Symptom Based Error Detection

Margaret Ann Francel
Spencer Rugaber

Georgia Institute of Technology
Atlanta, Georgia

1. INTRODUCTION.

A program *failure* is a departure of program operation from program requirements. Program failures are caused by program faults, which in turn are caused by programming errors. A programming *error* is a defect in the human thought process made while trying to write a program. A program *fault* is a manifestation of an error in the program code. [E1]

A failure is identified by an outside source, called an *oracle.* This oracle may be a human tester or an automated regression testing system. The failure is identified because external program expectations are not met or are violated. These expectations may have been expressed by the program specification or by pairs of input and output vectors describing accurate computations. From the observation that the execution failed, the oracle concludes that the program contains faults. That is, some part of the program is not executing in the manner that the programmer intended. This part of the execution is made up of one or more actions, called *deviant actions.* These actions occur throughout the execution until finally one or more incorrect values filter into output statements.

The process of finding program faults is called *debugging* [E1] and is usually labor intensive and time consuming. Because of this, it is desirable to automate the process as much as possible. For example, a debugger might be used to display intermediate results. Or a program slice might be identified, giving the oracle a subset of the program that can not be causing the fault. Each debugging method has the underlying purpose of trying to simplify fault detection for the oracle.

Most debugging methods identify programs faults by examining or analyzing program code. The main purpose of this paper is to show that the debugging process can be improved by also examining the history of the program's execution and the validity of the program's output values. The model developed and the methods presented do this while at the same time reducing the amount of work on the part of the oracle.

In the paper, a formal definition of a program, its execution, and its *trace* are given. A model to be used for storage of execution data and validity information is defined, and algorithms for constructing it are presented. The construction methods are shown to be time efficient. An example is given to demonstrate that the model structure itself is helpful in the error detection process. This demonstration is through the analysis of the situation where a unique error exists in a striaght line program. The model is also shown to lend itself to detecting entire classes of errors.

In order to concentrate fully on the fault localization problem, we begin by examining a limited subset of all program faults. Throughout the paper we restrict the programming environment and the behavior of the faults by the following assumptions:

A1) Only straight line terminating programs are considered.

A2) The program execution does not correct itself.

A3) An oracle is used to provide correctness information about the output values. The information is assumed internally consistent.

2. THE PROBLEM MODEL.

As stated above, the main problem of interest in this paper is the detection of the location of faults in a program accomplished by the examination of the history of the program's execution along with the output produced by the program. In this section, we present a model for organizing this information that provides fast retrieval of the stored data as well as a structure that lends itself to easy analysis of interrelationships between execution actions.

2.1. A Program and its Execution.

A *program* is a sequence of statements that access and combine named program variables.

An example of a program containing a fault:

```
PROGRAM SWAP
1) GET(x)
2) GET(y)
3) x := y
4) y := x
5) PUT(x)
6) PUT(y)
```

Figure 2.1

Once written, a program can be executed any number of times by the computer. Each *program execution* is a map from a vector of data, called the input, to another vector of data, called the output.

Examples:

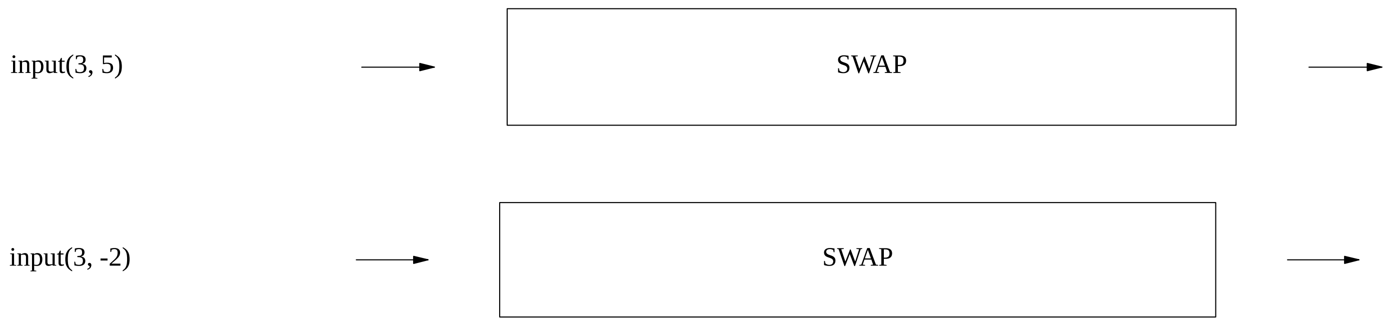

Figure 2.2

The actions carried out by an execution are prescribed by the named program. These actions can be summarized in an *execution trace.* For each execution action, the trace contains a record which consists of the number of the statement executed, the action taken, the values of all variables referenced and any newly assigned value.

Example:

PROGRAM SWAP EXECUTION TRACE input(3, -2) output(-2, -2)			
program statement executed	action taken	variable references	variable assignments
1	GET(x)		x/3
2	GET(y)		y/-2
3	x := y	y/-2	x/-2
4	y := x	x/-2	y/-2
5	PUT(x)	x/-2	
6	PUT(y)	y/-2	

Figure 2.3

As illustrated above, programming examples will be given in generic notation, and execution traces are limited to input, output and assignment actions. Input actions are represented by the GET statement, and output actions by the PUT statement. Both statements are restricted to naming a single variable.

2.2. Trace Graphs.

During program execution, an action can reference variables that have already been assigned values. In such a case, we say the trace record that recorded the use of the value is *directly affected* by the trace record that recorded the last assignment of the value. The collection of all records that directly affect a record r is called the *direct affect set* of r.

Example:

DIRECT AFFECT SETS FOR PROGRAM SWAP EXECUTION TRACE		
Statement Number	Action Taken	Direct Affect Set
1	GET(x)	{ }
2	GET(y)	{ }
3	x := y	{2}
4	y := x	{3}
5	PUT(x)	{3}
6	PUT(y)	{4}

Figure 2.4

The collection of direct affect sets of a trace have a natural representation as a directed graph. We call this graph a *trace graph.* The nodes of the graph are the records of the trace, and an ordered pair (r_i, r_j) is an arc in the graph if and only if r_j is directly affected by r_i. The graph for the trace illustrated above is given by:

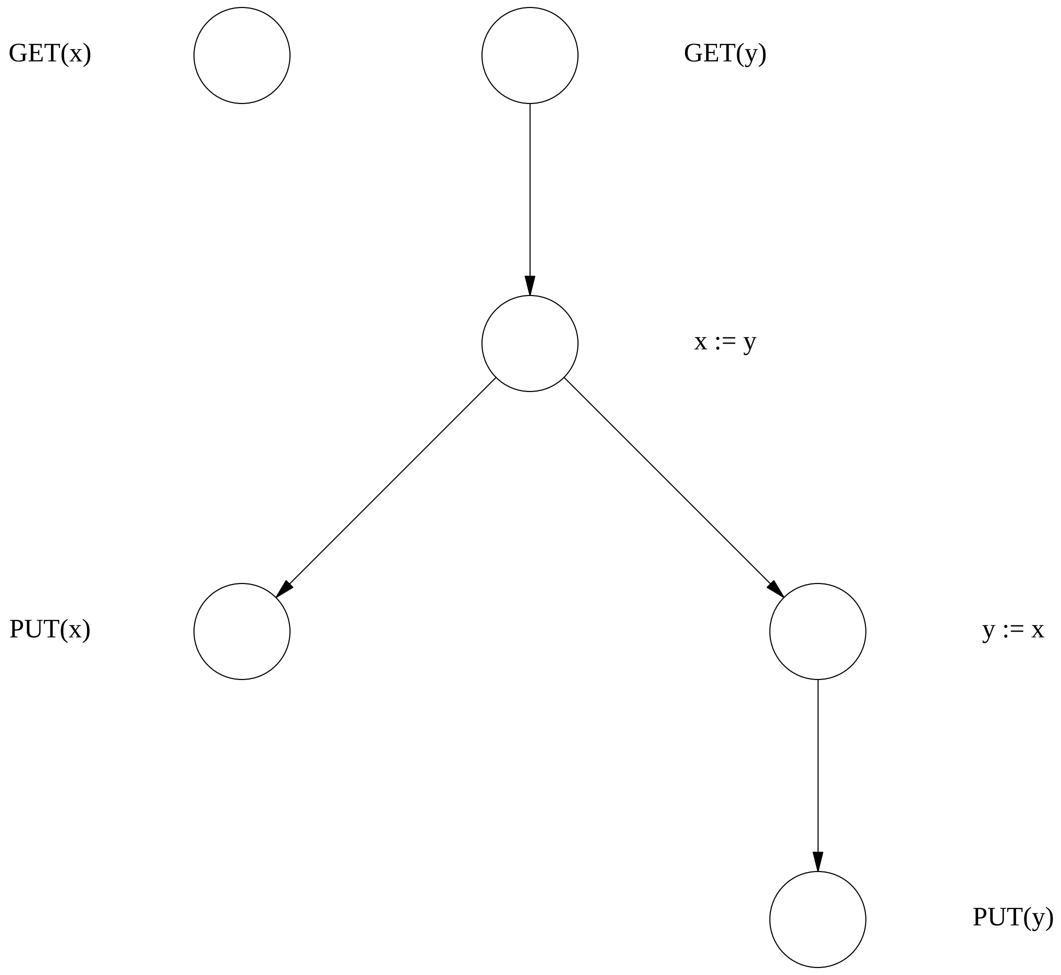

Figure 2.5

Note, all trace graphs are directed acyclic since the definition of arc specifies a relationship between execution actions rather than program statements. The execution order provides a topological ordering on the graph, where u < v if and only if (u,v) is an edge in the graph. Also, input nodes have no in-edges, output nodes have no out-edges, and the trace graph is finite if and only if the execution terminates.

The following algorithm provides a method for building a trace graph from a trace.

Algorithm 2.1: Building a trace graph.

Input. TRACE, an execution trace. Each record in TRACE contains the fields:

program_statement_executed	:	integer,
action_taken	:	(input, output, assignment),
variables_referenced	:	list of variables,
variable_assigned	:	variable.

Output. GRAPH, the trace graph of TRACE. Each record in GRAPH describes a node and is constructed by augmenting a record of TRACE with the following two fields:

in_edge_list	:	list of GRAPH records,
out_edge_list	:	list of GRAPH records.

Method. A table, referred to as HASH_TABLE, is used to keep track of value assignments to variables. Access to HASH_TABLE is through a hash function called HASH. During each step of the algorithm, HASH(X) represents the node of GRAPH in which X was last assigned a value. HASH_TABLE is initially empty.

```
for i := 1 to the number of records in TRACE do
    for every variable X referenced in TRACE[ i ] do
        if X is in HASH_TABLE
            then
                Add GRAPH[ i ] to the out-edges of HASH(X).
                Add HASH(X) to the in-edges of GRAPH[ i ].
    if TRACE[ i ].action is input or assignment
        then
```

```
if TRACE[ i ].variable_assigned is in HASH_TABLE
    then
        Update HASH(TRACE[ i ].variable_assigned) to have value i.
    else
        Insert TRACE[ i ].variable_assigned in HASH_TABLE with value i.
```

Lemma 2.1 If Algorithm 2.1 is processing record r, then HASH(X) represents the last trace record before r in which X was assigned a value.

Proof: Since the hash table is empty at the start of the algorithm, the statement is true before the first record is processed. If the statement is true before record r is processed, it will also be true after since trace records are processed in order, and the last step in processing r is to update the hash table to represent any assignment made in r. □

Theorem 2.1: Given a trace table, Algorithm 2.1 constructs the corresponding trace graph, (V,E), from the table in O(|E| + |V|) time

Proof: We first show that the trace graph and the graph constructed in Algorithm 2.1 are the same graph.

All edges of the trace graph are constructed by Algorithm 2.1. Assume (u,v) is an edge in the trace graph. Then v is directly affected by u. This says, there exists a variable x such that x is referenced in v and the last value assignment made to x before the execution of v was made in u. Since the last value assignment to x before the execution of v was made in u, Lemma 2.1 tells us that during the processing of v in Algorithm 2.1, HASH(x) will have the value u. Thus, Algorithm 2.1 constructs the edge (HASH(x),v) = (u,v) during the processing of v.

Algorithm 2.1 constructs no extra edges. Assume (u,v) is an edge constructed by Algorithm 2.1. Then (u,v) was constructed by the algorithm during the processing of v, and there exists a variable x referenced in v such that when v was processed HASH(x) = u. Since HASH(x) = u, we know the last value assignment made to x before the execution of v was in u. This implies v is directly influenced by u; therefore, it follows that (u,v) is an arc in the trace graph.

The complexity of Algorithm 2.1 is O(|E| + |V|). It is reasonable to assume that a hash table lookup can be done in unit time. The work of the algorithm is done in adding values to trace records and updating or inserting records in the hash table. In determining the complexity of Algorithm 2.1, we will count the number of values added to trace records and the number of updates and inserts made in the hash table. A new value, representing (u,v), is added to a trace record if and only if the record being processed is v. Since we have shown that each node in the trace graph is constructed exactly once, exactly 2 * |E| values are added to trace records. An insert or update to the hash table is done at most once per trace record. Thus, no more than |V| insert or updates are done during the algorithm. The above two facts tell us that the total amount of work done is 2*|E|+|V|. □

2.3. 0/1 Labeling a Trace Graph.

For a given program execution, the oracle has provided information that allows us to divide the output values of the execution into two sets, those that are correct and those that are incorrect. We project this information onto the trace graph by labeling correct output nodes and those nodes that directly affect correct output nodes with a "1", and labeling incorrect output nodes and those nodes that directly affect incorrect output nodes with a "0". We label a node with a "1" to indicate that the value assignment made by the represented action is correct. We label a node with a "0" to indicate that the value assignment made by the represented action is incorrect.

The correctness information about the output nodes can be propagated through the trace graph to give information about other actions. To explain how, we wish to use the concepts of path and Reach-In and

Reach-Out sets. There exists a *path* from node a to node b in a directed graph if there exists a sequence of nodes n_1, n_2, ..., n_k, such that n_1 = a, n_k = b, and for i = 1,...,k-1, (n_i, n_{i+1}) is an edge in the graph. The set of all nodes in a graph that appear on any path with terminal node n is called the *Reach-In* set of n. Similarly, the set of all nodes in a graph that appear on any path with initial node n is called the *Reach-Out* set of n.

Assumption A2 states that a program does not correct itself. That is, faults always lead to failures. From this, we can conclude that any path in the trace graph that leads to a correct value consists of valid actions, and any path that comes from an incorrect value assignment consists of deviant actions. To mark valid actions in the trace graph, we label each node in the Reach-In set of a correct output node with a "1". To mark deviant actions in the trace graph, we label each node in the Reach-Out set of a node that directly affects an incorrect output node with a "0".

To complete our labeling, we label any unlabeled input node with a "1" and any unlabeled assignment node with a "˜". The former is done to indicate that all input actions are assumed to be valid; the latter to indicate that at present we do not have any knowledge about the validity of the given node's action. The above trace graph labeling is called the *0/1 labeling* of the trace graph. Assumption A3 allows us to assume that a 0/1 trace graph labeling is well-defined.

The 0/1 labeled trace graph of program SWAP is shown below. Solid circles represent nodes labeled with a "1", while open circles represent nodes labeled with a "˜", and "x"ed circles represent nodes labeled with a "0".

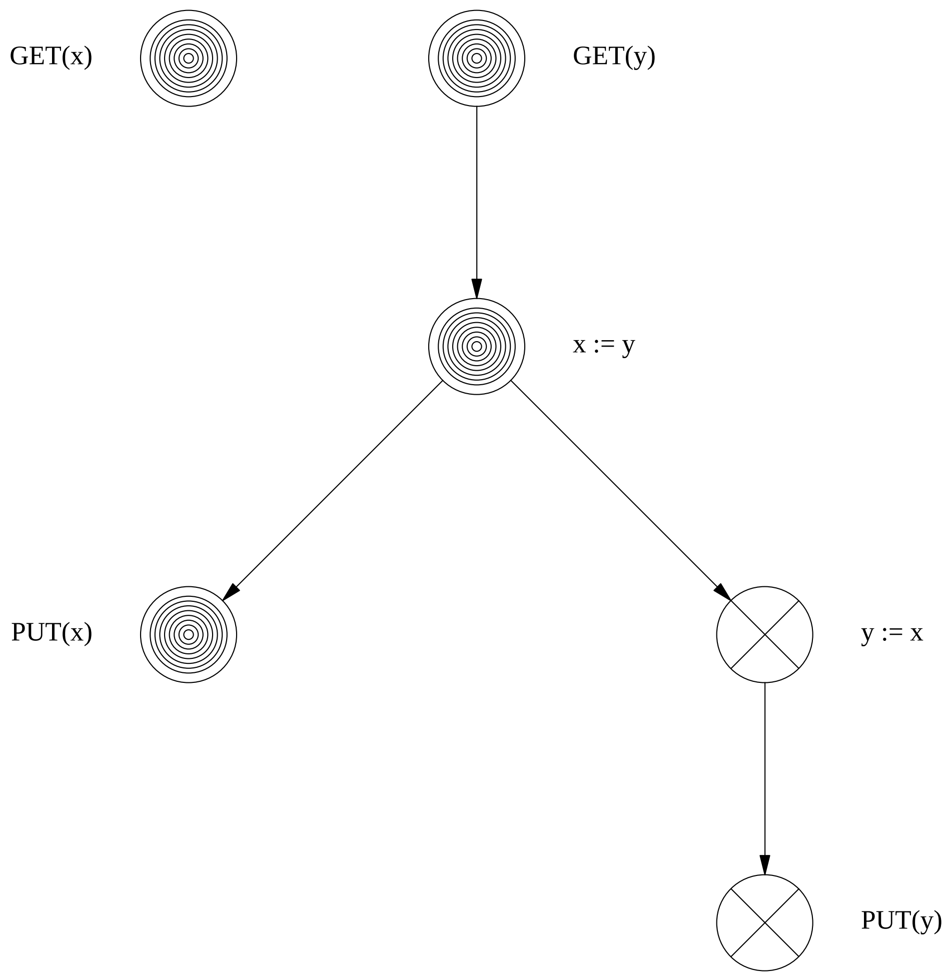

Figure 2.6

The following algorithm provides a method for 0/1 labeling a trace graph:

Algorithm 2.2: 0/1 labeling a trace graph.

Input. GRAPH, a trace graph obtained as output from Algorithm 2.1. Data from an oracle indicating which output results are correct and which are incorrect.

Output. A trace graph in which each node contains a field indicating the node's 0/1 label.

Method. Two queues are used; ONE, which holds unprocessed nodes that will be labeled with a "1" and ZERO, which holds unprocessed nodes that will be labeled with a "0". Both queues are initially empty.

```
for i := 1 to the number of records in GRAPH do
    case of (GRAPH[ i ].action) do
        input:
            Label GRAPH[ i ] with a "1".
        output:
            Label GRAPH[ i ] as indicated by the data from the oracle.
            Insert the node that directly affects GRAPH[ i ] in ONE or ZERO depending on
            label GRAPH[ i ].
        assignment:
            Label GRAPH[ i ] with a "~".
while ZERO is not empty do
    Remove node from ZERO.
    Label it with a "0".
    Insert all of its unlabeled children in ZERO.
while ONE is not empty do
    Remove node from ONE.
    Label it with a "1".
    Insert all of its unlabeled parents in ONE.
```

Theorem 2.2: Given a trace graph, (V,E), Algorithm 2.2 constructs from it in O(|E| + |V|) time a labeled trace graph.

Proof: We first show that the labeling produced by the algorithm and the 0/1 labeling are the same. Nodes are shown to be labeled with a "1" if and only if they are input nodes or in the Reach-In set of a correct output value. Nodes are shown to be labeled with a "0" if and only if they are in the Reach-Out set of a node that directly affects an incorrect output value.

If n is in the Reach-In set of a correct output value, then n is labeled "1". Assume $n, n_1, ..., n_k$ is a path from n to a correct output value. When Algorithm 2.2 processes n_k in the **for** loop, it will put n_{k-1} on the ONE queue. When n_{k-1} is deleted from the queue, n_{k-2} will be checked. If it is labeled "1", that says it already has been on the ONE queue; if it is not labeled, it will be put on the ONE queue, and by the assumption that faults always lead to failures we know that the node is not labeled "0". From the preceding we can conclude that n_{k-2} at some time appears on the ONE queue. We can continue arguing in this manner to show that n will appear on the ONE queue and hence will get labeled "1".

If n is labeled "1", then n is an input node or n is in the Reach-In set of a correct output value. Assume n is not an input node. Then the only way that n could have been labeled "1" is if it is a correct output value or if it appeared on the ONE queue. If it appeared on the ONE queue, then it is the parent of a correct output node or the parent of a node that appeared on the ONE queue. Call this node n_1. Now in an argument similar to the above, we see that n_1 is a correct output node or the parent of a node that appeared on the ONE queue. But since the graph is finite acyclic, we know the path of nodes we are producing must end at some correct output value. Thus, n is in the Reach-In set of a correct output value, namely the one at the end of the produced path.

If n is in the Reach-Out set of a node that directly affects an incorrect output value, then n is labeled "0". The argument to show this is symmetric to the argument that showed that if n is in the Reach-In set of a correct output value, then n is labeled "1".

If n is labeled "0", then n is in the Reach-Out set of a node that directly affects an incorrect output value. The argument to show this is symmetric to the argument that showed that if n is labeled "1" and n is not an input node, then n is in the Reach-In set of a correct output value.

The complexity of Algorithm 2.2 is O(|E| + |V|). The work of the algorithm is done in accessing nodes, thus in determining the complexity of Algorithm 2.2, we will count the number of different node accesses made. In the **for** loop, each node is accessed at most twice: once directly and possibly once as the node that is directly affecting an output node. By assumption A3, the validity values input by the oracle are consistent. We are assuming that this means no node can be in both the Reach-In set of a correct output value and the Reach-Out set of a node that directly affects an incorrect output value. This implies no node will appear on both the ONE and ZERO queues. For each node on a queue, either its children or its parents are examined. Therefore, the number of node accesses done during execution of the two **while** loops can’t exceed twice the number of edges in the graph. □

3. STRAIGHTLINE PROGRAMS CONTAINING A SINGLE FAULT.

One question of interest concerning the model presented in the last section is: Can graph properties help us to detect program faults? In this section, we look at the structure of the graph to see what it can tell us about the number of faults in a program.

The discussion in section 2.3 implies that actions that represent program faults can not exist in the subgraph of the trace graph labeled with "1"’s. Does this imply that the size of the subgraph labeled with "0"’s and "˜"’s determines the number of faults in the program? The program shown in Figure 3.1 is simple enough so that by merely looking at it one can determine that it contains a single fault: namely in statement 4 absolute values of differences, not the differences themselves are needed. When we examine its trace graph, we see that only the node containing the fault and the node outputting the incorrectly calculated value are labeled with "0" or "˜".

```
PROGRAM AVG. & S.D.
1)  GET(t1)
2)  GET(t2)
3)  avg := (t1 + t2)/2
4)  sd := ((t1-avg) + (t2-avg))/2
5)  PUT(avg)
6)  PUT(sd)
```

input(87 93)
AVG. & S.D.

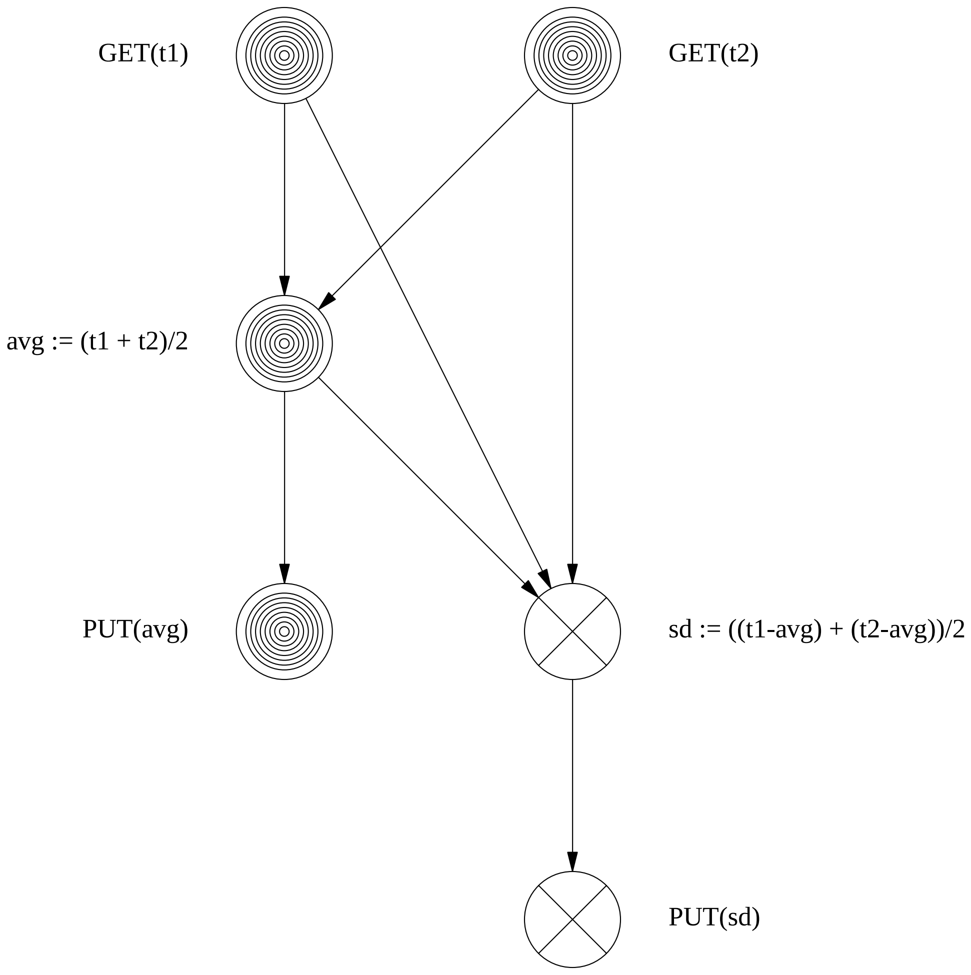

Figure 3.1

In Figure 3.2, we have the exact opposite situation. Again the program is simple enough so that by looking at it one can see that it contains a single fault: in statement 2 the variable a is uninitialized. But when we examine its trace graph, we see that all but one of the nodes are labeled with a "0" or with a "~".

```
PROGRAM DIVIDE & ROUND
1)  GET(b)
2)  x := a/b
3)  y := (x+.005)*100
4)  z := truncate(y)
5)  w := z/100
6)  PUT(w)
```

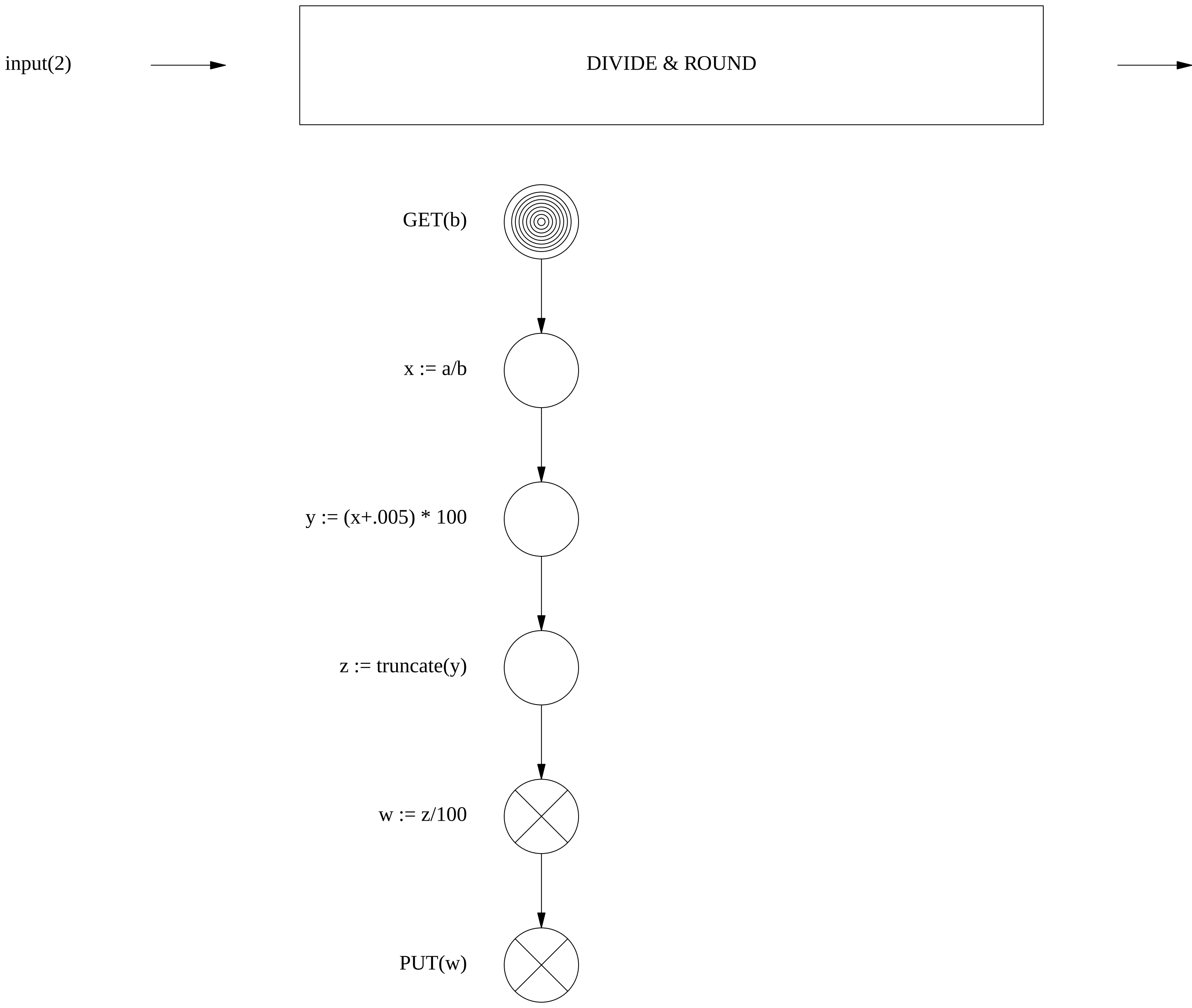

Figure 3.2

Since both of the above programs contain a single fault, we can conclude that the number of faults in a program does not depend solely on the number of nodes labeled "0" or "~" in the corresponding trace graph. This gives rise to the question: What properties about a 0/1 labeled trace graph do imply that all the deviant execution behavior is being caused by a single program fault?

Since actions that represent program faults can not exist in the subgraph of the trace graph labeled with "1"'s, we know that any node that represents a faulty statement must be labeled with a "0" or a "~". Also, in any trace graph, each deviant action must reach back to some action that represents a program fault. In the trace of a straight line program, each node represents a different statement. Thus, in a straight line program with a single fault, all deviant actions must reach back to some single node, namely the one that represents the faulty statement.

If we know a straight line program contains a single fault, to find it, we need to look for a node with the following two properties:

P1) It must be labeled as "0" or "~".

P2) Every node in the graph labeled "0" must be reachable from it.

In the labeled trace graph of Figure 3.3, the only node that satisfies the two properties needed for a node to represent the single faulty statement, in a straight line program is n_2. Thus, if the graph represents a program with a single faulty statement the statement is represented in the graph by n_2.

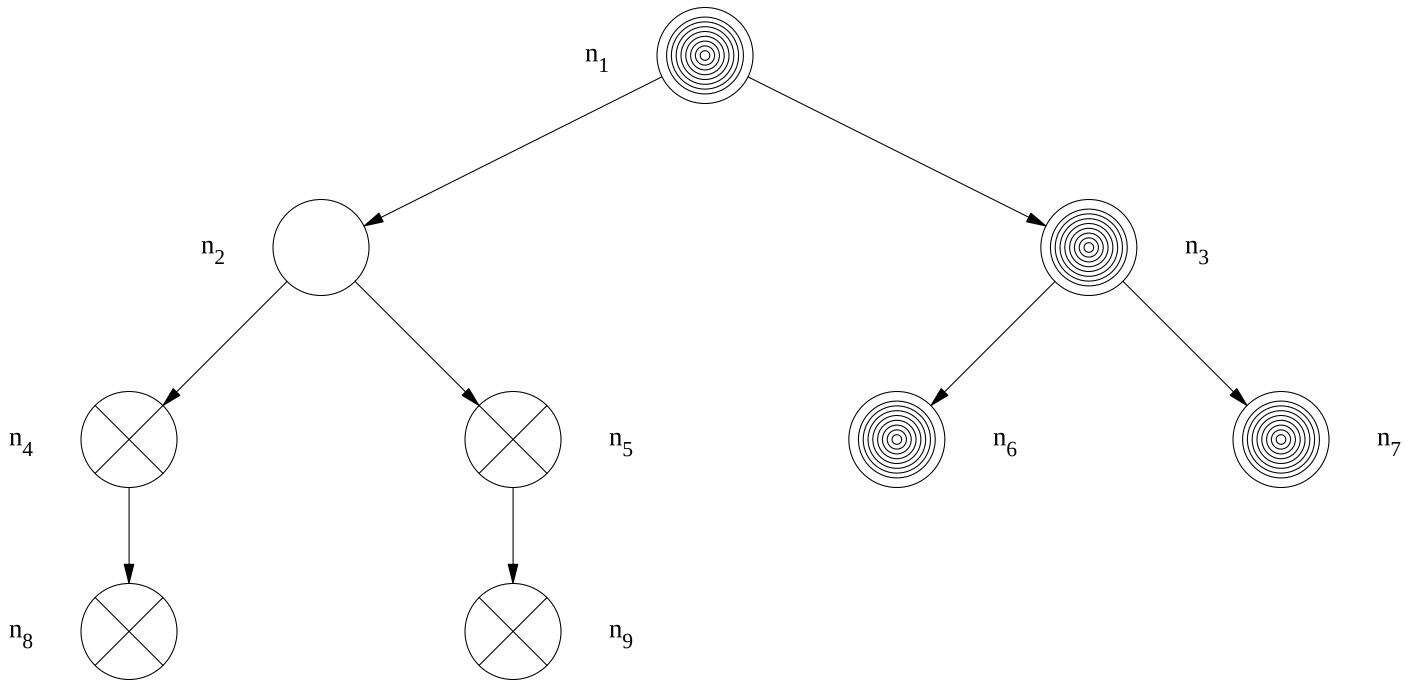

Figure 3.3

In the labeled trace graph of Figure 3.4, four nodes satisfy the two properties needed for a node to represent the single faulty statement in a straight line program. Thus, without querying an oracle for further information, all we can conclude is: If the labeled trace graph of Figure 3.4 represents a straight line program with a single fault, then one of the nodes n_2, n_4, n_5, or n_8 represents the faulty program statement.

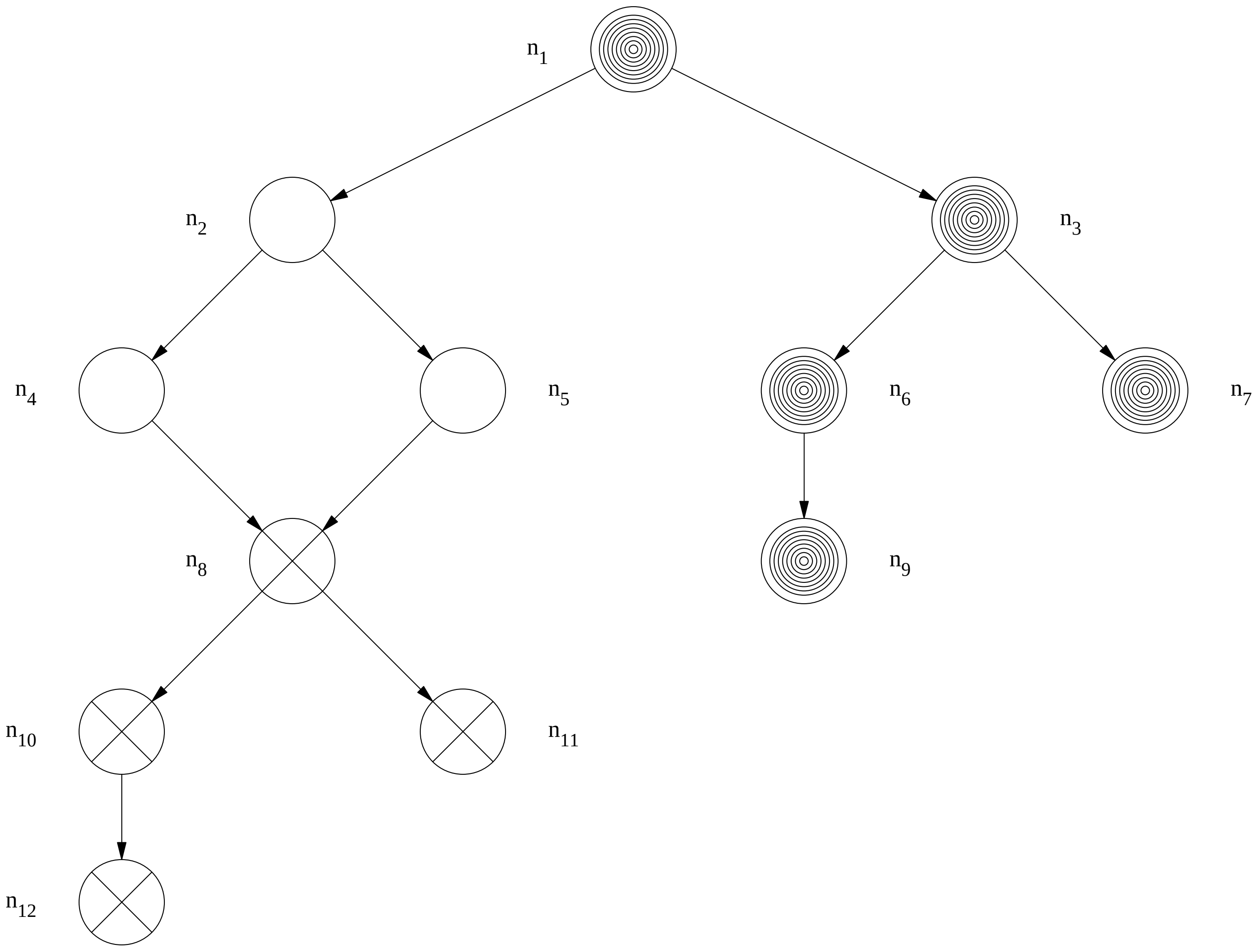

Figure 3.4

In Figure 3.5, no node of the labeled trace graph satisfies the two properties needed for a node to represent the single faulty statement in a straight line program; therefore, the program represented contains more than one fault. We know the program can not be fault free since there exists nodes in the trace graph which represent incorrect actions.

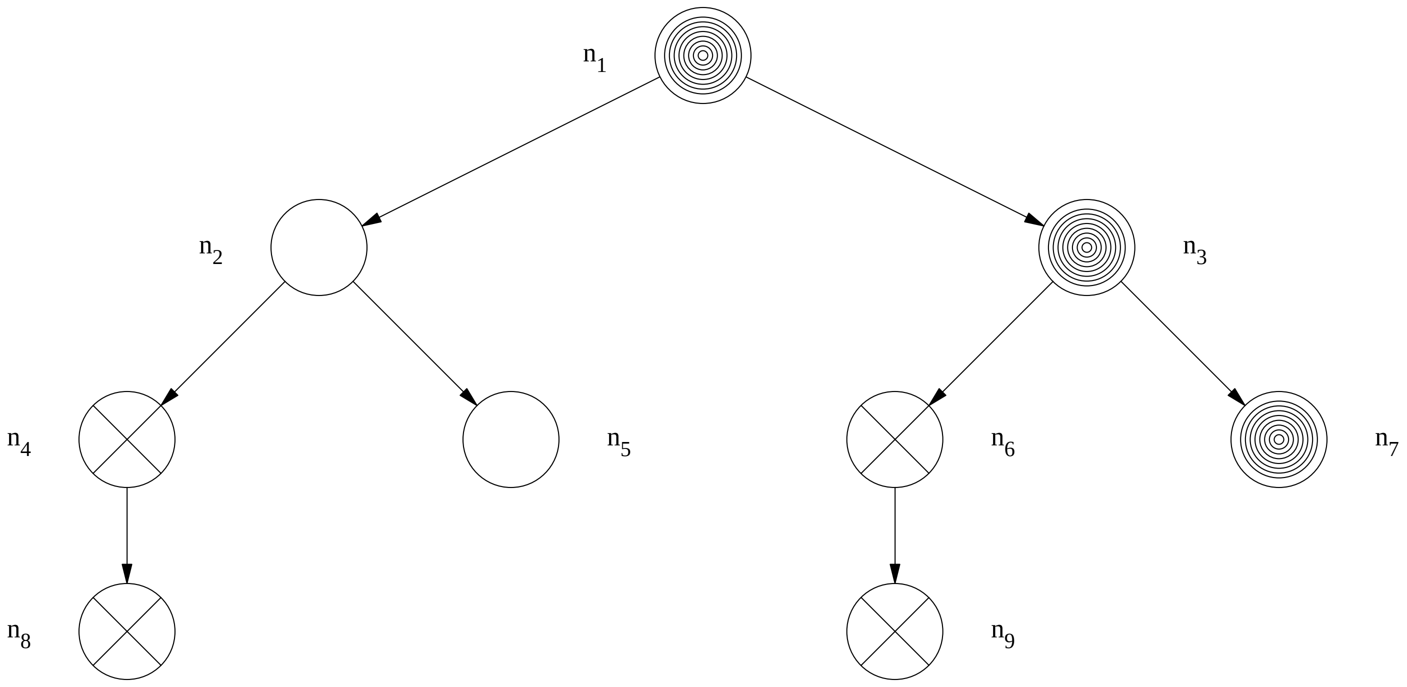

Figure 3.5

The following algorithm provides a method for finding all nodes in a 0/1 labeled trace graph that satisfy the properties needed for a node to represent the unique fault in a straight line program.

Algorithm 3.1: Locating nodes that could represent the faulty statement in a straight line program that contains a single fault.

Input. GRAPH, a labeled trace graph as output from Algorithm 2.2.

Output. POSSIBLE, a set containing all nodes that could represent the single faulty statement in a straight line program containing a single fault.

Method. A queue called TO_PROCESS is used. It contains nodes that could possibly represent the program fault and need to be checked further. OUT_REACH[i] represents the set of nodes in the trace graph labeled with a "0" and reachable from GRAPH[i]. FAULTY is the set of all nodes in the graph labeled "0".

```
Initialize FAULTY to the empty set.
for i := 1 to the number of records in GRAPH do
    case of (label of GRAPH[ i ]) do
        "0":
            Enter GRAPH[ i ] in the set FAULTY.
            Mark GRAPH[ i ] as still_possible.
            Initialize OUT_REACH[ i ] to {Graph[ i ]}.
        "~":
            Mark GRAPH[ i ] as still_possible.
            Initialize OUT_REACH[ i ] to the empty set.
        "1":
            Mark GRAPH[ i ] as not_possible.
            Initialize OUT_REACH[ i ] to the empty set.
Insert all nodes marked still_possible in TO_PROCESS in reverse topological sort order.
while TO_PROCESS is not empty do
    Remove a node from TO_PROCESS, call it $r_i$.
    Set OUT_REACH( $r_i$ ) to OUT_REACH( $r_i$ ) U ( $\bigcup_{r_j \text{ child of } r_i}$ OUT_REACH($r_j$)).
for i := 1 to the number of  records in GRAPH do
    if OUT_REACH[ i ] = FAULTY
        then
            Enter GRAPH[ i ] in the set POSSIBLE.
```

Lemma 3.1: A node n is in POSSIBLE as output by Algorithm 3.1 if and only if n satisfies the following two properties:

P1) It is labeled as "0" or "~".

P2) Every node in the graph labeled "0" is reachable from it.

Proof: FAULTY is the set of nodes in the trace graph that are labeled with "0". Entries are made to FAULTY only in the first **for** loop of the algorithm and only if the node is labeled with a "0".

Every node in POSSIBLE is labeled with a "0" or a "~". If a node is labeled with a "1", then in the first **for** loop of the algorithm it is marked as *not_possible* and its OUT_REACH set is initialized to the null set. No node marked as *not_possible* is ever put on TO_PROCESS, and its OUT_REACH set is never changed from the initialized value. Thus, in the second **for** loop its OUT_REACH set will never be equal to FAULTY. (Assuming the input graph has at least one incorrect output or there would be no need to look for program faults.) and the node will never be entered in POSSIBLE.

If n is an element of some OUT_REACH set, then n is labeled with a "0". This is true directly after the **for** loop of the algorithm has been executed since the only OUT_REACH sets that are not initialized to the null set are those where the nodes are labeled with a "0". In the remainder of the algorithm, each change made to an OUT_REACH set is through a union of already defined OUT_REACH sets, so no elements that weren't in some OUT_REACH set initialized in the **for** loop will appear in the new OUT_REACH sets.

If n is an element of OUT_REACH[i], then n is reachable from GRAPH[i]. This is clearly true directly after the **for** loop of the algorithm has been executed since either the OUT_REACH set is the null set or contains only GRAPH[i]. But if it is true before an execution of the **while** loop, it will also be true after an execution of the **while** loop since "reaches" is a transitive property, and an OUT_REACH set is changed in the loop by combining the old OUT_REACH set with the OUT_REACH set of a child of the node.

If n_k is a "0" labeled node reachable from a "0" labeled node n_1, then n_k is in OUT_REACH of n_1. Assume n_k is reachable from n_1, then there exists a path n_1, n_2, ..., n_k, ..., n_m, where n_m is a node with no outedges. Since n_1 is labeled with a "0", assumption A3 tells us none of the nodes on the path are labeled with a "1". This implies that each node on the path was marked as *still_possible* in the first **for** loop of the algorithm. Hence, each path node is in TO_PROCESS, and for $i < j$, n_j will be processed before n_i. When n_{m-1} is processed, its OUT_REACH set is updated and recorded. When the update is made, any node in OUT_REACH of n_m becomes a part of the OUT_REACH set of n_{m-1}. We can continue arguing in this manner to show that n_k will be in the OUT_REACH set of n_1. □

Lemma 3.2: The complexity of Algorithm 3.1 is $O(|V|^3)$, when the trace graph under consideration is (V,E).

Proof: We are assuming that sets will be kept in bit vector notation. If this is the case, we can further assume that every set operation will take at most time |V|, where V is the set of all possible set elements.

The work of the algorithm is done in operating on sets, thus in determining the complexity of Algorithm 3.1, we will count the number of set operations performed.

In the first **for** loop, a set is initialized for each node in the graph. In the **while** loop, for each node on the queue, a union is performed for each of the node's children. A node can appear on the queue once and can have at most |V| children. Thus, an upper bound for the number of set operations done in the **while** loop is $|V|^3$. In the second **for** loop, a set comparison is done for each node in the graph. Thus, the total amount of work done is $(2*|V|+|V|^3)*|V|$. □

Assume the program presented in Figure 3.6 contains a single fault. From the above discussion, we know the fault must occur in statement 3, 4, or 5. The program is simple enough so that knowing this allows us to conclude that the formula being evaluated, (a+b)/(a*b), is not correct. What is the correct formula? If we guess (a-b)/(a*b), the fault occurs in statement 3. If we guess (a+b)/(a-b), the fault occurs in statement 4. Or if we guess (a+b)-(a*b), the fault occurs in statement 5. No available information will help us to decide which of our three guesses, if any, is correct. If we are to decide where the program fault is occurring, we must query an oracle for additional information about certain trace actions.

```
PROGRAM
FORMULA EVALUATION.
1)  GET(a)
2)  GET(b)
3)  x := a + b
4)  y := a * b
5)  z := x / y
6)  PUT(z)
```

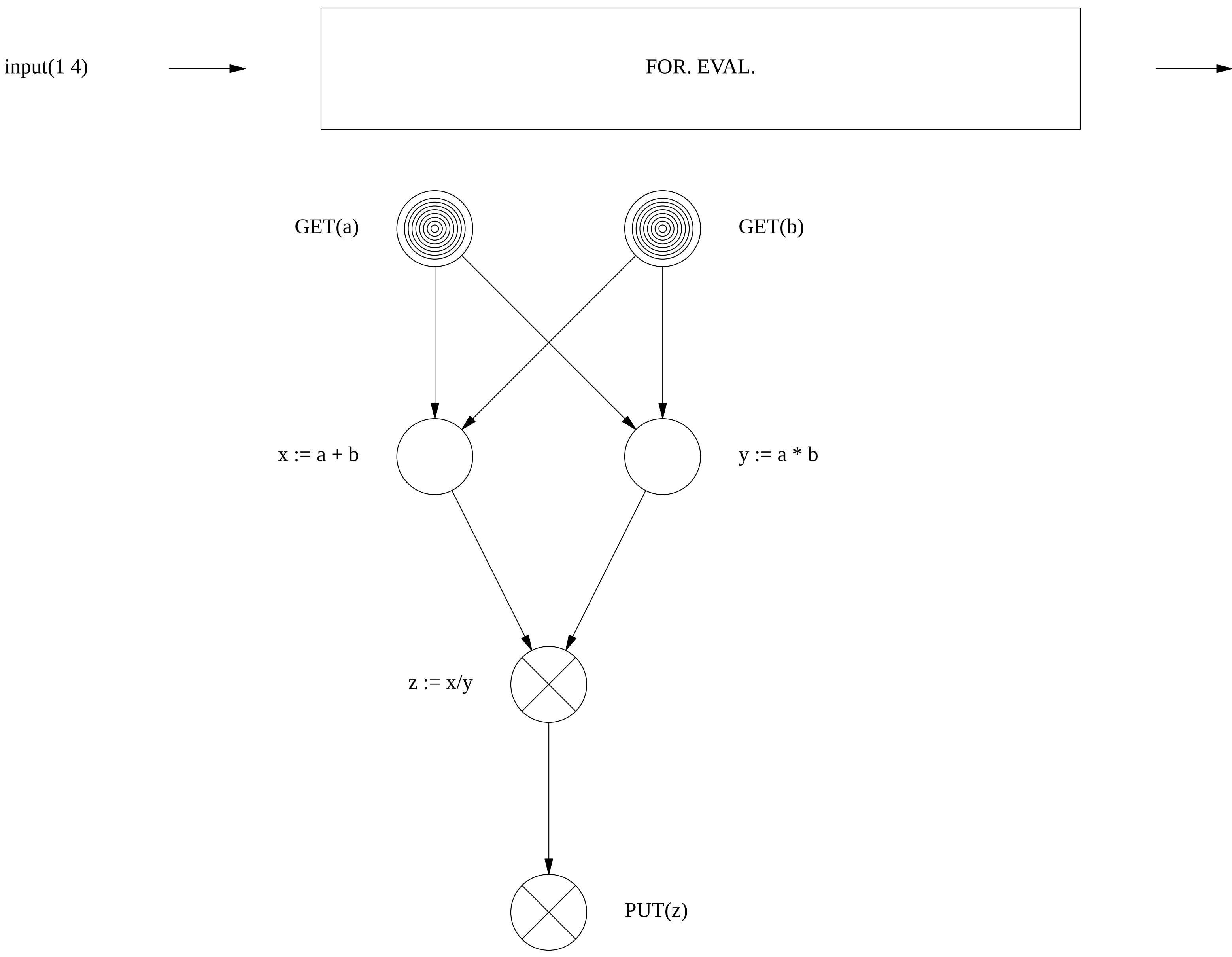

Figure 3.6

The above observation along with Lemma 3.1 gives us:

Theorem 3.1: Let P be a program with trace graph T. If POSSIBLE is the set constructed by Algorithm

3.1, when given input T, then

1) If POSSIBLE is empty, then P contains more than one error.
2) If POSSIBLE = $\{n_i\}$, then P can contain a single error, and if it does, it is represented in T by n_i.
3) If | POSSIBLE | > 1, then P can contain a single error, and if it does, it occurs at one of the nodes contained in POSSIBLE. Without further information from an oracle, the specific node cannot be identified.

4. FINDING UNINITIALIZED VARIABLES AND DEAD STATEMENTS.

Another question of interest concerning the model is: Given a class of errors, what information does a trace graph provide that helps to locate program errors from this class. In this section, uninitialized variables and dead statement error classes are examined. An error from the latter class is shown to be detectable from the interrelationship information provided by the trace graph, while an error from the former class is shown to be detectable during the construction of the trace graph.

4.1. Uninitialized Variables.

If a variable is referenced before it has been assigned a value, it is said to be an uninitialized variable. In Algorithm 2.1, when a variable reference is encountered, the hash table is checked. If there exists an entry in the hash table for the referenced variable, an edge is constructed; whereas, if no entry exists, no action is taken. Since a hash table entry is made for each value assignment at the time of the value assignment, not finding a table entry represents finding an uninitialized variable. Instead of taking no action when a reference does not exist in the hash table, an entry could be made to an error list stating what variable reference was missing from which record. The revised algorithm is presented below:

Algorithm 2.1 (revised): Building a trace graph.

Input. Trace an execution trace. Each record in TRACE contains the following fields:

program_statement_executed	:	integer,
action_taken	:	(input, output, assignment),
variables_referenced	:	list of variables,
variable_assigned	:	variable.

Output. A list of uninitialized variable errors and GRAPH, the trace graph of TRACE. Each record in GRAPH describes a node and is constructed by augmenting a record of TRACE with the following two fields:

in_edge_list	:	list of GRAPH records,
out_edge_list	:	list of GRAPH records.

Method. A table, referred to as HASH_TABLE, is used to keep track of value assignments to variables. Access to HASH_TABLE is through a hash function called HASH. The function HASH maps a variable X to the node of GRAPH in which X was last assigned a value. HAS_TABLE is initially empty.

```
for i := 1 to the number of records in TRACE do
    for every variable X referenced in TRACE[ i ] do
        if X is in HASH_TABLE
            then
                Add GRAPH[ i ] to the out-edges of HASH(X).
```

```
                    Add HASH(X) to the in-edges of GRAPH[ i ].
                else
                    Add X and TRACE[ i ] to list of uninitialized variable errors.
        if TRACE[ i ].action is input or assignment
            then
                if TRACE[ i ].variable_assigned is in HASH_TABLE
                    then
                        Update HASH(TRACE[ i ].variable_assigned) to have value i.
                    else
                        Insert TRACE[ i ].variable_assigned in HASH_TABLE with value i.
```

The list of uninitialized variable errors output in Algorithm 2.1 (revised) can be used as input to Algorithm 2.2 to construct a more complete labeling of the graph. Nodes that are currently labeled by the algorithm as unknown can potentially be labeled by the revised algorithm as deviant.

A revised labeling algorithm is presented below:

Algorithm 2.2 (revised): 0/1 labeling a trace graph.

Input. BAD_REFERENCE, a list of nodes referencing uninitialized variables. GRAPH, a trace graph obtained as output from Algorithm 2.1. Data from an oracle indicating which output results are correct and which are incorrect.

Output. A trace graph in which each node contains a field indicating the node's 0/1 label.

Method. Two queues are used: ONE, which holds unprocessed nodes that will be labeled with a "1", and ZERO, which holds unprocessed nodes that will be labeled with a "0". Both queues are initially empty.

```
for i := 1 to the number of records in GRAPH do
    case of (GRAPH[ i ].action) do
        input:
            Label GRAPH[i] with a "1".
        output:
            Label GRAPH[i] as indicated by the data from the oracle.
            Insert the node on in-edge list of GRAPH[i] in ONE or ZERO depending on label
            GRAPH[i].
        assignment:
            if GRAPH[i] in BAD_REFERENCE
                then
                    Insert in GRAPH[i] ZERO.
                else
                    Label GRAPH[i] as unknown.
    while ZERO is not empty do
        Remove node from ZERO.
        Label it with a "0".
        Insert all of its unlabeled children in ZERO.
    while ONE is not empty do
        Remove node from ONE.
        Label it with a "1".
        Insert all of its unlabeled parents in ONE.
```

To illustrate the amount of information that can be gained by marking uninitialized variables during trace graph construction, we build and label the trace graph of the DIVIDE & ROUND program presented in Figure 3.2 using the revised algorithms. In the trace graph built using the revised algorithms, (see Figure 4.1) no node is labeled as having unknown validity. Recall that in the original labeling, half of the nodes were labeled as having unknown validity. If we assume that the program contains a single fault, Algorithm 3.1, when given the labeled trace of Figure 4.1, will identify the error as being in statement 2. When given the labeled trace graph of Figure 3.2, the algorithm says a single error can be causing the faulty output and if so it is in statement 2, 3, 4, or 5.

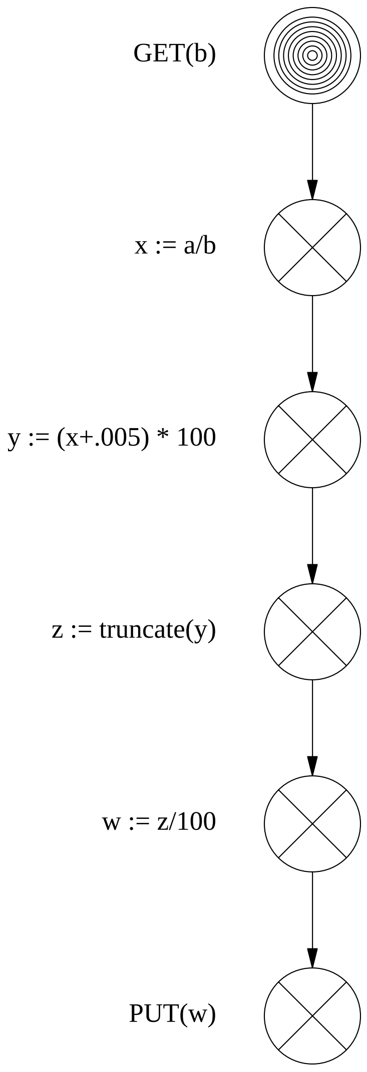

Figure 4.1

Since an execution trace represents only a single execution of the program, no claim is made that Algorithm 2.1 (revised) finds all uninitialized program variables. The best that can be said is that no uninitialized variables exist in any program statement appearing in the given trace.

4.2. Dead Statements.

We now look at a second class of errors that needs a completely built trace graph to obtain any helpful information. An assignment or input statement is said to be a dead statement if the variable which is assigned a value in the statement is never referenced after the assignment has been made or if the variable is reassigned a new value before being referenced. A variable reference in a trace graph is represented by an arc. The arc (r_i, r_j) in a trace graph indicates that a reference in r_j has been made to the value assigned in r_i. Thus, if an assignment or input node in a trace graph has no out-edges it can not have been referenced during the program's execution. For example, the statement GET(x) in Figure 2.1 is never referenced since the value of x assigned in the GET statement is overwritten with the value of y before a value for x is ever needed. In Figure 2.5, this shows itself as GET(x) having no out-edges.

Given a trace graph, it is straight forward to check each assignment and input node to see if its out-edge list is empty. If so, the node can be marked as a potential dead statement.

As was the case with uninitialized variables, no universal statement about dead variables can be made. Clearly, no remark about statements that do not appear in the program execution can be made. Also statements that are referenced in other executions, but are unreferenced in the execution being used, will be marked as dead.

5. FUTURE WORK.

The ideas and results in the previous sections open the way to a wealth of interesting problems. Below we mention several of these.

Relaxing The Assumptions: In the paper we have demonstrated how to model the execution of straight line programs using a labeled directed graph. Most programs make use of loops, conditionals, and subprogram calls. Each of these concepts needs to be explicitly provided for in the model.

In straight line programs, a program fault appears as an action in the trace graph at most once. When loops appear in the program a single program fault may be executed more than once, and hence, will be represented in the trace graph zero or more times. This turns error location into a two step process. In step one, the trace actions that represent faulty program statements must be identified. In step two, these must be "reduced" to the programming statements represented. One might want to help provide for this second step in the graph structure. A second type of arc could be used to link nodes that represent actions stemming from the same program statement. Although simplifying the error location problem, this type of link adds to the complexity of the graph construction process.

When a conditional is executed, the boolean expression is evaluated in exactly the same way as expressions included in assignment statements are evaluated when an assignment is made. Thus, many of the faults that appear in assignment statements can also appear in conditional statements. This indicates that conditionals should appear in some form in the trace. Exactly what form is what needs to be determined. Questions like "should some marking be included in the graph to indicate a path choice was made here" need to be addressed along with "should only the expression evaluation be represented or should the "whole" conditional be represented in some way?"

Subprogram calls appear to be the easiest concept to add to the present model. They can be treated as a series of complex assignment statements. A mapping between parameters and the actual arguments needs to be defined. This could then be used to alter HASH_TABLE, of Algorithm 2.1, before and after the subprogram statements have been added to the graph.

Accepting Nonconsistent Data: One of the underlining assumptions made about the program environment was that the data provided by the oracle lead to a consistent labeling of the graph. It is possible for a human oracle to violate this condition, see Figure 5.1.

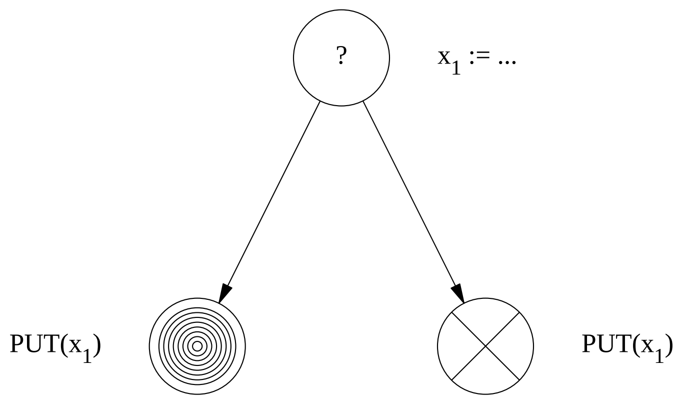

Figure 5.1

Because of this; some of the responsibility for seeing that the oracle is providing consistent data needs to be shifted from the oracle to the system. There are two ways in which this can be handled: batch or incrementally. In a batch system, the data is checked for consistency after all the data has been input. In an incremental system, the data is checked for consistency as it is being entered.

A modification of Algorithm 2.2 could be used in either type system to do the consistency checking. But the major work in a system is not in the checking of the data supplied by the oracle but in the generating of information to help the oracle. If the oracle has provided the system with one inconsistent set of validity values, then before the oracle is asked for another such set, information should be provided about the problems in the first set. The checking done by this type of algorithm is time consuming and involved. For example, if the graph in Figure 5.2 represents an execution for which the oracle is supplying values, and x_1 has been indicated as incorrect, then each of x_2, x_3, and x_4 must be input as incorrect. The system must discover this, inform the oracle of it, and monitor the system to see that the oracle does supply this value..

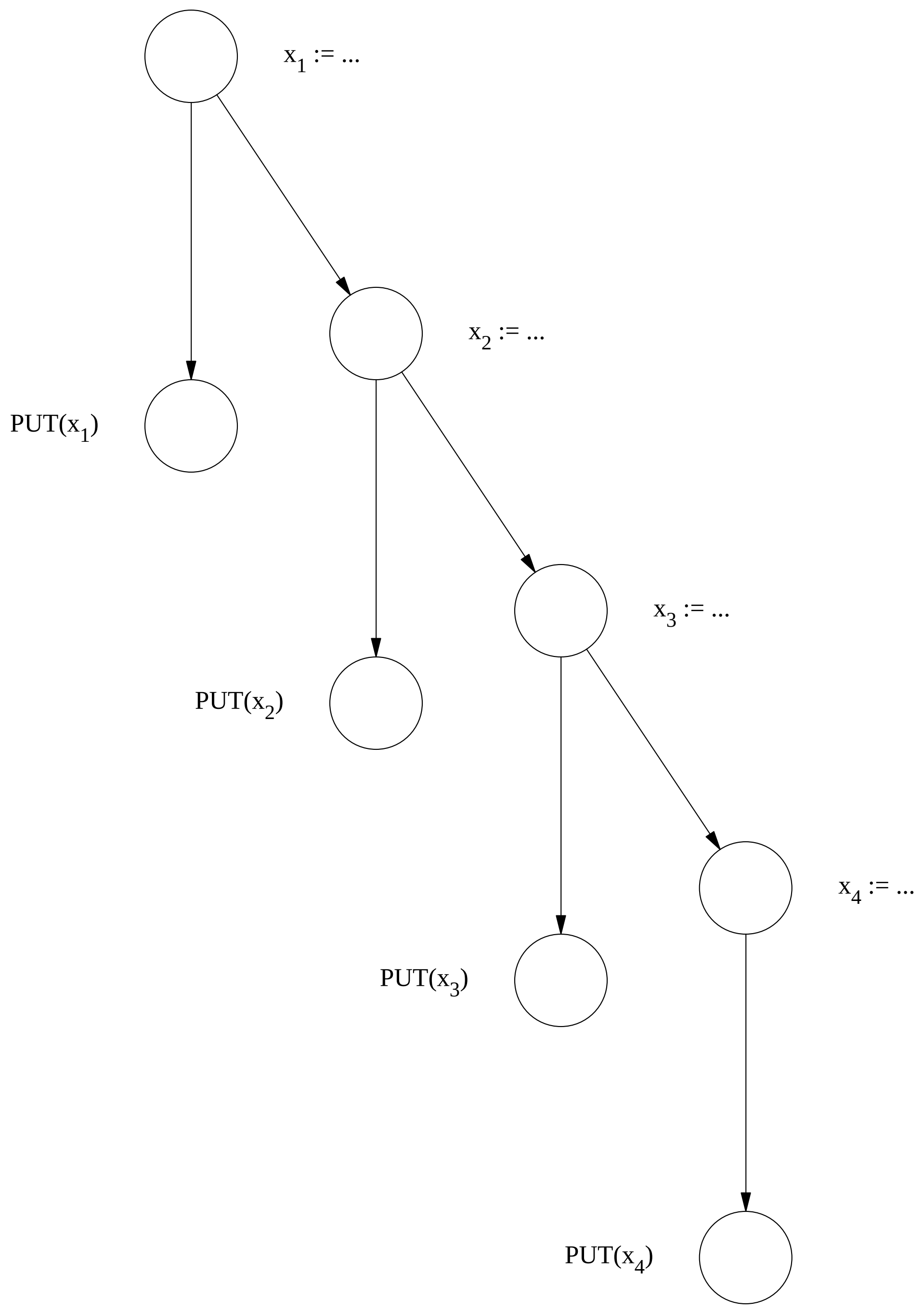

Figure 5.2

When an inconsistency is entered, or a value entered that leads to conclusions about other values unexceptable to the oracle, the oracle must be allowed to make changes. We use Figure 5.2 to show that this can have a rippling effect. Assume all output nodes are currently assumed incorrect, and that the oracle decides x_3 is correct. Then x_1 and x_2 must also be changed to correct. How does a system deal with this?

Finding the minimum number of errors. In section 3, we asked when a single fault could exist in a straight line program. A more general question would be to ask what is the minimum number of faults that can occur? In Figure 5.3, we see that two faults could be causing all the deviant actions. One would occur in n_5 and the other in either n_2 or n_4.

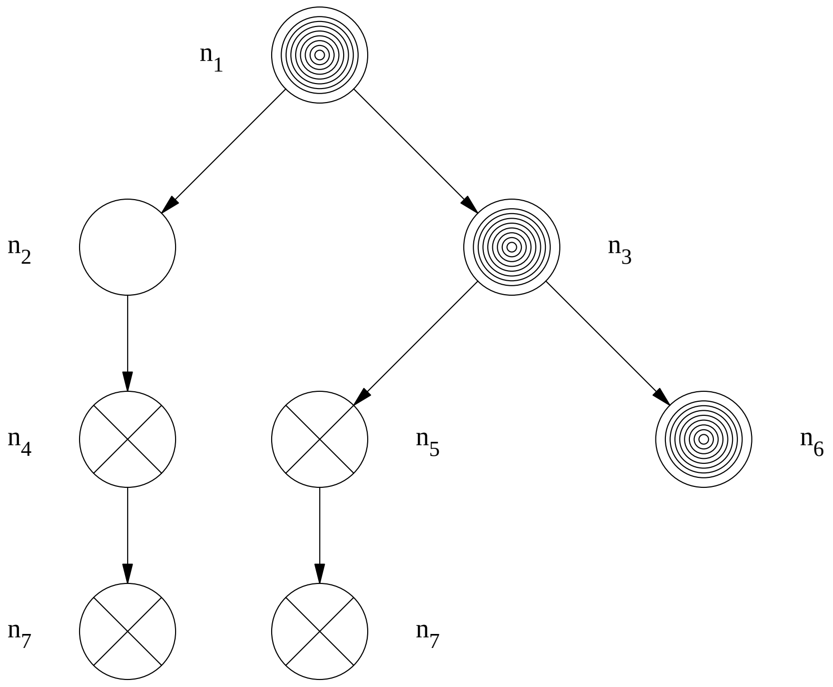

Figure 5.3

From this example, we see that the minimum fault situation will parallel the single fault situation in that the exact locations of the faults can not always be identified. In the example, one fault was pinpointed exactly while the other only identified as being part of a particular subgraph.

In finding a solution to the minimum error question, Reach-Out sets will need to be used. To generate these, a generalization of Algorithm 3.1 could prove useful. It also appears that some of the methods used in data flow analysis and dynamic programming will be applicable.

Of course once the multi-error problem is settled for straight line programs, the results need to be expanded so the same results can be determined when you have any program.

Oracle queries: As demonstrated several times in the paper, a fault position can not always be pinpointed exactly. It can however be restricted to a subset of the graph nodes. To pinpoint it more takes an oracle. Since one of the aims of the system is to minimize the work of an oracle, we want to determine what questions to ask the oracle that will produce the most results from the least amount of work. One problem to be considered is which node to ask the oracle about. Included in this is what happens when the action asked about is identified as deviant versus what happens when when the action is identified as correct. Figure 5.4 illustrates the two extreme situations that will need to be considered. Clearly there is no simple definitive answer to the question.

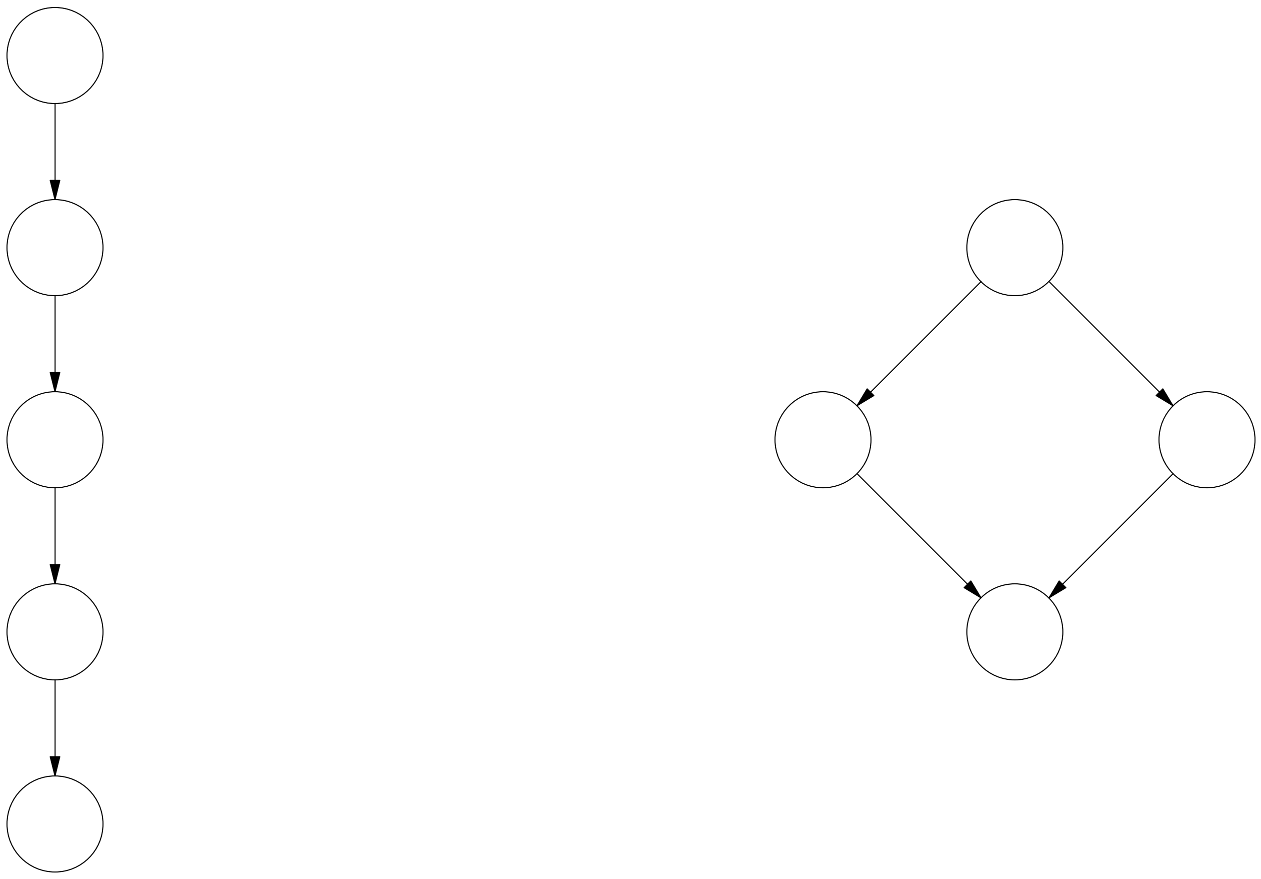

Figure 5.4

Another problem to be considered when querying an oracle is the amount of work needed to be done by the oracle to answer a question. If the position of two nodes in a graph provide the same amount of information, the one that takes less work on the part of the oracle to identify as correct or incorrect should be examined first.

Classes of errors: In mutation theory (see section 6 for more details) classes of faults are identified and discussed in relationship to testing. What can we say about these classes in relationship to fault identification? What other classes might be identified and discussed? This type of question is what is addressed in section 4. Much more needs to be done.

Multiple Traces: Since one trace has proven helpful in the fault detection process, can more than one trace provide us with even more information? Traces that produce trace graphs with identical structures can be used to identify differences in the labelings. A node with two different labels represents a deviant action rather than a faulty statement. What other information can be gained from having more than one trace? Can the analysis of a trace graph generate suggested test cases?

Real Programs: Some empirical studies need to be made. What types of graphs are produced by actual programs? Of special interest, is the average number of children and parents of a node because of Algorithm 3.1. It appears that the bound given is actually in practice more on the order of c * $|V|^2$, where c is a bound on the number of children and parents of a node.

6. THE LITERATURE.

Below we present several projects which have some relationship to the work presented in the paper.

In [S1], Shapiro develops algorithms to diagnose and correct errors in logic programs. A program is viewed as a series of procedure calls having inputs and outputs. During execution, the diagnosis system interprets a procedure and queries an external oracle, which determines if the procedure has produced the correct output on the given input. When done, the system returns the name of the incorrect procedure and the input that makes it fail. It is claimed that the number of oracle queries produced by the system is proportional to the logarithm of the number of procedure calls. This result is shown to to be minimal in the expected number of queries required.

In [R1], Renner presents the algorithms of [S1] adapted to work with Pascal programs. The relevant differences between Pascal and logic programming languages are identified, and the Shapiro algorithms are altered to deal with the problems caused by the differences. The complexity results of [S1] are not proven for the altered system, and in fact no longer appear to be true.

A slice is an independent program guaranteed to faithfully represent an original program within a domain of a specified subset of behavior [W1]. Starting from a subset of a program's behavior, a program can be reduced to a minimal form which still produces that behavior. This reduced program is the slice. In [W1], it is claimed that finding a slice is in general unsolvable. However, a dataflow algorithm is presented for approximating a slice when the behavior subset is specified as the values of a set of variables at a statement. In [W3], a dice is defined as a slice on incorrect variables from which slices on correct variables have been removed. Both [W2] and [W3] present empirical results on slicing and dicing. In [W2], experimental evidence is given to show that programmers mentally slice during debugging. A second experiment presented in [W3] further states that programmers using a dicing tool debugged their programs significantly faster than unaided programmers.

Program mutation is a technique for measuring the adequacy of test data. The technique is error-based. In error-based testing, the goal is to construct test cases that reveal the presence or absence of specific errors. Classes of errors have been defined, and case studies of mutation testing which investigate the effectiveness of mutation testing to uncover these classes have been done. For example, in [A1], a number of examples are provided to show how mutation operators can be used to uncover simple statement errors, dead code errors, domain errors, dead branch errors, data flow errors, special values errors, and coincidental correctness errors. We are interested in analyzing the error classes defined for mutation testing in our system; see section 4 for examples.

7. REFERENCES.

[A1] Acree, A.T., Budd, T.A., DeMillo, R.A., Lipton, R.J., and Sayward, F.G., :Mutation Analysis," Report GIT/ICS-79-08, Georgia Institute of Technology, 1979.

[E1] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 729-1983, Inst. Electrical and Electronics Eng., New York, 1983.

[R1] Scott Renner. "Diagnosis of logical errors in Pascal programs", Report UIUC-DCS-F-84_915 University of Illinois, Urbana, Apr.1984.

[S1] T.E. Shapiro. Algorithmic Program Debugging. MIT Press, Cambridge, MA. 1983.

[W1]
Mark Weiser. "Program Slicing". Proceeding of the Fifth International Conference on Software Engineering, San Diego, CA, March, 1981.

[W2]
Mark Weiser. "Programmers Use Slices When Debugging". Communications of the ACM 25, 7, pp. 446-452, July, 1982.

[W3]
Mark Weiser and Jim Lyle. "Experiments on Slicing-Based Debugging Aids". Emperirical Studies of Programmer, pp. 187-197, Ablex Publishing, Norwood, New Jersey, 1986.