (d: nat):
                n > 0 AND p <= q AND d = n - 1 - q AND
                subarray_perm(A, B, 0, n-1) AND
                (FORALL (k: below(n)): (k < p OR q < k) IMPLIES A(k) = B(k))
              IMPLIES subarray_perm(A, B, p, q)

  recur_satis: LEMMA sat_recur_rel(solution, recurrence)

  END sorted_array_recur
```