Data Flow Anomaly Analysis

Contents

- Data flows and data flow anomalies
- State machine for data flow anomaly analysis
- Example without loops
- Example with loops

Data Flows and Data Flow Anomalies

- The data flow anomaly analysis guarantees the identification of certain faults (so-called data flow anomalies)
- The data flow w.r.t. to a certain variable on a particular execution path can be described by its sequence of definitions, references (p-uses and c-uses) and undefinitions (see data flow testing)
- Rules for data flows
  - A value must not be assigned twice to a variable (dd-anomaly)
  - An undefined variable must not be referenced (ur-anomaly)
  - The value of a variable must not be deleted directly after the value has been assigned (du-anomaly)
- These data flow anomalies can be detected by static analysis

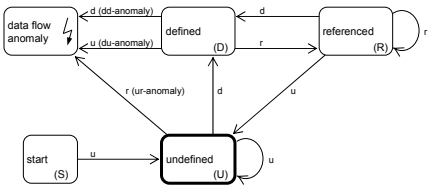
Data Flows and Data Flow Anomalies

- x is defined: d (defined)
  - The variable x is assigned a value (e.g. **x** = 5;)
- x is referenced: r (referenced)
  - The value of the variable x is read in a computation or in a decision, i.e., the value of x does not change (e.g. y = **x** + 1; or if (**x** > 0) ...)
- x is undefined: u (undefined)
  - The value of the variable x is deleted (e.g., deletion of local variables within a function or procedure at its termination). At program start all variables are undefined
- x is not used: e (empty)
  - The instruction of the node under consideration does not influence the variable x. x is not defined, referenced or undefined

Data Flows and Data Flow Anomalies

- Let us consider the two following data flows w.r.t. a variable (u: undefinition, d: definition, r: reference)
  - 1: u r d r u
  - 2: u d d r d u
- Sequence 1 begins with the pattern ur. The variable has a random value at the time of the reference, as it was not defined before. There is a data flow anomaly of the type ur; the reference of a variable with undefined, random value
- Sequence 2 contains two successive variable definitions. The first definition has no effect, as the value is always overwritten by the second definition. The data flow anomaly is of the type dd
- Sequence 2 ends with a definition followed by an undefinition. The value assigned by the definition is not used, as it is immediately deleted afterwards. This data flow anomaly is of the type du

State Machine for Data Flow Anomaly Analysis



- If the state *data flow anomaly* is reached or at the end of a data flow anomaly analysis the state *undefined* is not reached, a data flow anomaly is detected. The state machine defines a regular grammar. Such grammars are a standard case for compiler construction. In compilers they serve as a basis for the lexical analysis. Thus, data flow analysis can be integrated into compilers (which is the case in some compilers ➔ you should check whether your compiler can do it)

Example without Loops

- The operation MinMax gets two numbers via an interface which are to be returned ordered according to size

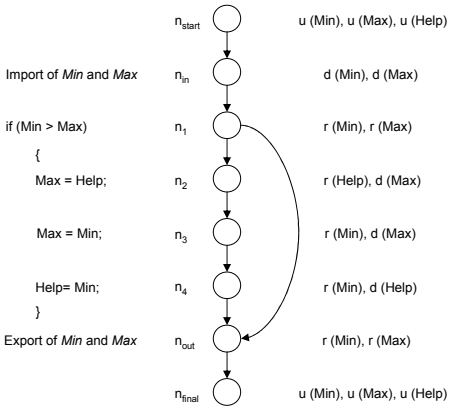
```
void MinMax (int& Min, int& Max)
{
    int Help;

        if (Min > Max)
        {
                Max = Help;
                Max = Min;
                Help = Min;
        }
    end MinMax;
}
```


Example without Loops

- Assignment of the data flow attributes w.r.t. the variables to the nodes of the control flow graph
  - u – deletion of a value (undefine)
  - d – value assignment (define)
  - r – reading a value (reference)
- Analysis of the data flows for the variables on the paths of the control flow graph

Example without Loops

- Control flow graph of MinMax



Example without Loops

- Data flows of MinMax

Variable \ Path	n_{start}	n_{in}	n_1	n_2	n_3	n_4	n_{out}	n_{final}	n_{start}	n_{in}	n_1	n_{out}	n_{final}
Min	u	d	r		r	r	r	u	u	d	r	r	u
Max	u	d	r	d	d		r	u	u	d	r	r	u
Help	u			r		d		u	u				u

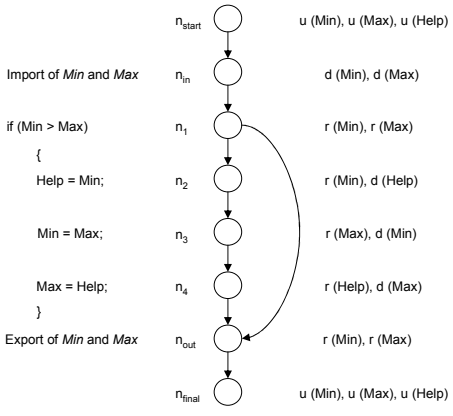
u – undefine d – define r - reference

Example without Loops

- The corrected version of the operation reads as follows

```
void MinMax (int& Min, int& Max)
{
    int Help;
    if (Min > Max)
    {
        Help = Min;
        Min = Max;
        Max = Help;
    }
    END MinMax;
}
```


Example without Loops



Example without Loops

Variable \ Path	n_{start}	n_{in}	n_1	n_2	n_3	n_4	n_{out}	n_{final}	n_{start}	n_{in}	n_1	n_{out}	n_{final}
Min	u	d	r	r	d		r	u	u	d	r	r	u
Max	u	d	r		r	d	r	u	u	d	r	r	u
Help	u			d		r		u	u				u

u – undefine d – define r - reference

Example with Loops

- Assumption: Data flow anomaly analysis must be done for all paths ➔ if the number of paths is to large, data flow anomaly analysis may not be feasible (reason: loops, see path testing)
- Fortunately this assumption is not correct
  - Concerning the data flow anomaly analysis it is sufficient to analyze the paths up to the first iteration of loops (the second execution of the loop body)
  - If no data flow anomalies occurred until then, it is ensured that also on the paths with a higher number of loop iterations no anomalies will occur

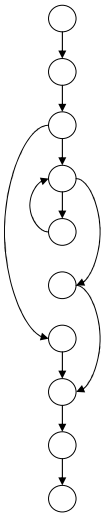
Example with Loops

- An operation uses Newtonian iteration as an approximation procedure in order to determine the square root
  - The operation should determine the square root for the non-negative inputs
  - For negative inputs the value 0.0 is to be returned
- Due to the approximation procedure it is difficult to give an estimation for the maximum number of iterations. This may cause an infinite number of paths (remark: If the loop is well-designed it should terminate for every input, so the number of iteration will be finite; but in this special case, this is hard to prove.)

Example with Loops

```
double Sqrt(double X)
{
    double returnValue;
    if (X > 0.0)
    {       double W;
            while (ABS(W*W-X) > 0.01)
            {
                    W = W - ((W*W-X) / (2.0 * W));
            }
            returnValue = W;
    }
    else
    {       returnValue = 0.0;
    }
    return (returnValue);
}
```

Example with Loops



n_{start}	u (X), u (W), u (returnValue)
n_{in}	d (X)
if (X > 0.0) ...	r (X)
while (ABS (W * W) – X) > 0.01 ...	r (X), r (W)
W = W – ((W * W – X) / (2.0 * W))	r (X), r (W), d (W)
returnValue = W	r (W), d (returnValue)
else returnValue = 0.0	d (returnValue)
return (returnValue)	r (returnValue)
n_{out}	
n_{final}	u (X), u (W), u (returnValue)

Example with Loops

- The analysis of the path which is passed through for non-positive input values is relatively simple
  - X: udru
  - W: uu
  - returnValue: udru
- None of these data flows contains anomalies
- For positive inputs the loop is executed. The data flow for the variable X begins with the sub-sequence udrr up to the loop decision. If the loop is not entered, the sub-sequence u follows directly. If the loop is entered, the sub-sequence rr edges itself in. This sub-sequence repeats with every further loop execution. Thus, the data flows on these paths can be given in complete form. The data flow for the variable X is: udrr$(rr)^n$u, with n >= 0. Value n represents the number of loop executions. For the variables W and returnValue also complete expressions for the data flows are received similarly
- X: udrr$(rr)^n$u, n>=0
- W: ur$(rdr)^n$ru, n >=0
- returnValue: udru

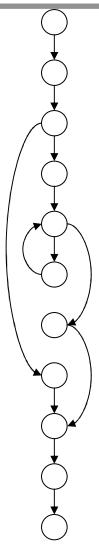
Example with Loops

- Question: Which values n have to be considered w.r.t. data flow anomaly analysis
  - Certainly the case n=0 has to considered, as a new sequence results due to the disappearance of the bracketed sub-sequence
  - The case n=1 also has to be considered, as two new sub-sequence emerge at the beginning of the bracketed expression and at its end
  - Furthermore the case n=2 is to be considered. For clarification the sequence ... $r(drd)^n r$ ... should be looked at which for n=0 and n=1 has no data flow anomalies, but for n=2 (... rdr**dd**rdr ...) shows a dd-anomaly
  - For greater values n no potential new data flow anomalies result. If no data flow anomaly on the paths up to the second loop execution has occurred yet, none will occur actually. The infinite number of paths has no influence on this

Example with Loops

- The operation sqrt for the variable W shows a data flow anomaly. The data flow $ur(rdr)^n ru$ begins with a ur-anomaly. The value of the variable W is not initialised yet at the time of the first reading access. However, the operation works correctly for random positive initial values of W, so that dynamic testing does not detect the fault reliably. For negative initial values of W the negative root is determined. If W by accident is initially equal zero, the program crashes, as a divide by zero occurs. While by dynamic testing this fault can be detected only unreliably, it is identifiable by data flow anomaly analysis reliably and at very low costs

Example with Loops



n_{start}	u (X), u (W), u (returnValue)
n_{in}	d (X)
if (X > 0.0) ...	r (X)
W = 1.0	d (W)
while (ABS (W * W) – X) > 0.01 ...	r (X), r (W)
W = W – ((W * W – X) / (2.0 * W))	r (X), r (W), d (W)
returnValue = W	r (W), d (returnValue)
else returnValue = 0.0	d (returnValue)
return (returnValue)	r (returnValue)
n_{out}	
n_{final}	u (X), u (W), u (returnValue)