The Path-wise Approach to Data Flow Testing with Pointer Variables

Delia I.S. Marx
Mathematics and Computer Information Science
Mercy College
555 Broadway
Dobbs Ferry, New York 10522
e-mail: marx@west.poly.edu

Phyllis G. Frankl *
Computer and Information Science
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11201
e-mail: phyllis@photon.poly.edu

Abstract

This paper describes a new approach to performing data flow testing on programs that use pointer variables and a tool based on this approach. Our technique is based on the observation that, under certain reasonable assumptions, we can determine which dereferenced pointers are aliased whenever control reaches a given program point *via a particular path*. Furthermore, we can group together paths which behave similarly and represent them by regular expressions. The resulting test requirements demand that the test data execute representatives of particular sets of paths between variable definitions and uses.

1 Introduction

Many researchers view data flow testing techniques as promising approaches, which may offer better fault-detecting ability than techniques based purely on control flow analysis, at fairly reasonable cost. It is reasonable to conjecture that data flow testing tools based on more accurate data flow analysis will be better at detecting faults than those based on cruder analyses. In particular, more careful analysis of pointer variables could enhance effectiveness, especially in those cases where the fault involves misuse of pointer variables. Recognizing this, Ostrand and Weyuker developed an extension to the theory of data flow testing and a tool based on more careful analysis of pointers than that provided by previous tools [11]. In this paper we propose an alternative approach to handling pointers and describe a tool, `APODOS`, based on this new approach. We call our approach a *path-wise* approach because, in contrast to previous *point-wise* approaches, it explicitly distinguishes among different paths that pass through a given pair of points in the program text.

Data flow testing is based on data flow analysis, similar to that performed by optimizing compilers. Loosely speaking, the intuition motivating data flow testing is that if there is a way for the value assigned to a variable at one program point to be used at another program point, the test data should execute some path on which that happens. Occurrences of variables in a program are classified as being *definitions* in which a value is assigned to a variable and *uses* in which a value is fetched from a variable. A definition-clear path with respect to $\mathtt{v}$ from statement d to statement u is a path from d to u along which $\mathtt{v}$ is not redefined. A definition of variable $\mathtt{v}$ in statement d *reaches* a use of variable $\mathtt{v}$ in statement u if there is a definition-clear path with respect to $\mathtt{v}$ from d to u. A *definition-use association (*`dua`*)* is a triple $(d, u, \mathtt{v})$ where d is a statement in which variable v is defined, u is a statement or predicate in which variable v is used, and the definition in d reaches the use in u. If $(d, u, \mathtt{v})$ is a definition-use association then control may reach u with variable $\mathtt{v}$ having the value which was assigned to it in d. The *all-uses* data flow testing criterion [14] demands that each such association $(d, u, \mathtt{v})$ be exercised during testing, by execution of a test case which causes the execution of a path containing a definition-clear subpath from d to u with respect to $\mathtt{v}$. Other data flow testing criteria are defined in [4, 5, 6, 9, 10, 15].

There are well-known algorithms for finding the definition-use associations in a program, provided that there is no aliasing [1]. Aliasing occurs when two program variables refer to the same memory location. One common way that aliasing occurs is through the use of pointer variables. Let ρ and σ be pointer variables and let $\rho\uparrow$ and $\sigma\uparrow$ be corresponding *dereferenced pointers*,

*Supported in part by NSF grant CCR-9206910.

i.e., the memory cells to which they point. If at some point during execution, $\rho = \sigma$ then $\rho\uparrow$ and $\sigma\uparrow$ are aliased at that point, and any definition or use of $\rho\uparrow$ is also a definition or use of $\sigma\uparrow$. Furthermore, the same variable $\rho\uparrow$ may refer to a different location at another point in the program's execution. This considerably complicates data flow analysis. For example, suppose there is an assignment to $\rho\uparrow$ in statement i, an assignment to $\sigma\uparrow$ in statement j, an occurrence of $\rho\uparrow$ on the right-hand side of an assignment at statement k, and no other occurrences of $\rho\uparrow$ on any path from i to k. The definition of $\rho\uparrow$ in i *may* reach the use of $\rho\uparrow$ in k. However, several phenomena can interfere with this.

- If there is an assignment to ρ between i and k, $\rho\uparrow$ may reference different cells at these points. For example, the definition and use of `q`$\uparrow$ in nodes 2 and 7 in the example in Figure 1 refer to different locations *if* the path from 2 to 7 passes through node 5.
- If control passes through j between i and k and the value of σ at j is the same as the value of ρ at i then the path from i to k is not definition clear with respect to $\rho\uparrow$. For example, the assignment to `q`$\uparrow$ in node 7 will kill the definition in `p`$\uparrow$ in node 6 *if* control passed through node 5 before reaching node 6.

In addition, a use of $\sigma\uparrow$ at k may or may not be reached by the definition of $\rho\uparrow$ at i. For example, the definition of `q`$\uparrow$ in node 7 reaches the use of `p`$\uparrow$ in node 8, if control passed through node 5 before reaching node 7.

Previous data flow testing tools have used data flow analysis algorithms that were originally developed for optimizing compilers. These are *point-wise* algorithms, meaning that they focus attention on the relationship between a pair of points in the program text, without regard to the particular path taken between those points. In the context of compilers this point-wise analysis is a necessity: one cannot make a program transformation that only preserves program semantics in the case that certain particular paths are followed. On the other hand, for the purposes of testing it is perfectly legitimate to take a *path-wise* approach which distinguishes between different kinds of paths between two program points. In this approach one requires the test data to execute a path which goes from one program point to another, following a representative of a particular set of paths. As we shall see below, distinguishing between different kinds of paths alleviates many of the problems caused by pointer variables and other forms of aliasing and also provides other useful information.

Section 2 of this paper explains the main ideas of the path-wise approach and compares it to the point-wise approach. Section 3 defines *extended definition-use associations (*`eduas`*)*, the test requirements arising in this approach, introduces a program representation called the *Alias Automaton*, shows how to express the `eduas` in terms of the Alias Automaton, shows how to use the Alias Automaton to check which `eduas` are covered by the test data, and gives an algorithm for building the Alias Automaton. Section 4 describes the design and implementation of `APODOS`, a tool based on this approach. Section 5 focuses on how the tool implements inter-procedural analysis. Section 6 discusses future work.

2 Path-wise analysis

In this section we briefly review the point-wise approach to data flow testing, introduce the path-wise approach and compare the two. We begin by considering the two approaches when dereferenced pointers are not of interest, then consider analysis of dereferenced pointers.

A *program point* is a point in the program text, or, equivalently, in the program's flow graph. The points of interest for data flow analysis are the tops and bottoms of nodes in the graph, where, depending on the details of the particular problem, nodes represent statements or basic blocks. Pointwise data flow analysis algorithms *implicitly* consider all paths between a given pair of program points and report the existence or non-existence of a path satisfying certain properties of interest. For example, a report that a definition of `v` at i reaches a use at j is a report that some path from i to j is definition-clear with respect to `v`; the output does not include any information about *which* of the paths from i to j satisfy the property. Efficient point-wise algorithms for many data flow analysis problems have been developed for use in optimizing compilers and have subsequently been used in testing tools. The efficiency of these algorithms is due in part to the fact that they do not explicitly examine the individual paths between the points in question. However, this efficiency comes at a price: as we shall see below, certain information that is useful for testing is sacrificed.

Several data flow testing tools based on the all-uses criterion and its variants have been developed using the point-wise approach. The first data flow testing tool developed was ASSET [3, 2] which accepts programs written in Pascal. Although it accepts programs containing pointer variables, it makes no attempt to identify definition-use associations involving the dereferenced pointers accurately. ASSET's front end uses a point-wise data flow analysis technique to derive a list of definition use associations (`duas`) and produces a modified subject program which has been augmented with probes that outputs a trace of the path executed by each test case. The back end checks which of the definition-use associations have been covered by the test data by searching the traces for patterns corresponding to the definition-use associations. The ATAC system takes a similar approach to data flow testing for C programs [7]. Harrold and Soffa developed a system that incorporates inter-procedural data flow analysis including treatment of aliasing arising from call-by-reference parameters [5]. TACTIC [11], discussed further below, includes analysis of dereferenced pointers.

In contrast to the point-wise approach, the *path-wise* approach is based on an explicit representation of sets of paths through the program. Suppose there is a definition of `v` at program point i and a use of `v` at j. We can derive a regular expression that explicitly describes the set of paths that have a definition-clear subpath with respect to `v` from i to j. As we shall see below, the set of paths passing through i and j can be further subdivided so that each subset contains paths which have similar behavior with respect to aliasing. In a testing tool based on the path-wise approach, the testing requirements are expressed as sets of paths, represented by regular expressions (or equivalently, by finite state automata), and the program traces are matched against them. Even if one is not concerned with analyzing dereferenced pointers, the path-wise approach offers certain benefits:

- It allows the flexibility to easily define different kinds of test requirements, (e.g., chains of `dua`s, `dua`s with additional requirements, etc.).
- In the case where the test data does not cover all of the test requirements, it provides information needed in order for the user to derive test data to cover remaining requirements. If the person using the tool is only told which definition-use associations have not been covered, she must derive the additional information about the paths in order to generate test cases that will satisfy the requirements (or to determine that the requirements are infeasible) [2].

Ostrand and Weyuker [11] address the problems caused by pointers by distinguishing between possible and definite definitions, possible and definite uses, and possible and definite definition-clear paths. For example, $\rho\uparrow$ has a possible definition at a program point i if there is a definition of $\sigma\uparrow$ at i, and it is possible for ρ and σ to have the same value at i. Definition-use associations are classified as being either strong, firm, weak, or very weak, depending on whether the definitions, uses, and definition-clear paths involved are definite or possible. They recommend that testers begin by requiring that all the strong definition-use associations be exercised, then move on to the firm ones, and so on, as time permits.

The TACTIC system [11] uses an inter-procedural point-wise algorithm similar to those in [8, 12] to identify the various kinds of def-use associations. Test cases are executed in an environment which records memory addresses accessed as well as the sequence of statements executed. To check whether a firm, weak, or very weak `dua` involving a definition of $\rho\uparrow$ at i and a use of $\sigma\uparrow$ at j was covered, TACTIC analyses these richer traces, looking for a path from program point i to program point j such that memory cell m was defined at i and used at j and was not redefined between i and j.

In contrast to Ostrand and Weyuker's point-wise approach, we address the aliasing problem through a path-wise approach. Note that in the example in Figure 1, although we don't know whether an arbitrary path connecting 6 to 7 will cover the corresponding definition-use association, we do know for any test case which follows a path from 5 to 6 to 7, the same memory location that is updated at statement 6 will be accessed at statement 7. More generally, under certain restrictions on the way pointer variables are used, we can associate with each definition-use association a set of paths describing the conditions under which that definition-use association is exercised. We will refer to a `dua` along with such a set of paths as an *extended definition-use association (*`edua`*)*. A more precise definition is given in section 3.

Although the path-wise approach is potentially more expensive than the point-wise approach, the more detailed information that it supplies is useful.

- In the path-wise approach it is not necessary to trace memory addresses. Thus less overhead is incurred in executing test cases and the approach can be applied in a wider variety of run-time environments.

- When using the point-wise approach, at a certain point the user may need to derive data intended to cover a particular `dua` or to check whether that `dua` is executable. Doing so requires examination of the set of paths which cover that `dua`. This is precisely the information contained in the corresponding `edua`. In this sense, the path-wise approach can be considered to complement the point-wise approach.

Figure 1: Example program

3 Eduas and the Alias Automaton

In this section we define extended definition-use associations and show how they can be represented in terms of an intermediate structure called the Alias Automaton. We then show how the Alias Automaton can be used to check which `edua`s have been exercised by the test data. We then present algorithms for constructing the Alias Automaton and deriving `edua`s from it.

The focus in this section is on the intra-procedural analysis. We consider aliasing through the use of pointer variables a more general problem than inter-procedural analysis, since aliases through pointer variables can be created and modified at any point in the program, whereas aliases through parameters are established and broken at the entry and exit of a block.

In this section we analyze a single non-recursive program unit Q which may be part of a larger program. We assume that we know which variables are aliased with one another at entry to Q. If Q calls other program units, we assume that the user supplies information about how those calls effect aliasing behavior. We will refer to Q as the subject program unit, or, for brevity, the subject program. The extension of our approach to inter-procedural analysis is discussed in section 5.

An *execution path* is a path from the entry to the exit of the program unit and an *execution point* is the top or bottom of a node on an execution path. At a given execution point the *Live Alias relation* describes which dereferenced pointers are aliased with one another. A pair $< \rho, \sigma >$ of pointers is an *alias pair* at a given execution point if ρ and σ refer to the same memory location. The Live Alias relation at that execution point is the set of alias pairs. Note that the Live Alias relation is an equivalence relation. For notational convenience we will denote a Live Alias relation R by a set of pairs whose reflexive, symmetric, transitive closure is R. For example, given three pointer variables `p`$\uparrow$, `q`$\uparrow$, `r`$\uparrow$, we will write $\{< p, q >\}$ to denote the relation $\{< p, p >, < q, q >, < r, r >, < p, q >, < q, p >\}$.

Note that for a given execution path, a given *program point* may correspond to several different execution points, each having different values for the Live Alias relation. For example, consider the path $(1, 2, 3, 4, 6, 7, 3, 4, 5, 6, 7, 3, 8)$ through the program in Figure 1. There are two execution points on this path that correspond to program point 7. The Live Alias relation of the first of these is $\emptyset$ and that of the second is $\{< p, q >\}$.

An *extended definition-use association (*`edua`*)* is a 4-tuple $(d, u, \mathtt{v}, S)$ where d and u are nodes, $\mathtt{v}$ is a variable, and S is the set of execution paths π having the following properties:

1. π has an execution point d_1 corresponding to program point d which has a definition of the memory location m of variable `v`,

2. π has an execution point u_1 corresponding to program point u which has a use of a variable `v'`, where the memory address m of `v` at d_1 is the same as that of `v'` at u_1, and

3. Along the sub-path of π from d_1 to u_1, there is no execution point at which either

   (a) the contents of the cell at address m is modified, or

   (b) the binding between the name `v` and the location m is broken.

For example, in the program in Figure 1 let S be the set of execution paths that pass through node 5. Then $(6, 7, \mathtt{p}\uparrow, S)$ is an `edua`. This is because along any such path, control passes through nodes 6 and 7 with the definition of `p`↑ in node 6 accessing the same memory location as the use of `q`↑ in node 7. The contents of that memory location is not modified between nodes 6 and 7 and the binding between `p`↑ and that location is not broken.

Let S' be the set of execution paths that *do not* pass through node 5. Then $(6, 8, \mathtt{p}\uparrow, S')$ is an `edua`. Note that for any path that *does* go through node 5, the Live-Alias relation at node 7 contains $< p, q >$, so the definition of `q`↑ at node 7 modifies the relevant memory cell. Also $(2, 7, \mathtt{q}\uparrow, S')$ is an edua. For any path that *does* go through node 5, the binding between `q`↑ and the memory cell is broken.

Note that if `v` is a variable which is never aliased with any other variables and $(d, u, \mathtt{v})$ is a definition-use association, then $(d, u, \mathtt{v}, S)$ is an extended definition-use association, where S is the set of paths having definition-clear subpaths from d to u. (In this case, `v'` is `v`).

In the remainder of this section, we describe how `edua`s are derived and how their coverage is checked. We restrict attention to the use of pointer variables as in Pascal. That is, the only operations which can be performed on pointers are assignment of one pointer to another (`p := q`); assignment of an unused memory address to a pointer (`new(p)`); release of the memory address corresponding to a pointer (`dispose(p)`); and the dereferencing operation (`p`↑), after which any operations appropriate to the type of the dereferenced pointer can be performed. In particular, performing arithmetic operations on pointers to yield other memory addresses (as permitted in the C language) is not allowed. Our approach considers arrays as aggregates, consequently it does not detect aliases that occur through array components (i.e. if array elements a[i] and a[j] are referenced at a point in the program when i = j). We also make the following assumptions:

- No double indirection is present. Double indirection occurs when a dereferenced pointer (or a field thereof) is another pointer or when the elements of an array are pointers.

- No hidden aliasing occurs through dangling references.

- The Live Alias relation is known at entry to the procedure being analyzed and any changes to that relation arising in called procedures are known.

Under these assumptions, given a single path through the program, we can compute the Live Alias relation at each point on the path. Given a set of paths represented by a regular expression, the paths may or may not have the same Live Alias relations at corresponding points.

3.1 The Alias Automaton

In order to group together paths which have the same Live Alias relation at corresponding points and to distinguish between different Live Alias relations that can arise at a given program point, we construct an intermediate representation called the *Alias Automaton.* The Alias Automaton arising from the program in Figure 1 is shown in Figure 2. Each state of the automaton is associated with a Live Alias relation and each transition is labeled by a statement in the subject program. A given Live Alias relation R may be associated with more than one automaton state and a given program statement may label more than one transition. Subscripts on statement numbers are included to distinguish different occurrences of the same statement number. The Alias Automaton has the following properties:

1. The start state of the automaton is associated with the initial Live Alias relation and the only transition out of the start state is labeled by the first statement in the subject program. States with no out-edges are accepting states.

2. Let s_1 and s_2 be automaton states associated with Live Alias relations R_1 and R_2 and let n be a statement labeling a transition from s_1 to s_2. Then if the Live Alias relation is R_1 immediately before executing n, it will be R_2 immediately after executing n.

3. A sequence of statement numbers is accepted by the automaton if and only if it corresponds to a path through the flow graph.

States with two successors represent decision points in the program. Loops in the program give rise to cycles

Figure 2: Alias Automaton

in the automaton, as discussed below. Let π be an execution sub-path beginning at the program unit entry. Execution of the automaton on this path will lead to a state s associated with Live Alias relation R such that the Live Alias relation after executing π is precisely R. Thus the Alias Automaton distinguishes between different paths from program entry to a given program point when those paths result in different Live Alias relations at the point. The construction of the Alias Automaton is discussed in section 3.2.

Just as one can augment a program flow graph with indications of which variables are defined or used in which statements, we can augment the Alias Automaton with such information. Consider state s with an out-edge labeled n. If there is a definition of $\rho\uparrow$ in statement n then the edge has a definition of $\rho\uparrow$ *as well as* definitions of any $\sigma\uparrow$ for which $< \rho, \sigma >$ is in the Live Alias relation at s. Uses are treated analogously. If there is a definition of ρ in statement n, there is a definition of $\rho\uparrow$ on the edge. In addition to modifying the corresponding Live Alias relation, we consider this to be a definition of $\rho\uparrow$ because $\rho\uparrow$ may refer to a different cell after executing statement n, thus can have a different value.[1]

For example, in the automaton in Figure 2 edge 7_1 has a definition and use of $\texttt{q}\uparrow$, whereas edge 7_2 has definitions and uses of *both* $\texttt{q}\uparrow$ and $\texttt{p}\uparrow$ because $< p, q >$ is in the Live Alias relation of the preceding state.

Having associated definition-use information with the edges of the automaton, we can now describe `eduas` in terms of the automaton. An `edua` can be represented by a triple $(d, u, \texttt{v})$ where d is an automaton edge having a definition of variable $\texttt{v}$ (which may be a dereferenced pointer, such as $\texttt{p}\uparrow$), u is an automaton edge with a use of $\texttt{v}$ and there is a definition-clear path with respect to $\texttt{v}$ from d to u in *the automaton*. For example, the definition of $\texttt{p}\uparrow$ in 6_1 reaches the use of $\texttt{p}\uparrow$ in 8_1 but the definition of $\texttt{p}\uparrow$ in 6_2 does not reach the use of $\texttt{p}\uparrow$ in 8_2, because it is killed in 7_2. We can compute the `eduas` by slightly modifying standard reaching definitions algorithms to account for the fact that definitions and uses in the automaton are on edges, rather than nodes. A list of all the `eduas` from the example program is shown in Table 1.

In principle, one could explicitly generate regular expressions representing the `eduas`, then use a pattern matching tool, such as *egrep* to check whether one of the program traces matches each regular expression. However, a more direct approach is to use the Alias Automaton directly to check the program traces. We "run" the program trace through the automaton to translate it into an *automaton trace*, i.e., the sequence of automaton edges traversed by this program path. We then check whether the automaton trace has the desired pattern.

a.	$(1_1, 8_1, \texttt{p}\uparrow)$	e.	$(6_3, 7_3, \texttt{p}\uparrow)$	i.	$(7_2, 8_2, \texttt{q}\uparrow)$
b.	$(6_1, 8_1, \texttt{p}\uparrow)$	f.	$(7_3, 8_2, \texttt{p}\uparrow)$	j.	$(6_3, 7_3, \texttt{q}\uparrow)$
c.	$(6_2, 7_2, \texttt{p}\uparrow)$	g.	$(2_1, 7_1, \texttt{q}\uparrow)$	k.	$(7_3, 8_2, \texttt{q}\uparrow)$
d.	$(7_2, 8_2, \texttt{p}\uparrow)$	h.	$(6_2, 7_2, \texttt{q}\uparrow)$	l.	$(7_1, 7_1, \texttt{q}\uparrow)$

Table 1: Extended definition use associations

For example, suppose the test data traversed the path $(1, 2, 3, 4, 5, 6, 7, 3, 4, 6, 7, 3, 8)$. The corresponding automaton trace is

$$(1_1, 2_1, 3_1, 4_1, 5_1, 6_2, 7_2, 3_2, 4_2, 6_3, 7_3, 3_2, 8_2).$$

This covers `eduas` c,e,f,h,j, and k listed in Table 1.

3.2 Construction of Alias Automaton

To construct the Alias Automaton, we begin with a tree representation of the regular expression representing the set of all execution paths in the subject program unit. Each leaf node in this tree is labeled with a symbol corresponding to a statement in the program or the null symbol λ. Internal nodes are labeled with the operator symbols: '*' for loop, '+' for conditional, '.' for concatenation, corresponding to the flow of control in the program. Sequence (.) nodes have two children. Conditional (+) nodes have two subtrees, representing the paths through the true and false branches. Loop (*) nodes have only a left subtree, which represents the path through the body of the loop. The expression tree for the example in Figure 1 is shown in Figure 3.

The Alias Automaton is constructed during a depth-first traversal of the expression tree[2]. As it is constructed, two sets of automaton states, the *pre-states* and the *post-states* are associated with each tree node. The *pre-states* of a node t represent all of the Live Alias relations that could occur after executing some sub-path that matches the regular expression up to just prior to t. They are constructed prior to visiting t and are passed down through the tree. The *post-states* of t represent Live Alias relations that could occur immediately after processing all of t's subtrees. They are constructed while processing t and its subtrees and are passed up through the tree.

A new state is added to the automaton when a leaf node of the tree (representing a statement n in the program) is processed. The Live Alias relations of the new

[1] As a consequence, in a case such as `p↑ := x; q:=p; p:=r; p:=q; y:=p↑`, our algorithm will not consider this to be a definition-clear sequence with respect to $\texttt{p}\uparrow$.

[2] Subtrees representing loop bodies are traversed multiple times.

Figure 3: Expression Tree

state is obtained by updating the relation at the preceding state according to how n affects the aliasing, as shown in function `Update` in Figure 4.

Pseudo-code for the traversal of the expression tree is given in procedure `Construct-Automaton` in Figure 5. One can think of the algorithm as "growing" the automaton from the initial state by maintaining a "wave" of newly created states representing different Live Alias relations that could occur at any execution point corresponding to the current point in the regular expression. This wave is represented by the post-state of the subtree which has been most recently processed.

There may be more than one state in the wave if the portion of the regular expression processed contains + and * nodes. It may be the case that some of the states in the wave of new states have the same Live Alias relation. Such states are equivalent in the following sense. Let s_1 and s_2 have the same Live Alias relation R and correspond to the same program statement n (i.e, point to the same point in the regular expression tree). If π_1 is a path from entry to s_1 and π_2 is a path from entry to s_2, then for any subsequent path π_3, the Live Alias relation after executing $\pi_1 \cdot \pi_3$ is the same as that after executing $\pi_2 \cdot \pi_3$. Such equivalent states can be merged (either by introducing a dummy "join state" or by adding an edge with null label from s_2 to s_1.)

Having processed subexpression α, the automaton for $\alpha \cdot n$ is obtained by adding new states as successors of each state in the wave.

The result of processing $\alpha \cdot (\beta + \gamma)$ is that a sub-automaton arising from β and a sub-automaton arising from γ are both added to each node in the "wave" that exists after processing α. After processing the sub-trees, the equivalent states are merged.

In processing $\alpha \cdot \beta^*$, the set S of new states after processing α represents Live Alias relations that can occur after zero iterations of the loop represented by β. These are each propagated through β to obtain a set S' of states representing Live Alias relations that can occur after one iteration of the loop. Some of the states in S' may have the same Live Alias relations as some of the states in S. If state $s \in S$ and state $s' \in S'$ have the same Live Alias relation, this indicates that, for our purposes, paths which go from the initial state to s are equivalent to those that go from the initial state to s'. Thus we add an edge (labeled by the empty string) from s' to s, creating a cycle in the automaton. Those states in S' which do not match any state in S represent situations that can occur at the top of the loop and that have not already been analyzed. These new states are propagated through β, the resulting final states are compared to states which have already been analyzed, and so on.

In our example, the first time the * node is processed, there is only one pre-state (state 3), which has the empty Live Alias relation. After propagating this through the loop, two post-states result, states 9 and 13. State 9 has the empty Live Alias relation, as did state 3, so we add an edge from 9 to 3. State 13 has Live Alias relation $\{< p, q >\}$, which is new. Next, state 13 is propagated through the loop, resulting in post-state 22, which has the same Live Alias relation as 13. Since no new Live Alias relations remain, analysis of the loop is complete.

Although there are infinitely many paths through the loop, this process is guaranteed to terminate, since there are at most 2^{m^2} Live Alias relations possible on a set of m pointer variables[3]. Thus after at most that number of iterations, there will be no new situations to analyze. Although the upper bound is super-exponential, preliminary experience suggests that the number of iterations will be much more reasonable in practice.

4 Design and implementation of `APODOS`

We have designed and implemented `APODOS`, **A** **P**ath-**O**riented **D**ata fl**O**w **S**ystem. [4]. `APODOS` is based on the path-wise approach to analyzing pointer variables. It takes as input a Pascal program P, indication from the user as to which subprograms are of interest and a set

[3] A tighter, but still super-polynomial bound can be obtained by observing that the Live Alias relation is an equivalence relation

[4] *Apodos* is the Spanish word for *nicknames* or *aliases*.

```
function Update(n: statement ,R:Live Alias relation): Live Alias relation;
R' := R;
case statement type of
Allocation or de-allocation, new(p) or dispose(p):
      Remove all alias pairs <p,r> and <r,p> where r ≠ p from R'
Assignment p := q:
      Remove all alias pairs <p,r> and <r,p> from R'
      Add the pairs <p,q> to R'
      R' := reflexive, symmetric, and transitive closure R'
other: no operation
end-case
return(R')
```

Figure 4: Pseudo-code for updating the Live Alias Relation

of test data T. The tool executes the program P using T and provides the user with information as to which `eduas` in P still need to be covered by additional test data to satisfy the all-uses criterion. The user then examines the output of the subject program to determine if the program has behaved correctly and whether there are `eduas` that still need to be covered. The user may continue the testing process by providing additional test data.

The current version of `APODOS` accepts almost any Pascal program, with a few syntactic restrictions, such as no `goto` statements and no nested `with` statements. For those programs that violate the restrictions on how pointers are used (see section 3) the analysis may be inaccurate. If a subprogram Q is designated as being "of interest" then all subprograms in which Q is nested must also be so designated. Recursive subprograms may not be designated as being of interest.

The overall architecture of our tool is similar to that of ASSET [3, 2]. `APODOS` consists of the following modules which communicate with each other via files. (A) Front End, (B) Propagator, (C) Obligation Generator, (D) Compiler, (E) Tester, (F) Coverage Analyzer.

The **Front End** takes as input a syntactically correct Pascal program P, which we will call the subject program. It produces the following:

- numbered subject: a copy of the subject program that includes unique numbers for each statement. This version of P is used to establish points of reference with the user. The paths that need to be covered are conveyed in terms of the statement numbers.
- tracer: a modification of the subject program, which produces a trace of the execution of P. It includes probes, which when executed, will write the corresponding statement number to a file called the program trace.
- regular expressions which represent the control flow (the set of all paths) of the main program , and of each subroutine in the following way: if the subroutine is to be ignored by the analysis, calls are represented by the corresponding statement number, otherwise the control flow of the caller is expanded by inserting in the position of each call, the regular expression corresponding to the called subroutine.
- alias-def-use table: which lists, for each statement, which variables are being aliased, defined and used. Statements in which either pointer variables are defined or which contain calls to included subroutines are points at which the Live Alias relation can change.

The front end was built using the Purdue Compiler Construction Tool Set (pccts)[13], which inputs an LL(k) grammar and outputs a top-down parser. We augmented the Pascal grammar supplied with pccts to include actions for inserting statement numbers and probes, constructing regular expressions, constructing a symbol table, and constructing an alias-def-use table containing pertinent information on alias creation, definitions and uses of variables.

The **Propagator** constructs the Alias Automaton, using the algorithm in section 3.2. It takes as input the alias-def-use table, the regular expression representing the set of paths in the subject program expanded with the included subroutines, and assumes the initial Live Alias relation is empty. Note: the regular expressions

```
procedure Construct-Automaton(t: expression-tree-node);
case node-type of
leaf representing statement n:
     for each pre-state s with Live Alias relation R do the following:
         create a new state s' with Live Alias relation R' = Update(n, R);
         create an edge from s to s' labeled n;
         add s' to post-states of t;

· :  pre-states of left child := pre-states of t;
     Construct-Automaton( left child);
     pre-states of right child := post-states of left child;
     Construct-Automaton( right child);
     post-states of t := post-states of right child;

+:   pre-states of left child := pre-states of t;
     Construct-Automaton( left child);
     pre-states of right child := pre-states of t;
     Construct-Automaton( right child);
     {note that each pre-state of t will get two successors, one
         coming from the left subtree and one coming from the right subtree}.
     post-states of t := union of post-states of left child and post-states of right child;
     merge post-states which have identical Live Alias relations;

⋆:   pre-states of left child := pre-states of t;
     post-states of t := pre-states of t;
     while pre-states of left child not empty do
         Construct-Automaton( left child);
         pre-states of left child := empty; {prepare to accumulate new states}
         for each post-state s of left child
             if s has same Live Alias relation as some post-state s' of t
               then add an edge from s to s' with null label
                     { merge, creating a cycle in the automaton }
               else {this is a new state, which must be propagated through loop }
                     add s to post-states of t;
                     add s to pre-states of left child ;
             end if
     end-while
```

Figure 5: Pseudo-code for construction of Alias Automaton

may contain call statements to other subroutines, which are treated as single statements. The module produces as output an automaton representing the different execution paths in P where the alias information is precise. The Propagator also constructs an extended-def-use table which describes which variables are defined and used on each automaton edge, as described in section 3.1.

The **Obligation Generator** inputs the Alias Automaton, and the extended-def-use table and derives the set of `edua`s.

The **Compiler** is an ordinary Pascal compiler, which takes the tracer and produces an object file, which we call the tracer object code.

The **Tester** takes as input the tracer object code and a set T of test data. For each test case t in T, it executes the tracer with t as input. This produces the output $P(t)$ which would be produced by running P on t, and also appends the trace of the paths executed to the trace file. These traces are run through the Alias Automaton to convert them from program traces to automaton traces.

The **Coverage Analyzer** takes as input the list of `edua`s produced by the Obligation Generator, the extended-def-use table, and the trace file produced by the Tester, and produces as output the obligations that were not covered by the trace. All of the `edua`s can be checked in (simulated) parallel, using the checking routine used to check `dua` coverage from ASSET.

5 Inter-Procedural Analysis

The ideas presented so far can be extended to the inter-procedural setting, provided that no recursive subroutines are considered. Both types of subroutines, functions and procedures, are treated in a similar fashion. In the current version of `APODOS`, the front end requests from the user an indication of what subroutines are to be included in the analysis. For each subroutine that is of interest to the tester, the regular expression which represents the control flow of its body is embedded in the caller's regular expression at the point of the call.[5] Local variables, and parameters, are included in the alias-def-use table. Probes are inserted in the tracer only for the statements that are of interest (i.e. the main program and the included subroutines).

The execution point at the call represents a point where the Live Alias relation may change: for each call-by-reference parameter, a new alias pair is formed with the corresponding argument.[6] The actual parameters are used and the formal parameters are defined at the top of the entry to the subroutine's body. At the bottom of the last statement of the subroutine's body the Live Alias relation may change due to the return to the caller and the local variables becoming out of scope. The alias set is updated by removing the corresponding alias pairs.

The rest of the tool remains as described in the previous sections. We are investigating whether the Alias Automaton grows too large when subroutines are "in-lined" in this manner. Preliminary tests suggest that the tester can exercise control on the growth rate of the Automaton by including subroutines selectively.

6 Future Work

We have presented a new approach to data flow testing in the presence of aliasing through pointer variables and pass-by-reference parameters. In contrast to previous point-wise approaches, our path-wise approach explicitly distinguishes paths which have different behavior with respect to aliasing. The testing requirements in our approach are extended definition-use associations, each of which requires the test data to exercise a representative of a particular subset of the set of paths from program entry to a program point with a variable definition to a program point with a use of that variable. In contrast to previous approaches, our approach does not require that memory accesses be traced during test execution. In addition, the explicit representation of the subset of paths which covers a given definition-use pair provides information that is useful for test data generation and for identifying infeasible test requirements. We have designed and implemented a tool, `APODOS`, based on this approach for testing Pascal programs. While our approach has a super-polynomial worst case, preliminary experience indicates that the cost is reasonable in practice. We are investigating methods to improve the portion of the algorithm that merges equivalent Alias Automaton states.

The current algorithm sometimes produces `edua`s that are redundant, in the sense that they describe situations captured by other `edua`s. We are investigating an improved algorithm.

Our approach cannot do accurate analysis in the presence of arbitrary double indirections (pointers to other

[5] In the case of a function, the routine's regular expression will be embedded just prior to the statement using the functional expression.

[6] In the case of pointer variables, aliasing occurs for the call-by-value parameters. Call-by-reference pointer parameters are cases of double indirection.

pointers), but can be extended to handle certain special cases, including some important ones arising in programs with linked data structures. We plan to incorporate this analysis into the tool.

Perhaps the most important direction for future work is experimentation with the tool to explore the cost and effectiveness of this technique.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[2] P. G. Frankl. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD thesis, Courant Institute of Mathematical Sciences, New York University, Oct. 1987.

[3] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. ASSET – a system to select and evaluate tests. In *Proceedings of the IEEE Conference on Software Tools*, pages 72–79, Apr. 1985.

[4] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, Oct. 1988.

[5] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 158–167. ACM Press, Dec. 1990.

[6] P. Herman. A data flow analysis approach to program testing. *The Australian Computer Journal*, 8(3):92–96, Nov. 1976.

[7] J. Horgan and S. London. Data flow coverage and the C language. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, pages 87–97. ACM Press, Oct. 1991.

[8] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248. ACM Press, June 1992.

[9] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[10] S. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Nov. 1984.

[11] T. J. Ostrand and E. J. Weyuker. Data flow based test adequacy analysis for languages with pointers. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, pages 74–86. ACM Press, Oct. 1991.

[12] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[13] T. Parr and R. Quong. ANTLR: a predicated-LL(k) parser generator. *Software Practice and Experience.* To appear.

[14] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-14(4):367–375, Apr. 1985.

[15] H. Ural and B. Yang. A test selection method for protocol testing. *IEEE Transactions on Communications*, 39(4), Apr. 1991.