Orion: High-Precision Methods for Static Error Analysis of C and C++ Programs

Dennis R. Dams and Kedar S. Namjoshi
Bell Labs, Lucent Technologies
600 Mountain Ave., Murray Hill, NJ 07974.
{dennis,kedar}@research.bell-labs.com

ABSTRACT

We describe the algorithmic and implementation ideas behind a tool, *Orion*, for finding common programming errors in C and C++ programs using static code analysis. We aim to explore the fundamental trade-off between the cost and the precision of such analysis. Analysis methods that use simple abstract domains run the risk of producing a high number of false error reports. On the other hand, the use of complex domains reduces the number of false errors, but limits the size of code that can be analyzed. We propose a new middle ground, where a two-phase process uses reasonably simple and efficient domains for an initial analysis, followed by a deep analysis of the discovered error paths in order to filter out as many false errors as possible. We describe the algorithms used for these phases, and their implementation in a GCC-based tool. Experimental results on a number of software programs bear out the expectation that this approach results in a high signal-to-noise ratio of reported errors, at an acceptable cost.

1. INTRODUCTION

We consider the use of data-flow analysis (DFA) as a debugging aid, as implemented e.g. in tools like Flexelint [15], Coverity [7], PolySpace [25], and Uno [19]. The inherent approximate nature of DFA leads in this context to *false alarms*: bogus error messages, which often are far more numerous than genuine ones. Such poor *signal-to-noise ratios* may be the reason why debugging aids based on data-flow analysis are not routinely used. If this problem can be overcome, extended static checks can become a standard element of the regular build process, much like the type checking that is already performed by compilers. A prerequisite is that the additional time taken by the analysis remains acceptable.

Adopting Abstract Interpretation terminology, we view DFA as exploring an abstract state space where an abstract state is a symbolic representation of a set of concrete (run-time) states. At every step of this traversal, a representation of the set of successor states needs to be computed, relative to some program statement. Precisely keeping track of such sets along a sequence of statements may quickly lead to complex expressions, even when the statements are seemingly simple. Indeed, the necessary representation language may be too complex for algorithmic checks of key properties, such as emptiness, of these sets. This fundamental undecidability is at the root of DFA's approximate nature. Data-flow analyzers therefore restrict the representation of state sets (the *abstract domain*) to decidable theories. The effect of a program statement on an element of such a domain is then computed by a *solver* operating over an abstract domain. It computes a domain value representing an over-approximation to the actual successor set. High-precision analyses, such as those implemented in PolySpace, limit this loss of precision by employing "heavy-weight" domains, requiring more sophisticated solvers. Calling such a solver at every step of the abstract state space traversal may then quickly become forbiddingly expensive.

1.1 Two-phase, Path-oriented Analysis

We propose an approach to reduce the resources required by solver calls, by using richer abstract domains only when necessary. The idea is to first perform an abstract state space traversal as described above with *light-weight abstract domains*. This traversal is *path-oriented*, meaning that it readily produces execution paths corresponding to potential errors. The use of light-weight domains keeps the exploration relatively fast, at the cost of a potentially high ratio of false error paths. In order to bring down this number, each of the error paths reported is then submitted to a second, more precise analysis. Since the size of the path is usually small in comparison with the entire abstract state space, we can afford to use advanced solvers here. Scrutinizing a single path then acts as a *feasibility check*. If the path is infeasible, it is suppressed; otherwise it is reported as a potential error. In our implementation, these two phases are actually interleaved.

Note that this approach exploits the fact that DFA is applied in the context of debugging. In contrast, when DFA is used for compile-time optimization, using simpler domains in a first phase will result in missing optimizations. Since there is no way of knowing what is missed, there is no point in performing a subsequent more precise analysis.

Our two-phase path-oriented analysis has been implemented in the program analyzer *Orion*, whose goal is to produce

high-quality[1] error reports with an acceptable run time. We aim at a signal-to-noise ratio of 3:1 or higher — meaning at least 3 out of every 4 reported errors have to be real ones. This number has been suggested by software developers as being acceptable. A major research goal behind this tool is to investigate the trade off between (1) the weight of the abstract domains during the phase-1 exploration and (2) the thoroughness of feasibility check — with the aim of maximizing the signal-to-noise ratio, at a minimal cost in computational resources required by the analysis. Our experiments with Orion show that, in most cases, we are able to achieve signal-to-noise ratios that meet or exceed the target.

Phase 1: Light-Weight Data-Flow Analysis. The phase-1 data-flow analysis performs a depth-first search on the product of the flow graph and a light-weight abstract domain. This domain is represented by a product of finite-state automata, each representing a particular data-flow fact being tracked. The depth-first search scheme allows error paths to be produced instantly. The abstract domain keeps track of any information that is necessary for Orion's defect detection — for example, in order to identify uses of uninitialized variables, it needs to be known which variables have not been assigned a value along each path being explored. In addition, the "low-hanging fruit" among the infeasible paths can be weeded out by keeping track of some minimal amount of additional information, e.g. known and inferred ranges for boolean and other variables.

While the path exploration is based on a depth-first search, we have developed a number of optimizations to it. Our algorithm utilizes the notion of *covering*, known from the area of Petri Nets [20], to shortcut the search. In addition, we propose a novel notion called *differencing* to further optimize the algorithm.

Phase 2: Tunable Feasibility Checking. The feasibility check to determine whether a path can actually occur under some run-time valuation of data variables makes use of theorem-provers. Theorem-provers are powerful but require human interaction and patience. However, Orion uses them in an "incomplete" way, resulting in an approach that is fully automatic and fast. Namely, the prover is allowed a predetermined amount of time — only if it finds within this time that the path is infeasible, that path is ignored. This crude — but effective — approach provides a simple way to *tune* Orion's precision by trading the signal-to-noise ratio for analysis time. In addition, the architecture of Orion allows to plug in different provers so as to experiment with alternatives and profit from advances in the field.

Several alternative approaches exist to ameliorate the signal-to-noise ratio. One is to *rank* the errors reported, based on heuristic rules, such that errors with a high probability of being real occur first. This idea is e.g. implemented in Microsoft's PREfix and PREfast defect detection tools [4], and also in MC/Coverity [17]. The drawback is that in order to be effective, such rules must be partly specific to the area of application, and consequently it may take a domain expert to devise effective heuristics. Another technique used in the above-mentioned tools is to suppress certain errors based on similarity to previously reported ones. Being largely orthogonal to our two-phased approach, many of these techniques may be combined with our feasibility checking method to get the best of both worlds.

[1]A high signal-to-noise ratio only is of course not all that matters. Another issue is *completeness*, i.e. how many "real" errors the analyzer reports. We discuss this point later in the paper.

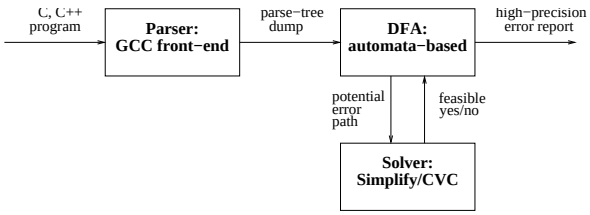


Figure 1: Orion tool architecture

1.2 Tool architecture

There are three main parts to Orion: the parser, the data-flow analyzer, and the solvers. The parser being used is the front-end of GCC, the Gnu Compiler Collection. Relying on this open source compiler has several advantages: it supports multiple languages, is widely used, and is being actively developed. The GCC version used is `3.5-tree-ssa`, a development branch that offers a uniform, simplified syntax tree for C and C++, and is intended to be the basis for future code transformations and static analyses to be built by GCC developers [23]. For example, work is currently in progress on integrating a front-end for Java.

GCC can dump the parse tree to a text file, which then forms the starting point for the analysis. The data-flow analyzer, together with several utilities, forms the core of Orion. This is written mostly in Objective Caml, about 18K lines of code altogether. The GCC dump is parsed into an OCaml data structure, from which a control-flow graph is constructed that is the main object of the path exploration. At every step of this search, each of a collection of *finite-state automata* is updated. Every automaton can be viewed as an *observer* of the sequences of statements being traversed in the path exploration, keeping track of the data-flow facts of a particular kind that hold at each control location that is reached. This modular set-up allows for an easy selection and combination of abstract domains in effect during the data-flow analysis.

Error paths, as flagged by one of the observer automata, are sent off to a solver during the path exploration. These solvers are theorem provers based mainly on decision procedures, aimed at providing automatic proofs for most questions submitted. Currently, two such provers have been interfaced to Orion: Simplify [26] and CVC [27].

1.3 Paper Outline

Sections 2 and 3 give details of the model-checking based abstract state space exploration of phase 1. In particular, they present the covering and differencing optimizations together with several theoretic and experimental results. Another optimization, *state aggregation*, is briefly discussed. Section 4 is concerned with the feasibility checking. It also explains how the interaction between depth-first search and

feasibility may affect completeness of the approach. Our experimental results on several programs are presented in Section 5. Section 6 discusses our contributions in the perspective of related work and concludes.

2. PATH EXPLORATION, COVERING, AND DIFFERENCING

The first phase of Orion's operation, as described in the introduction, can be viewed as model checking an abstract graph that is formed by partially interpreting program actions. Clearly, fully interpreting program actions leads to an infinite graph; thus, we need to perform an *abstract interpretation* [6]. Standard abstract interpretation calls for choosing an appropriate, lattice structured, abstract domain, and performing a fixpoint computation over the control flow graph (CFG) of a program. This computation collects abstract values at each control location until termination, which often can be ensured only through widening the fixpoint approximants. While our algorithms can be cast in this framework, they are better viewed as model checking-like algorithms, which perform the fixpoint iteration in a way that retains path information.

For our applications, it is important to know the control flow path that leads to an error, rather than knowing only the location of the error (such as reading an uninitialized variable). An error path provides the programmer with a clear and easily understandable justification for the error report. Moreover, such a path can be analyzed automatically to see whether it represents a feasible execution (i.e., whether it is executable for some values of input variables).

The simplest strategy is to just enumerate all paths through a control flow graph. This is clearly impossible, however: there may be infinitely many distinct paths due to the presence of loops. Tools such as PREfix/PREfast adopt a variant of this strategy [4], by limiting the number of paths that are considered, while attempting to cover as many CFG edges as possible. The alternative, which we employ, is to consider this analysis as a state-based, model checking problem on an abstract graph formed by the abstract interpretation of actions on the CFG, where the search is for reachable error states.

Perhaps the simplest model checking strategy is to use depth-first search (DFS) on the abstract graph. However, there is a significant potential for improvement through the use of a *covering* based search, where certain states generated by the DFS are *not* explored further if their future behavior is "covered" by that of other — already generated — states. The roots of the covering algorithm go back at least to work by Karp and Miller [20], but it has been reformulated several times in connection with various abstract domains (cf. [13, 12, 18]). We present below a new, very general formulation, which is a natural extension of the highly non-deterministic White-Grey-Black coloring search strategy followed, for instance, in [11]. Note that depth-first search, as well as other strategies, can be obtained by restricting the non-determinism in this algorithm.

The goal is to search a finite graph for the presence of a reachable error state. Throughout, let $G = (S, I, R)$ be the graph being searched, where S is a finite set of states, $I \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a transition relation. Let $E \subseteq S$ be a set of error states. We say that a state t is reachable in G from a state s, and denote it by $s \xrightarrow{*} t$, iff either $s = t$, or there is a finite sequence (a *path*) $s_0, s_1, \ldots, s_n$ with $n > 0$, where $s_0 = s$, $s_n = t$, and $(s_i, s_{i+1}) \in R$ for all i, $0 \leq i < n$. The set of reachable states, *Reach*, is given by $\{t \mid (\exists s : I(s) \ \wedge \ s \xrightarrow{*} t)\}$.

Algorithm I: Basic White-Grey-Black Search. We first describe the very general search strategy, attributable to Dijkstra, which starts with all states colored white, and re-colors states as grey or black during execution. The intuition is that white states are unexplored, grey states are partially explored, and black states are fully explored. The program is given below. The color of a state is given by its entry in the array "color". We abbreviate $(color[s] = W)$ by $white(s)$, and similarly for $grey(s)$ and $black(s)$. We use upper case symbols, e.g., $Grey, Black$, to indicate the set of states with that color. Actions are non-deterministically chosen (indicated by $[]$) guarded commands, where each guarded command has the form $guard \longrightarrow assignment$. The notation $([]s, t : \ldots)$ indicates a set of actions indexed by variables s, t of the appropriate types.

var color: array [S] of (W,G,B)
initially $(\forall s : white(s))$
actions
  $([]s : I(s) \ \wedge \ white(s) \longrightarrow color[s] \ := \ G)$
[] $([]s, t : grey(s) \ \wedge \ R(s,t) \ \wedge \ white(t) \longrightarrow color[t] \ := \ G)$
[] $([]s : grey(s) \ \wedge \ (\forall t : R(s,t) \ \Rightarrow \ \neg white(t))$
    $\longrightarrow color[s] \ := \ B)$

The key properties are: (a) invariantly, $Grey \ \cup \ Black$ is a subset of *Reach*, and (b) if *Grey* cannot increase, then $(Grey \ \cup \ Black)$ is the set of reachable states, and the size of *Grey* must decrease through the last action. Hence, we obtain the well-known theorem below.

THEOREM 0. *Algorithm I terminates with Black = Reach.*

Algorithm II: Covering-based Search. The basic WGB search terminates with $Black = Reach$, so to determine whether there is a reachable error state, one can check if $Black \cap E$ is non-empty. (Of course, this check could be made during the execution of the algorithm, but in the worst-case, the algorithm may have to explore all reachable states.) The goal of a covering-based search is to explore *fewer* than all reachable states, while still determining the exact answer. This is done by exploiting a "covering" relation between states where, informally, if a state s covers state t, then any path to an error state from t has a "matching" path to an error from s. Thus, there is no harm in stopping the exploration from t if it is covered by an already explored state.

The precise property of the covering relation that ensures soundness is, however, a bit more subtle, especially as our goal is to create a highly nondeterministic algorithm. Given is a *subset-covering* relation (usually a pre-order), $\sqsupseteq \subseteq 2^S \times 2^S$, with the following property: for any subsets X, Y of S,

if $X \sqsupseteq Y$ (read as X *covers* Y), then: for any $k > 0$, and any y in Y that has a path of length k to an error state, there is x in X that has a path of length *at most* k to an error state. We also introduce an additional color, Red (R), to label states that have been covered. The new algorithm is given below – it extends the WGB algorithm with the final indexed action.

var color: array [S] of (W,G,B,R)
initially $(\forall s : white(s))$
actions
$([]s : I(s) \ \wedge \ white(s) \longrightarrow color[s] \ := \ G)$
$[]$ $([]s, t : grey(s) \ \wedge \ R(s,t) \ \wedge \ white(t) \longrightarrow color[t] \ := \ G)$
$[]$ $([]s : grey(s) \ \wedge \ (\forall t : R(s,t) \ \Rightarrow \ \neg white(t))$
$\longrightarrow color[s] \ := \ B)$
$[]$ $([]s : grey(s) \ \wedge \ ((Grey \setminus \{s\}) \ \cup \ Black) \sqsupseteq \{s\}$
$\longrightarrow color[s] \ := \ R)$

The new action colors s red if it is covered by the other explored, but *uncovered* (i.e., non-red), set of states. Once a state is colored red, exploration is stopped from that state, as if it is colored black, but with the difference that red states cannot be used as part of a covering set. Both path matching and the distinction between red and black states are important: there are simple examples showing that dropping either one leads to an unsound or incomplete search method. As the algorithm proceeds, all white initial states are examined and turn grey, and grey states turn red or black. Thus, the final result is given by the following theorem. Note that $(Black \ \cup \ Red)$ is generally a *strict subset* of the reachable states, yet it suffices to detect errors.

THEOREM 1. *Algorithm II terminates, and there is a reachable error state iff one of the states in* $(Black \ \cup \ Red)$ *is an error state.*

Algorithm III: Adding Differencing Mechanisms. In the previous algorithm, a state t is covered, intuitively, because *all* error paths from t have matching paths from the set of states that cover it. It may, however, be the case that for some states t that remain uncovered, *most*, but not all, of the error paths from t are matched by already explored states, while the remainder can be matched from a small set of as yet unexplored states. These new states represent, in a sense, the small "difference" of the error behavior of t and that of the already explored states. One may then choose to stop exploring t, and explore the difference states instead. We modify the last action of the previous program to enable this choice, with "Diff" being the choice of the difference set. (The previous action can be recovered by setting Diff to the empty set, with the convention that $T \sqsupseteq \emptyset$ is true for all T.)

$([]s, Diff : grey(s) \ \wedge \ (Diff \subseteq White) \wedge$
$((Grey \setminus \{s\}) \ \cup \ Black \ \cup \ Diff) \sqsupseteq \{s\} \ \wedge \ \{s\} \sqsupseteq Diff$
$\longrightarrow color[s] \ := \ R; ([]t : t \in Diff : color[t] \ := \ G))$

The new action chooses a difference set *Diff* for some grey node s, and uses it to provide a covering. Then s is colored red, and all the states in *Diff* are colored grey. The choice of a difference set cannot, however, be completely arbitrary: it cannot include red states, and $\{s\}$ must cover *Diff*. This last constraint ensures that, even though Diff may contain unreachable states, any errors found from Diff states are also errors from $\{s\}$ and, inductively, from a reachable state. The full proof of correctness takes this into account, and otherwise is essentially identical to the proof for the covering only search.

THEOREM 2. *Algorithm III terminates, and there is a reachable error state iff one of the states in* $(Black \ \cup \ Red)$ *is an error state.*

3. APPLICATION TO CONTROL-FLOW GRAPHS

In Orion, the differencing-based algorithm is applied in phase-1 of the analysis to control flow graphs of individual functions. Formally, a control flow graph (CFG) is a tuple (V, V_0, Σ, E), where V is a finite set of *control locations*, $V_0 \subseteq V$ is a set of *entry locations*, Σ is a set of *program statements*, and $E \subseteq V \times \Sigma \times V$ is a finite set of *program transitions*, where a transition (n, a, n') can be viewed as a control flow transition from n to n', labeled with statement a. Orion adopts the model checking approach to analysis: thus, analysis properties are represented by finite state automata, which operate on Σ, and reject if an erroneous execution is found. An analysis automaton is a tuple $(S, I, \Sigma, \Delta, F)$, where S is a finite set of *states*, $I \subseteq S$ is a set of *initial states*, Σ, as above, is the *input alphabet*, $\Delta \subseteq S \times \Sigma \times S$, is the *transition relation*, and $F \subseteq S$ is the set of *rejecting states*. The (synchronous) composition of K such automata, $A_1, \ldots, A_K$, with a CFG (V, V_0, Σ, E) can be viewed as the single automaton with: $V \times S_1 \times \ldots \times S_K$ as the state set, $V_0 \times I_1 \times \ldots \times I_K$ as the initial states, Σ as the input alphabet, with a transition relation Δ defined by $((n, s_1, \ldots, s_K), a, (n', s'_1, \ldots, s'_K)) \in \Delta$ iff $(n, a, n') \in E$, and $(s_i, a, s'_i) \in \Delta_i$ *for all* i in $[1, K]$, and $(n, s_1, \ldots, s_K)$ being a rejecting state iff *for some* i in $[1, K]$, s_i is in F_i.

The model checking problem is to determine whether there is a path in this product to an rejecting state (recall that rejecting automaton states signal program errors). In order to use covering and differencing during the search, we need a general mechanism by which covering and differencing operations for individual automata are combined to provide such operations for a tuple of automata. We show how this can be done below.

We assume the existence of individual covering relations, $\sqsupseteq_i$, for each automaton, and construct the *point-wise global covering* relation: $(n, s_1, \ldots, s_K) \sqsupseteq (n', s'_1, \ldots, s'_K)$ if and only if $n = n'$ and $s_i \sqsupseteq_i s'_i$ for all i in $[1, K]$. For example, one automaton could keep track of the current set of uninitialized variables, U, in its internal state. This can be done by letting each state of the automaton be one such set, and including a transition from a state U to a state U' on program action a, if U' results from U by removing variables defined in a and adding variables newly declared in a (e.g., local scope declarations). Another automaton could, similarly, keep track of an estimate, B, of upper and lower bounds for each defined variable. For the first automaton,

$U \sqsupseteq_1 U'$ may be defined as $U \supseteq U'$. If there is a sequence of program actions that result in an error because a variable, say x, in U' is accessed by the last action without being defined along the sequence, then the same sequence causes an error from U, as x is included in U. Similarly, $B \sqsupseteq_2 B'$ iff the bounds information in B' is more precise than that in B. Any error path from B' can be enabled from B, as the information in B is more approximate than that in B'. The joint covering relation ensures that $(\{a,b\},(c>0))$ covers $(\{a\},(c>1))$, since $\{a\}$ is a subset of $\{a,b\}$, and $(c>1)$ implies $(c>0)$.

Similarly, given functions $diff_i$ that choose an appropriate difference set for each automaton, we can define a *point-wise global differencing* function as follows: $diff((n,s_1,\ldots,s_K),(n',s'_1,\ldots,s'_K))$ is defined only if $n = n'$ (i.e., same control location), and is given by the set $\{(n,s) \mid (\exists i, d : d \in diff_i(s_i,s'_i) \ \wedge \ s = (s_1,\ldots,s_{i-1},d,s_{i+1},\ldots,s_K))\}$. I.e., for the common control location, the difference set is obtained by taking, in turn, the difference of one of the component automaton states, while keeping the others the same. For the example above, the difference function for uninitialized variable sets is just set difference, while that for bounds information is bounds difference. Thus, the global difference of $(\{a,b\},(c>0))$ relative to $(\{a,d\},(c>10))$ at control location n is given by the set $\{(n,(\{b\},(c>0)),(n,\{a,b\},(c>0 \ \wedge \ c \leq 10))\}$.

Aggregating States. An optimization made in Orion to enable fast covering and differencing calculations is to merge the set of currently reached automata states for each control location in an approximate, but conservative manner. This is done by defining merge functions per automaton domain, and applying them point-wise to states with the same control location. Thus, if $merge_i$ is the merge function for automaton i, then the *point-wise global merge* is defined as $merge((n,s_1,\ldots,s_K),(n,s'_1,\ldots,s'_K)) = (n, merge_1(s_1,s'_1),\ldots,merge_i(s_K,s'_K))$. In our example, the merge of uninitialized variable sets is done through set union, and that of bounds by widening bounds. Thus, $merge((n,\{a,b\},(c>0)),(n,\{a,d\},(c>10)))$ gives $(n,\{a,b,d\},(c>0))$. While the merge operator can be quite approximate, we have not observed it to lead to many false error reports, and furthermore, it does indeed considerably speed up the covering and differencing operations.

Relationship to Standard DFA. The covering algorithm, implemented for control flow graphs, is equivalent to the standard fixpoint calculation in data flow analysis. This is because, in DFA, the collecting semantics aggregates abstract information per control flow node. During an iteration, if new information flowing in over an edge is a subset of the collected information, no further changes are triggered; this is equivalent to a covering. Differencing represents an optimization also to the standard DFA. Depth-first search can be seen as a particular scheduling of the general chaotic iteration scheme for data flow analyses. However, our covering and differencing algorithms presented earlier are more general than DFA, since they do not require a control flow graph skeleton on which to execute the algorithm. It is important to note that classical DFA algorithms only collect information at program points; they do not retain path information, which is essential for an independent feasibility check.

Experimental Evaluation. Experiments with these algorithms show a clear advantage in execution times (at least 3-fold, usually more) of covering/differencing relative to pure depth-first search. The intuition behind the differencing method is that it can speed up the search by: (i) covering more states, thus exploring fewer states overall, and (ii) faster computations, as difference states are generally smaller than the original (e.g., a subset). Experiments so far show, however, that for the properties we check for with Orion, the speedup obtained with smaller representations is nearly matched by the cost of the differencing operation. Thus, the benefit in run times is — as yet — marginal, with a maximum speedup of about 10%. We continue to explore this issue, however, and to look for more efficient differencing implementations. An interesting theorem shows that the first benefit does not arise for monotonic abstraction functions (the second can, however).

THEOREM 3. *For abstraction functions on program statements that are* monotonic *relative to a covering preorder, the number — but not the values — of automaton states explored is identical whether or not differencing is used in addition to covering.*

4. FEASIBILITY CHECKING

As explained in the introduction, a potential finite error path detected during the phase-1 search is subjected to further analysis by a "feasibility" check, which determines whether it can correspond to a real execution. The integration of bounds information in the abstract state in phase-1 already permits ruling out some error paths as infeasible during phase-1. The feasibility check described here is applied in phase-2 to other error paths that are generated by the phase-1 search. In this section we describe the manner in which this check is performed, optimizations, and some surprising consequences for the completeness of the error detection strategy.

4.1 Checking for Feasibility with Weakest Preconditions

Any finite path through a control flow graph of a function is a sequence formed from assignment statements, function calls, and boolean tests. For example, $read(\&x);(x<0);y=x;(y>3);z=f(x,y)$ could be such a path, where x,y,z are integer valued variables, and f is a function. This path is infeasible: from the first three actions, one may infer that the value of y is negative; hence, the subsequent test $(y>3)$ fails. Such inference can be formalized in many equivalent ways; we do so primarily through the use of *weakest liberal preconditions* for statements, as introduced by Dijkstra in [10].

The weakest liberal precondition for statement S to establish property ϕ after execution, denoted by $wlp(S,\phi)$, is that set of states from which every terminating execution of S makes the program enter a state satisfying ϕ. Letting $s \xrightarrow{S} t$ denote the fact that execution of statement S from state s can result in state t, this set is formally defined as $wlp(S,\phi) = \{s \mid (\forall t : s \xrightarrow{S} t \ \Rightarrow \ \phi(t))\}$. The weakest pre-

condition can be calculated by substitution for simple assignments ($wlp(x = e, \phi(x)) = \phi(e)$), and by implication for tests ($wlp(g, \phi) = (g \Rightarrow \phi)$), and inductively for sequences of actions ($wlp((S_1; S_2), \phi) = wlp(S_1, wlp(S_2, \phi))$). Its relationship to feasibility checking is given by the following theorem.

THEOREM 4. *A finite path π is feasible if and only if $wlp(\pi, false)$ is not valid.*

PROOF. From the inductive definition of *wlp* for paths, one obtains the following set-based characterization of $wlp(\pi, \phi)$: it is the set $\{s \mid (\forall t : s \xrightarrow{\pi} t \Rightarrow \phi(t))\}$. A path π is feasible iff (by definition) there are program states s, t such that $s \xrightarrow{\pi} t$ holds. By the prior characterization of *wlp*, this is equivalent to saying that $wlp(\pi, false)$ is not universally true. □

For the example above, the *wlp* calculation proceeds as follows: $wlp(z = f(x, y), false) = false$; $wlp((y > 3), false) = ((y > 3) \Rightarrow false) = (y \leq 3)$; $wlp(y = x, (y \leq 3)) = (x \leq 3)$; $wlp((x < 0), (x \leq 3)) = ((x < 0) \Rightarrow (x \leq 3)) = true$; and $wlp(read(\&x), true) = true$. But *true* is trivially valid: hence, by the theorem above, this path is infeasible. Notice that as $wlp(\pi, true) = true$ holds for all π, infeasibility can be detected early, and the *read* statement does not really need to be examined. The weakest precondition calculates backwards; its dual is the strongest postcondition, *sp*, which calculates forwards, and is defined as follows: $sp(S, \phi) = \{t \mid (\exists s : \phi(s) \wedge s \xrightarrow{S} t)\}$. Their duality leads to the following theorem.

THEOREM 5. *A finite path π is feasible if and only if $sp(\pi, true)$ is satisfiable.*

PROOF. It is well known that *wlp* and *sp* are near-inverses (formally, adjoints in a Galois connection). Thus, $(\psi \Rightarrow wlp(\pi, \phi))$ is valid iff $(sp(\pi, \psi) \Rightarrow \phi)$ is valid. Substituting $\psi = true, \phi = false$, we obtain that $wlp(\pi, false)$ is valid iff $\neg(sp(\pi, true))$ is valid; thus, $wlp(\pi, false)$ is not valid iff $sp(\pi, true)$ is satisfiable. □

We thus have two equivalent ways of calculating feasibility: either perform a forwards, symbolic calculation of $sp(\pi, true)$ and apply a satisfiability solver, or perform a backwards, substitution-based calculation for $wlp(\pi, false)$ and apply a validity checker. The approach we have implemented in Orion is somewhat of a hybrid: we calculate aliasing and points-to information along π in the forward direction, and use it to reduce the intermediate results of the *wlp* calculation. A key point is that aliasing information is quite accurate along a single path, more so than when it is calculated for a control flow graph, where accuracy is lost when merging information for incoming edges at a CFG node. The need for such points-to information is due to the fact that *wlp*, when applied to assignments through pointer variables, results in a case explosion. For instance, computing $wlp(*p = e, \phi(x, y))$ requires a case split on whether p points to x, to y, or to neither. Formally, it is given by the expression below, where $pt(p, x)$ is a predicate that is true iff p holds the address of x.

if $pt(p, x)$ *then* (*if* $pt(p, y)$ *then* $\phi(e, e)$ *else* $\phi(e, y)$)
else (*if* $pt(p, y)$ *then* $\phi(x, e)$ *else* $\phi(x, y)$)

Without points-to information to contain this case splitting, the *wlp* expressions at intermediate points on the path can grow exponentially, resulting in a slowdown. Orion can also use bounds information gathered for this path during the first phase analysis of the control flow graph to similarly reduce arithmetic expressions early in the *wlp* calculation.

On computing the *wlp* for a path, Orion sends this expression to a validity checker — we currently use either Simplify [26] or CVC [27] as the checkers. Thus, the path analysis is carried out using the full semantics of the programming language, as encapsulated in the *wlp* definition for the language constructs. An interesting observation is that the use of *wlp* automatically provides a *slicing* of the path relative to the feasibility check, since an assignment that does not affect variables in the current post-condition is treated as a no-op by the *wlp* substitution mechanism (e.g., $wlp(z = e, \phi(x)) = \phi(x)$).

An example of the effect of Orion's feasibility check is shown in Figure 2. When run on the source code shown on the left, without feasibility check the error path on the right is produced. With feasibility checking enabled, no errors are reported. The example is somewhat contrived to demonstrate several aspects of Orion's reasoning power on a few lines of code.

4.2 Implications for Completeness

Orion is focussed on finding errors. Two important correctness (i.e., non-performance) aspects of such procedures, are (i) *soundness*: is every reported error feasible? and (ii) *completeness*: does the procedure find *all* real errors?

Although Orion uses feasibility checking, as described above, to filter out false error reports, soundness is weakened due to fundamental limitations in decision procedures. These include the exponential complexity of some decision procedures and, indeed, the non-computability of validity for certain logics (e.g., arithmetic with multiplication). To achieve a reasonable analysis time, Orion limits the time allowed for the solvers, inevitably permitting in some false error paths to be reported. Orion aims to achieve a high degree of – but not perfect – precision in its error reports, obtaining a kind of high-probability soundness.

An analysis algorithm can obtain perfect completeness by reporting all potential errors: all real errors are contained in this report. However, this comes at the expense of soundness. Thus, there appears to be a balance between the two aspects: it may be necessary to sacrifice some completeness in order to achieve a high degree of soundness. In what follows, we point out two such tradeoffs in Orion.

```
int f(int i)            // line 1
{                       // 2
  int r, a, *p = &a;    // 3
  int m = 1;            // 4
                        // 5
  if (i<2)              // 6
    r = m*i;            // 7
  else                  // 8
    m++;                // 9
  a = m*(i+1);          // 10
  if (*p>=6)            // 11
    r = m*6;            // 12
  return r;             // 13
}
```

```
example.c:13 (function example.c:f) ::
  use of un-initialized variable(s): r

Possibly feasible error path:
example.c:3:  p=(&a)
example.c:4:  m=1
example.c:6:  !((i<=1))
example.c:9:  m=(m+1)
example.c:10:  a=(m*(i+1))
example.c:11:  !(((*p)>5))
example.c:11:  ((void)0)
example.c:13:  return_value =r
```

Figure 2: With feasibility checking, the error path (on the right) is suppressed.

State-based vs. Path-based Search. In the previous section, we argued that path-based search was impractical, and presented three state-based search algorithms. While these are indeed more efficient, they may compromise completeness, when combined with feasibility checking. Consider, for instance, the program below.

```
int foo(int x)
{
  int u,v;
L1:  if (x > 0) v=x; else v=x+1;
L2:  if (x < 0)
L3:       return u;
L4:  else return x;
}
```

The return statement at L3 is a candidate for an uninitialized variable error, and there are two possible paths to L3: $P_1 :: (x > 0); v = x; (x < 0); \mathit{return}\ u$, and $P_2 :: (x \leq 0); v = x + 1; (x < 0); \mathit{return}\ u$. A search that tries all paths will consider both, and point out P_2 as a feasible error path. But now consider a depth-first search that only keeps track of the current set of uninitialized variables. If the search tries path P_1 first, it will find the potential error, but a feasibility check will reject the path as being infeasible. Backtracking to L1, the search tries the else-branch at L1. However, it enters L2 with the same set of uninitialized variables, $\{u\}$, as before, and must backtrack without exploring P_2 in full. Hence, the real error goes unreported, a failure of completeness!

Notice that the failure is due to the feasibility check: dropping the check will cause an error report to be generated, with a path that is infeasible — but, of course, at the cost of soundness. The fundamental problem is that *any* state-based search that uses a finite abstract domain can be "fooled" into not distinguishing between some of the possibly infinitely many different paths that reach a control point. Orion actually avoids the problem for this example, since it also keeps track of variable bounds information that serves to distinguish the prefixes of P_1 and P_2 at L2. However, it may be possible for a complex enough program to fool the bounds tracking procedure into considering distinct prefixes as indistinguishable from one another: [8] has a discussion of this phenomenon in the more general context of abstraction methods.

Covering vs. Depth-first Search. As described in the previous section, a covering-based search is far more efficient than pure depth-first search. However, this too, may come with a completeness cost. The covering property preserves the existence of error paths, but not specific errors: i.e., if state t covers state s, and there is a path to an error from s, there must be a matching path from t, but not necessarily to the *same* error location. This general problem does not hold for the case of control-flow graphs, since covering states share a common control location. However, we have observed that a covering based search can miss reporting some errors in our tests of Orion. In this case, it is due to a different, but related phenomenon: the potentially over-approximate aggregation. Such an over-approximation enables some states to be covered, while this would not have been the case with an exact aggregation — such false covering, if it occurs, leaves some paths unexplored. However, in our tests, the number of missed errors is small (usually one or two) and, in all the cases we encountered, the missed error paths are infeasible. There is a tradeoff here that can be exploited. For instance, one can try Orion with the fast covering-based search for initial testing, but once all reported real errors have been fixed, one may try Orion with the more comprehensive, but slower depth-first search to expose any missed errors.

5. EXPERIMENTAL RESULTS

The table below summarizes the result of running Orion on various publicly available and proprietary software packages. The errors being checked are for the use of uninitialized variables, null pointer dereferencing, and out-of-bounds indexing of arrays. The table shows, from left to right, the source with its primary coding language, the number of lines of code (in 1000s), and the results of running Orion with and without external solvers for the feasibility check. When using the external solvers, the time they get for analyzing a single error path is limited to 2 seconds here. The last column shows the number of errors, out of those reported, that were deemed to be real by manual inspection. We considered an error path to be real if it is feasible locally within a procedure. It may be that some of the "real" errors are infeasible when the full program is considered, if their feasibility depends on assumptions regarding the context of the procedure. The execution times are for the analysis only, i.e., they do not include the time taken in parsing. They are measured in seconds, but these measurements are not

directly comparable over the different entries, as they were collected in multi-user mode on different machines and operating systems.

Source	KLoC	No Feas.		With Feas.		Real
		time	errs.	time	errs.	Errs.
`jpeg-6b(C)`	29.5	36	6	36	0	0
`RATS(C,C++)`	58	183	26	198	19	15
`sendmail(C)`	73	151	6	176	3	3
`linux(C)`	369	340	24	333	12	11
`anonOS(C)`	1123	*	*	1613	105	*

The `jpeg-6b` program is a JPEG library that comes with the Ghostscript program. The `linux` sources are for version 2.4.18 . `anonOS` is a proprietary operating system for which we had access to the source. For this system, the error reports were not analyzed manually for feasibility, which is why the "Real Errors" column has the entry "*". The version of `sendmail` is 8.11.6, the same version as considered in [19]. `RATS` is a package for simulating optical routers, developed at Bell Labs.

In each case, the numbers with most significance are the number of reported errors, and the fraction of those that are deemed to be real. The numbers show that the addition of the feasibility check significantly reduces the number of false error reports, bringing the signal-to-noise ratio into the range that developers would consider acceptable, as outlined in the introduction.

6. DISCUSSION

6.1 Related Work

Run-time Checking and Dynamic Analysis. An alternative way to detect programming errors is to monitor the code during execution. Tools such as Purify (from Rational/IBM) and CCured [24] insert checks into the code to this effect, that get compiled and executed along with the program. VeriSoft [16] and JPF [28] on the other hand do not insert checking code, but can be seen as advanced schedulers that perform various checks on the underlying code. Since they have full control over the scheduling, several different executions can be tested, i.e. they can be viewed as bounded model checkers. What run-time checkers and dynamic analyzers have in common is that they explore an *under-approximation* of a program's run-time state space. In this sense they are orthogonal to the DFA approach. Indeed, Orion offers facilities for inserting run-time checks for array-bound violations.

Static Analyzers. Tools that approach Orion most closely in terms of usage, purpose, and underlying techniques are Uno [19], MC/Coverity [17], PolySpace [25], klocwork [21], ESP [9], and FlexeLint [15]. The distinguishing features of Orion are orthogonal to the techniques for ranking error messages based on heuristics and history information, and can be combined to get the best of worlds. They are:

- Its 2-phase approach, which results in excellent signal-to-noise ratios without serious time penalties. At the same time it gives the user control over the desired precision: High-precision can be achieved by allowing the solvers more time to perform a deep semantic analysis of the error paths returned by the first, more superficial phase. The experimental results reported show that this reduces the amount of false alarms in a significant way, without excessively burdening the analysis time.
- The uniform application of the covering and differencing optimizations. MC/Coverity uses similar techniques to handle aggregate state machines produced by the per-variable check paradigm adopted in that tool (cf. the handling of block summaries by the tool). Uno implements a form of covering and differencing as well. Our general treatment allows us to prove the correctness of covering and differencing independently of the search scheme and the covering relation used, and to pinpoint a limitation of the effect that differencing may have when used with abstraction functions that are monotonic w.r.t. the covering relation (Theorem 3). The current implementation of Orion uses a covering relation over a combined domain for tracking uninitialized variables and bounds information, as explained in Section 3. We are considering adding points-to information as well; such extensions are easy to add thanks to the generality of the implementation.

While the high degree of automation offered by these tools increases their acceptance by software developers, there will always be errors that escape such static analyses — this is due to the undecidability of the problem of showing the absence of errors. If correctness is a serious concern, like in case of safety-critical applications, tools that require *annotations* can offer more certainty. ESC [14] and LClint [22] are examples of such tools. Orion also allows insertion of a limited form of user-annotations, but this feature has not been used in the experiments.

Software Verification Tools. The architecture of Orion bears a strong resemblance to that of software verification tools based on symbolic processing, such as SLAM [2], BLAST [18], and BANDERA [5]. All these tools use various forms of abstraction combined with model checking, and perform feasibility checking of errors. However, there is an important distinction between verification and static error analysis: verifiers are targeted towards producing a yes/no answer for a particular query, whereas static error analyzers are targeted towards producing multiple error reports. This distinction forces verifiers to apply conservative abstractions, and make a substantial effort, in terms of using complex abstract domains, in order to prove correctness. As a result, they are limited in terms of the size of code that can be handled: usually, to about 10-20KLoC. Since static error analyzers are designed to return multiple answers, they can be afford to be unsound and include false error reports in the answer set, so long as the fraction of false reports is reasonably small.

Orion, therefore, can be seen as an effort to apply techniques from the model checking and verification field to the problem of improving the precision of static analysis to acceptable levels. Our experimental results appear to bear out the hypothesis that the 2-phase analysis procedure discussed in this paper is effective at performing high-precision static analysis. Some of the technical details of Orion's im-

plementation differ from, or extend, the algorithms used in the verification tools mentioned above. The covering search algorithm presented here is more general than the one used in BLAST, and the addition of differencing is novel. Orion analyzes paths using weakest preconditions, as in BLAST, but with a forward, path-specific, points-to analysis (SLAM uses symbolic processing — i.e., strongest postconditions — to analyze for feasibility). On the other hand, these verifiers include methods for automatic refinement of the initial abstraction, based on hints obtained from the infeasibility proofs for false error paths — such refinement is necessary to show correctness. So far, we have not found a need to add such abstraction refinement: the feasibility checking mechanism appears to do a good enough job of filtering out false errors and, as explained before, we do not aim to prove correctness of code.

6.2 Ongoing and Future Work

Orion can also perform an interprocedural analysis for uses of uninitialized global pointer variables. The implementation is based on standard techniques for model checking of recursive state machines [3, 1], and thus can deal with recursive functions in C and C++. No feasibility checks are performed in this case, since interprocedural paths tend to be very long and would lead to unacceptable analysis times. Improving the signal-to-noise ratio for such paths is a focus of ongoing work. Another issue that we plan to consider is the extension of the analysis techniques to concurrency.

7. ACKNOWLEDGEMENTS

We would like to thank Gerard Holzmann for sharing insights into the implementation of Uno. We would also like to thank Nils Klarlund for suggesting several improvements to the presentation. This work is supported in part by grant CCR-0341658 from the National Science Foundation.

8. REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV*, volume 2102 of *LNCS*, 2001.

[2] T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, 2001.

[3] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. icalp 2001: 652-666. In *ICALP*, volume 2076 of *LNCS*, 2001.

[4] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[5] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H.Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, 2001. `http://www.cis.ksu.edu/santos/bandera`.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[7] Coverity. `http://www.coverity.com`.

[8] D. Dams. Comparing abstraction refinement algorithms. In *SoftMC: Workshop on Software Model Checking*, 2003.

[9] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[10] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *CACM*, 18(8), 1975.

[11] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An excercise in cooperation. *C.ACM*, 21(11), 1978.

[12] E.A. Emerson and Kedar S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, 1998.

[13] A. Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89(2), 1990.

[14] C. Flanagan, K.Rustan M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[15] Flexelint. `http://www.gimpel.com`.

[16] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL*, 1997.

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, volume 2404 of *LNCS*, 2002.

[19] G.J. Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT 2002*, Pasadena, CA, USA, 2002. `http://www.cs.bell-labs.com/what/uno/index.html`.

[20] R.M. Karp and R.E. Miller. Parallel program schemata. *J.CSS*, 3(2), 1969.

[21] Klocwork. `http://www.klocwork.com`.

[22] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2001.

[23] J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *First GCC Developers Summit*, 2003. at `www.gcc.gnu.org`.

[24] G.C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, 2002.

[25] Polyspace. `http://www.polyspace.com`.

[26] Simplify. `http://research.compaq.com/SRC/esc/Simplify.html`.

[27] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[28] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICSE*, 2000. `http://ase.arc.nasa.gov/visser/jpf`.