

8-6-1993

Data Dependence in Programs Involving Indexed Variables

Borislav Nikolik
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Computer Sciences Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Nikolik, Borislav, "Data Dependence in Programs Involving Indexed Variables" (1993). *Dissertations and Theses*. Paper 4688.

<https://doi.org/10.15760/etd.6572>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

AN ABSTRACT OF THE THESIS OF Borislav Nikolik for the Master of Science in
Computer Science presented August 6, 1993.

Title: Data Dependence in Programs Involving Indexed Variables

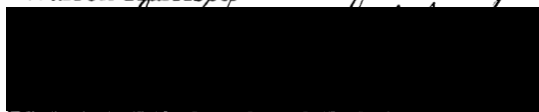
APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



Richard Hamlet, Chair



Warren Harrison



Michael Driscoll

Symbolic execution is a powerful technique used to perform various activities such as program testing, formal verification of programs, etc. However, symbolic execution does not deal with indexed variables in an adequate manner. Integration of indexed variables such as arrays into symbolic execution would increase the generality of this technique. We present an original substitution technique that produces array-term-free constraints as a counterargument to the commonly accepted belief that symbolic execution cannot handle arrays. The substitution technique deals with constraints involving array terms with a single aggregate name, array terms with multiple aggregate names, and nested array terms. Our approach to solving constraints involving array terms is based on the analysis of the relationship between the array subscripts.

Dataflow dependence analysis of programs involving indexed variables suffers from problems of undecidability. We propose a separation technique in which the array

subscript constraints are separated from the loop path constraints. The separation technique suggests that the problem of establishing data dependencies is not as hard as the general loop problem. In this respect, we present a new general heuristic program analysis technique which is used to preserve the properties of the relations between program variables.

**DATA DEPENDENCE IN PROGRAMS
INVOLVING INDEXED VARIABLES**

by

BORISLAV NIKOLIK

A thesis submitted in partial fulfillment of the
requirements for a degree of

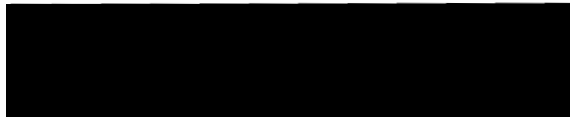
**MASTER OF SCIENCE
in
COMPUTER SCIENCE**

Portland State University

1993

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Borislav Nikolik presented August 6, 1993.



Richard Hamlet, Chair



Warren Harrison



Michael Driscoll

APPROVED:



Leonard D. Shapiro, Chair, Department of Computer Science



Roy W. Koch, Vice Provost for Graduate Studies and Research

TABLE OF CONTENTS

	PAGE
LIST OF FIGURES.....	v
CHAPTER	
I	INTRODUCTION..... 1
	Static Array Data Dependence..... 2
	Data Dependence in Programs Involving Pointers..... 4
	Thesis Organization..... 7
II	BACKGROUND..... 8
	Symbolic Execution..... 8
	Array Values in Symbolic Execution..... 10
	Data Dependence Analysis..... 13
	Input Constraints..... 16
	Dataflow Testing..... 16
III	SUBSTITUTION TECHNIQUE..... 19
	Multiple Array Terms..... 21
	Homogeneous Array Terms Constraints..... 21
	Nested Homogeneous Array Terms..... 23
	Heterogeneous Array Terms 25
IV	ARRAY SUBSTITUTION..... 26
	Evaluation of Expressions..... 26
	Array Substitution..... 29

		iv
	Substitution Algorithm.....	29
V	INFINITY OF PATHS.....	36
	Anomalies in Constraints.....	36
	Partial Symbolic Execution.....	38
	Explicit Naming of Paths.....	42
	Partial Paths.....	46
	Loop Forcing.....	50
VI	CONCLUSION.....	57
	REFERENCES.....	60

LIST OF FIGURES

FIGURE		PAGE
1.	Selection Sort[23], p. 25.....	4
2.	Pointer manipulation function.....	6
3.	Program fragment with two execution paths.....	9
4.	Path condition and functional description.....	9
5.	Array term and array subscript constraint.....	12
6.	Array manipulation program.....	12
7.	Program that exchanges two array elements.....	14
8.	Bubble sort function[24], p. 66.....	15
9.	Array manipulation function.....	17
10.	Predicate trans.....	28
11.	Translation of atoms to CLP(R) variables.....	28
12.	Substitute predicate.....	30
13.	Eliminate array terms from a set of equations.....	30
14.	Replacement predicate.....	31
15.	Search predicate.....	33
16.	Auxiliary predicates.....	34
17.	Array manipulation function.....	37
18.	Partial symbolic execution.....	39
19.	Deficiencies of PSE.....	40

20.	Deficiencies of PSE.....	41
21.	Irrelevant complex paths.....	42
22.	Irrelevant complex paths.....	43
23.	Node repetition required for data dependence.....	45
24.	Infeasible array data dependence.....	46
25.	Complex paths resulting from a loop.....	48
26.	Array manipulation function.....	51
27.	Bubble sort function[24], p.66.....	53
28.	Infeasible array subscripts.....	54
29.	Program segment without satisfiable ASC.....	55
30.	Bubble sort program.....	55

CHAPTER I

INTRODUCTION

Many problems in program analysis consist of identifying data-flow dependencies in programs. Dataflow dependence information is needed for data-flow testing, program optimization, program slicing, program equivalence, etc. The analysis presented in this thesis examines the existence and identification of a single path that establishes data dependence between two program statements. Therefore, the analysis is useful for some variants of data-flow testing, program optimization, program slicing, etc. The analysis is not appropriate for problems such as program equivalence which require data-flow properties for all the paths in a program.

The identification of data dependencies consists of showing that variable assignments propagate data between statements. In the case of indexed variables such as arrays, symbolic execution is used to remove execution-state differences between the variable indices, so that the indices can be tested for equality. Symbolic execution in a program text involving indexed references may result in constraints containing indexed terms. Our approach of solving constraints involving array terms is based on an original substitution technique that examines the relationship between the array subscripts. Depending on the outcome of the examination, different or the same simple variables are substituted for the indexed variables. The technique deals with constraints involving array terms of different kinds, nested terms, and multidimensional arrays. We present this technique as a counterargument to the commonly accepted belief that symbolic execution cannot handle arrays.

Software analysis of programs involving arrays suffers from problems of undecidability and unsolvability. In particular, a potential infinity of paths through looping

constructs needs to be considered to detect data dependence between statements containing array terms; two array indices could be different on an arbitrary number of loop traversals, then turn up the same. To deal with the infinity of paths problem a technique that separates the array subscript constraint from the loop path constraints is proposed. This approach allows distinct techniques to be used to derive the constraints necessary for establishing data dependence. The array subscript constraints are obtained on simple paths, and the loop path constraints are obtained on complex paths. Finally, the separation technique suggests that the array data dependence problem is not as hard as the general loop problem. In particular, rather than trying to obtain precise semantics of the loop, the loop is examined for paths that preserve data dependence.

STATIC ARRAY DATA DEPENDENCE

Program analysis involves identifying dependences between program statements. A particular type of program dependence is data-flow dependence. If there is a sequence of variable assignments that propagate data from S_1 to S_2 , then the statement S_2 is data-flow dependent on S_1 . The case involving simple variables requires the same variable name to occur in both of the statements. The case involving array data-flow objects is more complex; it consists of showing that two array references (possibly in the same program statement) could represent the same array element. In particular, the data dependence identification consists of determining whether the array subscripts can ever be the same. Consider the Selection sort routine given in Figure 1. Data could be propagated between the statement at line 6 (S_6) and the statement at line 8 (S_8); `small` acquires a value at line 6 and that value is used in the conditional statement at line 8. Therefore, the statement S_8 is data-flow dependent on the statement S_6 . In the array case, S_6 would be data dependent on S_{12} if $A[i]$ and $A[k]$ could represent the same array element, and therefore refer to the same location. To compare the subscripts i and k , the execution-state differ-

ences between statements S_6 and S_{12} need to be removed so that both indices refer to a common state. First the indices are symbolically executed on a common path so that the symbolic subscript expressions refer to a common execution state, and then the subscripts are checked for equality.

However, the constraints obtained by equating the symbolic expressions of the array indices appearing in some statements S_i and S_j may contain array terms. Furthermore, the path condition of the path connecting S_i and S_j may also contain array terms. For example, the path 12-13-4-5-6-7-8-9-10 in Figure 1 has $A[j] < \text{small} \wedge j \leq N \wedge k \leq N-1$ as path condition. Before such constraints can be mechanically solved by a constraint solver, care needs to be taken to properly represent array terms. In this thesis we formalize a general substitution technique and give the substitution algorithm that produces array-free constraints. The technique is general in the sense that it treats array terms with the same aggregate names, array terms with different aggregate names, nested as well as multidimensional array terms. Detecting data dependence between statements involving array terms consists of determining whether a set of array terms refers to the same memory location. With programs involving looping structures, this implies considering a potential infinity of paths through loops. Given an arbitrary program, the problem of determining whether two array indices can ever be the same is undecidable. To deal with the infinity of paths problem, an approach in which the array subscript constraint is separated from the loop path constraint is proposed. The former constraint is obtained on simple paths, paths for which symbolic execution is effective. Unfortunately, the latter constraint is derived from complex path analysis which suffers from problems of undecidability and unsolvability. Two heuristic techniques that deal with complex paths, partial symbolic evaluation and explicit naming of paths, are investigated. Then we propose a new heuristic technique: loop forcing.

```
void SelectionSort(A, N)
    int *A, N;
    {
        int small, i, j, k;
4       for(k = 1; k ≤ N - 1; k++) {
5           i = k;
6           small = A[k];
7           for(j = k + 1; j ≤ N; j++)
8               if(A[j] < small) {
9                   i = j;
10                  small = A[j];
11              }
12          A[i] = A[k];
13          A[k] = small
14      }
15  }
```

Figure 1. Selection sort [23], p. 25.

DATA DEPENDENCE IN PROGRAMS INVOLVING POINTERS

Treating arrays allocated at run time, or static arrays manipulated through pointers, raises difficult problems associated with pointers. Apart from the problems present in the static array analysis, additional complications result from anonymous dynamic arrays or static arrays manipulated through pointers. The difficulty arises from the fact that pointer variables are associated with a set of possible objects. In general, the exact object dereferenced through some pointer is not known because of the infinity of possible paths

through loops. Note that the object could be a collection of objects such as aggregate data structures. In this case, neither the exact collection nor the exact element in the collection is known. The sets of objects associated with different pointer variables are not disjoint. Therefore, a set of pointer variables could refer to the same object. Furthermore, the set of objects referenced by a pointer, in general, cannot be statically determined. Figure 2 shows a function that involves array manipulation through pointer arithmetic. To establish data dependence between S_{14} and S_{18} , the indices j and i at lines 12 and 18 need to be equal; this is the same difficulty that occurs in the static array case. Statement S_{18} is data dependent on S_{13} if p and q could not only have the same offset, but reference the same array — a case not present in the static array analysis. In particular, two distinct pointer variables, q (line 13) and p (line 18) could dereference the same object if i and j at lines 7 and 12 could be equal on some feasible path that connects lines 13 and 18. This case does not occur in the static case because two distinct array aggregate names always reference distinct objects. Determining the exact set of objects associated with a pointer variable is undecidable because it requires considering infinity of paths through loops. In the static case, the set of objects associated with an array variable is known, and determined at compile time rather than at run time.

Existing data flow testing tools such as ASSET[21] are based on simplistic approaches in which pointer and array variables are treated in the same manner as simple variables. A more sophisticated approach has been implemented in TACTIC[22]. Rather than considering paths that exercise data dependencies, program points are identified with alias sets, sets of variables associated with pointer variables. The alias set of a pointer variable at some program point is an approximation of all the objects possibly referenced by that variable. Whenever some variable V belongs to an alias set of some pointer variable p , then dereferencing p is a possible use of V .

```
int fun(int_array)
int *int_array;
{
int *p, *q,
i, j, done = TRUE;

4     scanf("%d %d", &i, &j);
5     while(done) {
6         if(i > j) {
7             p = &int_array[i];
8             q = p;
9             *p = j;
10        }
11        else {
12            p = &int_array[j];
13            *q = i;
14            *p = j;
15            done = FALSE;
16        }
17    }
18    i = p[i];
19    return int_array[i];
20 }
```

Figure 2. Pointer manipulation function.

However, computing alias sets at individual lines causes spurious associations; variable pairs are identified as associations, but they are located on an infeasible path. Many of these associations could be detected and eliminated by solving path constraints.

In this thesis, we do not address the issue of pointers in data dependence analysis. However, some of the techniques presented in this thesis are extensible to pointers. For instance, the substitution technique described in Chapter III could be used to deal with the case of indexed pointer variables. The loop forcing technique described in Chapter V could be used to test whether certain relations between pointer variables hold.

THESIS ORGANIZATION

Chapter II provides background about symbolic execution and data dependence. In Chapter III, we first present our technique for solving constraints involving array references and then discuss homogeneous and heterogeneous array terms. The substitution algorithm used to eliminate array terms from equations is given in Chapter IV. Chapter V addresses the problems involved in identifying data dependencies in programs containing arrays, discusses the separation of partial paths from complex paths and considers two heuristic techniques that deal with complex paths. Chapter V ends with a discussion of separation of satisfiable and unsatisfiable subscript constraints on simple paths and techniques to improve the data dependence analysis. Finally, our conclusions and suggestions for further work are presented in Chapter VI.

CHAPTER II

BACKGROUND

SYMBOLIC EXECUTION

Symbolic execution is a powerful technique that is used to produce an algebraic formula representing the output of a class of conventional executions. Instead of executing a program on individual inputs, symbolic inputs are given to the symbolic executor which derives a formula describing the relationship between the input and the output variables. Symbolic execution of a given path P results in two pieces of information: a *path condition* and *functional description*. The path condition is a set of constraints that describe how input variables are constrained by various control flow conditions along the path P . The functional description expresses output variables in terms of input variables for the given path P . For example, symbolic execution of the program fragment in Figure 3 produces the result given in tabular form in Figure 4.

Symbolic execution can be done in two ways: forward or backward chaining. Forward chaining starts with an expression at the top of the path and symbolically executes every statement along the path; conditional statements collected along the path affect the path condition, and assignment statements affect the functional description. Backward chaining is based on Hoare's axiom of assignment starting with an expression at the bottom of the path and symbolically executing only the statements that affect the expression. Forward chaining allows early detection of infeasible paths, but requires that the intermediate symbolic values of all variables be computed and retained.

```

int x, z;
...
2   scanf("%d", &x);
3   x = x + 1;
4   if(x > 0) {
5       x = x + 1;
6       z = x + 1;
7   }
8   else {
9       z = x;
10  }
11  z = z + x;
12  printf("%d",z);

```

Figure 3. Program fragment with two execution paths.

path	path condition	functional descr.
2-3-4-5-6-11-12	$x > -1$	$z = 2 * x + 5$
2-3-4-8-9-11-12	$x \leq -1$	$z = 2 * (x + 1)$

Figure 4. Path condition and functional description.

Symbolic execution can be used to perform various software analysis tasks. For instance, symbolic testing captures the intuition of experienced testers and aids in test-case generation and test-case coverage[12]. A test data generation system symbolically executes a path and attempts to generate test data that would cause execution of the selected path[14]. Symbolic execution can also be used in loop analysis and program correctness verification[13]. In data-flow testing of arrays, the execution state differences

between array subscripts are removed by symbolic execution[2]. However, symbolic execution tools either deal with arrays in an infeasible manner that requires extensive user involvement, intense computations or extensive usage of memory resources. Because of these problems, arrays have been excluded from symbolic execution or treated in an inappropriate manner. What is needed is a general technique that would allow integration of arrays in symbolic execution. Such a technique would be able to handle arrays of different kinds, multidimensional arrays, and nested array terms. Dynamic data structures can also be incorporated in symbolic execution theory[8].

Array Values in Symbolic Execution

Many attempts have been made to incorporate arrays in symbolic execution. At one end of the spectrum, the solutions are purely static, and at the other end, they are a mixture of static and dynamic analysis. The static solutions require an actual value when an ambiguous reference is encountered [4]; or, N parallel computations are performed, where N is the size of the array [3]. In [3] an exhaustive case analysis is performed, similar to the unresolved IF statement, in which each time a symbolic reference is encountered, each array element is involved in one of the N path and functional computations. Other static approaches consist of representing the value of the array elements by conditional expressions[9], or considering the case in which the subscripts of the unresolved references are equal or not equal to the subscripts of previous array assignments[16]. In [9], for example, the value of some $A[j]$ might be represented as “if $j = 5$ then y else if $j = 0$ then t else if $j = t - k$ then k else m ”. Note that j , t , k and m might involve array terms themselves. The main problem with the approach in [9] is that at some point, $A[j]$ might be used to resolve some conditional statement; the problem arises for the fact that (potential) array term expressions such as j , t , k and m would be parts of the path condition. The approach in [16] consists of imposing additional constraints (hypotheses) to the path condition so that the number of cases considered grows rapidly as substitution proceeds. Another problem with

the approach in [16] is the fact that the previous array assignment statements might have as right hand sides expressions involving (other) array terms. If that is the case, the resulting path conditions are not array-term-free.

The mixed solutions consist of delaying the substitution of the unresolved references until array subscripts are known[15]. The approach in [15] involves representing the array references in a compact form, evaluating the subscripts with numeric values and considering previous assignments to that array element.

These solutions are, unfortunately, infeasible for arrays of large sizes. Because of the infeasibility of these approaches, arrays have been widely excluded from symbolic execution. Array variables can be bound to a set that contains every symbolic value that the variable can have and the constraints under which each value will hold[8]. The constraints reflect the relationships between the array indices. Rather than keeping the state of all the symbolic values and the constraints under which array variables can have these values at each point in the program, symbolic execution can obtain path and functional constraints containing array terms. It is important to note that all the symbolic execution approaches described so far, except [4], involve array term constraints at some point. Once the constraints are obtained, they could be freed from array terms and given to an equation solver.

In general, array references appear in constraints when expressions to be symbolically executed are calculated using array references. Such constraints present a problem since the relationship between the array terms is not known. For example, symbolic execution of the path 4-5-6-7-8 in the code fragment of Figure 6 introduces the path constraint and the array subscript constraint presented in Figure 5.

The presence of array terms makes the above constraint inadequate for solving. The inadequacy arises from the fact that the array terms are not mathematical variables. But, the array terms could be eliminated by determining which array terms refer to

$$\text{arr}[y+1] + 1 + \text{arr}[x] + \text{arr}[y] > \text{arr}[\text{arr}[y+1] + 1]$$

$$0 \leq x \leq 9$$

$$0 \leq y \leq 9$$

Figure 5. Array term and array subscript constraint.

```

int x, y;
int arr[10];
...
4  scanf("%d %d", &x, &y);
5  x = arr[x] + arr[y];
6  y = arr[y+1] + 1;
7  x = y + x;
8  if (x > arr[y]) {
9      y = y + 1;
10     printf("%d" arr[y]);
11 }
12 else {
13     x = x + 1;
14     printf("%d", arr[x]);
15 }
16 ...

```

Figure 6. Array manipulation program.

For example, the array constraint from Figure 5, could be represented as:

$$J + 1 + K + L > M$$

$$0 \leq x \leq 9$$

$$0 \leq y \leq 9$$

The change of representation from constraints involving array terms to array-free constraints is described in Chapter III.

Array-reference-free constraints can be given to an equation solver. However, solving a general system of equations is an unsolvable problem. Therefore, many constraints resulting from subscript comparisons and array-free constraints will be unsuccessfully attempted regardless of the equation-solving algorithm used. Some statistical techniques for solving general sets of constraints have been proposed; the constraint variables are sampled and the samples are used to inspect the satisfiability of the constraints under consideration[15]. The probability that the set of constraints is not satisfiable is proportional to the number of unsuccessful samples tried, given that no success was observed.

DATA DEPENDENCE ANALYSIS

Much of the data dependence analysis work has been done in the field of parallel and vector processing. Optimizing compilers use data dependence graphs to detect parallelism in programs. Data dependence graphs represent data usage patterns in programs and are a useful source of information underlying program analysis and program testing techniques[18]. Data dependence analysis is necessary in order to perform various program transformations that would allow parallel execution of loop structures as well as taking advantage of architectural features such as cache memories. In order to carry out program restructuring transformations such as loop interchanging, loop skewing and loop rotation, data dependence analysis identifies the legal transformations that are semantically equivalent to the original program[17].

The data dependence problem for arrays can be characterized as follows: given two statements containing array references and a finite number of nested loops surrounding these statements, determine whether the array reference indices could be equal. Given that the array subscripts are linear combinations of the loop indices with constant coefficients, the data dependence problem consists of solving simultaneous diophantine equations [10, 25], which is an unsolvable problem in general. For example, in the code fragment of Figure 7, data dependence between statements 5 and 6 is established if the diophantine equation:

$$2 + 3 * i + 4 * j + 5 * k = 3 + i + j + k$$

has a solution in the integers.

```

int A, i, j, k;
...
2   for(i = 0; i < N; i++)
3   for(j = 0; j < M; j++)
4   for(k = 0; k < L; k++) {
5       A[2 + 3 * i + 4 * j + 5 * k] = A[j];
6       A[j] = A[3 + i + j + k];
7       A[3 + i + j + k] = A[2 + 3 * i + 4 * j + 5 * k];
8   }

```

Figure 7. Program that exchanges two array elements.

However, in general, the loop structure is more complex, consisting of an infinity of paths through the loop, with array subscripts that are not linear combinations of the loop indices.

Consider the Bubble sort function in Figure 8 that sorts an array **Arr** of **N** elements in descending order.

```
void BubbleSort(Arr, N)
    int *Arr, N;
    {
    int temp, i, j;
3    for(i = 2; i < N; i++)
4        for(j = N; j < i; j--)
5            if( Arr[j] < Arr[j + 1]) {
6                temp = Arr[j];
7                Arr[j] = Arr[j + 1];
8                Arr[j + 1] = temp;
9            }
10    }
```

Figure 8: Bubble sort function[24], p. 66.

Suppose we want to determine if $A[j]$ and $A[j + 1]$ at line 7 are data dependent. The array subscripts are linear combinations of the loop indices. Equating the indices and satisfying the equation is not sufficient since data dependence is not only a function of the loop indices, but of the conditional and assignment statements as well. The approach of testing indices for equality without considering the loop details does well when the loop has no additional conditional structures and no assignment statements of the subscript relevant variables. This approach amounts to disregarding the control structure within the loop which could potentially restrict the dependence equation. Furthermore, the assignment statements that appear within the loop body affect the subscripts, and could invalidate the subscript equation.

Input Constraints

The variables from the input domain are constrained by the various program control structures. In particular, for the execution to follow a path through the program, some constraints involving input variables need to be satisfiable in order for the path to be executed. Subscript comparisons involve the equality operator. In this thesis, we are concerned with constraints consisting of symbolic values of program variables and source language (C) operators. Since the dependence analysis is concerned with array subscript comparisons, the solutions to these constraints are searched for in the integers — some of the constraints may be over the reals (the ones involving array terms of type real), but most involve the more difficult integer domain.

DATAFLOW TESTING

Data-flow testing is a path testing strategy in which the paths to be covered are chosen in such a way that they exercise interesting dataflow properties of programs. A path is covered if a test case causes the execution to follow that path. Instead of covering all the paths in a program (full path testing), in data-flow testing the paths are variants of def-use paths for variables.

The precise data flow definitions used in this document are taken from[1]. A *path* is a sequence of nodes (n_1, \dots, n_j) , $j \geq 2$, such that there is an edge from n_i to n_{i+1} , where $i=1, \dots, j-1$. In this definition the path is defined with respect to an uninterpreted control flow graph. A *simple path* is a loop-free path (no node repetitions), but n_1 and n_j may be the same. A *complex path* is a nonsimple path which allows node repetitions. A def-use association for some variable V connects a node in which V is assigned a value (def) and a node (possibly the same one) in which V is used. A def-clear path does not allow redefinition of V in some intermediate node. Given a predicate node p that contains a statement of the form **condition** (x_1, x_2, \dots, x_n) , where **condition** is a conditional predicate involving

variables x_1 through x_n , and two successor nodes of p , i and j , the edges (p, i) and (p, j) contain p -uses of x_1, \dots, x_n . *Array subscript relevant variables* of some array term $A[i]$ are program variables that affect the subscript expression i .

Data flow testing systems usually treat arrays as aggregate objects in which differentiation between distinct array elements has not been made. In this approach, definitions and uses of distinct array elements are treated as references to the same object. For example, in the code fragment of Figure 9 aggregate array analysis would detect def-use dependence between the array pairs 4–5, 4–6 and 7–8. In fact, the only correct def-use dependence pair is 4–8 which is not identified by the aggregate analysis since the path falsely appears as interrupted by the assignment statement at line 7.

```

    int *A, x, y;
    ...
3     scanf("%d", &x);
4     A[x] = y;
5     y = A[x + 1];
6     z = A[x + 2];
7     A[x - 1] = y;
8     printf("%d", A[x]);
9     ...

```

Figure 9. Array manipulation function.

Aggregate data-flow testing suffers from problems of false path inclusion when there are no data dependent paths, and from problems of correct path omission when a path appears to be interrupted by an intermediate assignment when, in fact, it is not. These problems can be eliminated by element-wise analysis. Extensive element-wise analysis is

presented in [2].

CHAPTER III

SUBSTITUTION TECHNIQUE

Symbolic execution does not do well with indexed variables and looping structures. In this Chapter we present a solution to the indexed variable problem. The substitution technique handles array constraints produced by symbolic execution of any fixed path through a program.

Array terms are eliminated from constraints by analyzing the relationship between the array term indices. The relationships determine the kind of mathematical variables needed for the substitution.

The simplest case involves constraints containing a single array term T . Then T is itself unconstrained (except for subscript bounds), and the substitution of T with a variable that does not occur in the equation gives the correct constraint.

Constraints with multiple array terms with the same aggregate name complicate the substitution.

Let ξ be a constraint containing two array terms $A[\Phi_1]$ and $A[\Phi_2]$, where Φ_1 and Φ_2 are arbitrary expressions. To carry out the substitution, two cases need to be considered:

1) $\Phi_1 \neq \Phi_2$ is satisfiable. In this case the array terms might represent different memory locations, and so might hold different values. Therefore the array terms are substituted with distinct simple variables.

2) $\Phi_1 = \Phi_2$ is satisfiable. In this case the array terms might represent the same memory

location, and hence must have a single value. Therefore the array terms must be substituted with the same simple variable.

If case 1) holds, distinct simple variables should be substituted for both terms. If case 1) does not hold and case 2) holds, the same simple variables should be substituted for the array terms. It is important to note that the solution set of the substituted constraint in case 2) forms a subset of the solution set of the substituted constraint in case 1). Considering case 2) when case 1) is satisfiable might introduce an unsatisfiable substituted constraint when, in fact, case 1) produces a satisfiable substituted constraint. Since every solution that satisfies the constraint in case 1) satisfies the constraint in case 2), the more general case 1) is used, and case 2) need not be tried. For example, given the constraint

$$\mathbf{brr}[i] + 1 = \mathbf{brr}[j], \text{ where } i \text{ and } j \text{ are symbolic values,}$$

the constraint $i = j$ is satisfiable, so $\mathbf{brr}[i]$ and $\mathbf{brr}[j]$ might be substituted with the same simple variable I . The substituted constraint

$$I + 1 = I$$

has no integer solutions. The case 1) substitution constraint $i \neq j$ is satisfiable, and the substituted constraint

$$I + 1 = K$$

is satisfiable, where I is substituted for $\mathbf{brr}[i]$ and K for $\mathbf{brr}[j]$.

MULTIPLE ARRAY TERMS

Constraints that involve array terms with a single aggregate array name are called homogeneous array term constraints. Heterogeneous array term constraints are constraints with multiple aggregate array name terms. For example, the constraint

$$A[i] + B[j] - A[B[j]] = C[k]$$

is a heterogeneous array term constraint, whereas the constraint

$$A[i] + A[A[j]] > A[k]$$

is a homogeneous array term constraint.

This section describes proper substitution of homogeneous, heterogeneous and nested array terms.

Homogeneous Array Terms Constraints

Constraints that involve array terms with identical aggregate array names are called homogeneous array term constraints. As the number of array terms in an equation increases, the correct substitution is determined by considering whether all the subscripts may differ, in which case all the array terms are substituted with distinct variables. If all the subscripts may not be different, the ones that may not be different, but may be the same, are identified and substituted with the same variable. Given n array terms $A[\Phi_1]$, $A[\Phi_2]$, ..., $A[\Phi_n]$ contained in some array equation, where $\Phi_1, \Phi_2, \dots, \Phi_n$ are arbitrary expressions, two cases need to be considered:

- 1) The term $A[\Phi_i]$ is given a distinct simple variable if for all $\Phi_j, i \neq j, \Phi_i \neq \Phi_j$ is a satisfi-

able constraint. In this case the array term $A[\Phi_i]$ might represent different memory location than the rest of $A[\Phi_j]$ terms in the equation, and so might hold a different value. Therefore the array term is substituted with a distinct simple variable; one not used to substitute any of the $A[\Phi_j]$ terms. Each array term $A[\Phi_i]$ for which case 1) holds is substituted with a distinct simple variable. In the extreme situation in which the constraint $\Phi_1 \neq \dots \neq \Phi_n$ is satisfiable, $A[\Phi_1], A[\Phi_2], \dots, A[\Phi_n]$ are substituted with distinct variables and no further analysis is needed.

2) The term $A[\Phi_i]$ is given some simple variable V used to substitute some $A[\Phi_j]$, $i \neq j$, if $\Phi_i \neq \Phi_j$ is not satisfiable, but $\Phi_i = \Phi_j$ is satisfiable. The array terms $A[\Phi_i]$ and $A[\Phi_j]$ whose subscripts may be the same, but not different are substituted with the same simple variable. Each array term $A[\Phi_i]$ for which case 2) holds is substituted with a variable used to substitute some other $A[\Phi_j]$ term.

Giving a distinct variable to some $A[\Phi_i]$ requires that all the possible pairs of subscripts are compared, because of the nontransitivity of the \neq operator. In general, $\Phi_1 \neq \Phi_2 \wedge \Phi_2 \neq \Phi_3 \Rightarrow \Phi_1 \neq \Phi_3$ is a false implication. For example, given the array equation:

$$A[i] + A[j] = A[i], \text{ where } i \text{ and } j \text{ are symbolic values,}$$

$$i \neq j \wedge j \neq i \Rightarrow i \neq i \text{ is a false implication.}$$

Giving a same simple variable to some $A[\Phi_i]$ does not require that all the possible pairs are compared. Since equality is transitive, there is no need to compare each pair; considering only one pair of array terms with subscripts that must be the same and not different, determines the substitution outcome for the pair. For example, consider the fol-

lowing array equation in which the homogeneous array terms are numbered for easier identification. The substitution analysis of the array equation:

$$A_1[i + 1] - A_2[j] + A_3[j] + A_4[i] = A_5[j]$$

where i and j are symbolic values.

The substitution analysis of the array equation determines that i in $A_4[i]$ and $i + 1$ in $A_1[i + 1]$ can be different from the rest of the subscripts, so $A_4[i]$ and $A_1[i + 1]$ are given variables that are not used for substituting any other array term in the equation. To substitute the $A_3[j]$ term, the subscript in the $A_3[j]$ term need only be compared to either one of the subscripts in the $A_2[j]$ or $A_5[j]$ term, because of the transitivity of the $=$ operator.

Nested Homogeneous Array Terms

In the case where the symbolic expression has array terms as subscripts to arrays, the substitution technique is applied recursively. In particular, let $A[\Phi]$ be an array term such that the subscript expression Φ is a function of n nested expressions, $\Phi_1, \dots, \Phi_{n-1}$, where each Φ_i , $i = 1, \dots, n-1$, is an arbitrary (*proper*) expression containing an array term whose subscript is Φ_{i+1} , and Φ_n is an array-term-free expression. An expression is proper if it contains a single array term. The technique first substitutes the most nested array term $A[\Phi_n]$, substituting it with a simple variable that becomes part of the array index expression Φ_{n-1} of the nesting array term $A[\Phi_{n-1}]$. The array-free expression Φ_{n-1} is constrained to the array bounds and used for further subscript comparisons. Consider the following constraint in which the homogeneous array terms are numbered for easier identification:

$$1 + Arr_1[Arr_2[i + 1]] = 1 + p$$

where l is symbolic value, $\Phi = \text{Arr}_2[\Phi_1]$ and $\Phi_1 = i + 1$.

Initially, the most deeply nested array term $\text{Arr}_2[i + 1]$ is substituted with some I . The constraint becomes

$$1 + \text{Arr}_1[I] = 1 + p$$

with I being the subscript of the nesting array term Arr_1 . An additional constraint stating that I is in array bounds is added to the set of constraints. Next, $I \neq i + 1$ is added to the set of constraints, indicating that Arr_1 and Arr_2 could represent different memory locations, and $\text{Arr}_1[I]$ is substituted with K , giving

$$1 + K = 1 + p$$

which has $K = p$ as a solution.

Substituting an array equation that has both nested and multiple array terms does not complicate the technique since extensions of cases 1) and 2) from the previous section are used. Whenever some $A[\Phi_i]$ is to be substituted, cases 1) and 2) are considered with respect to the whole substitution state; Φ_i is compared to the rest of the subscripts until either some Φ_j , $i \neq j$, is found for which $\Phi_i = \Phi_j$ holds and $\Phi_i \neq \Phi_j$ does not hold, or there is no Φ_j for which $\Phi_i = \Phi_j$ holds. Because the substitutions involve considering the whole substitution state, the satisfiability of the substituted equation is independent of the order of the substitution.

Heterogeneous Array Terms

Array terms with distinct aggregate array names (array terms of different kinds) are treated independently from array terms with same aggregate array names (array terms of one kind). Care needs to be taken not to confuse simple variables used for substitution of one kind with variables used for another kind. The same independent treatment applies to any combination of heterogeneous terms such as nested terms of different kinds. A *substitution variable set* is a collection of simple variables used for substitution of array terms of one kind. Given two nonempty sets of substitution variables S_{v1} and S_{v2} , and an array equation (to be substituted) with two array kinds A and B, the substituted constraint could be unsatisfiable if S_{v1} and S_{v2} are not disjoint; substituting with the same variable, when in fact, distinct variables could be substituted, could make a satisfiable constraint unsatisfiable. Since the elements of distinct array kinds represent different memory locations, different substitution variables could always be given. Note that there are cases in which S_{v1} and S_{v2} are not disjoint and the resulting substituted constraint is satisfiable. For example, $A[i] + 1 = B[j] + 1$ becomes $I + 1 = I + 1$ when $A[i]$ and $B[j]$ are substituted with the same variable I. However, the substitution sets of variables should always be disjoint.

In one-dimensional array equations for which the total number of array terms (n) is greater than the size of the array (N), N substitution variables are needed in the worst case, the case where N subscripts may differ. The case in which all the subscripts may differ requires $O(n^2)$ comparisons, exactly $(n^2 - n) / 2$ comparisons.

CHAPTER IV

ARRAY SUBSTITUTION

The goal of array substitution is to produce equations free from array terms. The substitution description consists of two parts, a set of substitution pairs called the *substitution state* and a predicate called the *substitution predicate*. Each $\langle \text{Ter}, \text{Var} \rangle$ pair in the substitution state represents a binding of an array term Ter with a variable Var . Ter consists of two parts: the array name A , and the array subscript of A , A_{sub} . Var is a simple variable used to substitute Ter . Let ξ be a set of constraints describing the relationship between A_{sub_i} and A_{sub_j} , $i \neq j$, for all the $\langle \text{Ter}, \text{Var} \rangle$ pairs in the substitution state. Let ε be a set of equations to be substituted. The *substitution space* is the Cartesian product $\sigma = \varepsilon \times \xi \times \eta$, where η is the substitution state. The substitution predicate is a transformation $\tau: \sigma \rightarrow \sigma$.

This Chapter describes the implementation of the substitution predicate in a constraint Prolog.

EVALUATION OF EXPRESSIONS

Solving a set of constraints requires a constraint solver. Constraint Logic Programming (over the field of Real numbers) combines logic programming with a constraint solving engine. Symbolic execution can be carried out by logic programming, and the constraints produced during symbolic execution can be solved by CLP(R)[5]. The array subscript constraints require solutions in the integer domain. An algorithm that determines whether a set of linear equations has any solutions, and gives a way to enumerate the solutions is needed[11].

Each equation in the set of constraints is considered as a collection of Prolog data objects; variables are treated as atoms and the operators are modified in order to avoid interpretation as Prolog operators. The modification is accomplished by inserting the \sim symbol in front of the Prolog operators. This serves as guard against premature solving of the parts of the equations. After the array terms are eliminated from the set of constraints, the substituted constraints are converted to CLP(R) variables and operators, which CLP(R) attempts to satisfy.

The equations given to the substitution predicate have a data form like $a::b \sim + a::b \sim + 1 \sim = a::a::b$ ($::$ is the array subscript operator). The corresponding array equation expressed in C-like syntax is $a[b] + a[b + 1] = a[a[b]]$. The substitution predicate returns substituted equations that do not contain the array subscript operator ($::$); $i \sim + i \sim + 1 \sim = j$, where $a::b$ and $a::a::b$ are substituted with i and j respectively. CLP(R) can solve the corresponding real equation $I + I + 1 = J$. To obtain the array free equation, the predicate `substitute(E, Res, State)` is used, where **E** is a set of constraints, **Res** is the substituted set of constraints and **State** is the substitution space.

The transformation from the meta constraints to the CLP(R) constraints is accomplished by the `trans(L, E, Tequ)` predicate. **E** is array term free equation. **Tequ** is the CLP(R) constraint created during the evaluation of **E**. During the evaluation of **E**, the meta operators are replaced by CLP(R) operators and the interpreted equations are attempted for satisfiability. The **L** argument holds the CLP(R) variables used to represent program variables. The transformation predicate is given in Figure 10. The first predicate in Figure 10 describes how to evaluate expressions that contain the $\sim +$ operator; the expressions on both sides of the equality operator are translated recursively, forming a CLP(R) equality between the translated expressions. The second rule considers the $(\sim \rightarrow)$ constraint. The third and the fourth rules translate variables and constants, respectively. The `trans` predicate in the third rule binds constraint variables to CLP(R) variables. A constraint set involving three variables: `low`, `medium` and `equal` would have the predicates of Figure 11.

```

trans(L, A ~ +B, Out) :-
  trans(L, A, AA),
  trans(L, B, BB),
  Out = AA + BB.

```

```

trans(L, A ~ >B, Out) :-
  trans(L, A, AA),
  trans(L, B, BB).
  Out = AA > BB.

```

```

trans(L, A, Out) :-
  atom(A),
  translate(L, A, Out).

```

```

trans(L, A, A) :-
  real(X).

```

Figure 10. Predicate trans.

```

translate([Var1, Var2, Var3], low, Var1).
translate([Var1, Var2, Var3], medium, Var2).
translate([Var1, Var2, Var3], high, Var3).

```

Figure 11. Translation of atoms to CLP(R) variables.

The first predicate in Figure 11 associates the CLP(R) variable **Var1** to the constraint variable **low**. Similarly, the other two predicates bind **medium** and **high** to **Var2** and **Var3**, respectively. The query:

?- trans([A, B, C], medium \sim > low \sim - high, Res).

causes the CLP(R) constraint $A > B - C$ to be attempted for satisfiability, and returns the same constraint in Res.

ARRAY SUBSTITUTION

The predicate `substitute(E, Res, State)` describes the transformation of equation E to the resulting, substituted equation Res. During the transformation process, the substitution state (η) and the set of array subscript constraints (ξ), which are kept internally by CLP(R), change. Initially, η and ξ are both empty. To substitute an equation $E1 = E2$, the resulting equation is obtained by concatenating the substituted lefthand side, equal sign and the substituted righthand side. In general, the result from substituting the expression $E1 \text{ op } E2$ is `concat(Res1, concat(op, Res2))`, where Res1 and Res2 are obtained from `substitute(E1, Res1, S)` and `substitute(E2, Res2, S)`, respectively. To substitute an expression `arr[x]`, first x is substituted, and then the whole array term `arr[Res]` is substituted, where Res is the result of the subscript substitution.

The substitution state contains the bindings of simple variables to array terms. Whenever an array term $A[j]$ is to be substituted, the substitution state is searched for an array term $A[i]$ such that the constraint $i = j$ is satisfiable and $i \neq j$ is not. If the search succeeds, $A[j]$ is substituted with Var_i where $\langle A[i], \text{Var}_i \rangle$ is an element of the substitution state. An unsuccessful search causes $A[j]$ to be substituted with some Var_j such that $\text{Var}_j \notin \eta$.

Substitution Algorithm

The substitution algorithm describes how sets of array term constraints are transformed in to array-free constraints. The main predicates used are: `substitute`, `eliminate_arr`, `replace` and `find_var`. The auxiliary predicates are: `simple_var`, `trans`, `translate`, `in_bounds` and `append`.

The equations to be manipulated by the substitution predicates contain modified operators so that CLP(R) will not attempt to prematurely solve parts of the equation. The **substitute** predicate from Figure 12 takes a set of array term equations **E**, the resulting, substituted equation set **Res** and the substitution state **State**.

```

substitute(E, Res, State) :-
  simple_var(Symbols),
  eliminate_arr(E, Res, [], State, Symbols).

```

Figure 12. Substitute predicate.

The **simple_var** predicate binds the CLP(R) variable **Symbols** to a list of simple variable symbols used to replace array terms. The **eliminate_arr** predicate from Figure 13 takes a list of data object equations **E**, a list of substituted equations **Res**, empty state space, state space after the substitution of **E** and a set of simple variables.

```

eliminate_arr([], [], State, State, _).

eliminate_arr([E|TE], [Res|Tres], State, NewState, Symbols) :-
  replace(E, Res, State, MState, Symbols, Not_used_Symb),
  eliminate_arr(TE, Tres, MState, NewState, Not_used_Symb).

```

Figure 13. Eliminate array terms from a set of equations.

First the head equation is replaced with simple variables, then the array elimination predicate is applied to the tail equations.

The **replace** predicate takes an equation of data objects **E**, the substituted equation **Res**, list of initial state bindings **BState**, list of final state bindings **FState**, source of initial substitution symbols **BSymb** and a list of remaining simple variables after the sub-

stitution **FSymb**. The first predicate in Figure 14 is the base case of variables and constants; atoms need not be substituted. The third predicate constructs the resulting equation by substituting array terms in the left and right subexpression recursively. The second predicate recursively replaces nested arrays, and searches the substitution space for a simple variable. The `=..` Prolog operator is used to decompose the array equation **E** into operator **Op**, left subexpression **L** and right subexpression **R**. After **L** and **R** have been substituted recursively, the resulting equation **Res** is constructed by `=..` operator.

```
replace(E, E, S, S, Symb, Symb) :-
```

```
    atomic(E).
```

```
replace(A::X, Res, BState, FState, BSymb, FSymb) :-
```

```
    replace(X, Xsub, BState, MState, BSymb, MSymb),
```

```
    find_var(A::Xsub, Res, MState, FState, MSymb, FSymb).
```

```
replace(E, Res, BState, FState, BSymb, FSymb) :-
```

```
    E = .. [Op, L, R],
```

```
    replace(L, LRes, BState, LState, BSymb, LSymb),
```

```
    replace(R, RRes, LState, FState, LSymb, FSymb),
```

```
    Res = .. [Op, LRes, RRes], !.
```

Figure 14. Replacement predicate.

The `find_var` predicate takes the array to be substituted **A::X**, the resulting simple variable **Simple_var**, list of initial bindings, list of final bindings, list of initial simple variables and a list of final simple variables. The first predicate in Figure 15 is considering the case in which the state space is empty; then the array term **M::X** is substituted with a fresh symbol **Simple_Var** from the symbol list. The new state space contains the pair

$\langle M::X, \text{Simple_Var} \rangle$. The second rule checks if all the subscripts may differ, in which case $M::X$ is given a symbol that has not been previously used for substituting some array term. The third predicate substitutes $M::X$ with a variable that has already been used to substitute some $M::Y$ since $X = Y$ is a satisfiable constraint. The fourth predicate handles the case of multiple heterogeneous array terms; the subscripts are not compared since the memory locations of distinct arrays are disjoint.

A CLP(R) query with multiple homogeneous terms like:

$?- \text{substitute}([a::(b \sim +1) \sim +1 \sim > a::b \sim +2 \sim *a::b], \text{Res}, \text{State}).$

returns with *yes* and gives the following result:

$\text{Res} = [i \sim +1 \sim > j \sim *k]$

$\text{State} = [a::b \sim +1, j, a::b \sim +2, j, a::b, k].$

The query with multiple heterogeneous terms:

$?- \text{substitute}([(a::b \sim +1) \sim +1 \sim > a::c \sim +2 \sim *b::a], \text{Res}, \text{State}).$

returns with *yes* and gives the following result:

$\text{Res} = [i \sim +1 \sim > j \sim *k]$

$\text{State} = [a::b \sim +1, j, a::c \sim +2, j, b::a, k].$

The nested array query:

$?- \text{substitute}([a::(a::b \sim +1) \sim +1 \sim > a::c], \text{Res}, \text{State}).$

returns with *yes* and gives:

$\text{Res} = [j \sim > k]$

$\text{State} = [a::b \sim +1, j, a::i \sim +1, j, a::c, k].$

```

find_var(M::X, Simple_var, [], FState, [Simple_Var[T], T) :-
    append([M::X, Simple_var], [], FState).

```

```

find_var(M::X, Simple_var, [M::Y, Var[T], [M::Y, Var[FState]], BSymb, FSymb) :-
    trans(L, X, Xeval),
    trans(L, Y, Yeval),
    in_bounds(M, Xeval),
    X <> Y,
    find_var(M::X, Simple_var, T, FState, BSymb, FSymb).

```

```

find_var(M::X, Simple_var, [M::Y, Simple_var[T], FState, FSymb, FSymb) :-
    trans(L, X, Xeval),
    trans(L, Y, Yeval),
    in_bounds(M, Xeval),
    Xeval = Yeval,
    append([M::X|Simple_var], [M::Y, Simple_var[T], FState).

```

```

find_var(M::X, Simple_var, [D::Y, Var[T], FState, BSymb, FSymb) :-
    find_var(M::X, Simple_var, T, MState, BSymb, FSymb),
    append([D::Y, Var], MState, FState).

```

Figure 15. Search predicate.

The auxiliary predicates are given in Figure 16. The `in_bounds` predicate requires a fact `size(Array, N)` to be present in the knowledge base; `Array` is the array name and `N` is the size of `Array`. The `append(List1, List2, ResList)` predicate is true if `List1` concatenated to `List2` gives `ResList`. The `<>` operator is the “not equality” operator. The

semantics of the CLP(R) “negation as failure” predicate does not allow its use as a disequality (\neq) operator. For example, the query:

?- not(I = I).

returns *no* as expected, but the query:

?- not(I = J).

returns *no* as well, when the constraint $I \neq J$ is satisfiable.

in_bounds(Array, Subscript) :-

size(Array, N),

Subscript >= 0,

Subscript < N.

append([], List, List).

append([H|T], L, [H|Z]) :-

append(T, L, Z).

Sub1 <> Sub2 :-

not_equal(Sub1, Sub2).

not_equal(Sub1, Sub2) :-

Sub1 > Sub2.

not_equal(Sub1, Sub2) :-

Sub1 < Sub2.

Figure 16. Auxiliary predicates.

In this Chapter we presented the substitution algorithm that describes how array terms are eliminated from constraints. The algorithm deals with constraints involving array terms of the same kind, array terms of different kinds and nested array terms.

CHAPTER V

INFINITY OF PATHS

ANOMALIES IN CONSTRAINTS

The deficiencies of symbolic execution and path analysis for loops carry over to the substitution technique. The array indices, after being symbolically executed, are expressed in terms of input variables, which could assume any value in the input domain. But, the symbolic index expressions might be incorrect depending on the technique used to eliminate the infinity of possible paths through loops. Consider the code segment in Figure 17. The symbolic expression for k at line 12 should be 0, but if the paths through the loop are restricted to simple paths[1] then a *symbol anomaly* occurs at line 12. Informally, a simple path is a sequence of nodes such that all the nodes in the sequence are distinct except, possibly, the starting and the terminating node. A symbol anomaly occurs when a symbol is falsely treated as unconstrained. A *data dependence anomaly* occurs when a variable is used without being first assigned a value[2]. When symbolic execution produces constraints with data dependence or symbol anomalies, the confusion with input variables can invalidate the substitution analysis described in Chapter III. For example, to determine def-use dependence between the definition of $A[j]$ at line 7 and the use of $A[x]$ at line 13, on the simple path 7–8–9–10–11–12, the following constraint is checked for satisfiability:

$$A[0] + 2 = A[k] + 1, \text{ where } k \text{ is a symbolic value.}$$

```

    int *A, i, j, k, x, y;
    ...
5     j = 0;
6     j = A[j] + 2;
7     A[j] = 0;
8     for(i = 0; i < 5; i++, j++) {
9         if(i == 4)
10            k = 0;
11    }
12    x = A[k] + 1;
13    y = A[x];

```

Figure 17. Array manipulation function.

The subscript k falsely appears as an unconstrained input symbol. The substitution constraint $0 \neq k$ is satisfiable resulting in substitution of different simple variables for $A[0]$ and $A[k]$. The substituted constraint $I + 2 = K + 1$ where $A[0]$ and $A[k]$ are substituted with I and K respectively, is satisfiable, and a false data dependence between statements at lines 7 and 13 is identified by the simple path data-flow analysis. Considering the loop the correct number of traversals leads to the the correct constraint

$$A[0] + 2 = A[0] + 1$$

for which the substituted constraint $I + 2 = I + 1$ has no solutions in the integers.

In programs involving arrays, the usual simple path data-flow approach is inadequate. For establishing data dependence in the array case, simple paths do not suffice in data dependence analysis. In general, all the paths (simple and complex) through loops

need to be considered to determine the correct data dependence. However, for programs with infinite execution trees the number of paths is infinite. With the goal of improving the data dependence analysis, we investigate two previously proposed heuristic techniques: partial symbolic execution and explicit naming of nodes. Then we propose a new heuristic technique: loop forcing.

Partial Symbolic Execution

Program verification techniques deal with infinite execution trees by inserting inductive assertions at certain program points, allowing proof of correctness using symbolic execution[13]. However, these assertions are often quite difficult to discover. Only for a limited set of programs can generation of inductive assertions be automated[19, 20]. Symbolic execution does not handle loops well; in general, the algebraic description of the loop cannot be produced. To handle loops a more general technique is needed.

Partial symbolic execution is a heuristic technique that constructs a regular expression representing possible program paths. Then, a generalized form of symbolic execution is applied to the regular path expression. This generalized form of symbolic execution derives a generalized algebraic description of the function corresponding to the regular path expression. The algebraic description is generalized in the sense that loop details are sacrificed, but a necessary condition for some path to be executed is derived. In general, partial symbolic execution determines:

- 1) The conditions under which a set of paths P will be executed (partial path condition).
- 2) The constraints on the relationship between the initial and final states when any $p \in P$ is executed.

Consider the code fragment in Figure 18.

```

    int x, y;
    ...
3   x = 0;
4   y = 1;
5   while(y != 0){
6       scanf("%d", &y);
7       x = x + y;
8   }
9   if(y > 20)
10      printf("%d", x);
11  else
12      printf("%d", x + 5);

```

Figure 18. Partial symbolic execution.

The path condition derived by partial symbolic execution for the path expression (3-4-(5-6-7)*-8-9-10) is $Y > 20$, where the subexpression (5-6-7)* identifies zero or more occurrences of the subpath 5-6-7, and Y is a fresh symbol corresponding to y . Path computation relates the symbol representing y (y_i) in its initial state, at line 4, to the symbol representing y (y_f) in the final state, at line 9 through the fresh symbol Y . Since $Y > 20$ and $y_f = Y$, it can be concluded that $y > 20$.

Partial symbolic evaluation of a path expression which represents a simple path is identical to symbolic execution of that path. During the evaluation of path expressions that represent more than one path (paths through loops), fresh symbols are assigned to any variable that is potentially redefined on any of the paths represented by the path expression. At the end of the loop sub-path expression, the redefined variables appear unconstrained when, in fact, they might be constrained depending on the loop details. These

falsely unconstrained variables when involved in constraints might invalidate the substitution technique described in Chapter III. For example, consider the code fragment in Figure 19.

```

    int *A, x, y, n;
    ...
2   scanf("%d", &x, &y);
3   y = A[1];
4   A[y] = x;
5   for(i = 0; i < n; i++){
6       ...
7       x = 1;
8   }
9   if(A[x] > A[1])
10  ...

```

Figure 19. Deficiencies of PSE.

If combined with array conditions such as “if(A[x] > A[1])” and used to derive the path condition for the path (3–4–(5–6–7)*–8–9–10), partial symbolic execution would produce the constraint $A[x] > A[1]$ instead of the correct constraint $A[1] > A[1]$. The incorrect constraint would lead the substitution technique to give distinct simple variables to both array terms since x falsely appears as an unconstrained variable. In general, whenever unconstrained variables appear in constraints, the substitution technique might be invalidated.

Partial symbolic evaluation is further complicated by path expressions representing loops involving array terms. Assigning a fresh symbol to an array term appearing on some path from the path expression is inadequate because it accounts to treating the ar-

rays as aggregates instead of element-wise. For example, in the code segment of Figure 20, the array references $A[x]$ and $A[y]$ at lines 4 and 9 respectively, could represent both the same or different memory locations depending on the path chosen.

```

    int x, y, *A;
    ...
3   scanf("%d", &x, &y);
4   A[x] = y;
5   while(x > 0) {
6       if(x > y)
7           y = x;
8       else
9           A[y] = x;
10  }
11  printf("%d", A[A[y]]);
12  ...

```

Figure 20. Deficiencies of PSE.

Given two array references $A[\Phi_1]$ and $A[\Phi_2]$ in a complex path expression, the decision whether to give distinct or the same variables depends on the relationship between Φ_1 and Φ_2 . To determine the relationship between Φ_1 and Φ_2 , a fixed path needs to be chosen, and this is exactly what partial symbolic evaluation does not do.

Partial symbolic execution appears to be of limited help when dealing with derivation of path constraints for programs involving array terms in an element-wise manner. The necessary path conditions given by partial symbolic execution may contain falsely unconstrained symbols, which might invalidate the substitution technique of Chapter III. Furthermore, partial symbolic execution is not well suited for deriving path conditions of

path expressions representing paths involving array terms. Since the array indices are potentially different on each loop iteration, different paths through a loop could involve different array elements. Therefore, assigning fresh symbols to array terms is inappropriate.

Explicit Naming of Paths

The static analysis of data dependencies for programs with simple variables requires only simple paths to be considered. Complex paths need not be considered since they either cause redefinition of the variables in consideration, or they introduce paths irrelevant to the data dependence association. To illustrate the former case consider the code segment in Figure 21 and a possible def-use of y between line 5 and 12. Paths obtained by multiple iteration through the loop are not definition clear for y . An example of the latter case is given Figure 22; the def-use of x between lines 5 and 12 is not affected by the loop.

```

    int x, y;
    ...
5   scanf("%d %d", &x, &y);
6   while(x) {
7       scanf("%d", &x);
8       y = y + x;
9   }
10  x = x + 1;
11  x = y + 2;
12  printf("%d", &y);
13  ...

```

Figure 21. Irrelevant complex paths.

```

    int x, y, N, i;
    ...
5      x = x + 1;
6      y = x + 1;
7      for(i = 0; i < N; i++) {
8          scanf("%d", &y);
9          y = 2 * y;
10     }
11     if (y != 0)
12         y = x + 1;
13     ...

```

Figure 22. Irrelevant complex paths.

In general, considering only simple paths gives rise to an intuitive anomaly. Static analysis would find no data dependent paths when, in fact, complex paths exist that establish data dependence[7]. This anomaly occurs because a simple path connecting a du association is not feasible, and there is a complex feasible path that connects the du association. The problem, in this case, is to determine the number of loop traversals that would make two references of some variable lie on some feasible nonsimple path. The problem of determining whether two array indices can ever be the same is an extension of the infeasible simple path problem — how many loop traversals are needed for the subscripts to be equal? In general, both of these problems are unsolvable.

To deal with the infeasible simple path problem, a technique that explicitly names data dependent paths has been suggested[7]. From the program control flow graph, the static analysis names sequences of nodes which represent potential du paths through loops. For example in Figure 23, the sequences of nodes Seq1 = 1 2 3 4 5 6 7 4 12 13 and Seq2 = 1 2 3 4 5 6 7 4 5 6 7 4 12 13 are paths for the same dupath from "x = 0;" to

`”printf(”%d”,x);”`. The identification is easy since they both contain a cycle with the same start and end node 4. The sequence name for the dupath designated by Seq1 and Seq2, `”1...4...13”` is obtained by excising the cycle. Dynamic counting involves noting a def of a variable at node 1, accumulating the node sequence until control is in the loop at loop point 4 and discarding the sequence, collecting sequences of tail nodes until the use of the same variable is encountered at node 13. Then the dupath `”1...4...13”` is identified as covered. The dupath names are created statically and counted dynamically. Redundant loops such as `4 5 6 7 4 5 6 7 4` in Seq2 are excised by a check sum function that hashes the sequences named to different numerical path names. The check sum function has the property that two sequences representing the same dupath, such as Seq1 and Seq2, have the same numerical check sum value. The precise counting of dupaths is sacrificed (the dupaths are undercounted) because of the collisions occurring from the check sum function. However, this is a deficiency associated with the implementation of the sequence naming approach, rather than with the approach itself.

This technique amounts to assuming that the variable under consideration is on a complex data dependent path which is to be identified statically and covered dynamically. This approach suffers from the infeasible path problem which is, in general, intractable. In particular, there might not be a data dependent path between two variable references regardless of the number of loop traversals. If this technique were used to determine whether a du pair is on some dupath, semi-decidable problems arise: a potential infinity of test points could be given that exercise arbitrary paths through loops, and it is hoped that some of the dynamically accumulated sequences would match the statically established sequence of nodes.

Whenever a data dependence between some pair $A[\Phi_1]$ and $A[\Phi_2]$ that occurs on a complex path is to be detected, not only must some complex path be found that connects them, but assurance that the same memory locations are involved is needed. The array indices for the array pair could be dynamically checked for equality.

```

    int x, y;
    ...
1   x = 0;
2   positive = 0;
3   scanf("%d", &y);
4   while(y != 0) {
5       if(y > 0) {
6           scanf("%d", &y);
7           positive++;
8       }
9       else
10          y = -y;
11    }
12    if(positive > 0)
13        printf("%d", x);
14    ...

```

Figure 23. Node repetition required for data dependence.

This means that the same node naming technique could be used for establishing data dependence between array variables, but it would correspond to an unbound search for a complex path that establishes data dependence. Unfortunately, there is no assurance that such a search would ever terminate, since a pair of array terms might not represent the same memory location. For example, consider the code segment in Figure 24. There are complex paths such as 2-3-4-5-6-7-8-5-6-7-8-9 that connect the pair $A[x]$ at line 3 and $A[1]$ at line 9, but none of these paths establishes data dependence between the pair. Apart from the problem of the infeasible path problem, inclusion of arrays introduces an additional

problem of infeasible subscript constraints. Explicit naming of paths if used to determine data dependence suffers from semi-decidable problems. Potential paths are identified statically, and it is hoped that the test points cover the statically identified paths. When there are no feasible paths, a potential infinity of test points must be tried. Array terms further complicate the technique; not only paths that connect two array terms should be covered, but covered with the right values — the ones that cause the same array element to be involved. If no such values exist (infeasible array subscripts), then potential infinity of test points need to be tried to establish infeasible array subscripts.

```

        int *A, x;
        ...
2       x = 0;
3       A[x] = 0;
4       x++;
5       while(i != 0){
6         ...
7         x++; }
8       if(x > 1)
9         x = A[1];

```

Figure 24. Infeasible array data dependence.

PARTIAL PATHS

Symbolic execution is effective for simple paths, the paths that are considered in def-use analysis of simple variables. However, simple paths are not sufficient for analysis of programmes with arrays[2]. In programs with loops, the problem of determining whether two array indices can ever be the same is undecidable. The problem arises from the fact that the array indices might be equal after an arbitrary number

of loop traversals, and determining the number of loop traversals needed for two indices to be equal is, in general, undecidable. Compiler optimization and parallelization techniques introduce restrictions on the loop structure and array indices so that the existence of loop indices that lie within index limits such that the array subscript expressions are simultaneously equal can be determined[10]. Rather than ignoring the loop control structure and loop path details, and imposing severe restrictions on the loop structure and the array indices, we consider partial paths through loop structures and array subscript constraints obtained on such partial paths.

In general, obtaining the correct data dependencies requires considering all the paths in a program. However, some programs require only a subset of the possible paths to be considered in order to establish precise data dependence. These programs contain redundant paths with respect to data dependence. Identifying redundant paths, as well as the minimal set of paths needed to assure certain semantic properties, is in general undecidable. We expect to improve the precision of the data dependence analysis over one that does not take any such paths into consideration, by separating array subscript constraints from the loop path constraints. In this section, we introduce the necessary vocabulary, and the implications of this approach are considered in the next section.

A *loop path constraint*(LPC_p) is a constraint obtained by symbolic execution of some path through a loop that has p as a subpath. *Partial paths*(PP) are all the simple paths that connect two array references. A *partial path constraint*(PPC_p) is a path constraint produced by symbolic execution of some partial path p . An *array subscript constraint*(ASC_p) is a constraint obtained by equating two array indices *normalized* on some path p . Two array indices Φ_1 and Φ_2 are normalized on path p when the execution state differences between Φ_1 and Φ_2 are removed by symbolically executing the indices on p . For example, the analysis of the code fragment in Figure 25 produces the following constraints:

$$\text{PPC}_{3-4-5-6-7} = x > 0 \wedge x > y$$

$$\text{ASC}_{5-6-7} = x = x$$

$$\text{LPC}_{4-5-6-7-11-4-5} = x > 0 \wedge x > y$$

where x and y are symbolic values.

```

int *A, x, y, z;
...
3   scanf("%d %d", &x, &y);
4   while(x > 0) {
5       A[x] = x;
6       if(x > y)
7           z = A[x] + 1;
8       else {
9           z = A[x + 1] + 1;
10          A[x + 1] = z;
11      }
12      if(x > y)
13          z = A[x + 1];

```

Figure 25. Complex paths resulting from a loop.

Given two array references $A[\Phi_1]$ and $A[\Phi_2]$ and the set PP of simple paths connecting them, three cases are considered:

- 1) Both, PPC_p , and the $\text{ASC}_p \Phi_1 = \Phi_2$ are satisfiable for some $p \in PP$.
- 2) There is no $p \in PP$ for which PPC_p is satisfiable.
- 3) There is some $p \in PP$ for which PPC_p is satisfiable, but the $\text{ASC}_p \Phi_1 = \Phi_2$ is not satisfi-

able.

Case 1) establishes the data dependence between $A[\Phi_1]$ and $A[\Phi_2]$ on some partial path p . Case 2) establishes non-existence of some feasible partial path connecting the array references $A[\Phi_1]$ and $A[\Phi_2]$. Case 3) indicates that the subscript constraint is not satisfiable on any partial path, but there might be some complex path on which $A[\Phi_1]$ and $A[\Phi_2]$ are data dependent. Consider the example in Figure 25. Case 1) holds for the path 3-4-5-6-7 and array terms at statements 5 and 7: the constraints $PPC_{3-4-5-6-7}$ and $ASC_{3-4-5-6-7}$ are simultaneously satisfiable. Case 2) does not hold for the association $A[x + 1]$ at lines 10 and 13. In particular, the only simple path through the loop that connects lines 10 and 13 has unsatisfiable $PPC_{6-8-9-10-11-12-13} x > y \wedge x \leq y$, and therefore there is no simple feasible path connecting statements 10 and 13. The path 5-6-8-9 establishes case 3); $PPC_{5-6-8-9} x \leq y$ is satisfiable, but $ASC_{5-6-8-9} x = x + 1$ is not satisfiable. If case 1) holds, it could happen that the corresponding LPC invalidates case 1) by imposing additional contradictory constraints to ASC and PPC constraints.

The array subscript constraint obtained by symbolic execution of array subscripts on partial paths between two array references can be separated from the loop path constraints. Techniques for solving PPC and ASC on partial paths do not suffer from the undecidable problems, and they are different from the techniques used for solving LPC. Equating the normalized array indices produces array subscript constraints, symbolic execution produces partial path constraints, and good loop path constraints could be obtained through a technique called loop forcing.

Separating the constraints obtained on simple paths, ASC and PPC from the LPC could potentially improve the data dependence analysis. Having a satisfiable ASC and PPC is an indication of potential data dependence. But, satisfiable ASC and PPC do not guarantee correct data dependence identification; the interaction between ASC, PPC and LPC could invalidate the satisfiable ASC and PPC constraints. Therefore, when ASC_p

and PPC_p are satisfiable, a satisfiable LPC_p that does not invalidate ASC and PPC is needed. Such LPC_p could be obtained through loop forcing.

Loop Forcing

Both partial symbolic execution and explicit naming of nodes do not deal with loops well; they ignore loop details and produce approximations of the LPC which could invalidate the dependence associations. The problem of establishing array term dependence, in general, requires precise semantic analysis of loops. However, we think that the problem of establishing data dependence in programs with arrays is not as hard as the general loop problem. In particular, the separation of ASC and PPC from LPC suggests that the LPC could be forced not to be inconsistent with a satisfiable ASC and PPC. Rather than obtaining general statements about what loops do (such as loop formula or path condition for exit), the loop could be forced to do what is required. Of course, what is required must be a subset of what the loop actually does.

The problem of identifying data flow dependencies in statements containing array references can be broken into two cases which allow independent treatment. One of the cases requires general loop statements, but the other is much easier. Therefore the problem of identifying data dependencies in programs with arrays is not as hard as the general loop problem. As an illustration consider the code fragment in Figure 26. Suppose we want to investigate whether the assignment statement S_3 and the output statement S_9 at lines 3 and 9 respectively, are data flow dependent. If the while loop were not there, S_3 and S_9 would be data dependent since the array subscript equation $x = x$ is satisfiable. The elimination of the loop prevents further restrictions of the subscript relevant variables to be imposed by the loop details which could make the subscript equation unsatisfiable. Similarly, if the loop did not contain any definitions of x than S_3 and S_9 would be data dependent regardless of the number of loop traversals.

```

    int *A, i, x;
    ...
2   scanf("%d, %d", &i, &x);
3   A[x] = i + 1;
4   while(i)
5       if(x < 10)
6           x = x + 1;
7       else
8           i = i + 1;
9   printf("%d\n", A[x]);
10  ...

```

Figure 26. Array manipulation function.

Furthermore, the path 2-3-4-5-7-8-4-9 does not contain a definition of x and therefore establishes data dependence.

In general, given two array references $A[\Phi_1]$ and $A[\Phi_2]$ and a loop L placed between them, Φ_1 and Φ_2 are normalized on some simple paths that lead to the loop boundaries, producing normalized subscript expressions ϵ_1 (at top) and ϵ_2 (at end). For example, in Figure 24, the normalized subscript expression for the array subscript x in $A[x]$ at line 3, normalized to line 4 (upper loop boundary), is x . Similarly, the normalized subscript expression for the array subscript x in $A[x]$ at line 9, normalized to line 8 (lower loop boundary), is x . Given two normalized array subscript expressions, ϵ_1 and ϵ_2 , two cases need to be considered:

1) $\epsilon_1 = \epsilon_2$ is satisfiable. If L does not contain definitions of variables in ϵ_2 then $A[\Phi_1]$ and $A[\Phi_2]$ are data flow dependent. Furthermore, the existence of any path through L on which

variables of Φ_2 are not defined, establishes data dependence. The existence of any path through L includes the important special case when the path avoids L altogether.

2) $\epsilon_1 \neq \epsilon_2$ is satisfiable. This case requires determining a path through L that would eliminate this disparity.

When array references appear within a loop, then paths on which PPC and ASC are satisfiable is searched for. If such a path exists, the LPC is forced to preserve the satisfiability of the original ASC and PPC constraints.

Considering case 1) before approximating case 2) would improve the data dependence analysis. Case 1) suggests an approach where a search is made for a def-clear path with respect to the subscript relevant variables. Before attempting to force a def-clear path through the loop, the program flow graph is inspected for definitions of the subscript relevant variables. If the inspection shows that the subscript variables are defined on each simple path through the loop then the search is not attempted. For example, the simple path 7-8-4-5-6-7 in Figure 27 is a du-path between the array terms at line 7. The program data-flow graph would show that all the paths taking the false branch at conditional 5, are def-clear with respect to j . Therefore, a search for a feasible def-clear path should be attempted. The simple paths that are not def-clear would be identified from the program data-flow graph, and whenever the path condition implies taking a path that is not def-clear, the path is abandoned and the loop is tried for alternate paths. Intermediate array appearances do not complicate the matter since symbolic execution can handle arrays. This technique is successful when there exists at least one feasible path on which the subscript relevant variables are not defined. If it were known that such a path existed in advance, then an unbound search could find it. However, there is no assurance that such a path exists, and finding out is an undecidable problem.

```

void BubbleSort(A, N)
  int *A, N;
  {
  int i, j, temp;
3   for( i = 2; i < N; i++)
4     for(j = N; j < i; j--)
5       if(A[j] < A[j + 1]) {
6         temp = A[j];
7         A[j] = A[j + 1];
8         A[j + 1] = temp;
9       }
10    }

```

Figure 27. Bubble sort function[24], p.66.

Therefore, it could happen that the loop is forced through an arbitrary sequence of simple paths that do not redefine subscript relevant variables, and then the path condition implies taking a path that further constraints the original subscript constraint. Consider the code segment in Figure 28. The subscript constraint between $A[i]$ at lines 4 and 12 is $i = i$. However, after 51 iterations of the loop, the true branch at line 8 is taken redefining the subscript relevant variable i at line 9.

Another serious problem is an unsatisfiable ASC. Since the ASC is obtained on simple paths, it could happen that the ASC is not satisfiable on any simple path. The importance of this problem lies in the fact that if there exists no path p for which ASC_p and PPC_p are satisfiable for some du association that lies on p , then the loop forcing technique will not succeed.

```

        int i, j, k;
3      ...
4      A[i] = 0;
5      k = 0;
6      for(j = 0; j < 100; j++)
7          k = k + 1;
8          if( k > 50)
9              i = i + 1;
10         else
11             printf("%d", k);
12             printf("%d", A[i]);
13     ...

```

Figure 28. Infeasible array subscripts.

In particular, ASC is not satisfiable if either the unnormalized subscripts cannot be equal, or they can be equal, but every path that connects them redefines (modifies) the subscript relevant variables. For example, in Figure 29, neither the ASC_{5-6-7} nor the $ASC_{7-8-4-5}$ is satisfiable. Therefore, the loop forcing technique fails; there is no subscript relevant def-clear path that connects lines 5 and 7. Despite of the problems inherent in this technique, it improves the precision of the du analysis in some cases. We analyzed the Bubble sort program given in Figure 30. There are total of 13 simple du-paths. The aggregate view falsely omits 7 and falsely includes 10 du-paths. The loop forcing technique gets all 13 of them right.

The ASC and PPC constraints are necessary conditions for the loop forcing technique to succeed. Whenever there is no path through the loop for which both ASC and PPC are simultaneously satisfiable, loop forcing is not attempted.

```

int i, j, k;
3   j = 1;
4   for(i = 2; i < N; i++) {
5       A[i] = i;
6       j = i + N;
7       k = A[j];
8   }
9   ...

```

Figure 29. Program segment without satisfiable ASC.

```

int i, j, temp, *A, N;
3   ...
4   for(i = 0; i < N; i++)
5       scanf("%d", A[i]);
3   for( i = 2; i < N; i++)
4       for(j = N; j < i; j--)
5           if(A[j] < A[j + 1]) {
6               temp = A[j];
7               A[j] = A[j + 1];
8               A[j + 1] = temp;
9           }
10  for(i = 0; i < N; i++)
11  printf("%d", A[i]);

```

Figure 30. Bubble sort program.

Loop forcing is a general software analysis technique. Given that two program variables x and y are related with some relation R such as “equal to”, loop forcing searches for a path on which the relation R is preserved. This technique could be used to improve data dependence analysis in programs involving arrays, as well as to partially lift the simple path restriction present in data-flow testing. Rather than merely considering simple paths through loops which suffer from problems of infeasibility, loop forcing could be used to establish feasible du-paths. The infeasible simple variable du-path problem is easier to deal with than the array du-path problem, because the subscripts might not be equal, whereas two simple variables are always equal.

CHAPTER VI

CONCLUSION

The generally accepted belief that symbolic execution cannot handle arrays is false. Symbolic execution tools either deal with arrays in an infeasible manner or they avoid arrays altogether. However, arrays can be handled in a general way by the substitution technique that eliminates array terms from constraints. This approach is based on the examination of the relationships between the array term indices. The relationships determine the kind of mathematical variables needed for the substitution. This substitution technique deals with constraints containing homogeneous array terms, heterogeneous array terms, and nested array terms. The substitution technique has quadratic computational complexity with respect to the number of subscript comparisons for constraints involving one-dimensional array terms. The array-term-free constraints could be given to an equation solver which could attempt to solve them up to its capabilities. The substitution technique solves the problem of indexed variables in symbolic execution without introducing any additional undecidable problems. Utilizing the substitution technique, symbolic execution can handle indexed variables on any fixed path through a program. However, symbolic execution does not do well with looping structures. Except for special cases, neither the formula resulting from a loop nor the the path condition for exit can be computed. Special problems with loops arise for programs containing indexed variables—a potential infinity of paths need to be considered through loops to establish data dependence. We solved the problem of indexed variables in symbolic execution, but the loop problem must remain intractable.

The inclusion of arrays in program analysis introduces generally undecidable problems of determining whether two array subscript can ever be the same. The problem arises from the fact that a potential infinity of paths through loop structures needs to be considered in order to determine data dependence. Separating the array subscript constraints from the path constraints resulting from looping constructs allows different treatment of array subscript constraints from loop path constraints. The array subscript constraints are produced by symbolic execution of simple paths, whereas loop path constraints are derived by heuristic symbolic execution techniques. The successful treatment of array subscript constraints comes from the fact that these constraints are obtained by symbolic execution of simple paths, the paths for which symbolic execution is effective. Unfortunately, the loop path constraints require consideration of infinite number of paths through loops.

The partial symbolic execution heuristic technique that deals with complex paths is unsuitable for analysing data dependencies in programs involving arrays. The approach of explicit naming of paths suffers from semi-decidable problems and does not involve the necessary precision when applied to programs containing array terms.

Identifying data dependencies requires precise information about the semantics of the loops. Except for special cases, this information cannot be obtained. However, the data dependence problem in programs with loops is not as hard as the general loop problem. Many array association would be identified if the problem of array data dependence is broken in two cases: 1) the array subscript equality constraint is satisfiable and 2) the array subscript equality constraint is not satisfiable. Establishing data dependence when case 1) holds is not as hard as when case 2) holds. For dependence to be established in case 1), the loop is searched for a path whose path constraint does not invalidate case 1). Establishing dependence when case 2) holds is a harder problem since a path is needed which makes the subscripts equal.

We presented a general loop forcing technique that could be used to

preserve certain properties of relations between program variables. The loop forcing technique, in the context of data dependence, is used when the subscript equality constraint is satisfiable. The loop is then forced through a path that preserves the properties of the equality relation. If such an equality preserving path exists through the loop, data dependence is established. We think that this technique could contribute to partial lifting of the simple path restriction present in data-flow testing. In particular, rather than introducing infeasible simple du-paths, when in fact, there exist feasible complex du paths, a def-clear path could be forced through the loop which interrupts du associations. We also described the necessary constraint for the loop forcing technique to succeed: ASC. Unsatisfiable ASC for all the simple paths connecting a du association are indication that the loop forcing technique will not succeed.

Arrays have been treated inadequately in the past. The aggregate array analysis suffers from the problems of false path inclusion and correct path omission. Treating arrays element-wise can correct both of these mistakes, but introduces undecidable problems. By considering arrays as element-wise data-flow objects on a limited subset of the total paths in programs, the precision of data dependence could drastically improve. Symbolic execution can be used for analysis of simple paths connecting array references, and loop forcing for preserving data dependencies.

We have not conducted a study of the second case (Chapter V) in which the array subscript equality constraint is not satisfiable. The question of unsatisfiable subscript equality constraints raises the issue of redundant paths with respect to the array subscript relevant variables. These redundant paths which do not affect the array subscript relevant variables could be removed from the collection of possible paths considered for establishing data dependence between two array references.

A more elaborate study of the loop forcing technique is needed. The effect of the integration of loop forcing, partial symbolic execution and explicit naming of paths needs to be further examined.

REFERENCES

- [1] S. Rapps and E. J. Weyker, "Selecting Software Tests Using Data Flow Information," IEEE Trans. Software Eng. SE-11 (April, 1985), 367-375.
- [2] D. Hamlet, B. Gifford and B. Nikolik, "Exploring Dataflow Testing of Arrays," ICSE 15, (May, 1993), 118-129.
- [3] J. C. King, "Symbolic Execution and Program Testing," CACM 19, (July, 1976), 385-394.
- [4] W. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Tran. Softw. Enging. 4, (1976), 266-278.
- [5] J. Jaffar and J-L. Lassez, "Constraint Logic Programming," POPL, Munich, January, 1987.
- [6] Phyllis G. Frankl, "Partial Symbolic Evaluation of Path Expressions," Polytechnic University Department of Computer Science Technical Report PUCS-110-90, July, 1990.
- [7] J. R. Horgan and S. London, "Data Flow Coverage and C Language," ACM, 1991, 87-97.
- [8] A. Coen-Posisini and F. De Paoli, "Array Representation in Symbolic Execution," Computer Languages 18 (1993), 197-216.
- [9] Kemerer, R. and Eckmann, S. "UNISEX a UNIX-based Symbolic EXecutor for Pascal," Soft. Prac. Exper. 15, 439-457, 1985.
- [10] Michael Wolfe, Chen-Wen Tseng, "The Power Test for Data Dependence," IEEE Trans. Distr. Sys. 5, (1992), 591-601.
- [11] D.E. Knuth, "The Art of Computer Programming," Vol. 2, Seminumerical Algorithms, Second ed. Reading, MA: Addison-Wesley, 1981.

- [12] John A. Darringer, James C. King, "Applications of Symbolic Execution to Program Testing," IEEE Trans. Softw. Enging., April, 1978, pp. 51–60.
- [13] S. L. Hantler and J. C. King, "An Introduction to Proving The Correctness of Programs," ACM Computing Surveys, Vol. 8, No. 3, September 1976, pp. 331–353.
- [14] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Soft. Eng. SE-2 (September, 1976), 215–222.
- [15] C. V. Ramamoorthy, Siu-Bun F. Ho, W. T. Chen, "On the Automated Generation of Program Test Data," IEEE Trans. Soft. Eng. SE-2 (December, 1976), 293–300.
- [16] R. S. Boyer, B. Elpas and K. N. Levitt, "SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution," Proc. Int. Conf. Reliable Software, 1975.
- [17] Michael Wolfe, "Data Dependence and Program Restructuring," Department of Computer Science, Oregon Graduate Institute, Technical Rep. No CS/E 90–007.
- [18] Mary J. Harrold, Brian Malloy and Gregg Rothermel, "Efficient Construction of Program Dependence Graphs," Proc. Int. Symp. on Soft. Testing and Analysis, June, 1993, 160–170.
- [19] Katz, S. M and Manna Z., "A Heuristic Approach to Program Verification," Proc. Third Intl. Joint Conf. on Artificial Intelligence, SRI Publications Dept. Stanford Calif., 1973, pp. 500–512.
- [20] Wegbreit, B., "The synthesis of loop predicates," Comm. ACM 17, 2, (Feb. 1974), pp. 102–112.
- [21] P. G. Frankl, S. N. Weiss, and E. J. Weyuker, "ASSET: A System to Select and Evaluate Tests," Proc. IEEE Conf. Software Tools, New York, (April 1985).

- [22] Thomas J. Ostrand, and E. J. Weyuker, "Data Flow-based Test Adequacy Analysis for Languages with Pointers," ACM SIGSOFT Symposium on Software Testing and Verification, October 1991, Victoria, B. C., Canada.
- [23] C. William Gear, "Computer Applications and Algorithms," Science Research Associates, Inc. 1986.
- [24] N. Wirth, "Algorithms + Data structures = Programs," Prentice-Hall, Inc., 1976.
- [25] U. Banerjee, "Dependence Analysis for Supercomputing," Norwell, MA: Kluwer Academic, 1988