

Data-flow Analysis Animation - 4ba7 Advanced Compiler Design Coursework

John Gilbert (00720836)

March 5, 2004

Contents

1	Introduction	1
2	Animation GUI description	2
3	Implementation	3
4	Conclusions	4
5	Annotated code listing	4

1 Introduction

Event-class problems can be described using systems of boolean equations and solved with data-flow analysis. An iterative algorithm exists for solving these systems and is demonstrated in the accompanying Vivio animation under a number of conditions. The animation is not a substitute for reading the associated reference material on the subject as it does not sufficiently demonstrate the 'general solution' nor all of its applications; rather is it a tool to aid students comprehend two concrete applications of the technique thus paving the way for a genuine understanding of the generalised algorithm and its use in the construction of a compiler. The animation has two phases: initially global live variable analysis is performed on a variable followed by redundant code suppression as described in [1].

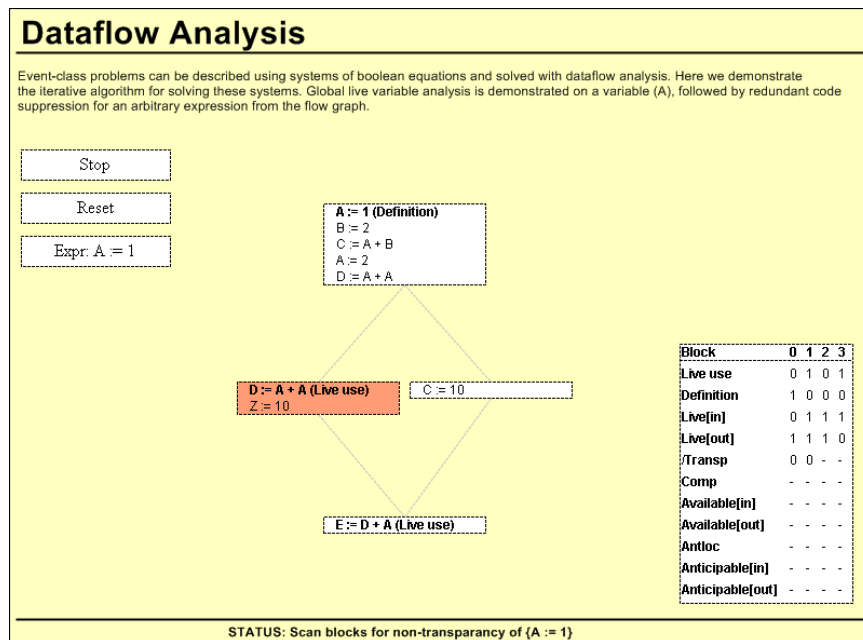


Figure 1: Scanning for local properties

2 Animation GUI description

Before global properties may be computed, associated local properties must be collected: this is indicated by highlighting the basic block which is being scanned and simultaneously updating the status indicator displayed at the bottom of the animation.

The computation of global properties is visually indicated by 'querying' the connection between a basic block and those which it is connected to. During the evaluation of global properties, intermediate values are shown on the flow graph at the entry and exit of each basic block in addition to being displayed in the properties table described next.

A table of boolean properties for both global and local properties is displayed where '0' indicates that a property is not present, '1' indicates a property is present and '-' indicates a property has not yet been computed.

To enable students work through examples step by step and convince themselves that the algorithm is correct, the internal property 'finished' of the iterative algorithm is displayed while the iterative algorithm is being executed. This coupled with the boolean table of properties provides all the information required to follow the algorithms steps.

While the animation is predominantly a static demonstration, users may select the expression upon which redundant code suppression will be performed by using the 'Expr' button on the left of the screen. The variable used for live variable analysis can not be changed by the user at run time.

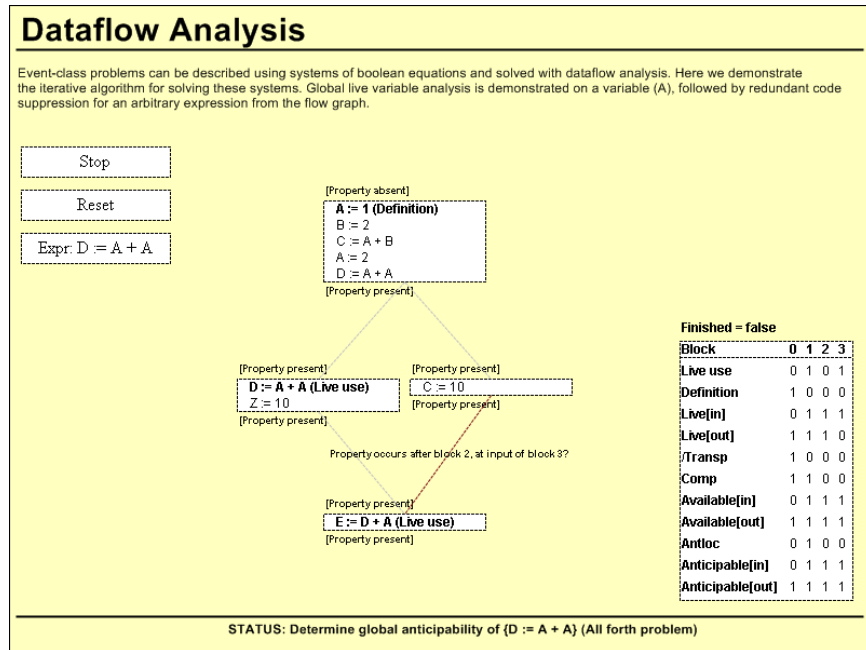


Figure 2: Computing global properties

3 Implementation

The underlying basic blocks for the algorithm and user interface are described using a jagged 2D array, where the first index indicates the basic block number and the second the tuple within that basic block. The arrays elements are instances of the Quadruple class. To indicate connectedness between the basic blocks, a square 2D array is used, where an index of (out, in) with value 1 states that control may pass from the output of block 'out' to the input of block 'in', while a value of 0 indicates that no such control pass can take place. A single integer variable stores the number of blocks within the flow graph, and a 1D array of integers stores the number of tuples in each basic block.

The Quadruple class encapsulates the required data for a single operation in the form of a quad. The value '_' is a default / no-value value, and a selection of get / set methods are provided for manipulating the classes member data.

For the determination of local properties a collection of helper-functions are provided: live_use, def, transp, not_transp, locally_available and locally_anticipable. These all return integer boolean results. The full signature for each of these is available in the annotated source code attached, and gives a description of what is computed.

The dfa_iterative function implements the generalised iterative algorithm mentioned above. As parameters it takes an array of the local events and anti-events for each block and the properties that characterise the problem as either a one / all and forth / back

problem. These properties are integer boolean values. Finally an index into the property table of results for this computation is taken, and should point to the position where properties at the input of basic blocks are to be stored, with a free space at location (index + 1) for the exit properties.

Simple code is included to layout and connect the visual representation of the flow graph. This generates lines based on the 'connectedness' array, and while for simple flow graphs proves sufficient will not currently show back edges particularly well.

Following this code the specific animation described in the previous section is implemented. This is straightforward and following the computation of all local and global boolean properties redundant code is identified for the expression under consideration and visually identified by striking the quad through in the GUI.

4 Conclusions

The Vivio animation presented here will be a useful tool for demonstrating data-flow analysis. In retrospect it might have been better to demonstrate global live variable analysis separately from redundant code suppression, to avoid overloading the function of a single demonstration tool. This would be simple to implement, requiring a duplicate copy of the current animation with minor modifications to the property table and the functions that are called. The reason the two applications were demonstrated together in the animation was to show a simple application before a more in-depth analysis.

Future work on the animation will allow users input arbitrary flow graphs: the current implementation contains a hard-coded flow graph which can only be changed by modifying the Vivio source code and then recompiling. This is a straightforward process but in order to make full use of the environment Vivio provides, more than a static demonstration would be nice.

An outstanding function required by the existing animation is to display back-edges on the GUI. While the algorithm as implemented will work perfectly, the current method used for connecting basic blocks on the GUI may cause confusion if a flow graph which uses back edges is inputted in the source code.

5 Annotated code listing

```
//-----
// Required includes for using a vivio 'SimpleButton'.
//
#include "standard.vin"
#include "simplebutton.vin"

//-----
// Quadruple {op_type, dest, left_operand, right_operand}
// : initially {-,-,-,-} where '-' denotes 'unused' or n/a
//
// operand types
const int ot_value = 0;
```

```

const int ot_name = 1;
const int ot_unused = 2;

// operand indexes
const int left_op = 0;
const int right_op = 1;

// class to represent a quadruple
class Quadruple()
{
    operator_type = "_";
    dest_name = "_";

    operand_name[0] = "_";
    operand_name[1] = "_";

    operand_value[0] = 0;
    operand_value[1] = 0;

    operand_type[0] = ot_unused;
    operand_type[1] = ot_unused;

    string visual_ref = "";

    // determine the type of an operand
    int function get_type(int index)
    {
        return operand_type[index];
    }
    end;

    // set the type and name of an operand
    function set_name(int index, string name)
    {
        operand_name[index] = name;
        operand_type[index] = ot_name;
    }
    end;

    string function get_name(int index)
    {
        return operand_name[index];
    }
    end;

    // set the type and value of an operand
    function set_value(int index, int value)
    {
        operand_value[index] = value;
        operand_type[index] = ot_value;
    }
    end;

    int function get_value(int index)
    {
        return operand_value[index];
    }
    end;

    function set_dest(string name)
    {
        dest_name = name;
    }
    end;

    string function get_dest()
    {
        return dest_name;
    }
    end;

    function set_operator(string type)
    {
        operator_type = type;
    }
    end;

    string function get_operator()
    {
        return operator_type;
    }
    end;

    // determine the number of operands used by the quadruple
    int function num_operands()
    {
        if((operand_type[0] != ot_unused) && (operand_type[1] != ot_unused))
            return 2;
        end;

        if((operand_type[0] != ot_unused) || (operand_type[1] != ot_unused))
            return 1;
        end;

        return 0;
    }
    end;

    // format the quadruple for printing on-screen
    string function as_string()
    {
        // need to make this more robust and take into account

```

```

// the operator.

no = num_operands();

if(no == 1)
  if(operand_type[0] == ot_name)
    visual_ref.format("%s := %s", dest_name, operand_name[0]);
  else
    visual_ref.format("%s := %d", dest_name, operand_value[0]);
  end;
return visual_ref;
end;

if(no == 2)
  if(operand_type[0] == ot_name)
    if(operand_type[1] == ot_name)
      visual_ref.format("%s := %s %s %s", dest_name, operand_name[0], operator_type, operand_name[1]);
    else
      visual_ref.format("%s := %s %s %d", dest_name, operand_name[0], operator_type, operand_value[1]);
    end;
  else
    if(operand_type[1] == ot_name)
      visual_ref.format("%s := %d %s %s", dest_name, operand_value[0], operator_type, operand_name[1]);
    else
      visual_ref.format("%s := %d %s %d", dest_name, operand_value[0], operator_type, operand_value[1]);
    end;
  end;
return visual_ref;
end;

return "[? No operands ?]";

end;

end;

//-----
// IR of a flow graph, upon which analysis will be performed. Changing this
// will require 'tweaking' of the layout code for the GUI, to property position
// basic blocks onscreen.
//
// number of blocks in the flow graph
const int num_blocks = 4;

// number of tuples in each block of the graph
num_tuples[0] = 5;
num_tuples[1] = 2;
num_tuples[2] = 1;
num_tuples[3] = 1;

// how the blocks are connected together
// [out, in] = 0 states that there is no connection from block_out to block_in
// [out, in] = 1 states that there is a connection from block_out to block_in
graph[0,0] = 0;
graph[0,1] = 1;
graph[0,2] = 1;
graph[0,3] = 0;
graph[1,0] = 0;
graph[1,1] = 0;
graph[1,2] = 0;
graph[1,3] = 1;
graph[2,0] = 0;
graph[2,1] = 0;
graph[2,2] = 0;
graph[2,3] = 1;
graph[3,0] = 0;
graph[3,1] = 0;
graph[3,2] = 0;
graph[3,3] = 0;

// A := 1
flow_graph[0,0] = Quadruple();
flow_graph[0,0].set_dest("A");
flow_graph[0,0].set_value(left_op, 1);

// B := 2
flow_graph[0,1] = Quadruple();
flow_graph[0,1].set_dest("B");
flow_graph[0,1].set_value(left_op, 2);

// C := A + B
flow_graph[0,2] = Quadruple();
flow_graph[0,2].set_dest("C");

```

```

flow_graph [0,2].set_operator("+");
flow_graph [0,2].set_name(left_op, "A");
flow_graph [0,2].set_name(right_op, "B");

// A := 2
flow_graph [0,3] = Quadruple();
flow_graph [0,3].set_dest("A");
flow_graph [0,3].set_value(left_op, 2);

// D := A + A
flow_graph [0,4] = Quadruple();
flow_graph [0,4].set_dest("D");
flow_graph [0,4].set_operator("+");
flow_graph [0,4].set_name(left_op, "A");
flow_graph [0,4].set_name(right_op, "A");

// D := A + A
flow_graph [1,0] = Quadruple();
flow_graph [1,0].set_dest("D");
flow_graph [1,0].set_operator("+");
flow_graph [1,0].set_name(left_op, "A");
flow_graph [1,0].set_name(right_op, "A");

// Z := 10
flow_graph [1,1] = Quadruple();
flow_graph [1,1].set_dest("Z");
flow_graph [1,1].set_value(left_op, 10);

// C := 10
flow_graph [2,0] = Quadruple();
flow_graph [2,0].set_dest("C");
flow_graph [2,0].set_value(left_op, 10);

// E := D + A
flow_graph [3,0] = Quadruple();
flow_graph [3,0].set_dest("E");
flow_graph [3,0].set_operator("+");
flow_graph [3,0].set_name(left_op, "D");
flow_graph [3,0].set_name(right_op, "A");

//-----
// Declarations of objects which will be used in the vivio animation GUI
//

// Font style declarations
const int bold = 1;
const int italic = 2;
const int underline = 4;
const int strike.through = 8;

// Size of the vivio animation
const int width = 800;
const int height = 600;
setviewport(0, 0, width, height, 0);

// Brush declarations
graybrush = SolidBrush(rgb(192, 192, 192));
blackbrush = SolidBrush(rgb(0, 0, 0));
lightyellowbrush = SolidBrush(rgb(250, 252, 224));
lightredbrush = SolidBrush(rgb(255, 155, 117));
darkredbrush = SolidBrush(rgb(121, 6, 10));
whitebrush = SolidBrush(rgb(255, 255, 255));
bbrush = SolidBrush( rgb( 255, 255, 192 ) );

// Pens declarations
thickblackpen = SolidPen(0, 2, rgb(0,0,0));
blackpen = SolidPen(1, 1, rgb(0,0,0));
graypen = SolidPen(1, 1, rgb(192, 192, 192));
darkredpen = SolidPen(1, 1, rgb(121, 6, 10));

// Font declarations
arial = Font("Arial", 12);
arial.italic = Font("Arial", 12, italic);
arial.bold = Font("Arial", 12, bold);
arial.stthrough = Font("Arial", 12, strike.through + bold);
arial.bold_small = Font("Arial", 9, bold);
arial.bold_hdr = Font("Arial", 30, bold);
arial.bold.italic = Font("Arial", 12, bold + italic);
arial.bold.italic.underline = Font("Arial", 12, bold + italic + underline);

// Basic UI (heading, status, background colour)
const int bblock_width_ui = 150; // width of a basic block on the GUI
Text(0, 0, 3, 5, 5, blackbrush, arial_bold_hdr, "Dataflow Analysis");
Line(0, 0, 1, thickblackpen, 5, 40, 790, 40);

```

```

Line(0, 0, 1, thickblackpen, 5, height - 35, 790, height - 35);
current_operation = Text(0, 0, 3, width/4, height - 30, blackbrush, arial_bold, "");
setbackground(bgbrush);

//-----
// Table of properties which will display results of local scanning
// for properties, aswell as those computed globally. While hard-coded for the
// number of properties used in this animation, it will scale if new
// basic blocks are added.
//
table_of_properties = Group();

const int box_x = 0;
const int box_y = 0;
const int indent = 100;

const int num_properties = 12;

prop_title[0] = "Block";
prop_title[1] = "Live use";
prop_title[2] = "Definition";
prop_title[3] = "Live [in]";
prop_title[4] = "Live [out]";
prop_title[5] = "/Transp";
prop_title[6] = "Comp";
prop_title[7] = "Available [in]";
prop_title[8] = "Available [out]";
prop_title[9] = "Antloc";
prop_title[10] = "Anticipable [in]";
prop_title[11] = "Anticipable [out]";

prop_table = Rectangle(0, table_of_properties, 0, blackpen, whitebrush, box_x, box_y,
    indent + (15 * num_blocks), 20 * num_properties);
divider_line = Line(0, table_of_properties, 0, blackpen, box_x, box_y + 15,
    box_x + indent + (15 * num_blocks), box_y + 15);

// property names
for(i = 0; i < num_properties; i++)
    Text(0, table_of_properties, 0, box_x, box_y + (20 * i), blackbrush, arial_bold, prop_title[i]);
end;

// block numbers
for(i = 0; i < num_blocks; i++)
    Text(0, table_of_properties, 0, box_x + indent + (15*i), box_y, blackbrush, arial_bold, "%d", i);
end;

// default values
for(i = 0; i < num_properties - 1; i++)
    for(j = 0; j < num_blocks; j++)

        // set to '-' in the box.
        prop_table_ui[i, j] = Text(0, table_of_properties, 0, box_x + indent + 15 * j,
            box_y + 20 * (i + 1), blackbrush, arial, "-");

        // initialize a 2d array for storing the results as INTs. this is ONLY valid for
        // results computed inside dfa_iterative, and is required for redundant code detection
        // as we must later verify where antloc[in] * available[in] is true.
        dfa_results[i, j] = 0;

end;
end;

table_of_properties.setpos(width - indent - (num_blocks * 20), height - num_properties * 20 - 50);

//-----
// 'Finished' indicator for the iterative algorithm. Only shown while running
//
fbox = Text(0, 0, 0, width - indent - (num_blocks * 20), height - num_properties * 20 - 70,
    blackbrush, arial_bold, "");

//-----
// Basic GUI -- build a manipulatable UI for the flow graph
//
// create a group for each basic block
for(i = 0; i < num_blocks; i++)

    flow_layout[i] = Group();
    line_num = 0;

    flow_graph_ui_boxes[i] = Rectangle(0, flow_layout[i], 0, blackpen, whitebrush, 0, 0, bblock_width_ui,
        15 * num_tuples[i]);

```

```

str_prop_in[i] = Text(0, flow_layout[i], 0, 0, -15, blackbrush, arial_bold_small, "");
str_prop_out[i] = Text(0, flow_layout[i], 0, 0, 15 * num_tuples[i] + 3, blackbrush, arial_bold_small, "");

for(j = 0; j < num_tuples[i]; j++)
    flow_graph_ui[i, j] =
        Text(0, flow_layout[i], 0, 10, line_num, blackbrush, arial, flow_graph[i, j].as_string());
    line_num = line_num + 15;
end;

end;

// position the basic blocks on the screen.
// -- until we have a graph layout algorithm we have to do it by hand :- (
left = 290;
flow_layout[0].setpos(left, 180);
flow_layout[1].setpos(left - 80, 345);
flow_layout[2].setpos(left + 80, 345);
flow_layout[3].setpos(left, 470);

// connect the basic blocks with lines to indicate flow
for(i = 0; i < num_blocks; i++)

    bottom_x = flow_layout[i].getx() + (bblock_width_ui / 2);
    bottom_y = flow_layout[i].gety() + flow_graph_ui_boxes[i].geth();

    for(j = 0; j < num_blocks; j++)
        if(graph[i, j] == 1)

            // for the moment we assume there are no back links (for layout)
            top_x = flow_layout[j].getx() + (bblock_width_ui / 2);
            top_y = flow_layout[j].gety();

            flow_graph_ui_lines[i, j] = Line(0, 0, 1, graypen, bottom_x, bottom_y, top_x, top_y);
            flow_graph_ui_lines[i, j].setfont(arial_bold_small);

            // need to deal with backlinks better!

        end;
    end;

end;

//-----
// Local Use: determines if variable_name is used live in block[basic_block]
//
int function live_use(string variable_name, int basic_block)

for(int tuple = 0; tuple < num_tuples[basic_block]; tuple++)

    // are either of the right or left operands used before being
    // defined in this quadruple (nb. must be of correct type for
    // comparison!) ?

    if(flow_graph[basic_block, tuple].get_type(left_op) == ot_name)
        if(flow_graph[basic_block, tuple].get_name(left_op) == variable_name)
            flow_graph_ui[basic_block, tuple].settext(flow_graph[basic_block, tuple].as_string() + " (Live use)");
            flow_graph_ui[basic_block, tuple].setfont(arial_bold);
            return 1;
        end;
    end;

    if(flow_graph[basic_block, tuple].get_type(right_op) == ot_name)
        if(flow_graph[basic_block, tuple].get_name(right_op) == variable_name)
            flow_graph_ui[basic_block, tuple].settext(flow_graph[basic_block, tuple].as_string() + " (Live use)");
            flow_graph_ui[basic_block, tuple].setfont(arial_bold);
            return 1;
        end;
    end;

    // if theres an assignment to the variable in this quadruple
    // then theres no live use in the block (would have returned 1 if
    // either operands of this quad were the variable being used-live)

    if(flow_graph[basic_block, tuple].get_dest() == variable_name)
        return 0;
    end;

end;

// if no reference was made then there is no live use here.
return 0;

```

```

end;

//-----
// Def: determines if a definition of a variable takes place in block[basic_block]
//
int function def(string variable_name, int basic_block)

// if the variable is the destination for any quadruple
// then the variable has a definition in this basic block

for(int tuple = 0; tuple < num_tuples[basic_block]; tuple++)
  if(flow_graph[basic_block, tuple].get_dest() == variable_name)
    flow_graph_ui[basic_block, tuple].setfont(arial_bold);
    flow_graph_ui[basic_block, tuple].settext(flow_graph[basic_block, tuple].as_string() + " (Definition)");
    return 1;
  end;
end;

// if nothing assigns to variable_name in this block return false
return 0;

end;

//-----
// Transp: determines if an expression (quadruple) is transparent in
// block[basic_block]. An expression is transparent in a block if its operands
// are not modified by execution of the commands in the block/
//
int function transp(Quadruple expr, int basic_block)

// for each operand, if the operand is a variable name,
// if there is an assignment to the operand then the block
// is not transparent for the expression.

for(int i = 0; i < expr.num_operands(); i++)
  if(expr.get_type(i) == ot_name)
    if(def(expr.get_name(i), basic_block) == 1)
      return 0;
    end;
  end;
end;

// otherwise it is.
return 1;

end;

// handy helper method

int function not_transp(Quadruple expr, int basic_block)

  if(transp(expr, basic_block))
    return 0;
  end;

  return 1;

end;

//-----
// Comp: determines if an expression is locally available in a block. An expr
// is locally available in a block if there is at least one computation of the
// expr in the block, and if its operands are not modified by the commands
// appearing after its last computation in a block.
//
// need these temps otherwise vivio goes mad.
string t1 = "", t2 = "", t3 = "", t4 = "";

int function locally_available(Quadruple expr, int basic_block)

// starting with the last tuple in the block, search backwards up the
// block until we hit a computation of the expression, or the first tuple.
// if a definition occurs of any of the expressions operands before we hit
// the computation, it is not locally available!

for(int i = 0; i < num_tuples[basic_block]; i++)

  // two expressions are equal if they have the same string representation

  t1 = expr.as_string();
  t2 = flow_graph[basic_block, num_tuples[basic_block] - (1 + i)].as_string();

```

```

    if(t1 == t2)
        return 1;
    end;

    // if an operand is modified, the computation is not locally available.
    for(int j = 0; j < expr.num_operands(); j++)
        if(expr.get_type(j) == ot_name)

            t3 = flow_graph[basic_block, num_tuples[basic_block] - (1 + i)].get_dest();
            t4 = expr.get_name(j);

            if(t3 == t4)
                return 0;
            end;

        end;
    end;

    // if no computation, nor any modification ==> not comp.
    return 0;

end;

//-----
// Antloc: Locally Anticipability..
// An expression is locally anticipable if there is a computation of the
// expr within the basic block, and if the commands appearing in the block before
// the first computation of the expression do not modify its operands.
//
int function locally_anticipable(Quadruple expr, int basic_block)

    for(int i = 0; i < num_tuples[basic_block]; i++)

        // two expressions are equal if they have the same string representation
        t1 = expr.as_string();
        t2 = flow_graph[basic_block, i].as_string();

        if(t1 == t2)
            return 1;
        end;

        // if an operand is modified, the computation is not locally available.
        for(int j = 0; j < expr.num_operands(); j++)
            if(expr.get_type(j) == ot_name)

                t3 = flow_graph[basic_block, i].get_dest();
                t4 = expr.get_name(j);

                if(t3 == t4)
                    return 0;
                end;

            end;
        end;

    // if no computation, nor any modification ==> not antloc.
    return 0;

end;

//-----
// Iterative Data Flow Analysis
//
// e[*] -- event takes place in block *
// ae[*] -- anti event takes place in block *
// x[*] -- property present at the input of block *
// x_o[*] -- property present at the output of block *
// prop_index -- base index into the property table for this result.
//
int x[*];
int x_o[*];

function dfa_iterative(int e[*], int ae[*], int forth, int one, int prop_index)

    // initialize the Xi
    // true for all problems
    // false for one problems

```

```

int finished = 1;
fbox.settext("Finished = true");

for(int k = 0; k < num_blocks; k++)

if(one == 1)
x[k] = x_o[k] = 0;
str_prop_in[k].settext("[Property absent]");
str_prop_out[k].settext("[Property absent]");
else
x[k] = x_o[k] = 1;
str_prop_in[k].settext("[Property present]");
str_prop_out[k].settext("[Property present]");
end;

prop_table_ui[prop_index, k].settext("%d", x[k]);
prop_table_ui[prop_index + 1, k].settext("%d", x_o[k]);

dfa_results[prop_index, k] = x[k];
dfa_results[prop_index + 1, k] = x_o[k];

end;

wait(5);

// solve the DFA equations iteratively
repeat

finished = 1;
fbox.settext("Finished = true");

for(int i = 0; i < num_blocks; i++)

int new_xi = e[i];
int yi = 0;
int first_pass = 1;

// in represents a successor of i if graph[i][in] AND its a forth problem ,
// // else it represents a predecessor if graph[in][i] AND its a back problem.
// yi calculates the presence of the property after/before this point
for(int in = 0; in < num_blocks; in++)
if( ( graph[i, in] == 1 && forth == 1 ) || ( graph[in, i] == 1 && forth != 1 ) )

if(forth == 1)
flow_graph_ui_lines[i, in].setpen(darkredpen);
flow_graph_ui_lines[i, in].settext("Property occurs after block %d, at input of block %d?", i, in);
else
flow_graph_ui_lines[in, i].setpen(darkredpen);
flow_graph_ui_lines[in, i].settext("Property occurs before block %d, at output of block %d?", i, in);
end;

int correct_x_val = -1;
if(forth == 1)
correct_x_val = x[in];
else
correct_x_val = x_o[in];
end;

if(first_pass == 1)
first_pass = 0;
yi = correct_x_val;
else

// operator is and for all problems, or for one
if(one == 1)
yi = yi | correct_x_val;
else
yi = yi & correct_x_val;
end;

end;

wait(3);

if(correct_x_val == 1)
if(forth == 1)
flow_graph_ui_lines[i, in].settext("Property occurs after block %d, at input of block %d? Yes", i, in);
else
flow_graph_ui_lines[in, i].settext("Property occurs before block %d, at output of block %d? Yes", i, in);
end;
else
if(forth == 1)
flow_graph_ui_lines[i, in].settext("Property occurs after block %d, at input of block %d? No", i, in);

```

```

else
    flow_graph_ui_lines[in, i].settext("Property occurs before block %d, at output of block %d? No", i, in);
end;
end;

wait(3);

if(forth == 1)
    flow_graph_ui_lines[i, in].settext("");
    flow_graph_ui_lines[i, in].setpen(graypen);
else
    flow_graph_ui_lines[in, i].settext("");
    flow_graph_ui_lines[in, i].setpen(graypen);
end;

end;
end;

// update ui based on the yi result

if(forth == 1)
    x_o[i] = yi;
    prop_table_ui[prop_index + 1, i].settext("%d", x_o[i]);
    dfa_results[prop_index + 1, i] = x_o[i];
    if(yi == 1)
        str_prop_out[i].settext("[Property present]");
    else
        str_prop_out[i].settext("[Property absent]");
    end;
else
    x[i] = yi;
    prop_table_ui[prop_index, i].settext("%d", x[i]);
    dfa_results[prop_index, i] = x[i];
    if(yi == 1)
        str_prop_in[i].settext("[Property present]");
    else
        str_prop_in[i].settext("[Property absent]");
    end;
end;

// if the anti-event does not occur, and the property is present
// after this point, then its present at this point.
// (t_la = transparent, live-after)
int t_la = 0;
if((ae[i] == 0) && (yi != 0))
    t_la = 1;
end;

new_xi = new_xi | t_la;

if(new_xi == 1)
    if(forth == 1)
        str_prop_in[i].settext("[Property present]");
    else
        str_prop_out[i].settext("[Property present]");
    end;
else
    if(forth == 1)
        str_prop_in[i].settext("[Property absent]");
    else
        str_prop_out[i].settext("[Property absent]");
    end;
end;

if(forth == 1)
    if(new_xi != x[i])
        finished = 0;
        fbox.settext("Finished = false");
        x[i] = new_xi;
        prop_table_ui[prop_index, i].settext("%d", x[i]);
        dfa_results[prop_index, i] = x[i];
    end;
else
    if(new_xi != x_o[i])
        finished = 0;
        fbox.settext("Finished = false");
        x_o[i] = new_xi;
        prop_table_ui[prop_index + 1, i].settext("%d", x_o[i]);
        dfa_results[prop_index + 1, i] = x_o[i];
    end;
end;

wait(3);

```

```

    end;

    until(finished == 1);

    wait(3);

    for(i = 0; i < num_blocks; i++)
        str_prop_in[i].settext("");
        str_prop_out[i].settext("");
    end;

    fbox.settext("");

end;

//-----
// Add interactive buttons to the GUI for basic manipulation of the animation.
//

start_button = SimpleButton(10, 130, 140, 30, whitebrush, gray192brush, blackbrush, 0, "Start");
running = 0;

when start_button.button.eventLB(int down, int xx, int yy)

    // ets just deal with when the button gets let go.
    if(!down)
        return;
    end;

    // standard toggle. if its on, turn it off else inverse!
    if(running == 0)
        running = 1;
        start_button.button.settext("Stop");
        start();
    else
        running = 0;
        start_button.button.settext("Start");
        stop();
    end;
end;

reset_button = SimpleButton(10, 170, 140, 30, whitebrush, gray192brush, blackbrush, 0, "Reset");

when reset_button.button.eventLB(int down, int xx, int yy)

    if(!down)
        return;
    end;

    // reset the current block and current tuple in the block
    setIntArg("CT", 0);
    setIntArg("CB", 0);

    // reset the animation
    reset();

end;

expression_button = SimpleButton(10, 210, 140, 30, whitebrush, gray192brush, blackbrush, 0, "");

// current block and current tuple are persistent across vivio
// reset() calls. this enables the handling of iterating over expressions
// yet resetting and restarting the animation when a change to the state is made.

current_block = 0;
current_tuple = 0;

getIntArg("CB", current_block);
if (num_blocks <= current_block)
    current_block = 0;
end;

getIntArg("CT", current_tuple);
if (num_tuples[current_block] <= current_tuple)
    current_tuple = 0;
end;

// what variable are we looking at for global live variable analysis?
string variable_name = "A";

// what expr are we looking at for redundancy analysis?
testq = flow_graph[current_block, current_tuple];
expression_button.button.settext("Expr: %s", flow_graph[current_block, current_tuple].as_string());

```

```

when expression_button.button.eventLB(int down, int xx, int yy)

    if (!down)
        return;
    end;

    if (num_tuples[current_block] > (current_tuple + 1))
        current_tuple++;
    else
        if (num_blocks > (current_block + 1))
            current_block++;
            current_tuple = 0;
        else
            current_block = 0;
            current_tuple = 0;
        end;
    end;

    setIntArg("CT", current_tuple); // These variables are persistent across reset() calls.
    setIntArg("CB", current_block); //
    reset();

end;

Text(0, 0, 3, 5, 55, blackbrush, arial, "Event-class problems can be described using systems of
boolean equations and solved with dataflow analysis. Here we demonstrate \nthe iterative algorithm
for solving these systems. Global live variable analysis is demonstrated on a variable (%s),
followed by redundant code\nsuppression for an arbitrary expression from the flow graph.", variable_name);

settps(2);
stop();
wait(1);

//-----
// Start global LVA and redundancy supression for the selected expression.
//
// local variables per block

int live[*]; // variable live in block i
int defined[*]; // variable defined in block i

int ntran[*]; // expr not transparent in block i? (ie. /ae)
int comp[*]; // expr locally available in block i?
int antloc[*]; // expr locally anticipable in block i?

// initialize local variables per block (done prop by prop, as its
// easier to see on-screen).

current_operation.setText("STATUS: Scan blocks for live use of %s", variable_name);
for(i = 0; i < num_blocks; i++)
    flow_graph_ui_boxes[i].setbrush(lightredbrush);
    live[i] = live_use(variable_name, i);
    prop_table_ui[0, i].setText("%d", live[i]);
    wait(5);
    flow_graph_ui_boxes[i].setbrush(whitebrush);
end;

current_operation.setText("STATUS: Scan blocks for definition of %s", variable_name);
for(i = 0; i < num_blocks; i++)
    flow_graph_ui_boxes[i].setbrush(lightredbrush);
    defined[i] = def(variable_name, i);
    prop_table_ui[1, i].setText("%d", defined[i]);
    wait(5);
    flow_graph_ui_boxes[i].setbrush(whitebrush);
end;

current_operation.setText("STATUS: Determine global liveness of %s (One forth problem)", variable_name);
dfa_iterative(live, defined, 1, 1, 2);

current_operation.setText("STATUS: Scan blocks for non-transparency of {%s}", testq.as_string());
for(i = 0; i < num_blocks; i++)
    flow_graph_ui_boxes[i].setbrush(lightredbrush);
    ntran[i] = not_transp(testq, i);
    prop_table_ui[4, i].setText("%d", ntran[i]);
    wait(5);
    flow_graph_ui_boxes[i].setbrush(whitebrush);
end;

current_operation.setText("STATUS: Scan blocks for local availability of {%s}", testq.as_string());
for(i = 0; i < num_blocks; i++)
    flow_graph_ui_boxes[i].setbrush(lightredbrush);
    comp[i] = locally_available(testq, i);

```

```

prop_table_ui[5, i].settext("%d", comp[i]);
wait(5);
flow_graph_ui_boxes[i].setbrush(whitebrush);
end;

current_operation.settext("STATUS: Determine global availability of {%s} (All back problem)", testq.as_string());
dfa_iterative(comp, ntran, 0, 0, 6);

current_operation.settext("STATUS: Scan blocks for local anticipability of {%s}", testq.as_string());
for(i = 0; i < num_blocks; i++)
  flow_graph_ui_boxes[i].setbrush(lightredbrush);
  antloc[i] = locally_anticipable(testq, i);
  prop_table_ui[8, i].settext("%d", antloc[i]);
  wait(5);
  flow_graph_ui_boxes[i].setbrush(whitebrush);
end;

current_operation.settext("STATUS: Determine global anticipability of {%s} (All forth problem)", testq.as_string());
dfa_iterative(antloc, ntran, 1, 0, 9);

current_operation.settext("STATUS: Redundant code suppression for {%s}", testq.as_string());
for(i = 0; i < num_blocks; i++)
  flow_graph_ui_boxes[i].setbrush(lightredbrush);

// if its anticipable at the input, and available at the input then its redundant in this block.
if(dfa_results[9, i] == 1)
  if(dfa_results[6, i] == 1)

    for(int t = 0; t < num_tuples[i]; t++)
      if(flow_graph[i, t].as_string() == testq.as_string())
        flow_graph_ui[i, t].setfont(arial_stthrough);
        break;
      end;
    end;
  end;

end;
end;

wait(5);
flow_graph_ui_boxes[i].setbrush(whitebrush);
end;
current_operation.settext("");

```

References

- [1] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.