Validation of Array Accesses: Integration of Flow Analysis and Program Verification Techniques*

*Raymond W. Lo*** <lo@cs.ucdavis.edu>
Karl N. Levitt <levitt@cs.ucdavis.edu>
Ronald A. Olsson <olsson@cs.ucdavis.edu>

Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
Phone: (916) 752-7004
Fax: (916) 752-4767

September 9, 1997

Abstract

A program that accesses an out-of-bound array element can cause unexpected behavior that is unacceptable to safety-critical or security-critical systems. Two traditional compile-time approaches to array bound checking are flow analysis and program verification.

This paper presents a new approach, *IFV*, that integrates flow analysis and program verification techniques. *IFV* is generally about as effective as program verification yet runs in about the same time as flow analysis. Its complexity is of the order of the product of the program size and the number of declared variables.

IFV matches loops to templates, which represent commonly occurring loop patterns, to automatically discover loop invariants, which it then uses to strengthen flow analysis. With only seven templates, it handles many common array-access patterns. Patterns not verified by flow analysis are processed with verification techniques, entirely automatically.

This paper also describes a prototype *IFV* system that performs compile-time array bound checking for programs in a subset of C.

Keywords

Software Analysis, Array Bound Checking, Range Analysis, Flow Analysis, Program Verification, Invariant Templates.

---

* This work is supported by the United States Department of Defense, Lawrence Livermore National Laboratory, and Deloitte Touche.

** Current address: Raymond W. Lo, Silicon Graphics, Inc., 2011 N. Shoreline Blvd., P.O. Box 7311, M/S 10U-178, Mountain View, CA 94039-7311.

1. Introduction

An array access is an expression of the form `array[expr]`. The access is correct when the subscript (the value of `expr`) is within the declared bounds of `array`. Demonstrating the correctness of array accesses is important because an undetected out-of-bound access often leads to incorrect output and can cause unexpected program behavior that results in safety or security problems. For example, a devious user might be able to circumvent a login program's normal password checking by giving input that overflows an array in the program. Array overflow in the UNIX `fingerd` (finger daemon) program was exploited by the Internet worm, and similar vulnerabilities exist in other security-critical programs. Dealing with accesses within loops is one of the key aspects in checking array bounds. Determining whether an array access is out-of-bounds is, in general, an undecidable problem, and therefore run-time bounds checking is necessary. Many language implementations do not validate the correctness of array accesses because of the additional run-time costs. In many cases, however, compile-time techniques can be successfully applied, obviating many run-time checks; the checks that are not validated at compile-time will be deferred until run-time. Of course, it is desirable to detect out-of-bounds accesses at compile-time when possible.

Two traditional compile-time approaches are flow analysis and program verification. Flow analysis techniques track the possible values assumed by index variables at various points in a program. Program verification techniques have the same goal, but they represent the values symbolically with assertions that, if proved, show the correctness of array accesses. Flow analysis techniques run in polynomial time. Automatic program verifiers are generally more effective at bounds checking than are flow analysis tools, but in some cases they run in exponential time or require substantial human assistance.

This paper presents a new approach that integrates flow analysis and program verification techniques. The approach is designated *IFV*, for *I*ntegrated *F*low Analysis and Program *V*erification. *IFV* is nearly as effective as program verification, yet runs in about the same time as flow analysis.

IFV matches loops to templates for commonly occurring loop patterns. With only seven templates, it handles many common array-access patterns. Using the templates, it generates simple loop invariants for the purpose of checking array accesses statically. These loop invariants capture the relationship among the variables that control execution of the loop as a recurrence relation; they do not, however, attempt to determine the overall result of executing the loop. The loop invariants strengthen flow analysis in two ways: fewer passes are required on the average and more checks are validated. Those checks that are not validated by range analysis are handled by applying program verification techniques. In generating the assertions, data flow information is used to select only relevant constraints and definitions, so that the time required to verify an array access depends only on the complexity of its associated recurrence relation and not on the size of the program. Therefore, *IFV* scales to large programs.

IFV further reduces the execution time of the verification phase by using data-flow information to generate verification conditions (VCs) and restricting the VCs to be in the form of propositional logic and linear inequalities. The verification condition generator (VCG) includes only relevant information in the VC and produces relatively simple theorems to be proved. As a result, the VCG also runs in polynomial time as opposed to the exponential time taken by straightforward traditional approaches. (Such an approach generates a VC for every possible path leading from the program entry to the point of an array reference, i.e., potentially an exponential number of such paths; more sophisticated approaches are possible and less expensive). Furthermore, theorems expressed in this restricted form can be proven automatically and without using heuristics.

Ignoring the theorem proving part, the overall average execution time is linear in the program size (number of nodes in the program graph) times the number of variables declared in the program. Compared with Cousot's approach (Cousot *et al.*, 1978) or Suzuki's approach (Suzuki, 1977), *IFV* handles some cases they cannot and they handle some cases *IFV* cannot; *IFV* runs in less expected time. *IFV* is much more powerful than Harrison's approach (Harrison, 1977) but runs with similar complexity. Compared with Gupta's approach (Gupta, 1990; Gupta, 1993) , *IFV* can remove all optimized checks and more (due to inductive information).

IFV is not as general as what can be done with a full-scale programmer-assisted program verifier, but it performs better than a pure flow analysis or verification approach. The benefits of an automated array bound checker outweigh its limitations. Furthermore, those few cases that *IFV* cannot handle can be deferred to a general program verifier.

A prototype system that uses the *IFV* approach has been implemented. It performs compile-time array bound checking for programs in a subset of C. The results from this prototype are promising.

The rest of this paper is organized as follows. Section 2 summarizes related work, particularly flow analysis and verification approaches. Section 3 presents *IFV*. Section 4 discusses the prototype implementation of *IFV* and its performance. Finally, Section 5 describes some advantages and disadvantages of *IFV*. Further details and applications of *IFV* appear in (Lo, 1992). A variant of the template approach has been applied in SGI's MIPS 7.x C compiler to normalize loops into a canonical form for easier analysis for optimizations (Liu *et al.*, 1996). The notion of showing that a program meets certain key properties, without requiring a complete proof of the program's correctness, has also been used in the work on property-based testing of security properties (Fink *et al.*, 1994).

2. Related Work

This section first describes previous approaches to array bounds checking. It then describes some of the problems in using those approaches and how *IFV* aims to solve those problems.

2.1. Other Approaches

Researchers have worked on the detection of out-of-bound accesses for many years. Approaches to detection fall under three broad categories: checking array subscripts at run-time, performing flow analysis at compile-time, and program verification. The run-time approach inserts code before array accesses to check the subscript value and to raise an error if the subscript value is out-of-bound. This approach is simple and straightforward to implement, but the code inserted may increase a program's execution time and code length unacceptably, particularly for scientific programs that involve many array accesses. Also, errors are unacceptable in many applications. One solution to reduce the overhead is to apply compile-time optimization techniques. Gupta (Gupta, 1990; Gupta, 1993) identifies two types of optimization that reduce the static and dynamic number of array bound checks. First, flow analysis determines checks that have been performed previously and are still valid; this information is used to eliminate redundant checks. Second, code motion (moving checks out of loops) reduces the dynamic number of checks. The first optimization may be combined with the program verification approach to reduce the number of checks to be verified. Recent work (e.g., (Asuru, 1992; Austin *et al.*, 1994; Chin *et al.*, 1995; Kolte *et al.*, 1995)) provides further optimizations and techniques building on Gupta's approach, but these still require run-time checks. Nevertheless, a run-time approach is not suitable for safety-critical applications where run-time errors cannot be tolerated.

Program verification proves array accesses correct and identifies out-of-bound accesses from unsuccessful proofs. The approach typically consists of three steps. The first step identifies loop invariants (user-defined assertions). The second step creates a verification condition (VC) for each array access by back-propagating the goal (the array check predicate) and collecting the branch conditions and loop invariants. The third step proves the VCs using a theorem prover. An array access is considered to be correct only if the corresponding VC is proved correct.

Boyer and Moore (Boyer *et al.*, 1981) developed a verification condition generator for array accesses in FORTRAN 66. A VC is generated for each appearance of an array access in a program. The disposition of these (and, of course, all other) VCs usually requires the creation of loop invariants.

Suzuki developed the induction iteration method (Suzuki, 1977) to generate candidate loop invariants automatically by back-propagating array access predicates through loops and combining them with various branch conditions. The validity of the candidate invariants must then be determined using a theorem prover that has the decision capability for linear arithmetic and boolean formulas. The invariant generation, VC generation, and theorem proving in Suzuki's method are performed at the same time—candidate loop invariants also serve as VCs.

Nagle developed a system (Nagle *et al.*, 1983) (which also detects integer overflows and undefined variables) to verify array accesses. This system requires the programmer to state all relevant assertions and loop invariants. VC generation is automatic, as is the case in all verification systems. Each array bound check is considered one goal to be proved and a VC is generated for every possible path leading to the goal. A check is validated if its VC can be proved by the system's theorem prover, which is rewrite-rule based and has built-in linear arithmetic and proposition logic capabilities. The required rewrite-rules are supplied by the programmer. Detlefs is developing the Extended Static Checking (ESC) system (Detlefs, 1996). ESC, like Nagle's system, uses program verification to verify array accesses. (It also detects null pointer problems and some problems, e.g., race conditions, in concurrent programs.)

Flow analysis, which is often used in compiler optimizations, is an alternate approach used for detecting out-of-bound array accesses. Harrison (Harrison, 1977) uses flow analysis to determine the bounds on the ranges of values assumed by integer variables at various points in the program. The ranges are then used to prove the checks. Some important information is ignored by flow analysis and therefore this result is weaker than that of program verification (see Section 3.3).

Cousot (Cousot *et al.*, 1978) also uses flow analysis to solve this problem, but this method propagates systems of linear inequalities that describe relations among variables. In each flow analysis step, the convex-hull of systems of linear inequalities are computed and simplified to a minimal system — i.e., one that contains no "irrelevant" inequalities, yet converges to the same result (Cousot *et al.*, 1978). Consider, for example, an initial system that contains some redundant inequalities: $a + b \leq 10$, $b > 0$, and $a \leq 10$. The algorithm would discover that $a \leq 10$ is redundant and so would exclude it from the minimal set. Although this method discovers more relations than Harrison's, it requires much more computation. For the convergence of the analysis, Cousot needs a heuristic (widening operator) (Cousot *et al.*, 1978) to discard the inequalities that cause the system not to converge within some prescribed time or number of iterations during the iterative analysis; discarding such inequalities generally reduces the precision of the results.

2.2. Problems with Other Approaches

A programmer-assisted program verification system can handle most programming patterns, but it demands considerable expertise and time from the programmer. A program verification system that automatically generates verification conditions can handle many cases, but it needs unrealistic computational resources for larger programs (e.g., Nagle's method) and may never terminate for programs with errors (e.g., Suzuki's induction iteration method). Flow analysis is less computationally intensive and suitable for use as a software engineering tool, but it fails to process loop-related and symbolic information (e.g., Harrison's method). The following lists some of the existing problems that are important to solve and some of the improvements that *IFV* achieves.

Gupta (Gupta, 1990; Gupta, 1993) observes that a large number of checks can be eliminated by optimization. Therefore, a large portion of checks performed by Nagle's method and by Suzuki's method is not necessary if the program is preprocessed using flow analysis. *IFV* uses flow analysis to perform the optimization before applying the VCG. As demonstrated below, in the examples presented by Nagle and Suzuki, a majority of the checks can be verified with Harrison's flow analysis; almost all are verified with *IFV*'s enhanced flow analysis. The use of the more expensive program verification techniques is not necessary in many cases.

Flow analysis does not discover loop invariants, and integer range flow analysis (Harrison, 1977) is not very useful without these invariants. Harrison (Harrison, 1977) strengthens this analysis by deriving simple inductive information in the form of single-variable monotonic (linear) recurrence relations, i.e., $var_{i+1} = const_1 \times var_i + const_2$. Still, the invariants so discovered are not strong enough to prove some array accesses that occur commonly in realistic programs. *IFV*'s template approach generates stronger invariants. For example, consider the simple, but representative programs in the Appendix. The invariants discovered by Harrison's method are too weak to prove all array accesses for Examples 1, 3, 4, and 5. By contrast, the invariants discovered by *IFV* are strong enough to prove all array accesses for Examples 1, 2, 3, and 4, as well as other programs containing loops that are represented by the examples' loops. (See Section 4 for further discussion.)

Consider the following computation using interval arithmetic.[1]

```
        /* bound(a) = (0, 10) */
    b=100-a;
        /* bound(b) = (100, 100) − bound(a) = (90, 100) */
    c=a+b;
        /* bound(c) = bound(a) + bound(b) = (0, 10) + (90, 100) = (90, 110) */
```

Interval arithmetic always assumes the worst case. Although the expression `c` should have the value 100, flow analysis computes the value using interval arithmetic and the result is (90, 110). *IFV* handles this pattern with verification techniques.

[1] Section 3.2 describes interval arithmetic in detail.

Harrison's flow analysis (Harrison, 1977) fails to obtain information from boolean expressions with subexpressions ordered in certain ways. For example, flow analysis can deduce $b \leq 100$ from the boolean expression $a \leq 100 \wedge b \leq a$, but it fails to deduce that from the equivalent expression $b \leq a \wedge a \leq 100$ because the flow analysis is always done from left to right. *IFV* solves the problem by evaluating the expressions repeatedly until no additional information can be deduced. Presburger arithmetic (Shostak, 1977), an alternative to *IFV*, solves a system of linear equalities, but it is potentially much more expensive than Harrison's method.

Nagle's method (Nagle *et al.*, 1983) verifies most correct array accesses because a programmer must provide the necessary invariants and rewrite-rules. The main argument against this method is that it is very hard for a programmer to write assertions; i.e., a programmer must be sufficiently proficient in logic to write the assertions needed to prove the goals.

Nagle's emphasis on covering every pattern generates an exponential number of VCs because this VCG produces one VC for each goal reachable from every path. That is, a program with x subscript checks following y if-then-else statements produces $x \times 2^y$ VCs. This exponential growth limits the system's usefulness to very small programs. *IFV* avoids the exponential growth by generating simpler VCs.

Suzuki's inductive iteration method may never terminate, particularly when the program is not correct with respect to array accesses. The size of the generated assertion doubles each time when it is back-substituted through a conditional statement. Contrasting with Nagle, Suzuki chose to generate one VC for one goal regardless of the number of paths. Therefore, the size of VCs instead of the number of VCs grows exponentially. Since the performance of the theorem prover is exponential in the size of formula, Suzuki must minimize its size with a theorem simplifier. The *IFV* loop invariant and VC generator runs in polynomial time.

Cousot's method is quite similar to Suzuki's inductive iteration method. Both methods suffer from exponential growth and require heuristics to cope with very large formulas.

3. An Integrated Approach

IFV uses program verification techniques to complement flow analysis. It uses deterministic and polynomial techniques. The array bound checker has four main steps (excluding the theorem prover). As shown in Fig. 1, the array bound checker converts a source program into an intermediate form, discovers loop invariants, computes integer ranges (as part of flow analysis), verifies the array checks, and finally generates a list of VCs for those accesses not verified. The VCs are then sent to the Boyer-Moore (Boyer *et al.*, 1988) theorem prover to be proved. Other, less powerful or special purpose theorem provers would also be suitable, e.g., those optimized for linear arithmetic, such as PVS (Owre *et al.*, 1992).

The first step translates the source program into an intermediate form that consists of a program graph and several kinds of data-flow information. (*IFV* builds on techniques and terminology common in compilers; Aho, Sethi, and Ullman's textbook (Aho *et al.*, 1986) contains relevant definitions and algorithms.) The intermediate form is used to further determine loops and their corresponding induction-variable candidates. An ***induction variable*** is a variable that is modified in a loop and depends on its value from previous iterations.

The second step generates loop invariants. *IFV* uses symbolic evaluation with the induction-variable candidates to derive their recurrence relations, if there are any. *IFV* then matches these recurrence relations against predefined templates and produces simple loop invariants, which are defined in propositional logic and systems of linear inequalities. The invariants so generated are always valid and are usually powerful enough to express the relations between integer expressions used in array subscripts. These invariants are passed to the third and fourth steps. As described below, the third step (Flow Analysis) uses them to validate some of the array checks; the fourth step (VCG) uses them to validate the array checks that remain to be validated after the third step.

The third step is to apply flow analysis techniques, including range propagation and range refinement. Unlike traditional flow analysis, *IFV* takes advantage of the availability of valid loop invariants and proof goals (array bound checks) to restrict the value ranges. German and Wegbreit (German *et al.*, 1975) proposed the use of output assertions (proof goal in *IFV* terminology) to generate loop invariants. Similarly, *IFV* uses the proof goals as invariants to inject inductive information into the flow analyzer to further narrow down the integer ranges (see Section 3.3.3.4). A problem in program verification is that a loop invariant needs to carry predicates that relate variables unchanged in the loop, so that the information can propagate to the later statements that follows the loop. This problem also happens in flow analysis—Harrison called it the interval constant problem (Harrison, 1977). The information flow paths in *IFV*'s flow analysis are slightly different to handle this special case (see Sections 3.3.3.1 and 3.3.3.2).

The fourth step generates the VCs and uses a theorem prover to prove them. In a traditional VCG, the verification of one goal (array check) usually results in many theorems to be proved because of the propagation of the goal through loops and branches in a program. Since *IFV* uses loop invariants that essentially break the cycles in the program, the VCG becomes straightforward—it just collects the predicates and loop invariants. *IFV* ignores the assertions that are not expressible in propositional logic and systems of linear inequalities. *IFV* uses only the propositional logic and linear capabilities of Boyer-Moore theorem prover for this purpose (Boyer *et al.*, 1988).

3.1. Step 1: Transforming the Program into the Intermediate Form

Recall that the purpose of Step 1 is to identify loops in the source program. Later, Step 2 identifies the induction-variable candidates for these loops. To identify loops, *IFV* uses traditional compilation techniques in which the source program is viewed as a program flow graph (Aho *et al.*, 1986). A node in a flow graph corresponds to a basic block in the program, i.e., a sequence of statements into which flow of control enters only at the beginning and leaves only at the end. An edge represents the flow of control, e.g., due to a conditional or looping statement, between basic blocks. A flow graph retains all the control information and the order of variable definitions and references present in the source program; it represents all potential executions of the program rather than any specific execution. Program flow graphs often fall into a class called reducible (Aho *et al.*, 1986) flow graphs, which can be partitioned into a hierarchical structure from which loops can be identified.

The details for Step 1 follow. The source program is translated into a program graph G. To distinguish the predicates associated with the two edges of a branch node, G is transformed further into SG (split graph) by splitting the branch node carrying a condition B into two null nodes carrying the assertions B and $\tilde{}B$, as shown in Fig. 2. All subsequent analysis will be carried out with SG.

IFV assumes that *SG* is a reducible flow graph (flow graphs of structured programs are reducible) for simplicity of discussion, although irreducible flow graphs can be converted to reducible flow graphs using node splitting (Aho *et al.*, 1986).

IFV further assumes that traditional data-flow analysis such as reaching definitions and dominators (Aho *et al.*, 1986) has been carried out and the following functions are provided:

- $mod(n)$ returns the set of variables modified at node n.
- $use(n)$ returns the set of variables used in the computation, assertion, and invariant associated with node n.
- $reach(n, v)$ provides the set of definitions of the variable v that reach node n. A reaching definition is represented by the ordered pair $var : node$.
- $dom(n)$ returns the set of nodes that dominate node n. A node d ***dominates*** node n if every path from the initial node of the flow graph to n goes through d. Two examples: the entry of a loop dominates all nodes in the loop and the testing of an if-then-else statement dominates its two bodies. In *SG*, the nodes dominated by a split branch node, b, is the predicate-region of b. The predicate-region (Ferrante *et al.*, 1983) of a predicate b is a region of influence of b, i.e., b stays valid in the region. *IFV* follows Ferrante's definition, but with a slight variation. Because G was split into *SG*, *IFV* differentiates the true-subregion and the false-subregion of a branch node.

IFV partitions *SG* into ***intervals***, as defined by the T1-T2 transformation (see below) (Aho *et al.*, 1986). The entry node of an interval is called its ***interval header***. An interval (Aho *et al.*, 1986) in the flow graph is a loop structure in which the nodes can reach the interval header plus a dangling structure in which its nodes cannot reach the interval header. A loop can be obtained from an interval by removing its dangling structure.

If a loop contains nested loops, each inner loop is abstracted into one node so that an abstracted loop contains no cycle that does not pass through the loop header (i.e., the corresponding interval header).

The algorithm follows:

- **Interval Partitioning by T1-T2 Analysis.**

  Copy *SG* to *IG*. Set $I(v) = \{v\}$ for each $v \in IG$. Apply the T1 and T2 transformations (defined below) repeatedly until neither can be applied. During the T1 transformation, in which any self-loop (v, v) is removed, the interval $I(v)$ is output. v is called the interval header. The T1 and T2 transformations are defined as follows:

  T1: Delete all self-loops (v, v) from *IG*.

  T2: If (v, w) is the only edge incident onto w in *IG*, and $w \neq v$, delete vertex w from *IG*. For every edge (w, x) in *IG*, add a new edge (v, x) if (v, x) is not already in *IG*, and then delete (w, x). Set $I(v) = I(v) \cup I(w)$.

  Note that T1 and T2 can be applied in any order and yield the same result (Aho *et al.*, 1986). (Because of this non-ordering and because T2 can introduce self-loops, the test $w \neq v$ is not superfluous.)

- **Identifying Loops from Intervals.**

  For each interval $I(v)$ headed by v, construct a subgraph $loop(v)$ from *IG* consisting of all the vertices in the interval. By removing those vertices in $loop(v)$ that do not reach the interval header v, $loop(v)$ now defines a loop entered at node v.

- **Transforming a Nested Loop into a Simple Loop.**
  Suppose *loop*(x) is a nested loop. For each inner loop *loop*(y) of *loop*(x), represent the whole inner loop *loop*(y) with a vertex $loop_y$ in *loop*(x).

Fig. 3 illustrates using the T1-T2 transformation to determine the intervals. Applying T2 to the edge (1,5) results in the interval (combined node) {1,5}. Applying T2 first to the edge (2,3) results in the interval {2,3}. Applying T2 again to the edge from the interval {2,3} to node 4 results in the interval {2,3,4}; this interval is headed by vertex 2. Every node in the interval can reach the interval header, therefore the loop is {2,3,4}. Applying T1 removes the self-loop. Applying T2 produces the interval {1,2,3,4,5} headed by vertex 1. Applying T1 again removes the self-loop. Since node 5 does not reach the interval header, the loop is {1,2,3,4}. In addition, {2,3,4} is an inner loop of {1,2,3,4}. By collapsing {2,3,4} into a single node $loop_2$, the outer loop becomes {1, $loop_2$}. Fig. 4 shows the collapsed loops.

3.2. Step 2: Generation of Simple Loop Invariants

Early research on program verification has attempted to combine algorithmic and heuristic techniques to generate loop invariants (German *et al.*, 1975; Katz *et al.*, 1973; Katz *et al.*, 1976; Wegbreit, 1973). The algorithmic approach generates invariants from assignment statements by propagating predicates and incorporating loop-exit tests. The heuristic approach is based on the assertions provided by a programmer and those generated with the algorithmic approach to generate new candidate invariants by strengthening and pushing them backward and forward through the program. It also uses domain-specific information and uses information from unsuccessful proofs to generate new invariants.

The algorithmic approach generates invariants that are valid but usually not very strong. The heuristic approach may generate stronger invariants but often produces too many invalid candidates. Template-matching in *IFV* uses the domain-specific information (i.e., recurrence relations of the loop induction variables) to generate invariants that are always valid and stronger than those produced with the algorithmic approach. The template-matching approach usually produces invariants strong enough to verify array accesses and with a small number of of templates.

In *IFV*, symbolic evaluation computes the recurrence relation of the induction variables. Those recurrence relations are in turn matched and unified with the predefined templates. An invariant is generated upon successful unification. The examples in Section 3.2.3 illustrate the above actions.

3.2.1. Induction Variables

With the loops identified, *IFV* locates the possible induction variables. The strategy is to first approximate the candidate set for every loop as all integer variables modified in the loop and then to eliminate the variables modified in the inner loops from the candidate set. (This heuristic works well in practice, but other heuristics that do not eliminate such variables could also be used.) The following algorithm determines the induction variables.

```
/*  IV is the candidate set of induction variables */
foreach loop L, working with inner loops first
   IV(L) = null;
    /* initialize the candidate set with integer variable modified in the loop */
   foreach n in nodes of L
      foreach v in mod(n) and v is an integer variable
         IV(L) = IV(L) ∪ {v};
   /* remove the variables that are modified in a inner loop */
   foreach M that is an inner loop of L
      IV(L) = IV(L) − IV(M);
```

Then *IFV* performs symbolic evaluation for exactly one iteration of the abstracted loop (i.e., an execution stops when it reaches the loop header). Symbolic values are assigned for candidate induction variables and variables not changed in the loop at the beginning of the loop. Variables modified in the loop but not in the candidate set have the *bottom* value. The results of symbolic evaluation depend on the type of computation: propositional calculus and linear arithmetic computations produce the usual formula; other, more complicated computations produce an undetermined (*bottom*) result; and computations that depend on a *bottom* value have *bottom* value. A candidate that has non-*bottom* values at the loop header is an induction variable. Its symbolic expression is the recurrence relation that will be matched with templates.

3.2.2. Simple Templates

A programmer can express a computation in many different ways (e.g., using different variable names and different order of execution). *IFV* abstracts the computation to recurrence relations and matches them with the templates listed in Table 1. Upon successful match, *IFV* unifies the recurrence relation with the matched templates and generates a loop invariant expressed in linear equations and propositional logic.

Table 1 uses the following notation. V_0 and W_0 represent initial values for variables V and W. In general, V_i and W_i represent values after execution of i iterations of the loop body. C, C_1, C_2, C_v, C_w represent interval constants, i.e., values that do not depend on the iteration of the loop. $X = any(Y, Z)$ means X can assume values of Y or Z in different passes through the loop.

Advantageously, the template matching method is extensible, as more templates can easily be added to the array bounds checker. In fact, template 1 has been generalized to $V_{i+1} = any(V_i, C_1, C_2, \ldots)$ and the invariant generated is $V_i = V_0 \vee V_i = C_1 \vee V_i = C_2 \vee \ldots)$. The $V_i + C$ part in templates 2 and 3 have been extended similarly. Note that the original template 2 is fairly weak, but it is still quite useful as the examples in the next section show.

3.2.3. Examples

Example 1 (program listings are in the Appendix) contains one loop, in which integer variables `i` and `j` are modified. They form the candidate set for induction variables. The result of symbolic evaluation on `i` and `j` is:

$$i_{l+1} = i_l + 1; \;\; i_0 = 0$$
$$j_{l+1} = any(j_l, j_l + 1); \;\; j_0 = 0$$

These relations unify independently with templates 2 and 5. Their unification with template 2 generates two invariants: $i \geq 0$ and $j \geq 0$. Their unification with template 5 generates one invariant: $(i - 0)/1 \geq (j - 0)/1$, which

	Recurrence Relation	Invariant Generated	Examples of Corresponding Loops
1	$V_{i+1} = any(V_i, C)$	$V_i = V_0 \vee V_i = C$	i=N; while (B) {S1; i=10; S2;}
2	$V_{i+1} = any(V_i, V_i + C)$ $C \geq 0$	$V_i \geq V_0$	for (i=0; B; i++); i=0; while (B1) {S2; if (B2) i++; S3;}
3	$V_{i+1} = any(V_i, V_i + C)$ $C \leq 0$	$V_i \leq V_0$	for (i=N; B; i- -) S; i=N; while (B1) {S2; if (B2) i=i-5; S3;}
4	$V_{i+1} = V_i + C_v$ $W_{i+1} = W_i + C_w$ $C_v \neq 0 \wedge C_w \neq 0$	$(V_i - V_0)/C_v = (W_i - W_0)/C_w$	for (p=0,q=10; B; p++,q++) S;
5	$V_{i+1} = V_i + C_v$ $W_{i+1} = any(W_i, W_i + C_w)$ $C_v \neq 0 \wedge C_w \neq 0$	$(V_i - V_0)/C_v \geq (W_i - W_0)/C_w$	for (p=0,q=10; B; p++) if (B1) q++;
6	$V_{i+1} = any(V_i, V_i \times C_1 + C_2)$ $V_0 \geq 0 \wedge C_1 \geq 1 \wedge C_2 \geq 0$	$V_i \geq V_0$	p=1; while (B) {S; if (B1) p=p*2 else p=p*2+1;}
7	$V_{i+1} = any(V_i, V_i/C)$ $C \geq 1$	$V_i/V_0 \leq 1$	p=N; while (B) {S; if (B1) p=p/2;}

Table 1. Loop Templates

simplifies to $i \geq j$. The results of both these unifications are used in the next step.

Example 2 has a nested loop. The inner loop modifies variables `thisone` and `first`, therefore the candidate set for the inner loop is `thisone` and `first`. The outer loop modifies variables `thisone`, `first`, and `top`. After eliminating the variables belonging to the inner candidate set, the candidate set for the outer loop contains only one variable: `top`.

The result of symbolic evaluation for the inner loop is:

$$thisone_{l+1} = thisone_l + 1;\ \ thisone_0 = top + 1$$
$$first_{l+1} = thisone_l;\ \ first_0 = top$$

Only one unification is possible with the seven templates. The unification of the first recurrence relation with template 2 produces one invariant: $thisone \geq top + 1$.

The outer loop in Example 2 only determines one recurrence relation: $top_{l+1} = top_l + 1;\ \ top_0 = 0$. Its only possible unification, with template 2, produces $top \geq 0$.

3.3. Step 3: Flow Analysis

The range flow analysis derives the worst case bound, valid for all possible execution paths, for each integer variable at each node. The worst case bounds of integer variables, called ***ranges***, are expressed in the format $constant1 \leq var \leq constant2$. The range analysis propagates existing ranges along edges in a program graph, generates a new range when a new value of an integer variable is defined in an assignment node, and then refines these ranges using invariants, assertions, and branch conditions. It continues these propagations and refinements until the ranges change no more. Information expressed as ranges can be computed efficiently using interval arithmetic, but certain information is lost (e.g., $a < b$ cannot be expressed).

A key aspect of flow analysis is that the result is always correct despite the initial ranges. The initial range can be the most pessimistic range $(-\infty, \infty)$ or the most optimistic range $(\infty, -\infty)$. In general, the optimistic initialization generates a stronger result (tighter bounds), but the pessimistic initialization has better convergence behavior. This prototype chooses the pessimistic range for the convergence benefit while strengthening the result with the introduction of assertions and loop invariants. As indicated previously, the ranges so derived are *weak* invariants of the program. That is, the invariants so derived are valid and can be used to verify the array accesses, but they are usually not strong enough to prove program correctness.

3.3.1. Flow Analysis Framework

The range analysis is a forward flow analysis problem. A given forward flow analysis problem is distinguished from others by:

(1) the set V of values to be propagated,
(2) the set F of transfer functions from V to V,
(3) and the confluence operator that combines values from incoming edges.

For example, two different flow analysis problems might differ by their specific confluence operators: one might use union and the other might use intersection.

A forward flow problem can be solved in a framework (Aho *et al.*, 1986; Kildall, 1973) using the following algorithm. Each node in *SG* carries two values—*in* and *out*.

Step 1: Assign $F_n(bottom)$ to the *out* value of each node n in *SG*.
Step 2: Compute the *in* value for each node by confluencing the *out* values of its predecessors.
Step 3: Update the *out* value with $F_n(in)$ for each node.
Step 4: Repeat step 2 and 3 until *in* and *out* do not change.

3.3.2. Range Propagation and Refinement

An ***interval*** is an ordered pair (min, max), where *min* and *max* denote the minimum and maximum value the interval represents. *min* and *max* can have integer values or symbolic values that represent negative infinity and positive infinity—($-\infty$ and ∞). Like integers, intervals can be added (+), subtracted (−), multipled (*), divided (/). Furthermore, they can be intersected (⊗) and unioned (⊕). The operations on intervals are defined as follows:

Expression	Result of Computation
$(a, b) + (x, y)$	$(a + x, b + y)$
$(a, b) - (x, y)$	$(a - y, b - x)$
$(a, b) * (x, y)$	$(min(a * x, a * y, b * x, b * y), max(a * x, a * y, b * x, b * y))$
$(a, b)/(x, y)$ where $x \le y < 0$	$(min(a/x, a/y, b/x, b/y), max(a/x, a/y, b/x, b/y))$
$(a, b)/(x, y)$ where $0 < x \le y$	$(min(a/x, a/y, b/x, b/y), max(a/x, a/y, b/x, b/y))$
$(a, b)/(x, y)$ where $x \le 0 \le y$	$(-\infty, \infty)$
$(a, b) \oplus (x, y)$	$(min(a, x), max(b, y))$
$(a, b) \otimes (x, y)$	$(max(a, x), min(b, y))$

The generic flow analysis framework from Section 3.3.1 can now be specialized for this specific problem.

In the flow analysis framework, a flow value of the range analysis (element of V) is a mapping from integer variables to intervals: a function *bound*: *var*$\rightarrow$(*min*, *max*), where *var* is an integer variable and (*min*, *max*) refers to the interval that the value of *var* may be in.

The transfer function F_n of vertex n can be described in two cases. The first case (range propagation) is when vertex n is an assignment node. The function computes and updates the range of the assigned variable. For example, given the assignment statement `J:=I*2;` and $bound(I) = (10, 20)$, the new range for `J` is $bound(J) = (20, 40)$. The second case (range refinement) is when vertex n is a branch node. *RangeRefine*(B, *env*), defined below, refines the ranges of variables used in the branch condition B. For example, if a path with condition `I<=J+K` is taken and the given *env* is $bound(I) = (10, 20)$, $bound(J) = (-5, 6)$, and $bound(K) = (0, 9)$, then the refined ranges are $bound(I) = (10, 15)$, $bound(J) = (1, 6)$, and $bound(K) = (4, 9)$. To illustrate how some of these refined bounds are obtained, observe that because the bound on I implies that $10 \leq J + K$ portions of the original bounds on J and K are eliminated. As one example, when I takes on its minimal value, 10, J can take on its maximal value, 6, allowing K to take on what is now its new minimal value, 4.

The confluence function is the worst case bounds from different incoming edges, computed by union-ing the intervals. For example, if $bound(H) = (0, 10)$ from one edge and $(20, 30)$ from the other, the confluenced range is $(0, 30)$. Note that how this process loses information: H is no longer known to not be between 10 and 20.

The following functions define the operations on flow values. The notation is: f, g, and *env* are flow values, v and Xs are some integer variables, E is an expression, and Bs are predicates.

$RangeUnion(f, g)(v) = f(v) \oplus g(v)$.

$RangeIntersect(f, g)(v) = f(v) \otimes g(v)$.

RangeExpr(E, *env*) computes the interval for the expression E using interval arithmetic given the current mapping *env* from variables to intervals. The evaluation of E follows the normal precedence of +,-,*,/. If E contains operators other than +,-,*,/, the *bottom* value is returned. Practice has shown that handling only the operators +,-,*,/ yields very good results.

RangeVar(X, B, *env*) computes the possible values for X given *env* and the inequality B consisting of $+, -, *, /, <, >, =, \leq, \geq$. For example, $RangeVar(K, I \leq J + K, env)$ generates the bound $(4, \infty)$ for K when $env(I) = (10, 20) \wedge env(J) = (-5, 6)$.

RangeRefine(B, *env*) uses the linear restraint B to generate tighter ranges for *env*. B is a normalized boolean expression consisting of AND, OR, and linear inequalities. *RangeRefine*(B, *env*) is defined as:

Case 1: B is of the form $And(B_1, B_2, \ldots)$

the fixed point of $env = RangeIntersect(RangeRefine(B_1, env), RangeRefine(B_2, env), \ldots)$

Case 2: B is of the form $Or(B_1, B_2, \ldots)$

$RangeIntersect(env, RangeUnion(RangeRefine(B_1, env), RangeRefine(B_2, env), \ldots))$

Case 3: B is a linear inequality of n variables: $X_1, X_2, \cdots, X_n$

a). more than two variables have *bottom* range *env*

b). $n - 1$ variables are defined. (i.e., X_i is *bottom*)

$RangeIntersect(env, RangeVar(X_i, B, env))$

c). all variables are defined

$RangeIntersect(env, RangeVar(X_1, B, env), RangeVar(X_2, B, env), \ldots)$

3.3.2.1. Integer Range Flow Analysis

After defining the operations on the flow values, a formal definition of the range flow problem follows:

(1) V is the set of functions env: $var \rightarrow (min, max)$, where *var* is an integer variable and *min* and *max* refer to the minimum and maximum integer value that *var* may have. The *bottom* value for V is defined as $env(x) = (-\infty, \infty)$.

(2) $F_n(env_{in})$ is the transfer function, where n is a node in *SG* and env_{in} is the input value. The input value env_{in} is refined to env'_{in} with several methods described in the next section.

Case 1: n is an assignment node of the format $X := E$.

The interval of X is changed. The new flow value is

$env_{out} = env'_{in}[X \rightarrow RangeExpr(E, env'_{in})]$

Case 2: n is a branch node with condition B.

B refines the flow value. The new flow value is

$env_{out} = RangeRefine(B, env'_{in})$

After the result is computed, the value env_{out} is further enhanced by the method described in the next section and produces env'_{out}.

(3) The confluence operator is *RangeUnion*.

3.3.3. Enhancements

The previous section presented the integer range problem in traditional flow analysis. This section improves the analysis using various information available from the intermediate program including loop invariants and interval constants. This section presents four changes to adopt this information. The first three enhancements apply to the *in*-values in the order they are presented in this paper. The fourth enhancement applies to the *out*-values. These improvements are necessary, and were implemented, in order to produce reasonable results because *IFV*'s flow analysis assumes a very pessimistic initial value.

3.3.3.1. Unchanged Variables

Interval constants refer to the variables not modified inside a loop. With pessimistic flow analysis, these variables have two paths from which range information comes from: the loop-entry edge and the loop-back edges. Since an interval constant is not defined in the loop, its ranges at the loop-back edges are *bottom*. When the confluence is computed at the loop header, the worst case (*bottom*) is taken. This situation is resolved by ignoring the loop-back edges when computing the intervals for the interval constants at the loop header. Specifically, the flow value at the loop entry is “fixed” after the confluencing step.

/* v is the loop header */
/* env_{entry} is the ranges from the entry edge of loop v */
/* env is the *in*-confluenced ranges of v */
/* env' is the enhanced ranges */
/* $unchanged(v)$ is the set of variables not changed in loop (v) */

$env'(x) =$
  if x in $unchanged(v)$
    $env_{entry}(x)$;
  else
    $env(x)$;

3.3.3.2. Conditionally Redefined Variables

As shown in Example 2 (see Appendix), the definition at line 016 of the variable `first` inside the loop is only conditionally executed. There exists a path in the loop body in which `first` is not modified. This path provides no information, hence a *bottom* value is associated with `first` for that path and subsequent computations. This problem is solved by ignoring the "mis-information" from collecting definitions in the loop.

/* v is the loop header */
/* env_{entry} is the ranges from the entry edge of loop v */
/* env is the *in*-confluenced ranges of v */
/* env' is the enhanced ranges */
/* env_w is the *out*-ranges of node w */
/* $changed(v)$ is the set of variables changed in loop (v) */
/* $definloop(x, v)$ is the set of nodes in v that variable x changed */

$env'(x) =$
  if x in $changed(v)$
    $env_{entry}(x) \oplus (\bigoplus_{w \in definloop(x,v)} env_w(x))$;
  else
    $env(x)$;

3.3.3.3. Loop invariants

IFV uses loop invariants to refine the ranges at each loop entry.

/* $invar_n$ is the loop invariant with loop header n */

$env'_{in} = RangeRefine(invar_n, env_{in})$

3.3.3.4. Array Check Predicates

Using array check predicates to refine the flow information has two advantages. First, it can introduce extra inductive information to verify more array checks. Second, it eliminates redundant checks.

The information is inserted in a node by refining the range information with the *out*-values.

/* env_{out} is the *out*-value at node n */
/* $check_n$ is the array check predicate at node n */

$env'_{out} = RangeRefine(check_n, env_{out})$

An array check predicate acts as a tentative assertion. Similar to an inductive proof, the check predicate is first assumed valid and generates an interval. The interval is then propagated, combined with other intervals, and refined by the loop invariants, etc. in the flow analysis. On the next pass of the flow analysis, if the new interval validates the array check predicate, then the predicate is an invariant of the loop.

This method does not introduce errors. When an array access is incorrect, the interval does not validate the check. For example, consider the following program, which contains an incorrect array access. Here, the assertion $bound(i) = (0, 2)$ cannot prove itself.

```
int i, sum, a[3];
i = 0;  sum = 0;
                    /* bound(i)=(0,0) */
while
                    /* bound(i)=(0,0) or (1,3).  It simplifies to bound(i)=(0,3) */
   (i < 6) {
                    /* bound(i)=(0,3).  The predicate (i < 6) does not further refine the bound */
   sum += a[i];
                    /* bound(i)=(0,2).  It is introduced by the array check. */
   i = i + 1;
                    /* bound(i)=(1,3) */
}
```

An array check predicate eliminates redundant checks. Using an array check predicate as an assertion effectively introduces the assumption that the array access is true; it is used to verify other accesses. The following section has further examples.

3.3.4. Examples

To illustrate the flow computation, consider again the program in Example 1. The loop invariant generated (see Section 3.2.3) for the loop starting at line 007 is $i \geq 0 \wedge j \geq 0 \wedge i \geq j$. The condition for entering the loop body is $i < 100$, ignoring the part involving non-integer variables. (Recall that *IFV* assumes induction variables are integers; see Section 3.2.1.) On the first pass, the range for `i` at line 007 from the initialization edge is $(0, 0)$ and the range from the loop-back edge is *bottom*. Hence the confluenced range is *bottom*. Using the loop invariant at line 007, the ranges become $bound(i) = (0, \infty)$, $bound(j) = (0, \infty)$. Entering the loop body, $i < 100$ refines the range for `i` to $(0, 99)$. The range for `i` shows that the array accesses `a[i]` at line 008 and 009 are correct. Propagating the ranges through the loop body, `i` is updated in both branches and its range becomes $(1, 100)$, `j` is updated in one of the branches and its range becomes $(0, \infty)$. On the second pass, the ranges $bound(i) = (0, 100)$, $bound(j) = (0, \infty)$ at line 007 combine with the invariant $i \geq j$, refining `j` to $(0, 100)$. Similarly, the ranges propagate through the loop. The analysis detects no changes at the third pass and stops. Unfortunately, the array access `a[j]` at line 009 is not verified and must be handled using verification techniques (see Section 3.4).

Example 2 shows how the flow analysis eliminates some of redundant checks. At line 018, the array access check $0 \leq top < 100$ refines the range for `top` to $(0, 99)$. The range propagates to line 019 and validates the check for `data[top]` there. Similarly, the array access check for `data[first]` at line 019 validates the one on line 020. A more complicated case occurs in line 019. The value of `first` at line 019 is either the value of `first` at line 015, `thisone` at line 015, or the value of `top` at line 013. The range for `top` is $(0, 97)$. The access check for `data[thisone]` at line 015 constrains the range for `thisone` to $(0, 99)$. Similarly, the access check for `data[first]` at line 015 constrains the range for `first` to $(0, 99)$. The confluence of these three ranges is $(0, 99)$, thus validating the access `data[top]` at line 019.

Example 3 shows another case that requires verification techniques. The proof of the access `str[i+j]` at line 012 requires symbolic evaluation. The range for `len`, after line 008, is $(0, 10)$. At line 012, after refinement by the branch conditions $i < 100 - len$ and $j < len$, `i` has the range $(0, 99)$ and `j` has the range $(0, 9)$. Taking the worst case bound, the range for the expression `i+j` is $(0, 108)$. Obviously, it is not strong enough to prove the required bound $(0, 99)$ for the array `str` and therefore requires verification techniques.

3.4. Step 4: Verification Condition Generation

In flow analysis, two integers denote the upper and lower bound of each range. This notation is not strong enough to represent certain information, e.g., the bounds generated from the predicate $X \neq 1$ is $(-\infty, \infty)$. Similarly, when two non-overlapping ranges merge together, the gap in between them is not represented. As seen in Section 3.3.4, the flow analysis is not able to handle symbolic constants (Example 3) and it loses some information during the sequential evaluation of the constraints (Example 1).

The VCG collects assertions that are valid at the point of array access check and passes the collected assertions to the theorem prover. The theorem prover considers the whole theorem at the same time, therefore the order of evaluation does not cause any problem. The symbolic capability of the theorem prover handles the symbolic computation problem.

Because general verification condition generation along with invariant generation can produce very complicated assertions, *IFV* limits the assertions generated to linear inequalities. To minimize the size of VC generated, *IFV* applies program slicing (Weiser, 1981) to include only the relevant variable definitions, branch predicates, and loop invariants. The predicate is then transformed into proper format and proved using the Boyer-Moore theorem prover (Boyer *et al.*, 1988). Program slicing (Weiser, 1981) produces a bona-fide program—a subset of the original program that behaves exactly the same with respect to the computation of a designated property. The only difference when applying slicing to VC generation is that *IFV* generates a bona-fide theorem by dropping irrelevant predicates—i.e., predicates that carry no information to prove the goal.

To distinguish distinct assignment and usage of variables, variables used are labeled with where they are used, and variables defined are labeled with the node at which they are defined, as discussed in the next section.

3.4.1. Slicing

For simplicity, the slicing is performed on a per node basis, i.e., either all of the assertions associated with a node or none are included. A VC is of the form $precondition \Rightarrow goal$, where *goal* is an array check predicate and *precondition* is the relevant assignment statements and loop invariants.

An identifier in a VC is represented as either $variable\colon node$ or $variable'\colon node$. The former refers to the use of *variable* at *node* and the latter refers to the definition of *variable* at *node*.

IFV starts the VC with $true \Rightarrow goal$. Suppose *uses* are the identifier in the VC. For each use $var\colon node \in uses$, $areach(var\colon node)$ determines where the variable v is defined.

IFV first relates the uses and definitions. Suppose the use $var\colon node$ has reaching definitions $v'\colon m$ and $v'\colon n$. The predicate $v\colon node = v'\colon m \vee v\colon node = v'\colon n$ is "and"-ed with the precondition.

Then *IFV* includes the new definitions that either come from assignment statement or loop invariants. If node m is an assignment node of the form `V=A`. The equality $V'\colon m = A\colon m$ is "and"-ed to the precondition. If node n is a loop header containing a loop invariant I, the invariant is "and"-ed to the precondition.

The branch predicates that dominate the array check and contain some of identifiers in the VC are also "and"-ed to the precondition. The new preconditions add additional variable use. The new use may bring in new preconditions. The process is repeated until no new additions occur.

3.4.2. Examples

Consider the following code with the goal being to validate the array check at line 6 (i.e., $0 \le i\colon 6 < 100$). $i\colon 6$ is defined at node 5, so the precondition includes $i\colon 6 = i'\colon 5$. The assignment statement at node 5 translates to $i'\colon 5 = i\colon 5 + 1$. $i\colon 5$ has a reachable definition from the loop invariant at node 3, therefore $i'\colon 3 \ge 0$ and $i\colon 5 = i'\colon 3$ are included. The predicate $(i < n)$ at node 4 dominates node 6. It produces the precondition $i\colon 4 < n\colon 4$, $i\colon 4 = i'\colon 3$, and $n\colon 4 = n'\colon 1$. $n'\colon 1$ is assigned at node 1, so $n'\colon 1 = 99$ is included. The VC is:

$$(i\colon 6 = i'\colon 5 \wedge i'\colon 5 = i\colon 5 + 1 \wedge i'\colon 3 \ge 0 \wedge i\colon 5 = i'\colon 3 \wedge i\colon 4 < n\colon 4 \wedge i\colon 4 = i'3 \wedge n\colon 4 = n'\colon 1 \wedge n'\colon 1 = 99)$$
$$\Rightarrow 0 \le i\colon 6 < 100$$

It can be proved with the Boyer-Moore theorem prover after translating into the proper format.

```
1   int a[100], n = 99;
2   i = 0;
3   while  /*  loop invariant is i ≥ 0 */
4      (i < n) {
5      i++;
6      a[i];
7   }
```

In the following example, suppose the flow analyzer does not verify the access on line 6. Then, the VC for the array access at line 6 is $((y\colon 6 = y'\colon 5 \vee y\colon 6 = y'\colon 3) \wedge y'\colon 3 = 1 \wedge y'\colon 5 = 2) \Rightarrow 0 \le y < 100$, which can also be proved.

```
1   int a[100];
2   if (B)
3      y = 1;
4   else
5      y = 2;
6   a[y];
```

Recall from Section 3.3.4 that the array access check for `a[j]` at line 009 in Example 1 is $0 \le j < 100$. The flow analysis verified $0 \le j$, but was unable to handle $j < 100$. The VC generated is:

$$(j\colon 9 = j'\colon 7 \wedge i\colon 7 < 100 \wedge i'\colon 7 = i\colon 7 \wedge 0 \le i;\colon 7 \wedge 0 \le j'\colon 7 \wedge j'\colon 7 \le i'\colon 7) \Rightarrow j\colon 9 < 100$$

The VCG applies slicing. It starts with the check $j\colon 9 < 100$ in which $j\colon 9$ is used. There is only one definition for j $(j\colon 7)$ reaching line 009; it produces the condition $j\colon 9 = j\colon 7$. The VCG then looks up the loop invariants and branch conditions that dominates $j\colon 7$ or $j\colon 9$. It identifies and includes into the condition the loop invariants $j'\colon 7 \le i'\colon 7$ and

$0 \le j'{:}7$ at line 007. The set of variable definitions now includes $i{:}7$. The VCG identifies and includes one more invariant, $0 \le i'{:}7$ and one branch condition $i{:}7 < 100$ involving $i{:}7$. The addition of these predicates does not add new variable definitions to the system, therefore the slicing stops.

Also recall from Section 3.3.4 that the array access `str[i+j]` at line 012 in Example 3 requires symbolic evaluation. The VCG collects related predicates and definitions into the verification condition. Its essential clause is:

$$0 \le len \le 10 \wedge i < 100 - len \wedge j < len \Rightarrow (i + j) < 100$$

4. Prototype Implementation

The current implementation is part of a source code analysis project for reasoning about security-related properties of programs. The tools developed for this project include static and dynamic tools for the analysis of programs. Included are a translator from C to intermediate form, in which control flow, alias information, and Def-Use information have been determined. A general flow analysis framework is also available for reasoning about various flow information among components of the program. The prototype is written in Common Lisp (Steele, 1984). The prototype has been tested on simple programs such as the examples in the Appendix.

Table 2 shows the execution times on a lightly-loaded SPARCstation-2 of processing the example programs from the Appendix. The execution times (CPU seconds) are averages measured. The program Combined combines all five examples and serves to indicate the cost of the algorithm applied to larger programs. (Because *IFV* runs on one procedure at a time, the Combined program consists of one procedure containing the code from the other examples.) Note that the cost of VC generation for Combined is higher than the sum of the costs of VC generation for the individual programs because there are more invariants to deal with.

	Execution Times (CPU seconds)			Numbers of		
	Invariant Generation	Flow Analysis	VC Generation	Lines	Integers	Checks
1 Compact String	0.05	0.18	0.01	19	2	3
2 Selection Sort	0.05	0.36	0.00	25	4	6
3 String Match	0.05	0.45	0.01	24	3	3
4 Tree Sort	0.11	1.4	0.00	53	6	18
5 Binary Search	0.03	0.46	0.02	24	4	3
Combined	0.25	5.4	0.35	145	19	33

Table 2. Performance results.

The *IFV* implementation uses the algorithm from Section 3.4 to eliminate useless conditions from the VCs. That algorithm is straightforward, but slow. It collects the predicates by testing each predicate to determine if it contains any variables used in the VC. The testing is done iteratively until no more predicates are added. The iterative collection process is then repeated for the next proof goal. A more elaborate algorithm could improve the performance here. In particular, the amount of computation can be reduced by precomputing the relation between predicates, e.g., collecting predicate *A* always implies collecting predicate *B*. Thus, the precomputation cost can be shared by all proof goals. The predicates can be viewed as nodes in a graph and their relations as edges; then the VCG becomes finding all reachable predicates from each proof goal in such a graph.

IFV proves all array accesses in the examples except the one at line 19 of the Binary Search (Example 5), and hence the corresponding one in Combined. Its problem is that `middle` is not initialized but defined in the loop. The value of `middle` at line 19 is valid only when the loop executes at least once. This information is unfortunately not captured by *IFV*. Perhaps a heuristic could be used to detect this and similar cases.

These performance results of the array bound checker agree with the expectations described earlier. In particular, based on these results, one can observe the following.

- The invariant generation time is essentially linear in the number of induction variables (program size).
- The execution time of a pass in the flow analysis grows linearly in the number of variables times and the program size. The number of passes is the depth of the program graph which increases slightly with the program size.
- The total execution time of the VCG depends on how many array checks are not verified by flow analysis. The time needed to generate a VC depends on the number of assertions, invariants, predicates, and definitions collected. Most of the generated VCs are simple.

5. Conclusion

The new approach, *IFV*, integrates flow analysis techniques and program verification techniques. It has several advantages.

The tool is entirely automatic. The loop invariant generation (template matching), flow analysis, VC generation, and theorem proving require no human assistance.

The simpler cases are handled by flow analysis, and the more complicated ones by the VCG and the theorem prover. For the examples given in the Appendix (and also other programs tested), more than 90% of the array checks were handled by flow analysis. The loops in these examples are representative of the kind found in real programs. Such savings are possible because of the use of inductive information in the flow analysis. Moreover, an equivalence of Gupta's optimization is performed by the flow analyzer (see Section 3.3).

The use of templates to generate invariants deterministically for array bound checking is new. (Templates are too weak for general purpose program verification, but they are ideal for the kind of integer analysis associated with most array index computations). The advantages of using templates is that all invariants generated are valid; heuristics generate assertions that require a theorem prover to justify or reject and hence require significant CPU time. Template matching is localized to recurrence relations involving loop induction variables. The cost of invariant generation is proportional to the number of templates times the number of induction variables. This paper has given seven templates, which cover many commonly-occurring loops. The number of induction variables is proportional to program size. Therefore, the cost of invariant generation increases linearly with program size. Furthermore, this approach is extensible with new templates.

The use of inductive information to refine flow analysis is very powerful and inexpensive. Flow analysis can be completed in approximately $O(depth)$ passes, where *depth* is the depth (Aho *et al.*, 1986) of the program graph. The depth (i.e., number of passes) is determined by a constant empirically (Knuth, 1971). Each pass has n steps, where n is the number of nodes in the program graph. Each step takes $O(v)$ time, where v is the number of declared integer variables. Therefore, the cost of flow analysis is proportional to the program size times the number of declared integer variables. The flow in the integer range analysis has been altered to handle some of the sequential evaluation problem and the interval-constant problem (see Section 3.4).

The *IFV* VCG generates one VC, in a restricted format, for each goal. The size of the VC is linear in the program size. The restricted format results in a loss of expressiveness, but the number of VCs and the size of VCs never grow exponentially. The *IFV* VCG includes only relevant predicates based on data-flow information, therefore the cost of VCG for a goal is linear in the size of the generated VC.

IFV is extensible due to the use of templates. A small library of templates corresponding to common loop structures has been produced. New templates for new applications can be added. Suzuki in (Suzuki, 1977) indicates that the approach developed there has problems dealing with traversal of data structure ending with a special value, e.g., null-terminated strings. In *IFV*, new templates to deal with such features can be created. The *IFV* approach can also work well in ensuring, at compile time, that many loops in Java programs do not cause array bounds overflow (Fritzinger *et al.*, 1996; Sun, 1995).

The techniques developed in this paper are useful for other applications. Flow analysis produces a lower bound and an upper bound for integer variables and expressions. When the bounds are equal, the variable is known to have a constant value. When the range of the expression in a switch-statement is small, a compiler can generate a more efficient indirect jump-table for it. Furthermore, branch conditions involving integer variables can sometimes be simplified. Branches not taken can then be removed. The techniques to recognize induction variables, which make use of local information (natural loops, data dependence, candidate set of induction variables), can make the pure abstract interpretation approach (Ammarguellat *et al.*, 1990) more efficient by eliminating unnecessary computations. The recognition of recurrence relations, which may be expressed in different statements or in different orders of execution, finds many applications in parallelization of numerical computations. Assigning a unique identity to each definition and each loop induction variable yields an acyclic description of the Def-Use chain, which allows a straightforward verification-condition generation, and reduces the complexity of the assertions generated this way.

IFV is not without shortcomings. It ignores certain control flow information (e.g., when information from incoming arcs merges, their flow paths are not distinguished) and uses propositional logic and linear inequalities to express assertions. This limits the analysis to proofs expressible in this paradigm. Many algebraic theorems such as $sqrt(4) = 2$, $sin(0) = 0$, and conversions between floating point and integers are not considered. Information about values read from input or files when the programs executed (sometimes available in the form of input assertions) is not used in *IFV*. Another situation *IFV* does not handle is when only part of an array is used. For example, when a string is placed in a character array in C, often accesses to the array should occur only up to the position of the terminating null character. One difficulty in addressing this problem is that sometimes such an array might (legitimately) be used to store more than one string. Another shortcoming of *IFV* is that it analyzes only one procedure at a time. The *IFV* approach can be extended to perform interprocedural analysis using known techniques (Sharir *et al.*, 1981). One final problem that *IFV* does not handle is integer overflow, e.g., 32767 + 1 overflows in a 16-bit machine. It seems that only the most general program verification techniques can solve all such problems and, furthermore, these kinds of problems did not occur in the programs examined during the course of this project, with respect to array access checks.

Acknowledgements

The anonymous referees and Lee White made many thoughtful comments, which helped to improve the presentation of this paper.

Figures

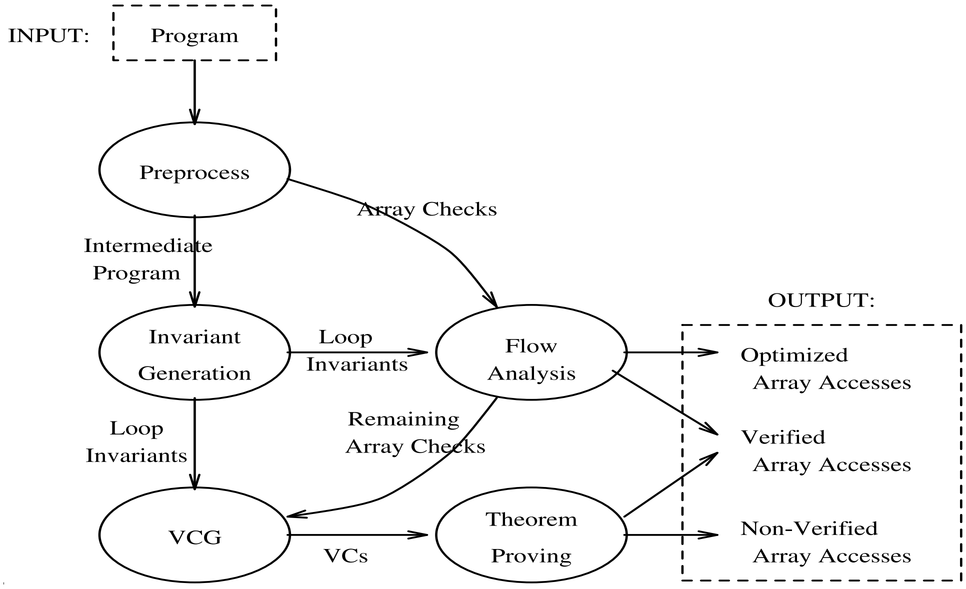


Fig. 1. Overview of integrated approach

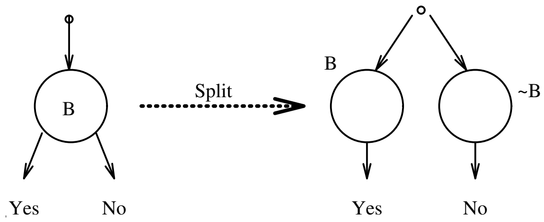
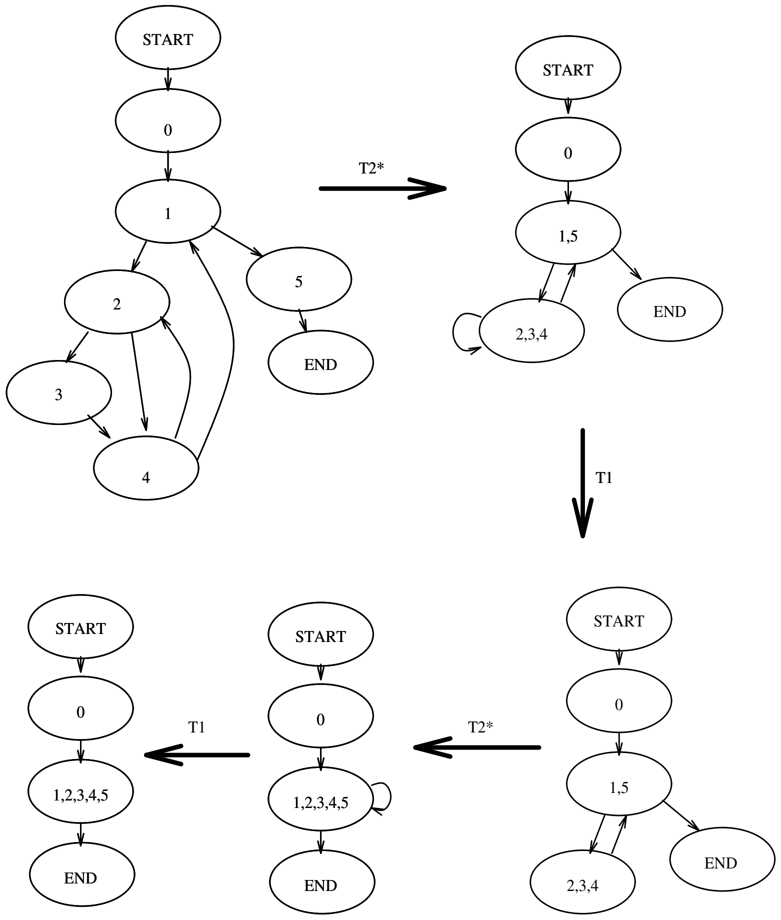


Fig. 2. Branch node splitting



T2* means a sequence of T2 transformations.

Fig. 3. T1-T2 tranformation

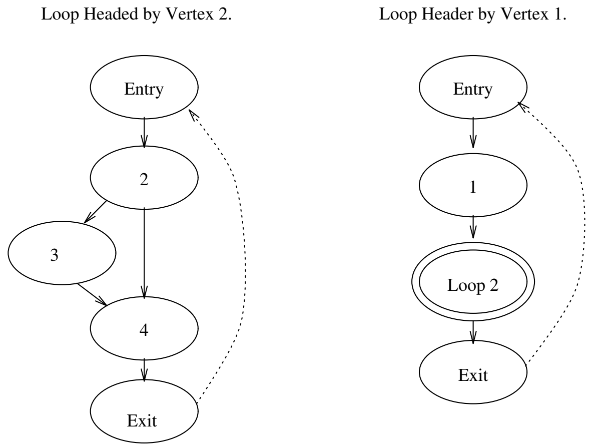


Fig. 4. Identifying loops

Appendix: Example Programs

```
/* Example 1: Compact String */
char a[100];

string_compact()
{
   int i = 0, j = 0;
   while (i < 100 && a[i] != '\0')
      if (a[i] != ' ') {
         a[j] = a[i];
         i++;
         j++;
      } else
         i++;
   if (j < 100) a[j] = '\0';
 }
```

```
/* Example 2: Selection Sort */
struct datatype {
   int key, val;
} data[100];

selection_sort()
{
   int top, bottom, first, thisone;
   struct datatype t;

   bottom = 99;
    for (top = 0; top < bottom-1; top++) {
      first = top;
      for (thisone = top+1;
         thisone < bottom; thisone++) {
         if (data[thisone].key < data[first].key)
            first = thisone;
      }
      t = data[top];
      data[top] = data[first];
      data[first] = t;
   }
 }
```

```
/* Example 3: Find pattern in string */
string_index()
{
   char  str[100], pattern[10];
   int i,j,len;

   for (len=0; len < 10; len++)
      if (pattern[len] == '\0') break;
   /* len is the length of the pattern string */
   for (i = 0; i < 100 - len; i++) {
      for (j = 0; j < len; j++)
         if (str[i+j] != pattern[j])
            break;
      if (j == len) break;
   }
   if (j == len)
      return i;
   else
      return -1;
}
```

```
   /* Example 4: Tree Sort
    */

   tree_sort()
   {
      int a[101];    /* 1 to 100 */
      int k, i, j, n;
      int copy, work;

      k = 50;
      while (2 <= k) {
            i = k;
            n = 100;
            copy = a[i];
            j = 2 * i;
            while (j <= n) {
               if (1 + j - n <= 0)
                     if (a[j] < a[j+1])
                        j++;
               if (copy < a[j]) {
                     a[i] = a[j];
                     i = j;
                     j = 2 * i;
               } else
                     j = n + 1;
            }
            a[i] = copy;
            k--;
      }
      k = 100;
      while (2 <= k) {
            i = 1;
            n = k;
            copy = a[i];
            j = 2 * i;
            while (j <= n) {
               if (j < n)
                     if (a[j] < a[i])
                        j++;
               if (copy < a[j]) {
                     a[i] = a[j];
                     i = j;
                     j = 2 * i;
               } else
                     j = n + 1;
            }
            a[i] = copy;
            work = a[1];
            a[1] = a[k];
            a[k] = work;
            k--;
      }
   }
```

```
/* Example 5: Binary Search */
int table[105];
main (key)
int key;
{
   int high = 100;
   int low = 1, middle;


   while (low <= high) {
         middle = (low + high) / 2;
         if (table[middle] == key)
            low = high + 1;
         else if (table[middle] > key)
            high = middle - 1;
         else
            low = middle + 1;
   }
   if (table[middle] != key)
         middle = 0;
   return middle;
}
```

REFERENCES

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffery D. (1986) *Compilers: Principles, Techniques, and Tools,* Addison-Wesley

Ammarguellat, Zahira and Harrison, W.L., III (1990) "Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation," *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 283-295

Asuru, Jonathan M. (1992) "Optimization of Array Subscript Range Checks," *ACM Letters on Programming Languages and Systems*, Vol. **1**(2), pp. 109-118

Austin, T.M., Breach, S.E., and Sohi, G.S. (1994) "Efficient Detection of All Pointer and Array Access Errors," *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 290-301

Boyer, Robert S. and Moore, J Strother (1981) "A Verification Condition Generator for FORTRAN," in *The Correctness Problem in Computer Science*, ed. Robert S. Moore and J Strother Moore,Academic Press, Inc.

Boyer, Robert S. and Moore, J Strother (1988) *A Computational Logic Handbook,* Academic Press, Inc.

Chin, Wei-Ngan and Goh, Eak-Khoon (1995) "A Reexamination of "Optimization of Array Subscript Range Checks"," *ACM Transactions on Programming Languages and Systems*, Vol. **17**(2), pp. 217-227

Cousot, Patrick and Halbwachs, Nicolas (1978) "Automatic Discovery of Linear Restraints Among Variables of a Program," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 84-96

Detlefs, David L. (1996) "An Overview of the Extended Static Checking System," *Proceedings of the Workshop on Formal Methods in Software Practice (ISSTA 96)*, http://www.research.digital.com/SRC/esc/Esc.html, pp. 1-9

Ferrante, Jeanne and Ottenstein, Karl J. (1983) "A Program Form Based on Data Dependency in Predicate Regions," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 217-236

Fink, George and Levitt, Karl (1994) "Property-based Testing of Privileged Programs," *Proceedings of the Tenth Annual Computer Security Applications Conference*, IEEE Computer Society Press, pp. 154-163

Fritzinger, J.S. and Mueller, M. (1996) "Java Security," *JavaSoft White Paper (http://java.sun.com/security/)*, Sun Microsystems, Inc.

German, Steven M. and Wegbreit, Ben (1975) "A Synthesizer of Inductive Assertions," *IEEE Transactions on Software Engineering*, Vol. **SE-1**(1), pp. 68-75

Gupta, Rajiv (1990) "A Fresh Look at Optimizing Array Bound Checking," *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 272-282

Gupta, Rajiv (1993) "Optimizing Array Bound Checks Using Flow Analysis," *ACM Letters on Programming Languages and Systems,*, Vol. **2**(4), pp. 135-150

Harrison, William H. (1977) "Compiler Analysis of the Value Ranges for Variables," *IEEE Transactions on Software Engineering*, Vol. **SE-3**(3), pp. 243-250

Katz, Shmuel and Manna, Zohar (1973) "A Heuristic Approach to Program Verification," *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 500-512

Katz, Shmuel and Manna, Zohar (1976) "Logical Analysis of Programs," *Communications of the ACM*, Vol. **19**(4), pp. 188-206

Kildall, Gary A. (1973) "A Unified Approach to Global Program Optimization," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 194-206

Knuth, D. E., (1971) "An Empirical Study of FORTRAN Programs," *Software—Practice and Experience*, Vol. **1**(4), pp. 105-134

Kolte, P. and Wolfe, M. (1995) "Elimination of Redundant Array Subscript Range Checks," *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 270-278

Liu, Shin-Ming, Lo, Raymond, and Chow, Fred (1996) "Loop Induction Variable Canonicalization in Parallelizing Compilers," *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, IEEE Computer Society Press, pp. 228-237

Lo, Raymond W. (1992) "Static Analysis of Programs with Application to Malicious Code Detection," *Ph.D. Dissertation*, Dept. of Computer Science, University of California, Davis

Nagle, John and Johnson, Scott (1983) "Practical Program Verification: Automatic Program Proving for Real-Time Embedded Software," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 48-58

Owre, Sam, Rushby, John, and Shankar, Natarajan (1992) "PVS: A Prototype Verification System," pp. 748-752 in *Lecture Notes in Artificial Intelligence (607) (Proceedings of the International Conference on Automated*

Deduction (CADE 11)), ed. D. Kapur,Springer-Verlag

Sharir, Micha and Pnueli, Amir (1981) “Two Approaches to Interprocedural Data Flow Analysis,” in *Program Flow Analysis: Theory and Applications*, ed. Steven J. Muchnick and Neil S. Jones,Prentice-Hall, Inc.

Shostak, Robert E. (1977) “On the SUP-INF method for proving Presburger formulas,” *Journal of the ACM*, Vol. **24**(4), pp. 529-543

Steele, Guy L. Jr (1984) *Common Lisp: The Language,* Digital Press

Sun (1995) “Java: Programming for the Internet,” *http://www.javasoft.com/*, Sun Microsystems, Inc.

Suzuki, Norihisa (1977) “Implementation of an Array Bound Checker,” *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 132-143

Wegbreit, Ben (1973) “Heuristic Methods for Mechanically Deriving Inductive Assertions,” *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 524-536

Weiser, Mark (1981) “Program Slicing,” *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439-449