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 analyses. Analysis methods that use simple dataflow 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.

Orion employs a two-level approach: potential errors are identified by an efficient search based on a simple domain; each discovered error path is then scrutinized by a high-precision feasibility analysis aimed at filtering out as many false errors as possible.

We describe the algorithms used 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 in tools like Flexelint [1], Coverity [2], Fortify [3] and Uno [4]. 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 DFA-based debugging aids are not routinely used. If this problem can be overcome, static error 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.

The standard answer of DFA to the false-alarm problem is to track more dataflow facts, or, in terms of Abstract Interpretation, to use more precise *abstract domains*. For example, by tracking known and inferred ranges for boolean and other variables, many false alarms can be avoided. However, the added precision may render the analysis forbiddingly expensive. A new generation of tools, including SLAM [5] and BANDERA [6], take an incremental approach in that they add additional precision only as needed, as identified by e.g. "Counter-Example Guided Abstraction Refinement" (CEGAR, [7]). The BLAST [8] tool

F.S. de Boer et al. (Eds.): FMCO 2005, LNCS 4111, pp. 138–160, 2006.

takes this even further by increasing the precision only for certain regions of the analyzed code. Still, these tools are limited in terms of the size of code that can be handled in a reasonable amount of time: usually in the order of 10-20KLoC (KLoC=1000 lines of source code), although there have been applications to larger code bases [9]. The reason is that these are software *verifiers*, targeted towards producing a yes/no answer for a particular query. This forces them to apply *sound* abstractions, and to make a substantial effort, in terms of using complex abstract domains, in order to prove correctness.

The aim of static error checkers, on the other hand, is to find errors with a high degree of accuracy within a reasonable amount of time and memory resources. Specifically, it is acceptable to sacrifice soundness, and miss a few errors, to the benefit of the signal-to-noise ratio and analysis time. We aim at a 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. Furthermore, analysis time should be in the same order of magnitude as build time.

1.1 Path-Oriented, Two-Level Analysis

Orion is a static error checker that achieves an excellent signal-to-noise ratio within a favorable analysis time. It reconciles these conflicting aims by using two precision levels for its analysis, as follows. It performs a DFA with *light-weight abstract domains* (level 1). Unlike traditional DFA, this is done with an automaton-based model checking algorithm. The advantage is that the search is *path-oriented*, meaning that it readily produces execution paths corresponding to potential errors. To bring down the high ratio of false error paths that would result, each of the potential-error paths found is then submitted to a separate, more precise analysis (level 2). Scrutinizing a single path acts as a *feasibility check*. If the path is infeasible, it is suppressed; otherwise it is reported as a potential error. The scope of this additional precision remains limited to the individual path, however, and does not get communicated back to any part of the level-1 analysis, as would be the case with the software verifiers mentioned above.

We note two consequences of this scheme. First, reasoning about a single ("straight-line") path is much easier than about code fragments that may contain branch/merge points and loops. As a result, we can afford to use very precise domains for level 2. Second, the overall two-level approach exploits the fact that we accept unsoundness of the analysis. This point will be discussed in Section 3.2.

Level 1: Path-Oriented, Light-Weight DFA. The level-1 data-flow analysis performs a depth-first search on the product of the control-flow graph and an observer automaton based on a light-weight abstract domain. This observer automaton, in turn, is represented by a product of automata, each representing a particular, usually simple, 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.

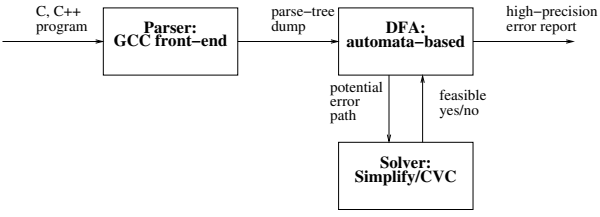

Fig. 1. Orion tool architecture

For example, in order to identify uses of uninitialized variables (use-before-def analysis), it needs to be known which variables have not been assigned a value along each path being explored. We sometimes refer to the automaton that tracks this information as the *error automaton*. In addition, by tracking a small amount of additional information, e.g. known and inferred ranges for boolean and other variables, many infeasible paths can be weeded out without reliance on level 2 checks. Such additional automata are called *information automata*. The automata, including the control-flow graph, typically depend on each other. For example, an error automaton that flags out-of-bounds array accesses depends on an information automaton that tracks variable ranges.

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 [10], to shortcut the search. In addition, we propose a novel idea called *differencing* to further optimize the algorithm.

Level 2: Tunable Feasibility Checking. The feasibility check that determines 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 one to plug in different provers so as to experiment with alternatives and profit from advances in the field.

1.2 Tool Architecture

There are three main parts to Orion: the parser, the data-flow analyzer (for level 1), and the solvers (for level 2), see Figure 1. 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 parse 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 [11].

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 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 [12] and CVC [13].

1.3 Paper Outline

Section 2 gives details of the model-checking based abstract state space exploration of level 1. In particular, it gives general formulations of the covering and differencing algorithms together with several theoretical results, and discusses their relation to standard data-flow algorithms. These algorithms are then specialized to the setting of automata-based analysis of control-flow graphs, and another optimization, *state aggregation*, is briefly discussed. Section 3 is concerned with the level 2 feasibility checking. It also explains how the interaction between depth-first search and feasibility may affect completeness of the approach. The experimental results on several programs are presented in Section 4. Section 5 discusses the contributions in the perspective of related work, and Section 6 concludes.

2 Path Exploration, Covering, and Differencing

The approach to DFA sketched above, namely an exploration of a graph in search for error states, is an instance of the view of *DFA as model checking of an abstract interpretation* [14]. In this view, the error automaton represents the (negation of the) property being checked, while the product of the flow graph and the information automata represents the abstract interpretation over which the property is checked. That is, each state of this abstract interpretation represents an overapproximation of the set of all run-time states that may be reached at the corresponding flow graph location. Traversing the overall product (of abstract interpretation and error automaton) state by state, in search for

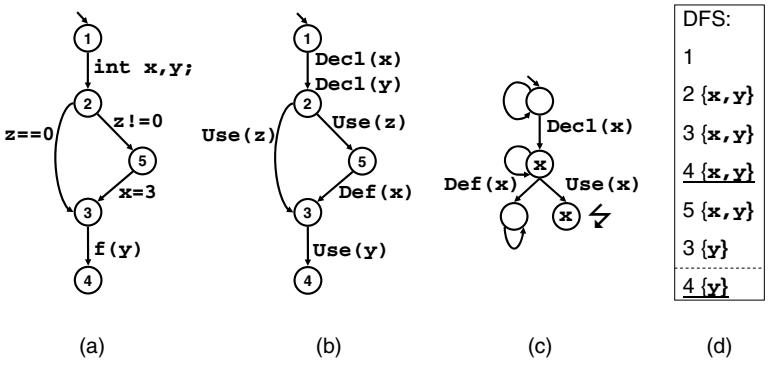

Fig. 2. Use-before-def checking

reachable error states (product states in which the error-automaton component is an error state), amounts to performing an explicit-state model check of a safety property.

While this view is conceptually appealing, applying it naively may lead to algorithms that are inferior to the traditional approach to DFA, in which a solution to a set of flow equations is computed iteratively. We illustrate this point by an example. Figure 2, part (a), shows a small C program as a control-flow graph whose edges are labeled with the program statements. Part (c) shows an error automaton that checks for use-before-def errors of the variable x (unmarked edges are taken when no other transitions are enabled). A similar automaton, not shown, is assumed that checks y. We think of these as a single error automaton defined by their product, whose states are thus subsets of $\{x, y\}$; e.g., the state $\{x\}$ represents all run-time states in which x has been declared but not yet defined. Thus, if a variable v is an element of some state automaton s, the automaton moves to an error state if a transition is taken that is labeled with Use(v). As its transitions depend on information about whether variables are declared, defined, and used in the program, we need the abstract interpretation of the program that is shown in part (b). The overall product graph to be explored has states (n, s) where n is the location in the graph of part (b) and s is the state of the product error automata.

Part (d) shows a sequence of states that is visited in a typical depth-first traversal; the error states are underlined. The point to note is that the use-before-def error of y (in the statement $f(y)$) is found twice along this sequence: once from state $(3, \{x, y\})$, where both x and y are tracked as being declared but not yet defined, and another time when only y is tracked, from state $(3, \{y\})$. This is an inefficiency: clearly, after having checked for use-before-def errors, "below" location 3, of any variable in $\{x, y\}$, when the search hits location 3 again with a *subset* of these variables, it could have backtracked safely, without missing any errors. In Figure 2 this is indicated by a dashed line, showing that state $(4, \{y\})$ does not need to be visited. In general, the savings from such early backtracking may be much more significant than in this example.

In an equation-solving approach this effect would not occur due to the fact that the first data-flow set to be associated with location 3 is $\{x, y\}$; when the set $\{y\}$ is then merged with it, it does not change ($\{x, y\} \cup \{y\} = \{x, y\}$), so no further propagation is needed [15]. In this section, we fix this shortcoming of the model-checking based approach to DFA by combining it with a notion of *covering*. In the example above, the search can backtrack from state $(3, \{y\})$ because, intuitively, it is "covered" by the already-visited state $(3, \{x, y\})$. While the addition of covering restores the efficiency of the algorithm in comparison to the equation-solving approach, we present a second optimization, called *differencing*, that may result in an additional performance improvement, and which has not been proposed in the context of the equation-solving approach, to the best of our knowledge. Both notions are formalized as extensions of a highly non-deterministic White-Grey-Black coloring search [16]. Depth-first search, as well as other strategies, can be obtained by restricting the non-determinism in this algorithm.

2.1 General Search with Covering and Differencing

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 Edsger W. 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 the "frontier" of partially explored states, 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 1. *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.

Formally, a *subset-covering* relation (usually a pre-order), $\sqsupseteq \subseteq 2^S \times 2^S$, is required to have the following property: for all subsets X, Y of S, if $X \sqsupseteq Y$ (read as X *covers* Y), then: for every $k > 0$, and every 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
$(\Box s : I(s) \ \wedge \ white(s) \longrightarrow color[s] \ := \ G)$
$\Box$ $(\Box s, t : grey(s) \ \wedge \ R(s,t) \ \wedge \ white(t) \longrightarrow color[t] \ := \ G)$
$\Box$ $(\Box s : grey(s) \ \wedge \ (\forall t : R(s,t) \ \Rightarrow \ \neg white(t)) \longrightarrow color[s] \ := \ B)$
$\Box$ $(\Box s : grey(s) \ \wedge \ ((Grey \setminus \{s\}) \ \cup \ Black) \sqsupseteq \{s\} \longrightarrow color[s] \ := \ R)$

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 2. *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 s is covered, intuitively, because *all* error paths from s have matching paths from the set of states that covers $\{s\}$. It may, however, be the case that for some states s' that remain uncovered, *most*, but not all, of the error paths from s' 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 s' and that of the already explored states. One may then choose to stop exploring s', 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; note that $T \sqsupseteq \emptyset$ is true for all T.)

$$
\begin{array}{l}
([]s, \mathit{Diff} : \mathit{grey}(s) \ \wedge \ (\mathit{Diff} \subseteq \mathit{White}) \wedge \\
\quad ((\mathit{Grey} \setminus \{s\}) \ \cup \ \mathit{Black} \ \cup \ \mathit{Diff}) \sqsupseteq \{s\} \ \wedge \ \{s\} \sqsupseteq \mathit{Diff} \\
\quad \longrightarrow \mathit{color}[s] \ := \ R; (\textbf{foreach} \ t : t \in \mathit{Diff} : \mathit{color}[t] \ := \ G))
\end{array}
$$

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 includes only white 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 3. *Algorithm III terminates, and there is a reachable error state iff one of the states in* $(\mathit{Black} \cup \mathit{Red})$ *is an error state.*

2.2 Application to Control-Flow Graphs

In Orion, the differencing-based algorithm is applied in the level-1 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 *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 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. This definition implies that the language of the combined automaton is the union of the (rejecting) languages of the components. While the automata can be run individually on the CFG, combining them can sometimes saves analysis time, while requiring more space. Moreover, the automata are sometimes combined in more complex ways, where the state of one automaton is used as an input to the other (this is referred to as the *cascade* or *wreath* product). For instance, array bounds checking is carried out by a combination of two automata: one that tracks upper and lower bounds on the values of program variables (this information is also used to filter out infeasible paths), the other simply applies the computed bounds to each array indexing operation.

The model checking problem is to determine whether there is a path in this product to a rejecting state (recall that rejecting automaton states signal program errors). Note that we focus on safety properties, which can be analyzed by checking reachability. 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 results in a use-before-def 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 (note that the same sequence is possible because $n = n'$). Similarly, $B \sqsupseteq_2 B'$ iff the bounds information in B' denotes a subset of the run-time states denoted by the information in B. Any error path from B' can be enabled from B, as the information in B is more approximate than that in B'. E.g., 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: $\mathit{diff}((n, s_1, \ldots, s_K), (n', s'_1, \ldots, s'_K))$ is defined only if $n = n'$ (i.e., same control location), and is given by the set $\{(n, s) \mid (\exists i, d : d \in \mathit{diff}_i(s_i, s'_i) \ \wedge \ s = (s_1, \ldots, s_{i-1}, d, s_{i+1}, \ldots, s_K))\}$. 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. An approximate merge may lead to missed errors due to early covering, this issue is discussed further in Section 3.2.

3 Feasibility Checking

As explained in the introduction, a potential finite error path detected at level 1 is subjected to further analysis by a feasibility check, which determines whether it can correspond to a real execution. By integrating bounds information in the abstract state at level 1 some error paths can already be ruled out at an early stage. The level-2 feasibility check described here is applied to other error paths that are generated at level 1. In this section we describe the manner in which this check is performed, optimizations, and some consequences for the completeness of the error detection strategy.

3.1 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 [17].

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 $wlp(S, \phi) = \{s \,|\, (\forall t : s \xrightarrow{S} t \ \Rightarrow \ \phi(t))\}$. The

weakest precondition 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 there exists a state s such that s is not in $wlp(\pi, false)$; i.e., that $wlp(\pi, false)$ is not a validity.

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.

$$\begin{aligned} &\mathit{if}\ pt(p,x)\ \mathit{then}\ (\mathit{if}\ pt(p,y)\ \mathit{then}\ \ \phi(e,e)\ \mathit{else}\ \phi(e,y)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ (\mathit{if}\ pt(p,y)\ \mathit{then}\ \ \phi(x,e)\ \mathit{else}\ \phi(x,y)) \end{aligned}$$

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.

Orion sends the expressions that represent *wlp*'s of paths to a validity checker — we currently use either Simplify [12] or CVC [13] as the checkers. An interesting observation is that the use of *wlp* automatically provides a *slicing* [18] 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 3. 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.

```
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 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
```

Fig. 3. With feasibility checking, the error path (on the right) is suppressed

3.2 Implications for Completeness

Orion is focussed on finding errors. Two important correctness (i.e., non-performance) aspects in this context[1] 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

[1] Note that the notions of soundness and completeness are defined opposite from those in the context of program *verification*.

limits the time allowed for the solvers, inevitably permitting in some false error paths to be reported. Orion thus aims to achieve a high degree of – but not perfect – precision in its error reports.

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. Conversely, an algorithm can achieve perfect soundness by not reporting any errors but, of course, at the cost of completeness. 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.

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: $P_1 :: (x > 0); v = x; (x < 0);$ *return* u, and $P_2 :: (x \leq 0); v = x + 1; (x < 0);$ *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 upper and lower bounds on variable values, which can 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: [19] 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.

4 Experimental Evaluation

Covering and Differencing. Experiments with the covering and differencing algorithms show a clear advantage in execution times relative to pure depth-first search: at least 3-fold, usually more. The intuition behind the differencing method is that it can speed up the search by: (i) potentially 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. The benefit in run times is marginal, with a maximum speedup of about 10%. We continue to explore this issue, however, and to look for more efficient differencing implementations.

Feasibility. The table in Figure 4 summarizes the result of running Orion on various publicly available software packages. The experiments were run on an AMD Opteron 2.6GHz dual core, 8GB, under Linux, except for the first (emacs-21.3), which had to be run on a considerably slower machine due to OS incompatibility issues. In any case, the measurements are not directly comparable over the different entries, as they were collected in multi-user mode. The errors being checked are use-before-def of variables variables, null-pointer dereferencing, and out-of-bounds indexing of arrays. These checks are made for individual functions in the program, making no assumptions about the context in which a function is called. The first three columns in the table show the name and version of the program analyzed, the number (in 1000s) of lines of code that were analyzed (this need not be the total amount of C code in the package, due to configuration options),

Source	KLoC	compile time (s)	compile + analyze time (s)	errs.	infeas.	real errs.
`emacs-21.3`	25.9	199	412	3	5	3
`jpeg-6b`	28.8	7	35	0	25	0
`libxslt-1.1.12`	31.2	27	103	2	1	0
`sendmail-8.11.6`	76.2	9	82	5	16	2
`libxml-2.6.16`	200.1	97	295	3	12	2

Fig. 4. Results on some open source packages

and the compilation time, in seconds, when Orion is not used. The next three columns give the results of compilation with the Orion checks enabled. The column "errs." lists the number of errors as reported by Orion, i.e., the number of error paths (as identified at level 1) that are determined to be *feasible* by Orion's feasibility check (at level 2). The column "infeas." lists the number of paths that show up as potential errors at level 1, but that are determined to be infeasible at level 2, and are thus not reported. The last column lists the number of real errors, among those reported by Orion, as determined by a manual inspection, see below. The analyses are run with precision 2, meaning that the time allotted for every feasibility check is 2 seconds. Increasing this time-out value does not show a significant decrease in the number of errors reported for these examples, while with precision 1 and lower, Orion does not invoke external validity checkers for feasibility checking, which leads to significantly more false alarms. For sendmail, jpeg, and emacs, a few additional customized options were given to indicate that certain functions should be considered as "exit functions" that do not return. This helped suppress a couple of false alarms.

The numbers of errors that are reported versus infeasible paths that are suppressed witness the effectiveness of the feasibility checking. An indication of the signal-to-noise ratio that is achieved may be derived by comparing the number of reported errors (column "errs.") to the number of those that are deemed real (column "real errs."). For the manual checks, we considered an error to be real if it was feasible locally within a procedure. It may be that some of these "real" errors are infeasible when considered in their interprocedural context. Still, arguably such cases indicate a lack of defensive coding (in no case did we find comments or assertions in the code that indicated the assumptions made).

5 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 [20] insert checks into the code to this effect, that get compiled and executed along with the program. VeriSoft [21], DART [22], and JPF [23] 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.

Static Analyzers. Tools that approach Orion most closely in terms of usage, purpose, and underlying techniques are Uno [4], MC/Coverity [24], PolySpace [25], klocwork [26], ESP [27], FlexeLint [1], and BEAM [28]. The distinguishing features of Orion are:

- Its 2-level, tunable 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 at the first, more superficial level. The experimental results reported show that this reduces the amount of false alarms in a significant way, without excessively burdening the analysis time.
- The application of the covering and differencing optimizations uniformly for all abstract domains. 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. The current implementation of Orion uses a covering relation over a combined domain for tracking uninitialized variables and bounds information, as explained in Section 2.2. We are considering adding points-to information as well; such extensions are easy to add thanks to the generality of the implementation.

The roots of the covering algorithm go back at least to work by Karp and Miller [10], but it has been reformulated several times in connection with various abstract domains (cf. [29,30,8]).

While the high degree of automation offered by all 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 [31] and LClint [32] 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.

Several alternative approaches exist to ameliorate the signal-to-noise ratio. One is to *rank* the errors reported, based on heuristic rules and history information, 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 [33], and also in MC/Coverity [24]. 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. Orion's distinguishing features are orthogonal to these techniques, and can be combined to get the best of worlds.

Software Verification Tools. The introduction already discussed the relation of Orion to software verification tools based on symbolic processing, such as SLAM [5], BLAST [8], and BANDERA [6], stressing the difference between *error checking* and *verification*. Orion 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 error analysis to acceptable levels. Our experimental results appear to bear out the hypothesis that the 2-level analysis procedure discussed in this paper is effective at performing high-precision static error analysis. Some of the technical details of Orion's implementation 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.

6 Conclusions and Ongoing Work

We have presented the static error checker Orion, which is aimed at producing error reports with a low false-alarm rate in reasonable analysis time. The approach that enables this is an automaton-based, path-oriented, two-level data-flow analysis, that uses powerful external solvers in a tunable fashion, and is optimized by the use of covering, differencing, and state aggregation schemes.

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.

Experiments on several programs shows that in most cases the targeted signal-to-noise ratio of 3:1 is achieved. A detailed inspection of the reasons that some infeasible paths are still reported as errors suggest two priorities for further work.

First, it turns out that the computation of expressions that represent weakest preconditions, tends to run out of the allocated resources in cases where the paths are very long (hundreds of statements). Orion can perform an interprocedural analysis for uses of uninitialized global pointer variables[2], and since interprocedural paths tend to be very long, no feasibility checks are performed in this case. We are currently investigating alternative solvers such as CVC-Lite

[2] The implementation is based on standard techniques for model checking of recursive state machines [34,35], and thus can deal with recursive functions in C and C++.

[36], and also looking into other approaches to feasibility checking such as the use of a SAT-based symbolic model checker (CBMC, see [37]) and of a testing-based tool (DART, see [22]).

Another source of false alarms is the out-of-bounds array check. We are currently working on an improved and generalized buffer-overflow checking module for Orion.

It has been pointed out that Orion is not a complete method for error detection. The interaction between the level-1 and level-2 checks may cause errors to be missed as explained in Section 3. Furthermore, when encountering certain language features such as long-jumps and function pointers, Orion favors analysis speed over completeness. The fact that it still finds a significant amount of errors in code that may be considered well-tested, confirms that such sacrifices are justified. Nevertheless, it is our intention to address the various sources of incompleteness by offering options to run Orion in a stricter mode, or at least to warn of the occurrence of language constructs that are not treated conservatively. In comparison, note that a static analyzer like Astrée [38], which is aimed at proving the *absence* of certain types of errors, comes with rather drastic restrictions on the allowed language constructs in order to guarantee its claims.

While the analysis times reported in our experiments are reasonable, in some cases they are an order of magnitude more than the time required to compile a program without error analysis by Orion. In addition, when software is analyzed that less well-tested, the number of errors can be significantly higher, leading to an increased analysis time due to more numerous feasibility checks. In order to address this, we have implemented an *incremental* algorithm in Orion; the results are reported elsewhere, see [39].

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

References

1. (FlexeLint) `http://www.gimpel.com`.
2. (Coverity) `http://www.coverity.com`.
3. (Fortify) `http://www.fortifysoftware.com/products/sca.jsp`.
4. Holzmann, G.: 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`.
5. Ball, T., Rajamani, S.: The SLAM toolkit. In: CAV. Volume 2102 of LNCS. (2001)
6. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., H.Zheng: Bandera: extracting finite-state models from Java source code. In: ICSE. (2001) `http://www.cis.ksu.edu/santos/bandera`.
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (2003) 752–794

8. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: CAV. Volume 2404 of LNCS. (2002)
9. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. (2004) 232–244
10. Karp, R., Miller, R.: Parallel program schemata. J.CSS **3**(2) (1969)
11. Merrill, J.: GENERIC and GIMPLE: A new tree representation for entire functions. In: First GCC Developers Summit. (2003) at `www.gcc.gnu.org`.
12. (Simplify) `http://research.compaq.com/SRC/esc/Simplify.html`.
13. Stump, A., Barrett, C., Dill, D.: CVC: a Cooperating Validity Checker. In: 14th International Conference on Computer-Aided Verification. (2002)
14. Schmidt, D., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: SAS. Volume 1503 of LNCS., Springer Verlag (1998)
15. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison–Wesley (1987)
16. Dijkstra, E., Lamport, L., Martin, A., Scholten, C., Steffens, E.: On-the-fly garbage collection: An excercise in cooperation. C.ACM **21**(11) (1978)
17. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. CACM **18**(8) (1975)
18. Tip, F.: A survey of program slicing techniques. Journal of programming languages **3** (1995) 121–189
19. Dams, D.: Comparing abstraction refinement algorithms. In: SoftMC: Workshop on Software Model Checking. (2003)
20. Necula, G., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: POPL. (2002)
21. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL. (1997)
22. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proc. of the ACM SIGPLAN. (2005)
23. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: ICSE. (2000) `http://ase.arc.nasa.gov/visser/jpf`.
24. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. In: PLDI. (2002)
25. (PolySpace) `http://www.polyspace.com`.
26. (Klocwork) `http://www.klocwork.com`.
27. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI. (2002)
28. Brand, D.: A software falsifier. In: International symposium on Software Reliability Engineering. (2000) 174–185
29. Finkel, A.: Reduction and covering of infinite reachability trees. Information and Computation **89**(2) (1990)
30. Emerson, E., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS. (1998)
31. Flanagan, C., Leino, K.M., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: PLDI. (2002)
32. Larochelle, D., Evans, D.: Statically detecting likely buffer overflow vulnerabilities. In: USENIX Security Symposium. (2001)
33. Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Software: Practice and Experience **30**(7) (2000) 775–802
34. Benedikt, M., Godefroid, P., Reps, T.: Model checking of unrestricted hierarchical state machines. icalp 2001: 652-666. In: ICALP. Volume 2076 of LNCS. (2001)

35. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: CAV. Volume 2102 of LNCS. (2001)
36. (CVC Lite) `http://chicory.stanford.edu/CVCL/`.
37. (CBMC) `http://www.cs.cmu.edu/~modelcheck/cbmc/`.
38. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In Sagiv, M., ed.: Proceedings of the European Symposium on Programming (ESOP'05). Volume 3444 of Lecture Notes in Computer Science., Edinburgh, Scotland, © Springer (2005) 21–30
39. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In Etessami, K., Rajamani, S.K., eds.: Computer Aided Verification. Number 3576 in LNCS (2005) 449–461

Appendix

We present detailed proofs of theorems in this section.

Algorithm I: White-Grey-Black Search

The algorithm is reproduced below.

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

Theorem 1. *Algorithm I terminates with Black = Reach.*

Proof. 1. Termination: to show termination, consider the progress measure $\rho = (|White|, |Grey|)$ under lexicographic ordering. As the graph is finite, this measure is finite, and the lexicographic order is well-founded. Notice that the first two actions, if executed, strictly decrease $|White|$, while the third action, if executed, keeps $|White|$ constant, but strictly decreases $|Grey|$. Thus, ρ decreases strictly for every executed action; hence, the program terminates.

2. Correctness: we show some auxiliary invariants first.

2a. $(Grey \ \cup \ Black) \subseteq Reach$ is invariant. (proof) this is true initially as the sets on the left are empty. The first action turns a white initial state grey, thus preserving the invariant. The second action turns a white successor of a grey state (reachable, by induction) grey, thus preserving the invariant. The third action changes the color of a grey state to black, thus keeping the union constant. (endproof)

2b. $post(Black) \subseteq (Grey \ \cup \ Black)$ is invariant. (proof) this is true initially as *Black* is empty. The first two actions increase only *Grey*, thus they preserve the invariant. The last action moves a state from grey to black, but the newly blackened state satisfies this condition. (endproof)

At termination, all actions are disabled. Thus: (i) from the disabling of the first action, all initial states are non-white, so $I \subseteq (Grey \cup Black)$; (ii) from the disabling of the second action, all grey states have non-white successors, so $post(Grey) \subseteq (Grey \cup Black)$; (iii) from the disabling of the third action, all grey states have at least one white successor. The second and third consequences together imply that $Grey = \emptyset$. Hence, from (i), $I \subseteq Black$, and from (2b), $post(Black) \subseteq Black$. Thus, $Black$ is a solution to the fixpoint equation for the set of reachable states. Since $Reach$ is the least solution, and from (2a) we have that $Black \subseteq Reach$, we get that $Black = Reach$.

Algorithm II: Covering-Based Search

The algorithm is given below.

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

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

Proof. 1. Termination: We use the same progress measure, $\rho = (|White|, |Grey|)$, as before. This decreases strictly for the first three actions, which are identical to those of the previous algorithm. The fourth action keeps $|White|$ unchanged while decreasing $|Grey|$.

2. Correctness:

2a. $(Grey \cup Black \cup Red) \subseteq Reach$ is invariant. (proof) this is true initially as the sets on the left are empty. The first action turns a white initial state grey, thus preserving the invariant. The second action turns a white successor of a grey state (reachable, by induction) grey, thus preserving the invariant. The third action changes the color of a grey state to black, while the last one changes the color of a grey state to red, thus keeping the union constant. (endproof)

2b. $post(Black) \subseteq (Grey \cup Black \cup Red)$ is invariant. (proof) this is true initially as $Black$ is empty. The first two actions increase only $Grey$, thus they preserve the invariant. The third action moves a state from grey to black, but the newly blackened state satisfies this condition. The fourth action only moves a state from grey to red. (endproof)

At termination, all actions are disabled. Thus: (i) from the disabling of the first action, all initial states are non-white, so $I \subseteq (Grey \cup Black \cup Red)$; (ii) from the disabling of the second action, all grey states have non-white successors, so $post(Grey) \subseteq (Grey \cup Black \cup Red)$; (iii) from the disabling of the third action, all grey states have at least one white successor. The second and third

consequences together imply that $Grey = \emptyset$. Hence, from (i), $I \subseteq (Black \cup Red)$, and from (2b), $post(Black) \subseteq (Black \cup Red)$.

However, we cannot claim, as in the previous proof, that $Black \cup Red$ satisfies the fixpoint equation. Indeed, we hope it does not, since this would mean that $Black \cup Red = Reach$, and we want this set of states to be a strict subset of the reachable states. At termination, red states can have successors that are any color except grey, as there are no grey states left. What we would like to claim is that $Reach \cap E \neq \emptyset$ if, and only if, $(Black \cup Red) \cap E \neq \emptyset$. To do so, we prove the stronger invariant below.

3. Invariantly, $Reach \cap E \neq \emptyset$ iff $(Grey \cup Black \cup Red \cup (I \cap White)) \xrightarrow{*} E$.

(proof) [right-to-left] by contrapositive. Suppose that there is no reachable error state. By (2a), it is not possible to reach an error state from $(Grey \cup Black \cup Red \cup (I \cap White))$, which is a subset of the reachable states.

[left-to-right] suppose that there is a reachable error state. We have to show that the right-hand expression is an invariant. The property is true initially as all initial states are white. The first transition only moves a state from $(I \cap White)$ to $Grey$, so error reachability is invariant. The second adds a $Grey$ state, hence reachability to an error state is preserved. The third moves a state from $Grey$ to $Black$, while the last moves a state from $Grey$ to Red, so again, reachability to error states is preserved. (endproof)

At termination, this invariant implies, as $Grey$ is empty, and all initial states are non-white (condition (i) above), that $Reach \cap E \neq \emptyset$ iff $(Black \cup Red) \xrightarrow{*} E$. This is not enough to imply the equivalence of $Reach \cap E \neq \emptyset$ and $(Black \cup Red) \cap E \neq \emptyset$: though the successors of a black state are colored only red or black, the error may be in a white successor of a red state, which remains unexplored on termination. We use the path-length constraint on the covering relation to show that this situation can occur only if a red/black state is itself an error state.

4. Invariantly, for any $k > 0$, if $Red \xrightarrow{=k} E$, then $(Grey \cup Black) \xrightarrow{\leq k} E$.

(proof) This is true initially as there are no red states, so the antecedent is false for all k. Assuming the claim to be true, we show that it is preserved by every transition. The first three transitions can only increase the set $(Grey \cup Black)$, without affecting red states, thus the property, being monotonic in $(Grey \cup Black)$, continues to hold. The last transition moves a state s from grey to red. By the path-length constraint on the covering relation, the implication holds for the newly red state s after the transition. Consider any other red state t and any k for which there is a path to error of length k. By the induction hypothesis, before the transition, there is a path from a grey/black state, t', of length k', where $k' \leq k$, to a state in E. If this path cannot be used as a witness after the transition, it must be because t' turns red after the transition; hence, $t' = s$. But then, by the covering property for s, there must be a path of length at most k' from one of the grey/black states after the transition to a state in E. (endproof)

Now we argue that $(Black \cup Red) \xrightarrow{*} E$ holds iff $(Black \cup Red) \cap E \neq \emptyset$. The direction from right to left is trivial. For the other direction, let k be the length of the shortest path to an E-node from $(Black \cup Red)$. If $k = 0$, we are

done. If $k > 0$, the start state of the path must be in *Red*, otherwise there is a shorter path by (2b) and the assumption that $Grey \neq \emptyset$. However, in that case, by (4) and the assumption that $Grey \neq \emptyset$, there is a path of length at most k from a black node to E; hence, again, there is a shorter path by (2b). Thus, k must be 0.

Algorithm III: Adding Differencing Mechanisms

The algorithm is identical to that presented before, but for a modified final action.

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

Theorem 3. *Algorithm III terminates, and there is a reachable error state iff one of the states in* $(Black \ \cup \ Red)$ *is an error state.*

Proof. Surprisingly, the proof of correctness is essentially identical to the one for the covering-only search.

1. Termination: this holds with reasoning identical to that in the previous proof.

2. Correctness: The only difference is that since Diff can include unreachable states, so (2a) no longer holds, so we have to adjust the proof of the right-to-left direction of (3). We give the new proof below; the rest of the argument is identical.

3. Invariantly, $Reach \cap E \neq \emptyset$ iff $(Grey \cup Black \cup Red \cup (I \cap White)) \stackrel{*}{\longrightarrow} E$.

[right-to-left proof] Suppose that there is no reachable E-state. We have to show that the r.h.s. is invariantly false. The r.h.s. is false initially as it reduces to $I \stackrel{*}{\longrightarrow} E$, which is false by assumption.

Suppose that it is false, we show that no action can make it true. The first action only colors a white initial state grey, so the r.h.s. stays false. Since there is no path to error from grey states, marking white successors of grey states as grey (the second action) cannot introduce a path to error. Similarly, the third action only colors a grey state black, so it does not make the r.h.s. true.

The fourth action, however, turns a grey state red *and* adds a set of — perhaps unreachable — states, *Diff*, to *Grey*. So the only way in which the property can be true after the transition is if $Diff \stackrel{*}{\longrightarrow} E$ holds. But the constraint $\{s\} \sqsupseteq Diff$ in the guard, together with the path-matching constraint on the covering relation, implies that $s \stackrel{*}{\longrightarrow} E$ is true. But this is known to be false, as s is a grey state before the transition. Hence, the property remains false after the transition.