Retaining Path-Sensitive Relations across Control-Flow Merges

DOUGLAS GREGOR
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180 USA
and
SIBYLLE SCHUPP
Dept. of Computing Science
Chalmers University of Technology
SE-41296 Gothenburg

We present a new value range merge operation that, unlike traditional value range merge operations, preserves arbitrary ordering relationships between program variables. We guide this new algorithm with two data structures that model interobject field relationships, enabling better precision in the analysis of object-oriented programs without prohibitive implementation or run-time costs. We discuss our own implementation of this algorithm in the STLlint static checker, which is able to verify C++ iterator loops that otherwise would require path-sensitive analysis.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation, Assertion checkers*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Path-sensitive analysis, value range propagation, dataflow analysis

1. INTRODUCTION

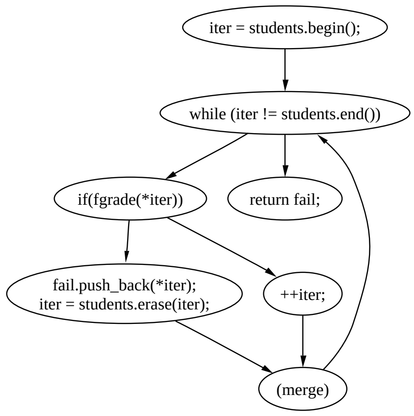
Static analysis seeks to discover semantic information about computer programs by analyzing a representation of the source code of the program. With this semantic information, various software engineering tasks may be performed automatically, including verification that a program behaves in a certain way or optimization of a program by transforming it into a semantically equivalent but more efficient program.

A particular static analysis algorithm represents a tradeoff between the efficiency of the algorithm and the precision of the information collected. Path sensitivity refers to the effect of the particular execution path taken through a program on the

```
vector<Student_info>
extract_fails(vector<Student_info>& students)
{
    vector<Student_info> fail;
    vector<Student_info>::iterator iter =
                              students.begin();

    while (iter != students.end()) {
        if (fgrade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);
        } else
            ++iter;
    }
    return fail;
}
```

(a) Extract failing students



(b) Control flow graph

Fig. 1. Precise analysis of a subroutine that erases students with a failing grade from a vector requires path sensitivity.

precision of a static analysis algorithm. A *path-insensitive* algorithm may follow two different paths from a conditional branch (flow sensitivity), but at a *control-flow join*, the point where the two paths rejoin, the results of the two paths will be merged into a single approximation of both paths. A *path-sensitive* algorithm separates the results from the two different paths and instead analyzes the program from the control-flow join several times; once for each incoming path. The merging behavior at a control-flow join may be specified within the description of an algorithm, as in abstract interpretation [Cousot and Cousot 1977], or may be an explicit part of the program representation, as in static single assignment (SSA) form [Alpern et al. 1988; Rosen et al. 1988]. Path sensitivity is generally considered intractable because the problem size grows exponentially with the number of conditional branches in the program. However, when the analysis domain, i.e., the representation of program state, is sufficiently simple (such as a small finite automaton), path-sensitive analysis can become practical [Engler et al. 2000] and run in polynomial time [Das et al. 2002].

Value range propagation [Harrison 1977] is a static analysis method that computes, for each program variable, the range of values that the program variable may attain for any program input. Value ranges are expressed as lower and upper bounds on the program variable, although the representation of these bounds differs greatly, varying from simple integral constant representations [Verbrugge et al. 1996] to symbolic expressions [Blume and Eigenmann 1995]. Value range propagation is expressible as either a path-sensitive or path-insensitive analysis, although existing formulations define path-insensitive analyses due to the exponential cost of path sensitivity with value ranges.

Simple, common programming idioms require path sensitivity. Fig. 1, borrowed from an introductory C++ text [Koenig and Moo 2000], illustrates one such idiom:

we wish to construct a static analysis that can verify the validity of the iterator `iter` in the loop, a problem referred to as the Concurrent Modification Problem in Java [Ramalingam et al. 2001]. An iterator is considered *valid* when it either references a valid element or is equal to the sentinel past-the-end iterator. Suppose we represent the iterator's position as an integral index into the container, explicitly represent the container's size, and then define the past-the-end iterator as any iterator with an index equivalent to the container's size. The problem of determining iterator validity is then ideal for value range propagation: we need only verify that the iterator's index does not exceed the container's size by comparing the value ranges. Within each loop iteration, the loop either erases the current student (decreasing the size of `students`), implicitly setting the iterator to the next student, or explicitly increments the iterator to the next student in the vector. Thus value range propagation can verify in each branch that the iterator remains valid, but once the value ranges have been merged following the if-then-else statement, iterator validity can no longer be proven because the value ranges of the iterator index and the container size will overlap. Thus a path-sensitive algorithm will verify iterator validity in Fig. 1 whereas a path-insensitive algorithm will not. Our goal is to create a static analysis that is able to properly verify programs such as that in Fig. 1 without suffering the exponential explosion in complexity, using information about the relationships between iterator objects and container objects to guide the analysis.

In this paper we will present an alternative merge operation for value ranges that preserves arbitrary sets of ordering relations (e.g., $<, \geq, =$) across control-flow merges with known precision. This merge operation is directed by data structures that capture the static and dynamic relationships between object fields in the program, which enables more precise analysis of object-oriented programs. Section 2 delves into the nature of the precision loss caused by value range merging. We then develop the basis for our relational merging algorithm in Section 3 and generalize from it to preserve arbitrary sets of relations in Section 4. We then delve into the nature of object-oriented programs and use the new static and dynamic field relation graphs to apply our value range merge algorithm to real problems, such as the verification of the example in Fig. 1, in Section 5. Finally, we discuss algorithmic complexity in Section 6 and our own implementation of the relational merge algorithm and associated data structures in Section 7.

2. PRECISION LOSS DUE TO RELATIONAL OVERLAP

To understand the inability of value range propagation to verify iteration validity in Fig. 1 with the traditional value range merge operation, we must understand why the merge operation does not preserve arithmetic relations. We begin with a general description of value range propagation. Let value ranges be denoted by a pair written $[a : b]$, where a is the lower bound and b is the upper bound (both inclusive), and let Range denote a set of ranges $[a : b]$. For a variable v, the notation $v \in [a : b]$ can be read as "the variable v has a value in the range $[a : b]$", where variables are names drawn from the set Variable.

We are interested primarily in the semantics of two types of program statements that affect value range propagation: assignment statements and control-flow joins. Assignment statements $v := e$ replace the value range of a variable v with the range computed by expression e. Control-flow joins, which occur at nodes in the control-flow graph with two or more incoming control-flow edges, replace the value range of each variable with a value range representing the join of all value ranges for that variable on the incoming flow edges.

Definition 1. Given a variable v whose value ranges are $[a_1 : b_1], [a_2 : b_2], \cdots, [a_n : b_n]$ on the n incoming control-flow edges, the value range merge operation ϕ : Variable $\rightarrow$ Range is defined as the union of the ranges on each incoming flow edge [Harrison 1977]:

$$\phi(v) = [\min(a_1, a_2, \cdots, a_n) : \max(b_1, b_2, \cdots, b_n)]. \tag{1}$$

We note two important properties of ϕ:

— ϕ is a join operation [Cousot and Cousot 1977], such that $\phi(v) \supseteq [a_i : b_i]\ \forall_{1 \leq i \leq n}$, meaning that the value range $\phi(v)$ includes the value ranges for v on all incoming flow edges.

— ϕ requires a partial ordering on the value range bounds, but otherwise does not restrict the bounds. Therefore, the value range bounds may comprise simple domains, such as integer or floating-point values [Verbrugge et al. 1996], or more expressive domains such as symbolic expressions [Harrison 1977; Blume and Eigenmann 1995].

Fig. 2 illustrates the effects of value range propagation on a very simple program utilizing an interval abstract data type. Each outgoing control-flow edge is annotated with the value ranges for the `lower` and `upper` fields of the interval `i`. For instance, the edge exiting the node corresponding to the `then` branch declares that `lower` $\in [1 : 1]$ and `upper` $\in [3 : 3]$ as a result of the two assignment statements in that branch. The edge exiting the `else` branch is similarly annotated. At the control-flow join, marked "merge", the value range propagation algorithm merges the values on each incoming control-flow edge, thus `lower` $\in [\min(1, 5) : \max(1 : 5)] = [1 : 5]$ and `upper` $\in [\min(3, 10) : \max(3, 10)] = [3 : 10]$ on the edge exiting the control-flow join.

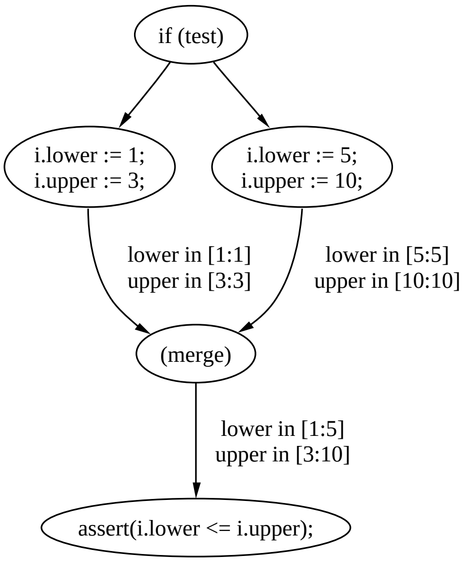
The value ranges exiting the control-flow join illustrate the loss of precision inherent in the definition of the merge operation. On the edge exiting the `then` and `else` branches, value range propagation can determine that the invariant `i.lower <= i.upper` holds by comparing the upper bound of `i.lower` to the lower bound of `i.upper`. Following the control-flow join, the relation is no longer provable, as $5 \nleq 3$: the conservative merge operation dictated by a path-insensitive analysis has degraded the precision such that the analysis can no longer statically determine the correctness of the following `assert` statement. Note that a path-sensitive analysis would not suffer from such an error, as it would in effect have two `assert` statements (one for each branch of the `if` statement).

We label the cause of the analysis' failure to detect the correctness of the `assert` statement as *relational overlap*. Given two variables, l and r, relational overlap occurs when the values of l and r are reassigned in two different control-flow paths such that $l \leq r$ is provable in each path individually but not provable after a control-flow merge. With value range propagation, relational overlap occurs when the merged ranges of l and r overlap (i.e., the intersection of the ranges is nonempty). This

```
class interval {
    lower :  integer;
    upper :  integer;
}

i :  interval;
if (test) then {
    i.lower := 1;
    i.upper := 3;
} else {
    i.lower := 5;
    i.upper := 10;
}
assert(i.lower <= i.upper);
```

(a) Merging an interval ADT



(b) Control flow graph

Fig. 2. Merging the value ranges representing the fields in an interval abstract data type results in an unacceptable loss of precision.

overlap occurs regardless of the underlying integral expression domain representing the lower and upper bounds within a value range. Thus, the problem occurs for simple value range bound representations, such as integral constants, and even very powerful symbolic value ranges. Similar representations, such as sets of disjoint value ranges [Patterson 1995], exhibit the same behavior.

The interval abstract data type example in Fig. 2 illustrates the simplest form of the relational overlap problem. Our motivating example from Fig. 1 fails due to relational overlap, but in a more disguised form. Suppose we analyze iterators as indices into a particular container of values, such that an iterator referencing the first element in a container would have index 0. Similarly, we represent the size of a container as an integral value, and then state that an iterator is *valid* if its index does not exceed the container's size. Consider the last iteration of the loop in Fig. 1, where the index of the iterator `iter` is one less than the size of the container: the `then` branch reduces the size of the container by one and leaves the iterator index at the same position, whereas in the `else` branch the size of the container remains the same and the iterator index is incremented by one. In both branches, the index is no larger than the container size, but at the control-flow merge this relation is no longer provable because the iterator index from the `else` branch exceeds the container size from the `then` branch.

Relational overlap occurs in the analysis of simple example programs and in common programming idioms, causing a loss of precision that inhibits verification and optimization. Relational overlap is caused by the value range merge operation ϕ, and cannot be solved directly with more expressive value range representations.

3. RELATIONAL MERGING ALGORITHM

Our approach to solving the relational overlap problem is to redefine the value range merge operation to *preserve* relations. A merge operation preserves a relation $x \oplus y$ if the provability of $x \oplus y$ on all incoming control-flow edges implies provability of $x \oplus y$ on the merged result. Section 2 illustrated that ϕ does not preserve the $\leq$ operation; similar examples show that ϕ does not preserve $<, >, \geq$, or $=$ (equality), either.

Definition 2. Given variables x and y with value ranges $[a_1 : b_1], [a_2 : b_2], \cdots, [a_n : b_n]$ and $[c_1 : d_1], [c_2 : d_2], \cdots, [c_n : d_n]$, respectively, on the n incoming control-flow edges, we define the merged value range of y *relative to* x as

$$\phi_{rel}(y, x) = [x + \min(c_1 - b_1, c_2 - b_2, \cdots, c_n - b_n) : x + \max(d_1 - a_1, d_2 - a_2, \cdots d_n - a_n)] \tag{2}$$

The function ϕ_{rel} utilizes the definition of value range comparison to preserve ordering relations across control-flow merges. Following a control-flow merge, the offset of y from x will be compared against zero when x and y are compared. By capturing the offsets between x and y in each control-flow edge at the time of the merge, ϕ_{rel} ensures that the offset precision isn't lost due to varying magnitudes of the values of x and y.

LEMMA 1. *Given variables x and y, the relative merge operation $\phi_{rel}(y, x)$ preserves the relation $x \oplus y$ for any $\oplus \in \{<, \leq, =, \geq, >\}$.*

PROOF. *Let x and y be variables with value ranges $[a_1 : b_1], [a_2 : b_2], \cdots, [a_n : b_n]$ and $[c_1 : d_1], [c_2 : d_2], \cdots, [c_n : d_n]$, respectively, on the n incoming control-flow edges. The difference $\phi_{rel}(y, x) - x = [\min(c_1 - b_1, c_2 - b_2, \cdots, c_n - b_n) : \max(d_1 - a_1, d_2 - a_2, \cdots d_n - a_n)]$ denotes the offset of y from x. The proof follows by examining all possible values of $\oplus$.*

(1) $x < y$: *The relation $x < y$ holds on control-flow edge i if $c_i - b_i > 0$. If $\forall_{1 \leq i \leq n}\ c_i - b_i > 0$, then $\min(c_1 - b_1, c_2 - b_2, \cdots, c_n - b_n) > 0$ and thus $x < y$ in the merged result.*
(2) $x \leq y$: *Analagous to the $x < y$ case.*
(3) $x > y$: *The relation $x > y$ holds on control-flow edge i if $d_i - a_i < 0$. If $\forall_{1 \leq i \leq n}\ d_i - a_i < 0$, then $\max(d_1 - a_1, d_2 - a_2, \cdots, d_n - a_n) < 0$ and thus $x > y$ in the merged result.*
(4) $x \geq y$: *Analagous to the $x > y$ case.*
(5) $x = y$: *The relation $x = y$ holds on control-flow edge i if $c_i - b_i = d_i - a_i = 0$. If $\forall_{1 \leq i \leq n}\ c_i - b_i = d_i - a_i = 0$, then $\min(c_1 - b_1, c_2 - b_2, \cdots, c_n - b_n) = \max(d_1 - a_1, d_2 - a_2, \cdots, d_n - a_n) = 0$ and thus $x = y$ in the merged result.*

□

Revisiting Fig. 2, we see that merging `upper` relative to `lower` results in `lower` $\in [1 : 5]$ and `upper` $= [$`lower` $+ 2 :$ `lower` $+ 5]$. The offset of `upper` from `lower` is clearly positive, so with the revised merge operation the analysis is able to prove statically that the assertion condition is true.

We now generalize relational merging to an arbitrary chain of relations that allows us to increase the precision of relational merging for more than two variables. That is, instead of a single relation $x \oplus y$ with two variables, we consider a chain of $m-1$ relations with m variables $v_1 \oplus_1 v_2 \oplus_2 \cdots \oplus_{m-1} v_m$. We arbitrarily select a value i with $1 \leq i \leq m$, call variable v_i an *anchor point* and let $v_i' = \phi(v_i)$. From the anchor v_i, we perform cascading relative merges outward, such that the neighbors of v_i, v_{i-1} and v_{i+1}, are merged relative to v_i; the merging continues recursively along the chain until no neighbors exist. More formally, let $1 \leq i \leq n$ and $v_i' = \phi(v_i)$. Then $\forall_{1 \leq j < i}$ we set $v_j' = \phi_{rel}(v_j, v_{j+1})$ and $\forall_{i < k \leq m}$ we set $v_k' = \phi_{rel}(v_k, v_{k-1})$.

LEMMA 2. *Given a set of variables $v_1, v_2, \cdots, v_m$ with value ranges on the n incoming control-flow edges, the chain of relations $v_1 \oplus_1 v_2 \oplus_2 \cdots \oplus_{m-1} v_m$ with $\oplus_i \in \{<, \leq, =, \geq, >\}$, and an integer anchor $1 \leq i \leq m$ the cascading chain merge algorithm preserves all relations $v_j \oplus_j v_{j+1}$ for all $1 \leq j < m$.*

PROOF. *For $1 \leq j < m$, either $v_j' = \phi_{rel}(v_j, v_{j+1})$, if $j < i$, or $v_j' = \phi_{rel}(v_j, v_{j-1})$, if $j > i$. In either case, the relation $v_j \oplus_j v_{j+1}$ is preserved by Lemma 1.* □

4. VARIABLE RELATION GRAPH

To accommodate the complex relationships among program variables in real programs, we expand the relational merging algorithm to support an arbitrary set of relations among variables. We represent the set of relationships as edges in an undirected *variable relation graph*, a data structure that serves as the basis for our relational merging algorithm.

We first formalize the notion of the variable relation graph. Given a set of m variables $v_1, v_2, \cdots, v_m \in$ Variable and a set of relations $R \subseteq$ Variable $\times$ Relation $\times$ Variable, where a Relation may be any one of $<, \leq, =, \geq$, or $>$, we construct a *variable relation graph*. The vertices V of the graph are the variables $v_1, v_2, \cdots, v_m$ (we will refer to variables and vertices interchangeably with these names). The undirected edges $E \subseteq$ Variable $\times$ Variable of the graph represent any relationship between two variables. The edge $(v_i, v_j) \in E$ if $i \neq j$ and there exists a $(v_i, \oplus, v_j) \in R$ for some $\oplus \in \{<, \leq, =, \geq, >\}$. The variable relation graph therefore captures the structure of the web of relationships among variables, but not the directionality of the relationships.

Definition 3. Given a variable relation graph $G = (V, E)$ with vertices $V = \{v_1, v_2, \cdots, v_n\}$, let the predicate Preserves(v_i, v_j) return true when $v_i \oplus v_j$ is preserved by the algorithm for $\oplus \in \{<, \leq, =, \geq, >\}$. The precision of a merge algorithm with respect to the variable relation graph G is $\frac{|P|}{|E|}$, where

$$P = \{(v_i, v_j) | (v_i, v_j) \in E \text{ and } \mathrm{Preserves}(v_i, v_j)\}.$$

A more precise algorithm will preserve more relations in the variable relation graph, and therefore will have a higher precision. The precision of ϕ is 0 (it is not guaranteed to preserve any relationships) and the precision of the chain merging operation is 1 according to Lemma 2 if we restrict the variable relation graph to a chain of relations.

$$\begin{array}{ll} \text{MergeVisitor} = & \\ \quad \text{start-vertex}(v) & \{v' = \phi(v);\} \\ \quad \text{tree-edge}(u, v) & \{v' = \phi_{rel}(v, u);\} \end{array}$$

Fig. 3. A depth-first search visitor that performs relative merges along a spanning tree in the variable relation graph

4.1 Tree properties

By first restricting the variable relation graph to a tree, we can define a merge operation suitable for variable relation trees with perfect precision. Variable relation trees are unrooted due to their direct connection with sets of variable relations (which have no such concept), so we arbitrarily select a vertex $v_s \in E$ as the root. The root v_s is equivalent to the anchor in the chain-merging algorithm in Section 3, so we let $v'_s = \phi(v_s)$ and perform cascading merges starting with the root and out to each child node. The cascading merges are performed by executing a depth-first search starting at v_s and applying ϕ_{rel} to the target and source of each tree edge encountered. Fig. 3 illustrates the depth-first search visitor that performs the variable relation tree merge: the start-vertex event is executed for the starting node of the depth-first search (v_s), and for each tree edge thereafter tree-edge(u, v) is executed as we cross the edge from u to v.

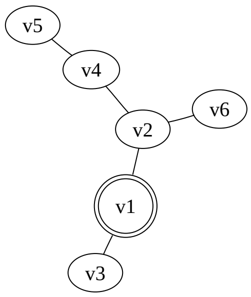
Fig. 4 illustrates a variable relation tree and the merges performed by our merge algorithm. The variable relation tree represents the set of relations $R = \{v_1 \leq v_3, v_1 \leq v_2, v_2 \leq v_6, v_4 \leq v_2, v_5 \leq v_4\}$, and we have arbitrarily selected vertex v_1 as the root of the tree. We see that for each tree edge, the target of the tree edge has been merged relative to the source of the tree edge.

LEMMA 3. *The merging algorithm for a variable relation tree preserves all relations in the tree. It has a precision of 1.*

PROOF. *Given a variable relation tree $G = (V, E)$, for each edge $(v_i, v_j) \in E$ the depth-first traversal executes either tree-edge(v_i, v_j) or tree-edge(v_j, v_i), thus performing the relative merge $\phi_{rel}(v_j, v_i)$ or $\phi_{rel}(v_i, v_j)$, respectively. Thus by Lemma 1, all relationships are preserved and the precision is 1.* □

The choice of the root of the tree, v_s, is arbitrary and does not affect the asymptotic behavior of the algorithm. However, depending on the domain of the value range bounds, it may affect the overall precision of the analysis. Paths through the variable relation tree could theoretically be very long, creating a long chain of dependencies. Calculating the actual value range requires a substitution step for each edge in the path, and most symbolic analyses limit the number of substitution steps to avoid infinite recursion. Once the substitution limit has been reached, an analysis substitutes the most conservative bound, e.g., $[-\infty : \infty]$ or $\top$, reducing the precision of the resulting value. For this reason, the tree center is a good choice for the root vertex v_s because it minimizes the number of substitutions that will need to be performed.

The depth-first visitor formulation of the merging algorithm in Fig. 3 applies equally well to a variable relation forest. The depth-first search executes start-vertex for each vertex that is not reachable from any edge of an existing tree in



(a) Variable relation tree

$$\begin{aligned} v_1' &= \phi(v_1) \\ v_2' &= \phi_{rel}(v_2, v_1) \\ v_4' &= \phi_{rel}(v_4, v_2) \\ v_5' &= \phi_{rel}(v_5, v_4) \\ v_6' &= \phi_{rel}(v_6, v_2) \\ v_3' &= \phi_{rel}(v_3, v_1) \end{aligned}$$

(b) Merges performed on variables in the tree

Fig. 4. A variable relation tree where vertex v_1 is arbitrarily selected as the root

the forest, and therefore effectively partitions the forest into separate trees merged independently.

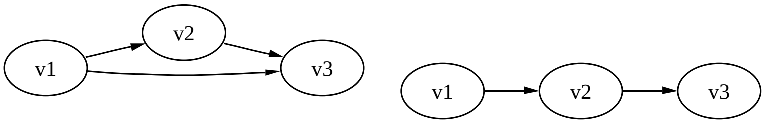
4.2 Graph properties

A depth-first traversal of a graph crosses the tree edges of a spanning tree in each connected component within the graph. We therefore construct our variable relation graph merge algorithm with two phases: the first phase selects a spanning tree in each connected component of the graph, and the second phase applies the depth-first—search visitor in Fig. 3 to each of these spanning trees.

The occurrence of cycles in the variable relation graph $G = (V, E)$ results in a loss of merge algorithm precision. Given two paths from a vertex v_i to another vertex v_j, the relations on both paths cannot be guaranteed to be satisfied, as at most one of the paths will become a part of the spanning tree. Thus the relations for edges not part of the spanning tree are not guaranteed to hold following the control-flow merge. The set of edges P whose relations are guaranteed to be preserved is equivalent to the set of edges in the spanning tree (or spanning forest). The precision of the algorithm is therefore $\frac{|P|}{|E|} = \frac{|V|-1}{|E|}$.

4.3 Spanning tree selection and pruning heuristics

The spanning tree selection in the first stage of the variable relation graph merge algorithm determines the set of relations that are preserved by the algorithm. We would prefer to select edges for relations that are more important for the goals of the analysis: for instance, an edge representing an invariant relation (i.e., one that is always true) may be more important than an edge representing a relation that would only benefit optimization. Similarly, the likelihood of a comparison actually being performed may weigh into the importance of an edge: one can count the static or dynamic occurrences of comparisons mapping to a particular edge as a measure of the importance of that edge. Once the application-specific importance measures have been coalesced, a maximal spanning tree algorithm may be used to select a spanning tree to be merged.



(a) Original graph (b) Transitive reduction

Fig. 5. The transitive reduction of a directed variable relation graph may reduce the effective precision of the merge operation.

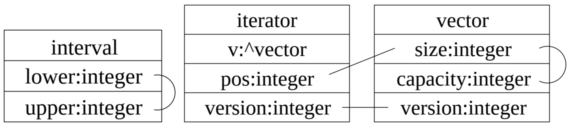
We may also prune unnecessary edges from the variable relation graph prior to spanning tree selection. We label a relation *implied* when the relation is a logical consequence of other relations, and therefore can be removed from the set of relations without affecting the final results. By analogy, an implied edge is an edge that represents an implied relation. Removing an implied edge from the variable relation graph increases the precision of the merge algorithm.

Implied edge candidates can be identified in a directed form of the variable relation graph. In this directed variable relation graph, an edge (v_i, v_j) denotes the relation $v_i \leq v_j$. Thus when there exists an edge (v_i, v_j) and another path from v_i to v_j (that does not contain (v_i, v_j)), the edge (v_i, v_j) may be implied. The variable relation graph, however, does not distinguish between relations that are invariant and those that are *transient*, i.e., may not hold in all paths at a particular control-flow merge. If the path from v_i to v_j contains only edges representing invariant relations, the edge (v_i, v_j) is implied and may be removed. Consider the directed variable relation graph and its transitive reduction in Fig. 5: if the relations (v_1, v_2) and (v_2, v_3) are invariant, then they must both be true at any program point and, therefore, (v_1, v_3) is implied and may safely be removed as in the transitive reduction of the graph. However, if the edge (v_1, v_2) is a transient relation, then it is possible that the relations represented by (v_2, v_3) and (v_1, v_3) hold on all incoming paths, but (v_1, v_2) does not; then our pruning will have made it impossible for the merge algorithm to guarantee the preservation of the relation represented by (v_1, v_3). Thus when (parts of) the variable relation graph represent intended program invariants, precision can be improved by computing the transitive reduction of the directed variable relation graph and removing the edge directions.

5. MERGING OBJECTS

The relational merge algorithm discussed in Section 4 depends on the construction of the variable relation graph. Thus far, we have assumed that variables correspond to program variables. While this assumption may be suitable for low-level internal representations that primarily consider integral or floating-point registers, it is not a particularly good assumption for higher-level internal representations, such as those used in the analysis of object-oriented programs.

In object-oriented programs, the majority of the important transient and invariant relations relate fields within an object or fields between objects. The example in Fig. 2 illustrates the most basic *intra*-object dependency, where a relation exists between two fields within the same object, e.g., `i.lower` $\leq$ `i.upper`



(a) Interval (b) Iterator and vector

Fig. 6. Static field relation graphs for the interval, iterator, and vector abstract data types

for object `i`, and Fig. 1 illustrates *inter*-object dependencies, where an important relation exists between a field of one object and a field of another object, e.g., $\texttt{iter.pos} \leq \texttt{students.size}$.

We introduce two *field relation* graphs (FRGs) that capture the relationships between fields. The field relation graph is similar to the role reference diagram in role analysis [Kuncak et al. 2002] because it captures relationships between fields of different objects, although the form of the information is different. There are two types of FRGs: the *static* field relation graph (discussed in Section 5.1), that describes relations between the fields of classes, such as the relationship between an iterator's position and a container's size, and the *dynamic* field relation graph (discussed in Section 5.3), that describes relations between fields of particular objects, such as the relationship between `iter.pos` and `students.size` in Fig. 1.

The two FRGs are directly related because they describe the same properties, i.e., relationships between fields, but we find that the static FRG is more readily constructed given a representation of the program source code (see Section 5.2) whereas the dynamic FRG is more useful for program analysis. In Section 5.4, we discuss a transformation that constructs a dynamic FRG from a static FRG, a process we call *instantiation*, drawing on the notion of access paths [Larus and Hilfinger 1988] to provide the crucial link between the two field relation graphs.

5.1 Static field relation graph

The static field relation graph captures the relationships between fields based on the class types involved. For instance, within the abstract data type `interval` of Fig. 2 there exists the relationship $\texttt{interval.lower} \leq \texttt{interval.upper}$ that must hold for any object of type `interval`. Similarly, there may exist dependencies between fields in different class types, e.g., $\texttt{iterator.pos} \leq \texttt{vector.size}$, as in Fig. 1. Fig. 6 illustrates two static field relation graphs. The first graph contains an *intra*-class relationship, i.e., a relationship that occurs between two fields within the same object, between the `lower` and `upper` fields of an `interval` type. The second graph contains an intra-class relationship between the `size` and `capacity` fields of an `vector`, but also an *inter*-class relationship, i.e., one that relates two fields from different object types, between the iterator's `pos` field and the vector's `size`. Like the variable relation graph, this graph is undirected and therefore describes only the fact that a relationship exists and not what the relationship means.

The nodes in the static field relation graph are fields of a particular class type and are written as $C.m$, where C is the class type and m is a field in that class. The undirected edges $(C_1.m_1, C_2.m_2)$ represent a relationship between field m_1 of some object o_1 of type C_1 and field m_2 of some object o_2 of type C_2.

The static relation graph can be constructed with information from several sources, and is discussed in Section 5.2. The graph can be used as input to the graph-based relational merging algorithm when objects of each class type can be trivially assigned. This occurs in several cases:

— When the analysis merges the states of all objects of a particular type into a single object of that type. In this case, objects are synonymous with classes, and the analysis proceeds as if fields were not instance-specific.

— When a connected component of the graph is restricted to fields within a single class type and the edges of the component relate to fields that will only be compared within a particular object, i.e., it is guaranteed that for the edge $(T.m_1, T.m_2)$ and objects o_1 and o_2 of type T, a comparison between $o_1.m_1$ and $o_2.m_2$ requires $o_1 = o_2$. Section 5.2 further discusses detection of this case.

5.2 Computing the static field relation graph

The static field relation graph can be computed with a flow-insensitive, context-insensitive analysis of a representation of the program source code or user annotations. Field relationships come from comparisons between fields in the program source, e.g., a comparison `a.m` $<$ `b.n` that appears in the source code would generate an edge between field `m` in the class type of object `a` and field `n` in the class type of object `b`. There are several ways that relations may be found in the program source representation, each of which provides slightly different semantic information:

— Relations may actually be user-specified class-level invariants. In this case, the comparison itself is implicitly represented by the existence of the invariant. Edges in the static field relation graph generated from these invariants are useful for relation graph pruning discussed in Section 4.3.

— Relations may come from assertion, precondition, or postcondition statements in the program code. The edges generated from these comparisons are locally invariant, because they are guaranteed to hold at the point of the assertion, precondition, or postcondition in a correct program.

— Relations may come from arbitrary conditionals in code, such as assignments to boolean variables or branch conditions. These edges detect only *controlling* relational properties, e.g., properties that affect the control flow of the program but not necessarily its correctness. Nonetheless, they do prove useful for enhancing the precision of analysis when the properties can be exploited locally.

During the construction of the static field relation graph, two types of edges are considered: intra-object relation edges and inter-object relation edges. An inter-object relation edge is an edge from a particular field in an object of some type to a field in an object of another (possibly different) type. An intra-object relation edge is an edge from a particular field in an object to another field in that same object. We note that an intra-object edge is an inter-object edge, so an analysis may conservatively classify all edges as inter-object. A comparison between fields within the program source code generates an intra-object edge when it is statically guaranteed that the two fields occur in the same object, e.g., the fields are accessed via the same object name as `o.m` and `o.n`.

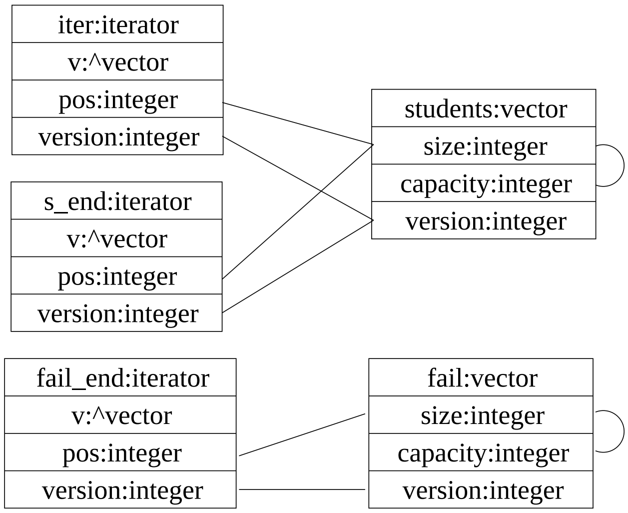


Fig. 7. Dynamic field relation graph for the objects in Fig. 1

When the static field relation graph contains a connected component consisting only of intra-object edges, these subgraphs of the static field relation graph can be used directly with the graph-based relational merge algorithm by applying the algorithm to the subgraphs for each object of the class type referenced by the subgraph. Moreover, because all of the information is static, graph pruning and spanning tree selection can be performed only once off-line.

5.3 Dynamic field relation graph

The static field relation graph captures relationships among fields in different class types, but fails to differentiate between objects that have the same class type. For instance, in Fig. 1, the `students` and `fail` vectors have the same type, but there are no interdependencies between them because the two are distinct objects. Perhaps more importantly, iterators have dependencies with the containers they reference but not with other containers of the same type. The distinction is crucial to the verification of the sample code: the `push_back` operation may invalidate iterators referencing the underlying vector. Thus, an analysis that does not differentiate between iterators that reference the vector `students` and iterators that reference the vector `fail` cannot prove that the iterators that reference `students` are not invalidated by a call to `fail.push_back`.

The dynamic field relation graph captures the relationships between instance-specific fields of objects, disentangling dependencies that appear to occur at the type level but do not occur at the object level, such as the relationships between iterators and the containers they reference. Fig. 7 illustrates the dynamic field graph for the important objects in our motivating example from Fig. 1, and here we see that the connected components for the `fail` and `students` vectors are distinct, even though the two vectors have precisely the same type. Thus the dynamic field relation graph can accurately model the relationships amongst objects in a program.

The dynamic field relation graph may be used directly with the graph-based relational merge algorithm. The value of a particular field in an object constitutes a variable in the variable relation graph, so the relational merge considers the edges in the dynamic field relation graph as edges in the variable relation graph, and therefore performs a control-flow merge operation that preserves important intra- and inter-object relationships. It is this formulation that provides the necessary framework to prove iterator validity in Fig. 1: the dynamic field relation graph provides an accurate model of the inter-object field relationships at each control-flow join and the relational merge algorithm preserves all relationships within the dynamic field relation graph, because in this case the graph (shown in Fig. 7) is a forest for which the relation merge algorithm guarantees perfect precision.

5.4 Instantiation

While the static field relation graph can be constructed from the program source code, the dynamic FRG cannot be synthesized so readily, because it must account for object references at each particular program point where the graph may be needed. However, analysis of our motivating example requires the precision of the dynamic FRG and therefore we describe an efficient algorithm for the construction of the dynamic FRG given an approximation of the references between objects as represented in a storage shape graph [Chase et al. 1990] and the static FRG, a process we refer to as *instantiation*.

The naïve instantiation algorithm assumes that for every edge $(C_1.m_1, C_2.m_2)$ in the static FRG, every object o_1 of type C_1, and every object o_2 of type C_2 there is a relationship between $o_1.m_1$ and $o_2.m_2$. This algorithm produces very dense dynamic field relation graphs, in the worst case producing a complete graph when given objects of a single class type. The resulting dynamic FRG will relate many objects that may otherwise be mutually unreachable in the storage shape graph. For instance, in addition to the edges in Fig. 7, the dynamic field relation graph produced by the naïve instantiation algorithm would include edges relating iterators from different containers and iterators with containers they cannot access. The naïve instantiation algorithm therefore reduces the precision of the relational merge algorithm by drastically increasing the number of edges in the graph. The reduced precision results in ineffective relational merging, because the important relations in the graph are not guaranteed to be preserved and the algorithm is no longer able to solve real problems, such as the verification of Fig. 1.

The goal of our instantiation algorithm is to capture only the relationships that can be useful at run time. To achieve this goal we determine the ways by which fields of different objects are compared. For instance, an iterator is considered valid if its index (`iter.pos`) does not exceed the size of the vector it references, which is stored in the object field `iter.v`. Thus, the real dependency is between `iter.pos` and `iter.v.size`, instead of between `iter.pos` and `vec.size` for some arbitrary vector object `vec`.

For each $(C_1.m_1, C_2.m_2)$ in the static field relation graph, we record the *access paths* α_1 and α_2 that result in the member accesses $C_1.m_1$ and $C_2.m_2$, respectively. An access path [Larus and Hilfinger 1988] is an expression mapping from an object through a (possibly empty) sequence of object field references and eventually to a field. Thus $o.m_1.m_2$ is an access path expression that begins with object o, projects to the object referenced by field m_1 in object o, and then projects to field m_2 of that object. The simplest form of an access path expression only follows field references, although for any particular language access paths can include other expressions, such as type casts, pointer dereferencing, or even pointer arithmetic. We will focus only on access path expressions involving field references.

Given access paths α_1 and α_2 that produce an edge $(C_1.m_1, C_2.m_2)$ in the static field relation graph, we calculate the longest common prefix α_p of α_1 and α_2 and note the class type C_p of the object resulting from the evaluation of the partial access path α_p. The class type C_p *generates* the edge $(C_1.m_1, C_2.m_2)$ if from some object of type C_p in the program we can establish a relationship between $C_1.m_1$ and $C_2.m_2$. We let β_1 and β_2 be the rest of the access paths, that originate at objects of class type C_p, such that $\alpha_1 = \alpha_p.\beta_1$ and $\alpha_2 = \alpha_p.\beta_2$. We record the access path pair (β_1, β_2) in a list of generated edges for class type C_p. The access path pair will later be used to construct edges within the dynamic FRG. This computation is performed off-line, and the result is a set of access path pairs for each class type. We note that when $C_1 = C_2$, $\beta_1 = m_1$, and $\beta_2 = m_2$ the edge $(C_1.m_1, C_2.m_2)$ is an intra-object dependency; otherwise, the edge is an inter-object dependency.

Given these access path pairs for each class type, we construct the edges of the dynamic FRG by following access path pairs from any object of the originating type. For each object o with class type C_p, we traverse the list of generated edges for that type C_p. At each access path pair (β_1, β_2), we traverse the access paths starting with object o (e.g., by following edges in the storage shape graph) to determine the two vertices in the dynamic field relation graph that may be related because they could be compared via the object o. We then add an edge between the two vertices in the dynamic field relation graph. This step is computed prior to a relational merge operation when a flow-sensitive storage shape graph is used, or can be computed once off-line if a flow-insensitive storage shape graph is available.

For the iterator validation example in Fig. 1, the access paths for the comparison $\texttt{iter.pos} \leq \texttt{iter.v.size}$ are $\alpha_1 = \texttt{iter.pos}$ and $\alpha_2 = \texttt{iter.v.size}$. The longest common prefix is $\alpha_p = \texttt{iter}$ with class type $C_p = \texttt{iterator}$. Then $\beta_1 = \texttt{pos}$ and $\beta_2 = \texttt{v.size}$, and the access path pair $(\texttt{pos}, \texttt{v.size})$ is added to the set of edges generated by `iterator`. When the dynamic field relation graph is computed, we consider the access path pair $(\texttt{pos}, \texttt{v.size})$ for all objects of type `iterator`, and will reconstruct all edges involving an iterator's `pos` field and a vector's `size` field that appear in Fig. 7, without creating any additional edges. The edges between `version` fields are generated analogously, completing the dynamic FRG for the example.

The instantiation algorithm described here efficiently constructs a precise dynamic field relation graph from a storage shape graph and the static field relation graph. The algorithm complexity is $\mathcal{O}(|O| \cdot |G| \cdot p)$, where $|O|$ is the number of objects (locations in the storage shape graph) whose type may generate any edges in the dynamic FRG, $|G|$ is the maximum number of access path pairs that generate edges from any particular class type, and p is the maximum length of any access path. In real programs, the access path length and the number of edges generated from any particular class type tend to be rather small, and in practice they are bound to small constants.

5.5 Pruning the dynamic field relation graph

The dynamic field relation graph for a particular program may become very large, as it contains all intra- and inter-object relationships that may affect the analysis. For a particular control-flow join, however, it is rare that the values differ across incoming control-flow edges for all fields in the graph, with many control-flow joins affecting only a small subset of the fields in the graph. Pruning is accomplished by constructing vertices only for field instances where the value differs on any two incoming control-flow edges; edges that would be introduced into the graph are only added if both vertices represent such fields, otherwise they are discarded.

The dynamic field relation graph that results from pruning during instantiation retains all relationships important to the static analysis that may not be preserved during a control-flow merge, but eliminates relationships that cannot change due to the control-flow merge. Pruning eliminates the cost of merging values that do not differ along the incoming control-flow edges and can drastically reduce the size of the dynamic FRG. Moreover, connectivity-based statistics for the full dynamic FRG, which are useful for spanning tree selection, can be calculated without building the full FRG, so that spanning tree selection heuristics do not suffer from pruning.

6. ALGORITHM COMPLEXITY

The algorithms presented for construction of the dynamic field relation graph and for relation-preserving value merging are executed once for each control-flow merge during the static analysis of a program. This frequency of execution requires that the algorithms be efficient in both time and space, and that they scale well with program size. Application of these algorithms occurs in two stages: construction of the dynamic FRG and merging along the edges of a spanning tree in the dynamic FRG.

The first stage, construction of the dynamic FRG as discussed in Section 5.4, iterates over each potential generator and each access path pair introducing edges into the dynamic FRG at each step. We let $|O|$ denote the number of objects in the program and note that $|O|$ is an upper bound on the number of potential generator objects, because every potential generator object must also be an object. We then let $|P|$ denote the maximum number of access path pairs for any particular generator class. For any particular access path pair, computing the field instance referenced by the access path from a particular generator is linear in the length of the access path, although we ignore this effect by bounding the maximum access path length to some arbitrary constant. Constructing the dynamic FRG therefore considers $\mathcal{O}(|O| \cdot |P|)$ edges, $|O|$ vertices, and executes in $\mathcal{O}(|O| \cdot |P|)$ time.

The second stage, which includes spanning tree selection and relation-preserving merging, traverses all edges in the dynamic FRG and applies either ϕ or ϕ_{rel} at each vertex. We denote the execution time complexity of ϕ and ϕ_{rel} by $\mathcal{O}(\phi)$ and do not specify it further because it differs greatly among different representations: very simple value range propagation, for instance, may require time proportional to the number of control-flow edges, whereas symbol range propagation may require cubic time in the number of program variables for each flow edge in the worst case [Blume and Eigenmann 1995]. The merge operation is applied to each field of each object, thus we let $|F|$ denote the maximal number of fields in any object type. The cost of relational merging along a depth-first traversal of the dynamic FRG is therefore $\mathcal{O}(|O| \cdot |P|) + |O| \cdot |F| \cdot \mathcal{O}(\phi)$.

The algorithmic complexity for $|O|$ objects that generate $|P|$ access path pairs each is therefore $\mathcal{O}(|O| \cdot |P| + |O| \cdot |F| \cdot \phi)$. Our algorithm is able to improve the precision of value range merging for object-oriented systems to enable verification of programs where traditional value range merging fails, such as the example in Fig. 1, with an efficient algorithm that does not suffer the exponential complexity explosion of path-sensitive algorithms. We conclude that our algorithm is effective for real problems yet efficient enough to be implemented in path-insensitive static analysis systems.

7. IMPLEMENTATION

We have implemented the relational merging algorithm presented here within the STLlint [Gregor and Schupp 2003] static verifier for C++ programs. With the aid of the dynamic field relation graph, STLlint is able to prove that programs involving iterator operations such as those in Fig. 1 are free of iterator usage errors. This goal was previously unattainable with symbolic range propagation, due to the loss of precision inherent in the symbolic range merge operation.

Introduction of the algorithm presented in this paper into STLlint required only two to three days of programming effort and less than a thousand lines of code. The majority of the work involved construction of the static FRG, including searching the program source code for object field comparisons and constructing access paths.

The STLlint implementation closely mirrors the algorithm presentation. The relational merging algorithm is implemented as specified by the depth-first search visitor in Fig. 3, using the Boost Graph Library [Siek et al. 2002]. Root vertex selection is performed using the maximal-degree heuristic based on the dynamic FRG (Section 4.3), although at present the spanning tree is arbitrarily selected.

The static FRG is constructed during a single pass over the source code, where undirected edges are introduced into the graph when a comparison expression is found that relates two object fields with a common root. Access paths within the static FRG implement the basic notion of access paths presented here, with several small extensions necessary for use with a more complex internal representation.

The dynamic FRG is constructed at each control-flow merge in two phases. The first phase determines the set of objects that have been modified on at least one of the incoming control-flow edges and marks each of these *live*. The second phase iterates through generator objects (as discussed in Secion 5.4), introducing edges between fields only when the fields reside within live objects (see Section 5.5).

Experience with this system has shown that the dynamic FRGs computed at control-flow merges tend to be small. In addition, the dynamic FRGs are often composed of several connected components (also small), indicating that the precision and performance problems associated with long dependency chains do not often occur. Additionally, even though STLlint does not implement a directed static FRG and the additional pruning opportunities it presents, we have discovered several examples where such a capability would be beneficial to the analysis. We expect that further application of the STLlint static verifier will uncover additional interesting properties of the static and dynamic FRGs.

8. RELATED WORK

Our work draws from the large body of existing work on value range propagation [Harrison 1977]. Value range propagation is used in many forms of static analysis, including bit width calculations [Stephenson 2000], static branch prediction [Patterson 1995], and loop dependency checking [Blume and Eigenmann 1995]. While the representations of value ranges vary greatly, from simple constants [Verbrugge et al. 1996] to complex symbolic domains [Blume and Eigenmann 1995] and even sets of disjoint, weighted value ranges [Patterson 1995], the underlying merge operation has remained essentially unchanged. We have expanded this area by considering a different merge operation with a different goal: instead of focusing on the precision of the value range itself, we focus on the precision of the value range with respect to other value ranges it will be compared against, resulting in an analysis that is similar to value range propagation but useful in different contexts, namely in the preservation of inter-object transient relationships.

The Value Name Graph (VNG) of Bodik and Anik [Bodik and Anik 1998] represents the path-sensitive relationships among the values of program variables without an exponential explosion in the state space. The approach differs from our own in two main ways: first, the information is encoded using a graph structure representing the relationships among variables at all points in the program, whereas our approach operates only at control-flow merge points regardless of the data structure used to store values on the incoming flow edges; second, our algorithm maintains path-sensitive equality and ordering relationships, whereas the VNG encodes only equality information.

Our use of the field relation graph shares a common goal with (related) field analysis [Aggarwal and Randall 2001; Ghemawat et al. 2000]: to model and prove intra- and inter-object field relationships. Related field analysis detects program invariants relating fields in two objects by verifying that the invariant is maintained by all subroutines that may access the fields in question. The key difference between our work and that of related field analysis is that our algorithm can detect local, transient properties in addition to program invariants, which is crucial to our analysis: verification of Fig. 1 requires that the transient relation `iter.pos` $<$ `students.size` be proven locally, but field analysis can only prove the weaker program invariant `iter.pos` $\leq$ `students.size` globally. While the latter is useful for many problems, it is not sufficient to verify the correctness of the iterator's behavior in the example.

Our static field relation graph may be viewed as a simplified form of the role reference diagram used in role analysis [Kuncak et al. 2002]. Role analysis studies the interconnections amongst objects, modeling references between objects via the role reference diagram to give a concise view of the object interactions in a system. Although our static FRG is similar to the role reference diagram, the two data structures are used in very different ways: the role reference diagram is a form of program specification against which the program is verified, whereas our static FRG is a model used to guide the control-flow merge algorithm and is extracted directly from the program source code.

We share the same implementation strategy as algorithms based on object sensitivity [Milanova et al. 2002], in that both our method and theirs use knowledge present in object-oriented systems to achieve a middle ground between sensitivity and insensitivity. Object sensitivity lies between context sensitivity and context insensitivity by being sensitive only to the object parameter of a method call, instead of sensitive to all parameters in a subroutine invocation, just as we are sensitive to certain inter- and intra-object relationships present in the FRG. Moreover, both analysis methods can be grafted onto existing analyses without requiring major changes to the infrastructure, because they are refinements of existing algorithms. The usefulness of the two approaches suggests that there may be further utility in studying the space between sensitive and insensitive algorithms, focusing on algorithms that provide sensitivity when it will impact the algorithm's precision and insensitivity when the additional precision gives little benefit.

Our STLlint static checker uses a similar approach to that of Extended Static Checking [Detlefs et al. 1998], which uses incomplete programmer-written specifications of program code and unsound verification methods to find various common problems in programs By reducing complex data types (e.g., iterators over a double-ended queue) to incomplete, unsound specifications (e.g., an iterator is merely an index into a container) in the vein of Extended Static Checking, we were able to apply the relational merge algorithm to C++ code such as that of Fig. 1, which may otherwise have required a much more powerful analysis.

The Concurrent Modification Problem (CMP) in Java occurs when an iterator from the Java Collections Framework [Chan et al. 1999] is used after the collection that it references has been modified. The CMP differs from iterator invalidation problems in C++ in that *every* collection modification in Java invalidates all iterators, whereas iterator modifications in C++ containers may invalidate some iterators only, depending on the position of the iterators and on the behavior of the container itself. For instance, erasure from a C++ `vector` invalidates all iterators after the point of erasure, but any iterators before the erased element remain valid. The CMP has been addressed using pointer must-alias relationships to verify that iterators reference the appropriate version of the container [Ramalingam et al. 2001]. However, this technique is not directly applicable to the C++ standard containers with more complicated, position-based invalidation rules.

9. CONCLUSION

We have presented an efficient control-flow merge operation that improves the precision of value range propagation with respect to arbitrary sets of numerical relationships among program variables. The resulting analysis achieves some of the benefits of path-sensitive analyses, such as the ability to determine when a particular relationship holds on all incoming flow edges, without the state explosion typical of path-sensitive analyses.

We have also illustrated the use of information regarding likely relationships between object fields gathered from program source code to direct the control-flow merge algorithm. With the use of the static and dynamic field relation graphs, we can model and preserve inter-object field relationships to improve analysis of object-oriented programs. We have found the field relation graphs to be crucial in the static verification of iterator loops seen in many C++ programs.

We have implemented the relational merge algorithm and field relation graphs in the STLlint static verifier. The implementation required only a modest effort within the existing symbolic interpreter, requiring less than a thousand lines of analyzer code and no major infrastructure changes. Our experience with STLlint implies that other static analysis systems could incorporate this algorithm with relative ease.

10. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation (NSF) NGS Grant 0131354.

REFERENCES

Aggarwal, A. and Randall, K. H. 2001. Related field analysis. In *Proc. of the SIGPLAN '00 Conf. on Programming Language Design and Implementation (PLDI)*. 214–220.

Alpern, B., Wegman, M. N., and Zadeck, F. K. 1988. Detecting equality of variables in programs. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*. ACM Press, 1–11.

Blume, W. and Eigenmann, R. 1995. Symbolic range propagation. In *Proc. of the 9th Inter. Parallel Processing Symposium*. 357–363.

Bodik, R. and Anik, S. 1998. Path-sensitive value-flow analysis. In *Proc. of the 25th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. 237–251.

Chan, P., Kramer, D., and Lee, R. 1999. *The Java Class Libraries: Supplement for the Java 2 Platform*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Chase, D. R., Wegman, M., and Zadeck, F. K. 1990. Analysis of pointers and structures. In *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation (PLDI)*. 232–245.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM Press, New York, NY, Los Angeles, California, 238–252.

Das, M., Lerner, S., and Seigle, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proc. of the SIGPLAN 2002 Conf. on Programming Language Design and Implementation (PLDI)*.

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. Tech. rep., COMPAQ Systems Research Center.

Engler, D., Chelf, B., Chou, A., and Hallem, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*.

Ghemawat, S., Randall, K. H., and Scales, D. J. 2000. Field analysis: Getting useful and low-cost interprocedural information. *ACM SIGPLAN Notices 35,* 5, 334–344.

Gregor, D. and Schupp, S. 2003. Making the usage of STL safe. In *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming*, J. Gibbons and J. Jeuring, Eds. IFIP Conference Proceedings, vol. 243. Kluwer, 127–140.

Harrison, W. H. 1977. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering SE-3,* 3 (May), 243–250.

Koenig, A. and Moo, B. E. 2000. *Accelerated C++*. Addison-Wesley.

Kuncak, V., Lam, P., and Rinard, M. 2002. Role analysis. In *Proc. of the 29th Annual ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages*. 1–16.

Larus, J. R. and Hilfinger, P. N. 1988. Detecting conflicts between structure accesses. In *Proc. of the ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation (PLDI)*. 21–34.

Milanova, A., Rountev, A., and Ryder, B. G. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. of the international symp. on software testing and analysis*. ACM Press, 1–11.

PATTERSON, J. R. C. 1995. Accurate static branch prediction by value range propagation. In *SIGPLAN Conf. on Programming Language Design and Implementation*. 67–78.

RAMALINGAM, G., WARSHAVSKY, A., FIELD, J. H., AND SAGIV, M. 2001. Deriving specialized heap analyses for verifying component-client conformance. Research Report RC22145, IBM Research Division. August.

ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1988. Global value numbers and redundant computations. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*. ACM Press, 12–27.

SIEK, J. G., LEE, L.-Q., AND LUMSDAINE, A. 2002. *The Boost Graph Library: User's Guide and Reference Manual*. C++ In-Depth Series. Addison-Wesley.

STEPHENSON, M. W. 2000. Bitwise: Optimizing bitwidths using data-range propagation. M.S. thesis, Massachusetts Institute of Technology.

VERBRUGGE, C., CO, P., AND HENDREN, L. 1996. Generalized constant propagation: A study in C. In *Proc. of the 6th Intern. Conf. on Compiler Construction (CC '96)*, T. Gyimothy, Ed. Lecture Notes in Computer Science, vol. 1060. 74–90.

Received Month Year; revised Month Year; accepted Month Year