2006

Range analysis of variables within a program

Yuanzhen Bao

The University of Montana

Follow this and additional works at: http://scholarworks.umt.edu/etd

Recommended Citation
Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

**Please check "Yes" or "No" and provide signature**

Yes, I grant permission

No, I do not grant permission

Author's Signature: Yuanchen Bai

Date: 08/28/2006

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.
RANGE ANALYSIS OF VARIABLES WITHIN A PROGRAM

By

Yuanzhen Bao

B.S., Southern Yangtze University, Wuxi, P.R.China, 2002

presented in partial fulfillment of the requirements

for the degree of

Master of Science

The University of Montana

July 2006

Approved by:
Chairperson
Dean of Graduate School
8-28-06

Date

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Range Analysis of Variables within a Program

Chairperson: Dr. Joel Henry

This thesis addresses the problem of identifying value ranges that variables may take within a program. A novel algorithm is introduced. It is based on static range propagation, and uses the constraint information derived from the source code. I will demonstrate how these ranges are tracked accurately at each point of the program. In particular, I will describe how loop iterations and conditional branches could be handled.

The solution provides lower and upper bounds for the values of variables. We believe this algorithm is useful for implementing program analysis and software verification tools.

Following is the outline of this thesis. Section 1 gives some background and introduces the problem we address. Sections 2 and 3 present our range analysis algorithm and demonstrate it more specifically through three examples in Section 4. Section 5 is a survey of related work, and finally, Section 6 gives our conclusions. In Section 7, I discuss some possible extensions and optimizations that might become promising plans for future work.
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Joel Henry, for the valuable direction and feedback he has given me on this thesis. The discussions with him were always helpful, and his suggestions enlightened me through the whole process of formulating and solving the problems.

I would also like to thank my dear parents who have been supportive of my many years of education and the time and resources it has cost. I would like to thank other faculty members and my classmates in Computer Science Department, who provided help on numerous occasions.
# Table of Contents

**INTRODUCTION** ......................................................................................................................... 1  
- 1.1 BACKGROUND & PROBLEM DESCRIPTION ................................................................. 1  

**SOLUTION** ................................................................................................................................. 4  
- 2.1 OVERVIEW .................................................................................................................. 4  
- 2.2 RANGE REPRESENTATION ...................................................................................... 5  
- 2.3 BASIC RANGE ANALYSIS ....................................................................................... 6  
- 2.4 Restricting Ranges at Conditional Points ................................................................. 9  
- 2.5 Handling Loop Iterations ....................................................................................... 10  

**ALGORITHM** .............................................................................................................................. 13  
- 3.1 Variable Struct ......................................................................................................... 14  
- 3.2 Get Subset of Variables ........................................................................................ 15  
- 3.3 Range Calculation .................................................................................................. 18  
- 3.4 Algorithm Correctness .......................................................................................... 23  

**EXAMPLE SOLUTIONS** ........................................................................................................... 24  
- 4.1 Example 1 ............................................................................................................... 24  
- 4.2 Example 2 ............................................................................................................... 35  
- 4.3 Example 3 ............................................................................................................... 41  

**RELATED WORK** ....................................................................................................................... 56  

**CONCLUSIONS** ......................................................................................................................... 58  

**FUTURE RESEARCH** ............................................................................................................... 60  

**REFERENCES** ........................................................................................................................... 60  

**APPENDIX A – EXAMPLE 3 PROGRAM CODE** ...................................................................... 62  

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF FIGURES

Fig 1: Control Flow Graph for Conditional Statements ........................................... 10
Fig 2: Variable Dependency ..................................................................................... 16
Fig 3: Variable Relation Tree ................................................................................... 16
Fig 4: Loop Control Flow ......................................................................................... 20
Fig 5: Create_Variable(x2) Finds \{x2,x1,b,a,disc,c\} ............................................. 29
Fig 6: Algorithm Execution: Create_Variable(x2) .................................................. 30
Fig 7: Algorithm Execution: Calc_Variable_Range .................................................. 35

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


**LIST OF ALGORITHMS**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg 1: Function Main</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>Alg 2: Function Create_Variable</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>Alg 3: Variable Struct</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>Alg 4: Function Calc_Variable_Range</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Alg 5: Function Evaluate_Instruction</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>Alg 6: Improvement for Loop Analysis</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF TABLES

Table 1: Rules for Range Calculation ................................................................. 7
Table 2: Variable inside a Loop Body ................................................................. 12
Table 3: Get Subset of Variables ....................................................................... 18
Table 4: Convert Loop into Straight-Line Code .................................................. 21
Table 5: Basic Operations Used in Range Analysis ............................................. 21
Table 6: Variable Ranges for Each Trip Count .................................................... 39
Introduction

1.1 Background & Problem Description

The solution introduced in this thesis is part of the research project in the field of Software and Safety Assurance sponsored by NASA.

Software systems are getting larger and more complex. In huge programs with more than 100,000 lines of code, it is very difficult to find faults without the support of program debugging or maintenance tools. NASA mission failures (e.g. Mars Polar Lander and Mars Orbiter) illustrate the importance of having an efficient verification and validation process for such systems. One approach would be trace of the values of variables inside a program. Since one cannot always know the exact value of a variable before executing the program, we need powerful analysis algorithms to approximate a range for the variable. The range is simply a lower bound and an upper bound on the values that a variable may take. It could be either numeric or symbolic in nature.

Algorithms in this area work on source language level or the program control flow graph. It centers upon the calculation of ranges at each point of the code. That is, the algorithm statically “run” the program by following its control flow path, updating the current ranges to reflect the effects of statements encountered. For example, consider the code below:
\[ y = x; \]
\[ a = y + z; \]

The ranges of \( x \) and \( z \) are known: \( x = [-5:5], z = 14 \). What about the range of \( a \)? It is obvious to human reader that \( a = [9:19] \). Subconsciously you probably used the following logic in coming to the result:

\[
\begin{align*}
y = x; & \quad \leftarrow x = [-5:5], \text{ so } y = [-5:5]. \\
a = y + z; & \quad \leftarrow y = [-5:5], z = 14, \text{ so } a = \left[ (-5+14) : (5+14) \right] = [9:19].
\end{align*}
\]

Our method uses exactly this logic, normally called as static range analysis, which provides an approximation of run-time properties of programs. In practice, this approximation has a trade-off between the accuracy of the information extracted from program text and the efficiency of the algorithm.

We assume some of the variable values/ranges are known in advance. This is reasonable in NASA's projects. For example, velocity is greater than 0, or less than some fixed value according to the aerospace science knowledge. It is also possible that initial range information comes from the output of another previously run software.

The resulting range can then be used to provide verification information, which may indicate where an error is possible or how the error may arise. I believe this research will greatly benefit the subsequent work on implementing software analysis tools.

We focus on scalar variables in this thesis, because the input data come from
space applications and most of them are physical numeric values. Furthermore, the result generated by Simulink output verification tool, which is another implementation of the project, serves as inputs of our algorithm. It guarantees the values limited to only integer and real numbers. Other types of variables, e.g. complex numbers, are out of the scope of this project at this time, and might be much more complicated to be implemented.
Solution

• 2.1 Overview

William Harrison published one of the first works on variable range analysis problem [1], which forms the basis of our approach. He described a range propagation algorithm similar to the standard constant propagation. Additionally, he proposed the framework that uses the data and the conditional structure of a program to derive inductive definitions for the ranges of variables.

In general, there are three types of program structure units:

1. Sequential Execution (statements ending with ";">"
2. Conditional Execution (if else, switch)
3. Iterative Execution (while, for loop, repeat)

Our solution will address these separately in following sections. Special methods will be given for handling ranges within loops and across conditional branches.

Briefly speaking, the algorithm works on the source language level, discovering values on all possible executions of a program. First the algorithm searches for individual instructions where range values can be immediately determined (e.g. variable declarations) and sets them. Then the algorithm propagates this range over other instructions and estimates a conservative range at each point, as far forward through the program as possible.
2.2 Range Representation

In this section, I would like to introduce the mathematical notation used to represent ranges.

A range of a variable is either empty or \([l:u]\) with \(l \leq u\), where \(l\) is the lower bound and \(u\) is the upper bound of the variable. The choice of representation depends upon the data type. In this thesis, we focus on integer values, which could be a continuous range, or possibly enumerated (that is, a range with holes, e.g. \([2, 5, 10, 13, 25 \ldots]\)). We define \(-\text{INF}, +\text{INF}\) as denotations for the minimal and maximal values of the relative data type, respectively.

If the range is a sequence, which is especially useful in loop control variables and array ranges. It can be described by a lower bound, upper bound, and increment (arithmetic step size). For example, \([6:22:2]\) represents the sequence \([6, 8, 10 \ldots 20, 22]\), where 6 is the minimum, 22 is the maximum and the whole sequence is increased by 2.

In addition, a good range representation should provide efficient encoding for less common cases such as ranges that can only be specified relative to another variable, e.g. "\(x\) is greater than \(y-10\)". A symbolic representation (e.g. \(y-10\)) is required at this time. Normally there are multiple variables (e.g. \(y+z\)), and would be too inefficient for implementation.

From a practical point of view, program range analysis is probably of most use in tracking numeric value ranges. A simple range representation capable of handling
arithmetic operations is sufficient.

2.3 Basic Range Analysis

Given the initial value range information, additional information can be derived from the program text:

1. Traverse the source code in a top-to-bottom direction.

2. For each instruction, the range of the left hand side output variable (if any) is determined based on the range(s) of the input variables.

If a variable is not the target of an assignment statement, its value will not be changed. The left hand side variable is reevaluated according to the expression, calculated from the variables, constants, and operators on the right hand side. We extend the usual arithmetic operations to handle value ranges. Basic rules are summarized in Table 1. Take "addition" for example, given the value range information of the inputs \((\text{In1} \text{ and } \text{In2})\), the range of output is: \(\text{Output}.\text{Min} = (\text{In1}.\text{Min} + \text{In2}.\text{Min})\) and \(\text{Output}.\text{Max} = (\text{In1}.\text{Max} + \text{In2}.\text{Max})\). If the input range is divided by zero, or a value very near zero, exception handling will be called for the overflow. The term Phi operation in this table is adopted from SSA (Static Single Assignment) form, which is a famous compile-time intermediate program representation developed in 1980's. Many program analysis algorithms use this form to enhance their performance. Phi-Function takes the union of all operand ranges. Details about its usage will be described in following sections.
Consider the following assignment statement with a single operator.

\[ b = a \times 10; \]

If the variable \( a \) is \([-2:2]\), the range of \( b, [-2:2]\times10 = [-20:20] \) is then analyzed. Below is another example with more than one operator.

\[ c = a \times 10 + b; \]

In order to calculate the range of \( c \), the range of \( a \times 10 \) is firstly calculated. Then, we
can obtain the maximum value of \( c \) by adding the maximum value of \( a \times 10 \) to that of \( b \). The minimum value of \( c \) can be calculated by adding the minimum value of \( a \times 10 \) to that of \( b \).

Next, let us consider the case where more than one assignment exists in a program. The variable ranges can be calculated by applying above analysis methods to each assignment successively from the top of the program. If the same variable appears on left hand side in more than one assignment, the minimum and maximum values are determined by the last expression. The example below shows a function with multiple assignment statements.

```c
int func(int a)
{
    int b, c;
    c = a * 10;    ---- (1)
    b = a + 5;    ---- (2)
    c = a * 2 + b;    ---- (3)
}
```

`func()` has three assignments (1), (2), and (3). In this example, assume that argument \( a \) is \([-2:2]\). The value ranges of local variables \( b \) and \( c \) are calculated in the following manner. First, the assignment (1) is analyzed and we can know that the range of \( c \) is \([-20:20]\). Next, by analyzing the assignment (2), we obtain the range \([-2+5 : 2+5] = [3:7]\) for \( b \). And next, after analyzing the expression (3), we can know that the range of \( c \) at this point is \([-2*2+3 : 2*2+7] = [-1:11]\). Notice that variable \( c \) has been calculated twice in `func()`, and the second result overrides what we get from line (1).
2.4 Restricting Ranges at Conditional Points

Obviously, conditional points in a program provide vital information for the range analysis, as stated in reference [1]. Our approach considers “then” and “else” branches independently. Let’s look at the following if-then-else statement, in which the ranges of \(y\) and \(z\) are \([1:5]\) and \([-15:-10]\).

```plaintext
def (a <= 5)
    x = y - 5;  //then
else
    x = z + 12;  //else
```

Fig 1 shows the control flow graph. The range for \(x\) would be \([-4:0]\) on the “then” branch, or \([-3:2]\) on the “else” branch. The result value is determined by which path the program has actually taken at run time. In this thesis, we add a “merge” operation at the place where program flow joins. To be consistent with the concepts defined in SSA form, see reference [5], we call it Phi-Function. It takes the union of ranges attained from each path, and merges into a conservative approximation. We will discuss this more specifically later in Section 3. Thusly, the range \([-4:2]\) is obtained for \(x\). Nested conditional statements can be analyzed in a similar fashion.
Conditional branches also provide range information for the variable being tested. Comparison(s) on which the branches depend behave, in fact, like a group of definitions for the tested variable’s value range on each path from the conditional point. In the previous example, the maximum value of $a$ is set to 5 in the “then” piece of code, and in the “else” piece of code, the minimum value of $a$ is set to 6.

- **2.5 Handling Loop Iterations**

Loop analysis is a large area of research that can refer to many different types of methods. The range of values generated by instructions inside a loop depends on the number of times the loop is iterated. If the analysis does not take into account any static knowledge of loop bounds, continuous iterations will produce the infinite ranges for any variable inside the loop. Consequently, loop trip count estimate is necessary.
We statically compute the number of loop iterations that will be executed at run time. This thesis focuses on loops (typically "for", "while" loops), where the iterator has the form \( i = i + s \) \((s \text{ is a constant and } i \text{ is the iterator})\). It includes, for example, loops of the form \( \text{for} (i = a; i < b; i = i + s) \). The iteration bounds \( a \) and \( b \), and has a unit or non-unit stride \( s \). In this case, if the loop is non-zero-trip, \((b-a)/s > 0\), the loop trip count can be calculated from \([b-a]/s\). Other more complex loops include nested loops, those have more than one iterator, or loops that use a comparison to finish. However, they may introduce limitations on the trip count estimate, depending on the source code being analyzed.

The variable ranges in the body of the loop can be calculated by traversing the loop iteratively (details described in Section 3). If loop bounds are unknown, then all variables modified inside the loop are conservatively assigned the maximum data range, \((-\text{INF} : +\text{INF})\), to ensure correctness.

A simple optimization, which can be applied when the loop body computations are linear, is to find a closed form solution for loop-carried expressions (see Reference [2], [3]) in terms of the loop trip count and the growth factor, and then aggregate all invariants added or subtracted from the loop variables of interest. As an example, consider the simple code fragment below:

```c
x = 0;
for (i = 0; i < 10; i ++) {
    x = x + 5;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In this case, the loop is bounded and the number of iterations is 10. During the first iteration, we compute the growth factor from expression “\( x = x + 5 \)”, see Table 2. Since variable \( x \) is added by 5 each iteration, we can then multiply this growth factor “5” by the number of loop iterations “10” to determine the final result of the variable’s range. The possible values \( x \) may take inside the loop are \([5:50]\). Finally, we get the range \([50:50]\) for \( x \) after the loop.

**Table 2: Variable inside a Loop Body**

<table>
<thead>
<tr>
<th>loop-carried expressions</th>
<th>( x = x + 5; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>before loop</td>
<td>([0,0])</td>
</tr>
<tr>
<td>iteration 1</td>
<td>([5,5])</td>
</tr>
<tr>
<td>inside loop</td>
<td>([5,50])</td>
</tr>
<tr>
<td>after loop</td>
<td>([50,50])</td>
</tr>
</tbody>
</table>
Algorithm

The top-level function of our algorithm works in two stages: function Create Variables and Calc Variable Range, and finally returns the result range, as shown in Alg 1.

```
function Main(variable v) : range
    varPtr = Create_Variable(v);
    Calc_Variable_Range;
    return varPtr.varRange;
end function

Alg 1: Function Main
```

As opposed to the conventional approach that analyzes all variables at one time, we have developed a lower-cost algorithm. It computes for a particular variable, which is requested in the function argument Main(variable v). Since many application techniques only want to know the ranges of a small subset of all variables, especially for those huge programs with more than 1,000 variables in total, this variable-targeted algorithm will greatly reduce the costs of range calculation.

We only maintain the information necessary to compute the desired range. The algorithm recursively calls function Create_Variable, see Alg 2, to build a “working” list of variables: the target variable, and the other variables that the target variable’s value may depend on.
function Create_Variable(variable v) : *VARSTRUCT
    varPtr.varName = v;
    varPtr.varRange = initial values or (-INF:+INF);
    Add v into variableList;
    Go through the program;
    for each instruction whose lhs includes v, do
        for each variable x on rhs, do
            if x is not in variableList, then
                xPtr = Create_Variable(x);
            end if
        end for
    end for
    return varPtr;
end function

Alg 2: Function Create_Variable

• 3.1 Variable Struct

The function Create_Variable(v) first creates a data structure for the argument variable v, and adds it into the working list.

Each variable is represented by a struct with fields: varName and varRange, as shown in Alg 3. The field varRange is to trace the value range of the variable and keep updating at each point of the program.

struct VARSTRUCT
{
    variable varName;
    range varRange;
} *varPtr;

Alg 3: Variable Struct

The initial value of varRange is determined from each variable's definition (e.g.
variable declarations). In assignment "v = constant", the value range of v can be immediately determined regardless of other instructions. It is set to the single constant value. Otherwise, if the definition is not an assignment (e.g. the variable is a formal parameter, an argument to a function call, or an I/O statement), then the variable’s initial range is set to unconstrained (-INF : +INF), which represents that the value cannot be predicted at this time.

• 3.2 Get Subset of Variables

The key responsibility of function Create_Variable(v) is to answer the question: What program variables and statements potentially affect the value of argument variable v?

First, let’s be acquainted with the concept of data dependency. It describes how variables interact with each other. For example, consider the statements in Fig 2, the value of variable v1 is defined at the first statement and referred to at the fifth statement, where variable v5 is calculated. So a data dependency exists from the first statement to the fifth statement, more specifically, exists between variable v1 and v5. In general, a statement defines and references variables, corresponding to writes and reads. Variable B is data dependent on A if A defines (declares or modifies) a variable that B uses (references for its value). In this example, the data dependency also exists between variable v4 and v5, v2 and v4, v3 and v4. Fig 3 illustrates a variable relation tree, which represents the set of relations R{ (v5,v1), (v5,v4), (v4,v2),
\( \{v4, v3\} \), and \( v5 \) is assumed to be the root of the tree (the target variable). Although \( v2 \) and \( v3 \) do not directly contribute to the value of \( v5 \), the indirect dependency relationships still exist via the node \( v4 \).

\[
\begin{align*}
1: & \quad v1 = 5; \\
2: & \quad v2 = 10; \\
3: & \quad v3 = 15; \\
4: & \quad v4 = v3 - v2; \\
5: & \quad v5 = v1 + v4 + 1; \\
\end{align*}
\]

**Fig 2: Variable Dependency**

![Variable Relation Tree](image)

**Fig 3: Variable Relation Tree**

Let \( \text{Set}(v) \) be the set of variables involved in generating the value of \( v \) in a program.

We use a simple and conservative approach to compute \( \text{Set}(v) \):

1. Search for all the statements that directly update the variable \( v \), i.e. those that define \( v \).
2. Get the variables that are included in the right hand side of these statements.
3. Repeat from Step 1, compute \( \text{Set}(v) \) for each variable in Step 2.
Our algorithm starts with Create_Variable(target variable). In the above example, \( v_5 \) is the target and Create_Variable\( (v_5) \) finds out all the variables that directly affect the computation of \( v_5 \), which are \( v_1 \) and \( v_4 \). Those indirectly dependent variables are created by recursive calls to function Create_Variable. We can see from Fig 3 that \( v_2 \) and \( v_3 \) are indirectly related to \( v_5 \). They are generated by calling Create_Variable\( (v_4) \).

All the related, directly or indirectly, variables are recorded in a working list for future use. Below is another example of getting subset of variables. (a) is the original code fragment and let \( x_2 \) be the target variable, (b) shows the code after the “slim” operation. Finally the variable list includes \( \{x_2, b, disc, a, c\} \). Other variables, e.g. realpart, imagpart, will stop here, excluded from the range calculation.
Table 3: Get Subset of Variables

<table>
<thead>
<tr>
<th>Program Code:</th>
<th>Program Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a, b, c, disc;</td>
<td></td>
</tr>
<tr>
<td>float x1, x2, realpart, imagpart;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>disc=b<em>b - 4</em>a*c;</td>
<td></td>
</tr>
<tr>
<td>if (disc &gt; 0)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>x1=(float)(-b+sqrt(disc))/(2*a);</td>
<td></td>
</tr>
<tr>
<td>x2=(float)(-b-sqrt(disc))/(2*a);</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>else if (disc &lt; 0)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>realpart=(float)-b/(2*a);</td>
<td></td>
</tr>
<tr>
<td>imagpart=(float)sqrt(-disc)/(2*a);</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

(a) | (b)

- **3.3 Range Calculation**

Alg 4 shows the major function for analyzing ranges. Calc_Variable_Range traverses the program line by line, checking each instruction. If any left hand side variable is included in the working list, call function Evaluate_Instruction to perform the range calculation.

```plaintext
function Calc_Variable_Range
    Go through the program;
    if the lhs variable \( v \in \text{variableList} \), then
        Evaluate_Instruction;
        if "loop", then
            count = number of loop iterations;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
for each possible value of count: count(i), do
    varPtr.varRange(i) = varPtr.varRange;
    for each loop iteration, do
        for each related instruction(v) in the loop, do
            Evaluate_Instruction, using varPtr.varRange(i);
        end for
    end for
    Phi-Function(all varPtr.varRange(i));
end if
if "conditional point", then
    for each outbranch b(i) from the test point, do
        if v is the tested variable, then
            tmpRange = Calc_Conditional_Range(v, b(i));
            varPtr.varRange(i) = tmpRange "n" varPtr.varRange;
        else
            varPtr.varRange(i) = varPtr.varRange;
        end if
        for each related instruction(v) inside, do
            Evaluate_Instruction, using varPtr.varRange(i);
        end for
    end for
    Phi-Function(all varPtr.varRange(i));
end if
end function

Alg 4: Function Calc_Variable_Range

As mentioned in Section 2, loops and conditional statements need special handling. We only consider bounded loops in this thesis. If the number of iterations is a constant, as in "for (i = 0; i < 5; i ++)", we simply use this trip count value to unwind the loop body. The instructions that are called in multiple iterations of the loop are repeated into a single iteration. However, at most times, the number of iterations is determined by another variable, as in "for (i = 0; i < n; i ++)", and we should first estimate the range of the variable. Since a loop count must be integer, we
are able to get a set of count values from the range. For each possible value, expand the loop body into straight-line program text, thusly variable ranges can be propagated through the loop. Below is a loop fragment, and assume n is \([4:5]\) at the point of loop entry.

```c
int m = 0, n;
for (i = 0; i < n; i++) {
    m = m / 2 + i;
}
```

From the range of \(n\), we know that the number of iterations is 4 or 5. (a), (b) illustrate the straight-line code after loop unwinding, when \(n = 4\) and \(n = 5\) respectively.

![Fig 4: Loop Control Flow](image-url)
Table 4: Convert Loop into Straight-Line Code

\[
\begin{align*}
n = 4, & \text{ straight-line code:} \\
& \text{int } m = 0, n = 4; \\
& m = m / 2 + 0; \quad \text{//iteration 1, } i = 0 \\
& m = m / 2 + 1; \quad \text{//iteration 2, } i = 1 \\
& m = m / 2 + 2; \quad \text{//iteration 3, } i = 2 \\
& m = m / 2 + 3; \quad \text{//iteration 4, } i = 3 \\
& m = m / 2 + 4; \quad \text{//iteration 5, } i = 4
\end{align*}
\]

\[
\begin{align*}
n = 5, & \text{ straight-line code:} \\
& \text{int } m = 0, n = 5; \\
& m = m / 2 + 0; \quad \text{//iteration 1, } i = 0 \\
& m = m / 2 + 1; \quad \text{//iteration 2, } i = 1 \\
& m = m / 2 + 2; \quad \text{//iteration 3, } i = 2 \\
& m = m / 2 + 3; \quad \text{//iteration 4, } i = 3 \\
& m = m / 2 + 4; \quad \text{//iteration 5, } i = 4
\end{align*}
\]

For conditional statements, function Calc_Conditional_Range(v, b) computes and returns the range of a given variable \( v \) at a given control-flow branch \( b \). For example, the function would return \([10 : +\infty)\), if \( b \) is the “then” branch of the conditional statement “if \( v \geq 10 \) then”.

The equality test places a lower bound on \( v \) along the true path and an upper bound on \( v \) along the false path. We implement this by taking the intersection (\( \cap \)) of the variable’s current range and the result of Calc_Conditional_Range for each outbranch respectively. Table 5 gives out the semantics of \( \cap \). It results in a range whose lower bound is the maximum of the lower bounds of the arguments’ ranges and whose upper bound is the minimum of the upper bounds of the arguments’ ranges.

Table 5: Basic Operations Used in Range Analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>([a:b] \cup [c:d] \rightarrow [\min(a,c) : \max(b,d)])</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersection (( \cap ))</td>
<td>([a:b] \cap [c:d] \rightarrow [\max(a,c) : \min(b,d)])</td>
</tr>
</tbody>
</table>

Alg 5 is the implementation of function Evaluate_Instruction. It handles the range calculation for each instruction.
function Evaluate_Instruction
    if rhs of instruction(v) are constants, then
        varPtr.varRange = Calc_Data_Range(constants);
    else if rhs of instruction(v) includes variables, then
        for each variable x in rhs, do
            xRange = xPtr.varRange;
        end for
        if rhs is a Phi-Function, then
            varPtr.varRange = "U"(all xRanges);
        else
            varPtr.varRange = Calc_Data_Range(xRanges);
        end if
    end if
end function

Alg 5: Function Evaluate_Instruction

The function Calc_Data_Range simply does the arithmetic range calculations. For an assignment statement, the range of a variable in the left hand side is computed from ranges of variables, constants, and operators in the right hand side. For example, Calc_Data_Range(2+[l:10]) would return [1+2 : 10+2] = [3:12]. If there is only one single constant, say 10, Calc_Data_Range(10) would return the range [10:10].

If the variable’s calculation contains a Phi-Function, its range is computed by unioning (U) all the argument ranges involved. The semantics for U operator is also listed in Table 3. The union results in a range whose lower bound is the minimum of the lower bounds of the arguments’ ranges and whose upper bound is the maximum of the upper bounds of the arguments’ ranges.

- 3.4 Algorithm Correctness
In order to prove the correctness of our algorithm, we established the following process:

1. Proof of concept with test cases. Three test case examples are described in Section 4, manually simulating how the algorithm pseudo code is executed. Given initial values/ranges of the variables, the algorithm produces final ranges. Different programs are used for different goals, verifying the handling of conditional branches and loops respectively.

2. In addition, since we have completed the software implementation, text file of code fragments are read into range analyzer software. Input the same initial values as above, and the result should be same as well.

3. Run the test program exhaustively with different combination of initial values for variables. We collect the results from each run to see whether they fall into the range generated by our algorithm.
Example Solutions

In this section, I'm going to illustrate the algorithm in detail with three examples.

- 4.1 Example 1

```c
#include "stdio.h"
#include "math.h"

main()
{
    int a, b, c, disc;
    float x1, x2, realpart, imagpart;
    scanf("%d, %d, %d", &a, &b, &c);
    printf("The equation ");

    //a = 0, not a quadratic equation
    if (a == 0)
        printf("is not quadratic\n");
    else
    {
        disc = b*b - 4*a*c;

        //b^2-4ac = 0, equal real roots
        if (disc == 0)
        {
            x1 = (float)-b/(2*a);
            x2 = x1;
            printf("has two equal roots: x1 = %8.4f, x2 = %8.4f\n", x1, x2);
        }
        else if (disc > 0) //b^2-4ac > 0, distinct real roots
        {
            x1 = (float)(-b+sqrt(disc))/(2*a);
            x2 = (float)(-b-sqrt(disc))/(2*a);
        }
    }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
printf("has distinct real roots: \(x_1 = %8.4f, x_2 = %8.4f\)\n", x1, x2);
}
else // if \(b^2 - 4ac < 0\), complex roots
{
    realpart = (float)-b/(2*a);
    imagpart = (float)sqrt(-disc)/(2*a);
    printf("has complex roots:\n");
    printf("%8.4f+%8.4fi\n", realpart, imagpart);
    printf("%8.4f-%8.4fi\n", realpart, imagpart);
}
} // end else

We have discussed a little bit about this code fragment in previous Section 3, as an example of getting the subset of variables. Now in this example, suppose we wish to compute the range for variable \(x_2\), and the known initial ranges are: \(a = [-5:5]\), \(b = [0:10]\) and \(c = 5\). The whole algorithm works as follows:

main\(x_2\)
Create\_Variable\(x_2\)
\(x_2.\text{varRange} = (-\text{INF}:+\text{INF})\)
add \(x_2\) into \text{variableList} \((x_2)\)
go through the program \((x_2)\)
\(x_2 = x_1;\)
\(x_1\) not in the \text{variableList}
Create\_Variable\(x_1\)
\(x_1.\text{varRange} = (-\text{INF}:+\text{INF})\)
add \(x_1\) into \text{variableList} \((x_2, x_1)\)
go through the program \((x_1)\)
\(x_1 = (\text{float})-b/(2*a);\)
\(b\) not in the \text{variableList}
Create\_Variable\(b\)
\(b.\text{varRange} = [0:10]\)
add \(b\) into \text{variableList} \((x_2, x_1, b)\)
go through the program \((b)\)
no more instruction\((b)\)
end Create_Variable(b)
a not in the variableList
Create_Variable(a)
  a.varRange = [-5:5]
  add a into variableList {x2, x1, b, a}
go through the program (a)
  if (fabs(a) <= 0)
    no more instruction(a)
  end Create_Variable(a)
x1 = (float)(-b+sqrt(disc))/(2*a);
b in the variableList
disc not in the variableList
Create_Variable(disc)
  disc.varRange = [-INF:+INF]
  add disc into variableList {x2, x1, b, a, disc}
go through the program (disc)
disc = b*b - 4*a*c;
b in the variableList
  a in the variableList
c not in the variableList
Create_Variable(c)
  c.varRange=[5:5]
  add c into variableList {x2, x1, b, a, disc, c}
go through the program (c)
  no more instruction(c)
  end Create_Variable(c)
if (fabs(disc) <= 0)
else if (disc > 0)
  no more instruction(disc)
  end Create_Variable(disc)
a in the variableList
  no more instruction(x1)
  end Create_Variable(x1)
x2 = (float)(-b-sqrt(disc))/(2*a);
b in the variableList
disc in the variableList
  a in the variableList
  no more instruction(x2)
  end Create_Variable(x2)

Calc_Variable_Range
go through the program
if (a == 0) is a test point
for each outbranch
branch[0]: if (a == Q)
  tmpRange = Calc_Conditional_Range = [0:0]
  a.varRange[0] = a.varRange AND tmpRange = [0:0]
  no more instruction(varList)
branch[1]: else (a == 0)
  tmpRange = Calc_Conditional_Range = (-INF:-1)OR[1:+INF]
  disc = b*b - 4*a*c;
  for each variable in rhs
    bRange = b.varRange = [0:10]
    cRange = c.varRange = [5:5]
  disc.varRange[1] = Calc_Data_Range(bRange, aRange, cRange)
    = [0:100]-4*([-5:-1]OR[1:5])*[5:5] = [-100:200]
  if (disc == 0) is a test point
  for each outbranch
    branch[0]: if (disc == 0)
      tmpRange = Calc_Conditional_Range = [0:0]
      disc.varRange[1][0] = disc.varRange[1] AND tmpRange = [0:0]
      xl = (float)-b/(2*a);
      for each variable in rhs
        bRange = b.varRange = [0:10]
      xl.varRange[0] = Calc_Data_Range(bRange, aRange) = [-10:0]/([-10:-2]OR[2:10])
        = [-5:5]
    branch[1]: else if (disc > 0)
      tmpRange = Calc_Conditional_Range = [1:+INF]
      xl = (float)(-b+sqrt(disc))/(2*a);
      for each variable in rhs
        bRange = b.varRange = [0:10]
        discRange = disc.varRange[1][1] = [1:200]
      xl.varRange[1] = Calc_Data_Range(bRange, discRange, aRange)
        = ((-10:0)+[1:sqrt(200)])/([-10:-2]OR[2:10]) = (-7.1:7.1]
      x2 = (float)(-b-sqrt(disc))/(2*a);
      for each variable in rhs
        bRange = b.varRange = [0:10]
        discRange = disc.varRange[1][1] = [1:200]
x2.varRange[1] = Calc_Data_Range(bRange, discRange, aRange)
    = ([10:0] - [1:sqrt(200)]) / ([10:-2] OR [2:10])
    = [-12.1:-0.1] OR [0.1:12.1]
no more instruction(varList)
branch[2]: else
    tmpRange = Calc_Conditional_Range = (-INF:-1)
disc.varRange[1][2] = disc.varRange[1] AND tmpRange = [-100:-1]
no more instruction(varList)
no more branch, Phi-Functions of the test point
disc.varRange[1] = Phi-Function(disc.varRange[1][0], disc.varRange[1][1],
        disc.varRange[1][2]), Take Union
disc.varRange[1] = [-100,200]
x1.varRange = Phi-Function(x1.varRange[0], x1.varRange[1]), Take Union
x1.varRange = [-7.1:7.1]
x2.varRange = Phi-Function(x2.varRange[0], x2.varRange[1]), Take Union
x2.varRange = [-12.1:12.1]
end test point if (disc == 0)
no more branch, Phi-Functions of the test point
a.varRange = Phi-Function(a.varRange[0], a.varRange[1]), Take Union
a.varRange = [-5:5]
disc.varRange = Phi-Function(disc.varRange[1]), Take Union
disc.varRange = [-100:200]
end test point if (a == 0)
no more instruction(varList)
end Calc_Variable_Range
return x2.varRange = [-12.1:12.1]
end Main(x2)

Starting with Create_Variable(x2), the function Create_Variable is recursively
invoked six times and finds out all the strongly connected variables, creating a struct
for each once it is found, see Fig 5. These related variables are added into a
variableList \{x2, x1, b, a, disc, c\} for further use in the next function call
Calc_Variable_Range.

28

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Fig 5: Create_Variable(x2) Finds \( \{x2, x1, b, a, \text{disc}, c\} \)
In Calc_Variable_Range, only those related statements will be considered, which
means its left hand side variable is included in the variableList \{x2, x1, b, a, disc, c\}. To further elaborate, let’s take variable x1 for example. I will describe how its range is computed within the function. Initially, on entry to Calc_Varable_Range, the range of x1 is (-INF : +INF). The first visit to x1 “x1 = (float)-b/(2*a);” appears inside the “else” branch of test point “if (a == 0)” and the “then” branch of test point “if (disc == 0)”. Its range is updated by function Calc_Data_Range. Notice the right hand side variables: there are test point constraints on variable a at this time. a’s range comes from the conditional statement, we call Calc_Conditional_Range and get [-5:-1] or [1:5] upon the “else” of “if (a == 0)”. The range of b is still the initial value. So the range of x1 at this point is calculated from:

\[ x1 = [-10:0]/(2*[5:-1]OR[1:5]) = [-5:5] \]

The second visit to x1 “x1 = (float)(-b+sqrt(disc))/(2*a);” appears inside a different branch “else if (disc > 0)” of the test point “if (disc == 0)”. Similarly, the range of variable disc [1:200] comes from test point constraints and we get x1’s range:

\[ x1 = ([10:0]+[1:sqrt(200)]) / (2*[5:-1]OR[1:5]) = [-7.1:7.1] \]

The first and second visits to x1 belong to different branches of the same test point, and their ranges are computed separately. So we use subscripts x1.varRange[0], x1.varRange[1] to distinguish. Finally, x1’s range after leaving the conditional
section “if (disc == 0)” is calculated by Phi-Function, getting the union of
x1.varRange[0] and x1.varRange[1]:

\[
x1.\text{varRange} = \Phi\text{-Function}(x1.\text{varRange}[0], x1.\text{varRange}[1]), \text{ Take Union}
\]
\[
x1.\text{varRange} = [-7.1:7.1]
\]

There’re no more statements about x1. When function Calc_Variable_Range
completes, x1’s range is stored in x1.varRange = [-7.1:7.1].
Calc_Variable_Range

go through the program
if (a == 0) test point

branch[0]
tmpRange = Calc_Conditional_Range = [0:0]
a.varRange[0] = a.varRange AND tmpRange = [0:0]
no more instruction

branch[1]
tmpRange = Calc_Conditional_Range = (-INF:-1] OR [1:+INF)
disc = b*b - 4*a*c;
for each variable on rhs
  bRange = b.varRange = [0:10]
cRange = c.varRange = [5:5]
disc.varRange[1] = Calc_Data_Range(bRange, aRange, cRange)
  = [0:100] - 4*([-5:-1] OR [1:5])*[5:5] = [-100:200]
if (disc == 0) test point

branch[0]
tmpRange = Calc_Conditional_Range = [0:0]
disc.varRange[1][0] = disc.varRange[1] AND tmpRange = [0:0]
x1 = (float)-b/(2*a);
for each variable in rhs
  bRange = b.varRange = [0:10]
x1.varRange[0] = Calc_Data_Range(bRange, aRange)
  = [-10:0]/([-10:-2] OR [2:10]) = [-5:5]
x2 = x1;
for each variable in rhs
  x1Range = x1.varRange[0] = [-5:5]
x2.varRange[0] = Calc_Data_Range(x1Range) = [-5:5]
no more instruction

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
branch[1]

tmpRange = Calc_Conditional_Range = [1:+INF)

\[ x_1 = \frac{-b+\sqrt{\text{disc}}}{2a} \]

for each variable in rhs

\[ bRange = b.varRange = [0:10] \]
\[ \text{discRange} = \text{disc.varRange}[1][1] = [1:200] \]
\[ \text{aRange} = a.varRange[1] = [-5:-1]OR[1:5] \]

\[ x_1.varRange[1] = \text{Calc_Data_Range}(bRange, \text{discRange}, \text{aRange}) \]
\[ = ([10.0]+[1:200] )/([-10:-2]OR[2:10]) = [-7.1:7.1] \]

\[ x_2 = \frac{-b-\sqrt{\text{disc}}}{2a} \]

for each variable in rhs

\[ bRange = b.varRange = [0:10] \]
\[ \text{discRange} = \text{disc.varRange}[1][1] = [1:200] \]
\[ \text{aRange} = a.varRange[1] = [-5:-1]OR[1:5] \]

\[ x_2.varRange[1] = \text{Calc_Data_Range}(bRange, \text{discRange}, \text{aRange}) \]
\[ = ([10.0]+[1:200] )/([-10:-2]OR[2:10]) \]
\[ = [-12.1:-0.1]OR[0.1:12.1] \]

no more instruction

branch[2]

tmpRange = Calc_Conditional_Range = (-INF:-1)
disc.varRange[1][2] = disc.varRange[1] AND tmpRange = [-100:-1]

no more instruction(varList)

no more branch, Phi-Functions of the test point

\[ \text{disc.varRange[1]} = \text{Phi-Function}(\text{disc.varRange[1][0],} \]
\[ \text{disc.varRange[1][1], \text{disc.varRange[1][2]},} \]
\[ \text{Take Union} \]
\[ \text{disc.varRange[1]} = [-100,200] \]

\[ x_1.varRange = \text{Phi-Function}(x_1.varRange[0], x_1.varRange[1]), \]
\[ \text{Take Union} \]
\[ x_1.varRange = [-7.1:7.1] \]

\[ x_2.varRange = \text{Phi-Function}(x_2.varRange[0], x_2.varRange[1]), \]
\[ \text{Take Union} \]
\[ x_2.varRange = [-12.1:12.1] \]

end test point if (disc == 0)
4.2 Example 2

\[ a = x; \]
\[ \text{for} \ (i = 0; \ i < y; \ i++) \]
\[ \quad a = a + z; \]

This is a quick and simple example primarily to illustrate loop analysis. Our target variable is \( a \) and initial values are: \( x = [-5:5], \ y = [1:5], \ z = 14 \). After calling Create_Variable(a), we get the variableList \( \{a, x, z\} \). The pseudo-code below shows how function Calc_Variable_Range works:

\[
\text{main}(a) \\
. \\
. \\
\text{Calc_Variable_Range} \\
\text{go through the program} \\
\quad a = x;
\]
for each variable in rhs
    xRange = x.varRange = [-5:5]
a.varRange = Calc_Data_Range(xRange) = [-5:5]
for (i = 0; i < y; i++) is a loop entry
    count = [1:5] -- loop iteration
    for each value of count
        count[0] = 1
        a.varRange[0] = a.varRange = [-5:5]
        for each iteration
            iteration 1
                a = a + z;
                for each variable in rhs
                    aRange = a.varRange[0] = [-5:5]
zRange = z.varRange = [14:14]
a.varRange[0] = Calc_Data_Range(aRange, zRange) = [9:19]
                no more instruction(varList)
        no more iteration
        count[1] = 2
        for each iteration
            iteration 1
                a = a + z;
                for each variable in rhs
                    aRange = a.varRange[1] = [9:19]
zRange = z.varRange = [14:14]
a.varRange[1] = Calc_Data_Range(aRange, zRange) = [23:33]
                no more instruction(varList)
        no more iteration
        count[2] = 3
        for each iteration
            iteration 1
                a = a + z;
                for each variable in rhs
zRange = z.varRange = [14:14]
a.varRange[2] = Calc_Data_Range(aRange, zRange) = [9:19]
iteration 2
  a = a + z;
  for each variable in rhs
    aRange = a.varRange[2] = [9:19]
    zRange = z.varRange = [14:14]
    a.varRange[2] = Calc_Data_Range(aRange, zRange) = [23:33]
  no more instruction(varList)

iteration 3
  a = a + z;
  for each variable in rhs
    aRange = a.varRange[2] = [23:33]
    zRange = z.varRange = [14:14]
    a.varRange[2] = Calc_Data_Range(aRange, zRange) = [37:47]
  no more instruction(varList)
  no more iteration

count[3] = 4
  for each iteration
    iteration 1
      a = a + z;
      for each variable in rhs
        zRange = z.varRange = [14:14]
        a.varRange[3] = Calc_Data_Range(aRange, zRange) = [9:19]
      no more instruction(varList)

    iteration 2
      a = a + z;
      for each variable in rhs
        aRange = a.varRange[3] = [9:19]
        zRange = z.varRange = [14:14]
        a.varRange[3] = Calc_Data_Range(aRange, zRange) = [23:33]
      no more instruction(varList)

    iteration 3
      a = a + z;
      for each variable in rhs
        aRange = a.varRange[3] = [23:33]
        zRange = z.varRange = [14:14]
        a.varRange[3] = Calc_Data_Range(aRange, zRange) = [37:47]
      no more instruction(varList)

    iteration 4
      a = a + z;
      for each variable in rhs
        aRange = a.varRange[3] = [37:47]
zRange = z.varRange = [14:14]
a.varRange[3] = Calc_Data_Range(aRange, zRange) = [51:61]
no more instruction(varList)
no more iteration

count[4] = 5

for each iteration
iteration 1
  a = a + z;
  for each variable in rhs
    zRange = z.varRange = [14:14]
    a.varRange[4] = Calc_Data_Range(aRange, zRange) = [9:19]
    no more instruction(varList)
iteration 2
  a = a + z;
  for each variable in rhs
    aRange = a.varRange[4] = [9:19]
    zRange = z.varRange = [14:14]
    a.varRange[4] = Calc_Data_Range(aRange, zRange) = [23:33]
    no more instruction(varList)
iteration 3
  a = a + z;
  for each variable in rhs
    aRange = a.varRange[4] = [23:33]
    zRange = z.varRange = [14:14]
    a.varRange[4] = Calc_Data_Range(aRange, zRange) = [37:47]
    no more instruction(varList)
iteration 4
  a = a + z;
  for each variable in rhs
    aRange = a.varRange[4] = [37:47]
    zRange = z.varRange = [14:14]
    a.varRange[4] = Calc_Data_Range(aRange, zRange) = [51:61]
    no more instruction(varList)
iteration 5
  a = a + z;
  for each variable in rhs
    aRange = a.varRange[4] = [51:61]
    zRange = z.varRange = [14:14]
    a.varRange[4] = Calc_Data_Range(aRange, zRange) = [65:75]
    no more instruction(varList)
no more iteration
end for each value of count

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In the function Calc_Variable_Range, we convert the loop body into straight line code by repeating the statement “a = a + z”, and i is the loop-index, which determines the number of iterations. According to the range of variable $y \ [1:5]$, there are five possible values for loop trip count: 1, 2, 3, 4, and 5. We analyze each value separately. For example, when the count equals 4, we propagate ranges on four “a = a + z” statements continuously. Since $a = [-5:5]$ and $z = 14$ before entering the loop, we get a.varRange[3] from $[-5:5] + 14 + 14 + 14 + 14 = [-5 + 14 + 14 + 14 + 14 : 5 + 14 + 14 + 14 + 14] = [51:61]$. Subscripts are used here to identify different possible values of loop trip count (depends upon which data structure is used, for array, the fourth possible value is indicated by a.varRange[3]). Thusly, the results after the loop body are listed below:

<table>
<thead>
<tr>
<th>Possible Values of Loop Trip Count</th>
<th>Variable Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>count[0] = 1</td>
<td>a.varRange[0] = [9 : 19]</td>
</tr>
<tr>
<td>count[1] = 2</td>
<td>a.varRange[1] = [23 : 33]</td>
</tr>
</tbody>
</table>

At the end, before leaving the loop, we take union of all the five ranges got from each loop analysis:
Based on the analysis above, an improvement is available for our algorithm. As we can see from the pseudo-code execution, the range calculation when loop trip count = 1 occurs again as a part of calculation when loop trip count = 2, 3, 4, 5, and the calculation for loop trip count = 2 is also duplicated in loop trip count = 3, 4, 5, and so on, the analysis for loop trip count = 5 includes all previous calculations. Suppose there are several possible values for the loop trip count, we only need pick the biggest value to do the calculation, and keep records of certain range values in the midway when the number of loops comes to any possible value of loop trip count. See Alg 6.

```plaintext
if "loop", then
  count[] = possible values of #loop_iterations;
  count_max = Max(count[]);
  int num = 0;
  for each loop iteration (trip count = count_max) do
    for each related instruction(v) in the loop, do
      Evaluate_Instruction;
    end for
    num++;
    if num exists in count[] and equals count[i] then
      varPtr.varRange[i] = varPtr.varRange;
    end if
  end for
  varPtr.varRange = "u"(all varPtr.varRange[i]);
end if
```

Alg 6: Improvement for Loop Analysis
4.3 Example 3

In this example, we select a real life program. Its source codes (vehicle.cc, vehicle.h) are attached as Appendix A. In order to investigate the range of variable speed, function Create_Variable(speed) generates a variable list \{speed, Aspeed, speed_temp, speed_limit, Aspeed_limit, accel, Aaccel, delta_time, decell, Adecel\}. This helps reduce lots of work, we only need consider these 10 variables instead of all, which are more than 50 variables. We performed an algorithm walkthrough to simulate what will happen when Main(speed) is triggered, as is shown below:

Initial Values: Aspeed = [1:18], Aspeed_limit = 20, Aaccel = [0:1], Adecel = [0:0.5].
Target: speed
main(speed)
Create_Variable(speed)
    speed.varRange = (-INF:+INF)
    add speed into variableList {speed}
go through the program (speed)
    speed = Aspeed;
    Aspeed not in the variableList
    Create_Variable(Aspeed)
        Aspeed.varRange = [1:18]
        add Aspeed into variableList {speed, Aspeed}
go through the program (Aspeed)
    no more instruction(Aspeed)
end Create_Variable(Aspeed)
if (speed > 0.0)
speed = speed_temp;
speed_temp not in the variableList
Create_Variable(speed_temp)
    speed_temp.varRange = (-INF:+INF)
    add speed_temp into variableList {speed, Aspeed, speed_temp}
go through the program (speed_temp)
    speed_temp = speed;
    speed in the variableList
if (speed_temp <= speed_limit)
    speed_limit not in the variableList
Create_Variable(speed_limit)
    speed_limit.varRange = (-INF:+INF)
add speed_limit into variableList {speed, Aspeed, speed_temp, speed_limit}
go through the program (speed_limit)
speed_limit = Aspeed_limit;
Aspeed_limit not in the variableList
Create_Variable(Aspeed_limit)
Aspeed_limit.varRange = [20:20]
add Aspeed_limit into variableList {speed, Aspeed, speed_temp, speed_limit, Aspeed_limit}
go through the program (Aspeed_limit)
no more instruction(Aspeed_limit)
end Create_Variable(Aspeed_limit)
no more instruction(speed_limit)
end Create_Variable(speed_limit)
speed_temp = speed + accel*delta_time;
speed in the variableList
accel not in the variableList
Create_Variable(accel)
accel.varRange = (-INF:+INF)
add accel into variableList {speed, Aspeed, speed_temp, speed_limit, Aspeed_limit, accel}
go through the program (accel)
accel = Aaccel;
Aaccel not in the variableList
Create_Variable(Aaccel)
Aaccel.varRange = [0:1]
add Aaccel into variableList {speed, Aspeed, speed_temp, speed_limit, Aspeed_limit, accel, Aaccel}
go through the program (Aaccel)
no more instruction(Aaccel)
end Create_Variable(Aaccel)
no more instruction(accel)
end Create_Variable(accel)
delta_time not in the variableList
Create_Variable(delta_time)
delta_time.varRange = (-INF:+INF)
add delta_time into variableList (speed, Aspeed, speed_limit, Aspeed_limit, accel, Aaccel, delta_time)
go through the program (delta_time)
delta_time = 1;
no more instruction(delta_time)
end Create_Variable(delta_time)

if (speed_temp > speed_limit)
speed_limit in the variableList
speed_temp = speed_limit;
speed_limit in the variableList
speed_temp = speed - decell*delta_time;
speed in the variableList
decell not in the variableList
Create_Variable(decell)
decell.varRange = (-INF:+INF)
add decell into variableList (speed, Aspeed, speed_temp, speed_limit, Aspeed_limit, accel, Aaccel, delta_time, decell)
go through the program (decell)
decell = Adecell;
Adecell not in the variableList
Create_Variable(Adecell)
Adecell.varRange = [0:0.5]
add Adecell into variableList {speed, Aspeed, speed_temp, speed_limit, Aspeed_limit, accel, Aaccel, delta_time, decell, Adecell}
go through the program (Adecell)
no more instruction(Adecell)
end Create_Variable(Adecell)
no more instruction(decell)
end Create_Variable(decell)
delta_time in the variableList
if (speed_temp < 0.0)
speed_temp = 0.0;
no more instruction(speed_temp)
end Create_Variable(speed_temp)
no more instruction(speed)
end Create_Variable(speed)

Calc_Variable_Range
go through the program
accel = Aaccel;
for each variable in rhs
  AaccelRange = Aaccel.varRange = [0:1]
accel.varRange = Calc_Data_Range(AaccelRange) = [0:1]
decell = Adecell;
for each variable in rhs
  AdecellRange = Adecell.varRange = [0:0.5]
decell.varRange = Calc_Data_Range(AdecellRange) = [0:0.5]
speed = Aspeed;
for each variable in rhs
    AspeedRange = Aspeed.varRange = [1:18]
speed.varRange = Calc_Data_Range(AspeedRange) = [1:18]
speed_limit = Aspeed_limit;
for each variable in rhs
    Aspeed_limitRange = Aspeed_limit.varRange = [20:20]
speed_limit.varRange = Calc_Data_Range(Aspeed_limitRange) = [20:20]
delta_time = 1;
delta_time.varRange = Calc_Data_Range = [1:1]
speed_temp = speed;
for each variable in rhs
    speedRange = speed.varRange = [1:18]
speed_temp.varRange = Calc_Data_Range(speedRange) = [1:18]
if(position < signal_pos) is a test point
for each outbranch
    branch[0]: if(position < signal_pos)
        if (speed_temp <= speed_limit) is a test point
        for each outbranch
            branch[0]: if (speed_temp <= speed_limit)
                for each variable in rhs
                    speed_limitRange = speed_limit.varRange = [20:20]
tmpRange = Calc_Conditional_Range = [-INF:20]
speed_temp.varRange[0][0] = speed_temp.varRange AND tmpRange = [1:18]
for (i=0; i<size; i++) is a loop entry
    count = 5 -- loop iteration
for each iteration

iteration 1

\[ \text{speed\_temp} = \text{speed\_temp} + \text{accel}\times\text{delta\_time}; \]
for each variable in \text{rhs}

\[ \text{speed\_tempRange} = \text{speed\_temp.varRange}[0][0] = [1:18] \]
\[ \text{accelRange} = \text{accel.varRange} = [0:1] \]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0] = \text{Calc\_Data\_Range(speed\_tempRange, accelRange, delta\_timeRange)} = [1:19] \]
no more instruction(varList)

iteration 2

\[ \text{speed\_temp} = \text{speed\_temp} + \text{accel}\times\text{delta\_time}; \]
for each variable in \text{rhs}

\[ \text{speed\_tempRange} = \text{speed\_temp.varRange}[0][0] = [1:19] \]
\[ \text{accelRange} = \text{accel.varRange} = [0:1] \]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0] = \text{Calc\_Data\_Range(speed\_tempRange, accelRange, delta\_timeRange)} = [1:20] \]
no more instruction(varList)

iteration 3

\[ \text{speed\_temp} = \text{speed\_temp} + \text{accel}\times\text{delta\_time}; \]
for each variable in \text{rhs}

\[ \text{speed\_tempRange} = \text{speed\_temp.varRange}[0][0] = [1:20] \]
\[ \text{accelRange} = \text{accel.varRange} = [0:1] \]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0] = \text{Calc\_Data\_Range(speed\_tempRange, accelRange, delta\_timeRange)} = [1:21] \]
no more instruction(varList)

iteration 4
speed_temp = speed_temp + accel*delta_time;
for each variable in rhs
  speed_tempRange = speed_temp.varRange[0][0] = [1:21]
  accelRange = accel.varRange = [0:1]
  delta_timeRange = delta_time.varRange = [1:1]
  speed_temp.varRange[0][0] = Calc_Data_Range(speed_tempRange, accelRange, delta_timeRange) = [1:22]
no more instruction(varList)
iteration 5
speed_temp = speed_temp + accel*delta_time;
for each variable in rhs
  speed_tempRange = speed_temp.varRange[0][0] = [1:22]
  accelRange = accel.varRange = [0:1]
  delta_timeRange = delta_time.varRange = [1:1]
  speed_temp.varRange[0][0] = Calc_Data_Range(speed_tempRange, accelRange, delta_timeRange) = [1:23]
no more instruction(varList)
no more iteration
if (speed_temp > speed_limit) is a test point
for each outbranch
branch[0]: if (speed_temp > speed_limit)
  for each variable in rhs
    speed_limitRange = speed_limit.varRange = [20:20]
tmpRange = Calc_Conditional_Range = (20:+INF)
  speed_temp.varRange[0][0][0] = speed_temp.varRange[0][0] AND tmpRange = (20:23]
speed_temp = speed_limit;
for each variable in rhs
  speed_limitRange = speed_limit.varRange = [20:20]
speed_temp.varRange[0][0][0] = Calc_Data_Range(speed_limitRange) = [20:20]
no more instruction(varList)

branch[1]: else
  for each variable in rhs
    speed_limitRange = speed_limit.varRange = [20:20]
tmpRange = Calc_Conditional_Range = (-INF:20)
speed_temp.varRange[0][0][1] = speed_temp.varRange[0][0] AND tmpRange = [1:20]
no more instruction(varList)
no more branch, Phi-Function of the test point
speed_temp.varRange[0][0] = Phi-Function(speed_temp.varRange[0][0][0], speed_temp.varRange[0][0][1]), Take Union
speed_temp.varRange[0][0] = [1:20]
end test point if (speed_temp > speed_limit)
no more instruction(varList)

branch[1]: else
  for each variable in rhs
    speed_limitRange = speed_limit.varRange = [20:20]
tmpRange = Calc_Conditional_Range = (20:+INF)
speed_temp.varRange[0][1] = speed_temp.varRange AND tmpRange = NULL
no more branch, Phi-Function of the test point
speed_temp.varRange = Phi-Function(speed_temp.varRange[0][0], speed_temp.varRange[0][1]), Take Union
speed_temp.varRange = [1:20]
end test point if (speed_temp <= speed_limit)
switch (signal) is a test point
for each outbranch
branch[0]: case red
branch[1]: case yellow
branch[2]: case stop
    if (stopping_distance > signal_pos-position_temp) is a test point
    for each outbranch
    branch[0]: if (stopping_distance > signal_pos-position_temp)
                if (speed > 0.0) is a test point
                for each outbranch
                branch[0]: if (speed > 0.0)
                    tmpRange = Calc_Conditional_Range = (0:+INF)
                    speed.varRange[0][0][0] = speed.varRange AND tmpRange = [1:18]
                    speed_temp.varRange[0][0][0] = speed_temp.varRange = [1:20]
                    for (i=0; i<size; i++) is a loop entry
                    count = 5 -- loop iteration
                    for each iteration
                    iteration 1
                    speed_temp = speed_temp - decell*delta_time;
                    for each variable in rhs
                        speed_tempRange = speed_temp.varRange[0][0][0] = [1:20]
                        decellRange = decell.varRange = [0:0.5]
                        delta_timeRange = delta_time.varRange = [1:1]
                        speed_temp.varRange[0][0][0] = Calc_Data_Range(speed_tempRange, decellRange, delta_timeRange) = [0.5:20]
                        no more instruction(varList)
                    iteration 2
                    speed_temp = speed_temp - decell*delta_time;
                    for each variable in rhs
                        speed_tempRange = speed_temp.varRange[0][0][0] = [0.5:20]
                        decellRange = decell.varRange = [0:0.5]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0][0] = \text{Calc\_Data\_Range(speed\_tempRange, decellRange, delta\_timeRange)} = [0:20] \]
no more instruction(varList)

iteration 3
\[ \text{speed\_temp} = \text{speed\_temp} - \text{decell*delta\_time}; \]
for each variable in rhs
\[ \text{speed\_tempRange} = \text{speed\_temp.varRange}[0][0][0] = [0:20] \]
\[ \text{decellRange} = \text{decell.varRange} = [0:0.5] \]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0][0] = \text{Calc\_Data\_Range(speed\_tempRange, decellRange, delta\_timeRange)} = [-0.5:20] \]
no more instruction(varList)

iteration 4
\[ \text{speed\_temp} = \text{speed\_temp} - \text{decell*delta\_time}; \]
for each variable in rhs
\[ \text{speed\_tempRange} = \text{speed\_temp.varRange}[0][0][0] = [-0.5:20] \]
\[ \text{decellRange} = \text{decell.varRange} = [0:0.5] \]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0][0] = \text{Calc\_Data\_Range(speed\_tempRange, decellRange, delta\_timeRange)} = [-1:20] \]
no more instruction(varList)

iteration 5
\[ \text{speed\_temp} = \text{speed\_temp} - \text{decell*delta\_time}; \]
for each variable in rhs
\[ \text{speed\_tempRange} = \text{speed\_temp.varRange}[0][0][0] = [-1:20] \]
\[ \text{decellRange} = \text{decell.varRange} = [0:0.5] \]
\[ \text{delta\_timeRange} = \text{delta\_time.varRange} = [1:1] \]
\[ \text{speed\_temp.varRange}[0][0][0] = \text{Calc\_Data\_Range(speed\_tempRange, decellRange, delta\_timeRange)} = [-1.5:20] \]
if (speed_temp < 0.0) is a test point
for each outbranch
branch[0]: if (speed_temp < 0.0)
    tmpRange = Calc_Conditional_Range = (-INF:0)
    speed_temp.varRange[0][0][0][0] = speed_temp.varRange[0][0][0] AND tmpRange = [-1.5:0]
    speed_temp = 0.0;
    speed_temp.varRange[0][0][0][0] = Calc_Data_Range = [0:0]
    no more instruction(varList)
branch[1]: else
    tmpRange = Calc_Conditional_Range = [0:+INF)
    speed_temp.varRange[0][0][1][1] = speed_temp.varRange[0][0][0] AND tmpRange = [0:20]
    no more instruction(varList)
no more branch, Phi-Function of the test point
speed_temp.varRange[0][0][0] = Phi-Function(speed_temp.varRange[0][0][0], speed_temp.varRange[0][0][1]), Take Union
speed_temp.varRange[0][0][0] = [0:20]
end test point if (speed_temp < 0.0)
no more instruction(varList)
branch[1]: else
    tmpRange = Calc_Conditional_Range = (-INF:0))
    speed.varRange[0][0][1] = speed.varRange AND tmpRange = NULL
    no more instruction(varList)
no more branch, Phi-Function of the test point
speed.varRange[0] = Phi-Function(speed.varRange[0][0], speed.varRange[0][0][1]), Take Union
speed.varRange[0] = [1:18]
speed_temp.varRange = Phi-Function(speed_temp.varRange[0][0][0]), Take Union
speed_temp.varRange = [0:20]
no more instruction(varList)
no more branch for if (stopping_distance > signal_pos-position_temp)
no more instruction(varList)
no more branch for switch (signal)
speed = speed_temp;
for each variable in rhs
    speed_tempRange = speed_temp.varRange = [0:20]
speed.varRange[0] = Calc_Data_Range(speed_tempRange) = [0:20]
    no more instruction(varList)
no more branch for if(position < signal_pos), Phi-Function of the test point
speed.varRange = Phi-Function(speed.varRange[0]), Take Union
speed.varRange = [0:20]
no more instruction(varList)
end Calc_Variable_Range
return speed.varRange = [0:20]
end Main(speed)

Final Result:
speed.varRange = [0:20]
The original program has 210 lines in total, and the following are lines reached by
function Calc_Variable_Range. Statements that do not affect the target variable’s
value are all eliminated in range calculation, which in this example, is about 86% of
the source code.

```
55 accel = Aaccel;
56 decell = Adecell;
59 speed = Aspeed;
60 speed_limit = Aspeed_limit;
79 deltatime — 1 ;
83 speed_temp = speed;
84 if(position < signal_pos)
85 {
87 if (speed_temp <= speed_limit) // try to accelerate
88 {
89     for (i=0; i<size; i++)
90         speed_temp = speed_temp + accel*delta_time;
91     if (speed_temp > speed_limit)
92         speed_temp = speed_limit;
93 }
97 switch (signal)
98 {
103 if (stopping_distance > signal_pos-position_temp)
104 {
105     if (speed > 0.0) // can not back up
106 {
107         for(i=0; i<size; i++)
108             speed_temp = speed_temp - decell*delta_time;
109         if (speed_temp < 0.0)
110             speed_temp = 0.0;
111 }
113 }
114 }
115 } // end switch
118 speed = speed_temp;
119 }
```

Obviously, our algorithm produces a much more efficient solution, especially for
those real programs, which have hundreds or thousands of lines. Although it takes
time scanning back and forth through the program text to find related variables, this
extraction dramatically reduces the computation in the following steps. Otherwise, we
need compute the range for every variable at every statement.
Related Work

A number of researchers have been working on problems related to variable range analysis, and many of them stem from a formal foundation – Abstract Interpretation [4] [5]. We briefly compare our approach with several similar techniques.

Constant propagation [8] is a well-known flow analysis problem whose goal is to identify expressions that are constant on every possible execution of a program and can therefore be evaluated at compile time rather than runtime. The constant information is simply propagated around the control flow graph. Our algorithm uses this mechanism, instead of computing constants, we want to determine the range of values a variable may take.

The recent literature on SSA (Static Single Assignment) form has concentrated on the use for various program optimizations, including variable range analysis. It was developed as a dataflow representation geared towards propagating values through source code. In our algorithm, we use its idea of Phi-Functions, which are inserted at the “joins” where more than one versions of a variable meet, e.g. the exit of if-then-else structure.

There are also efforts that work on symbolic range analysis [6] [7]. It represents the bounds of value ranges via symbolic expressions, where each symbol corresponds to a program variable that itself may have symbolic bounds. Blume [6] have developed algorithms to compare and compute the difference between arbitrary symbolic expressions, using constraint information derived from the program text.
Also related is work on adopting variable range analysis to statistically predict if a specific thread of code can be executed. The technique tracks the weighted value ranges of variables through a program, much like constant propagation. Branch prediction is then performed by simply consulting the value range of the appropriated variable.
Conclusions

This thesis presented an efficient algorithm for solving range constraints. The basic implementation has been completed and delivered to NASA for initial use.

- The algorithm is given in pseudo-code functions, so the implementation should be quite straightforward.
- Given initial range information, our technique statically steps through the source code to analyze values a variable may take at any given point of the program.
- For each instruction, evaluate RHS and update LHS. Three program-building structures have been covered in this thesis: assignment statement, IF condition and loop.
- We convert loop iterations into straight-line code by unwinding the loop body. Thus it can be treated same as a regular range propagation problem.
- For conditional points, we consider “then” and “else” branches independently and then take the union of resulting ranges from each condition.
- The strength of our algorithm is its efficiency for real programs. Before propagation, we first select out those “related” variables according to the data dependency graph, and other variables and statements will be excluded from the range calculation. Details see Section 3. This saves a lot of computing effort.
- According to the range of variables, our algorithm can detect and eliminate dead code, e.g. a conditional branch that will never be executed.

The solution still has limitations at this time:
• We focus on numeric (integer, floating) values. And the range calculations are limited to basic operations, e.g. plus, minus, multiplication, division, etc.

• The performance depends on the program to be analyzed. If most variables within the program are strongly related, and since each call to function Create_Variable (described in Section 3) requires a full traverse of program text from top to bottom, in this case the code parsing time increases without any reduction of range calculation. Our approach works best when the target variable is loosely dependent on other variables.

• The current algorithm focuses on discovering semantic information about the program without any run-time inputs, thus have the nature of precision loss during range propagation.

In conclusion, this thesis addresses the problem of identifying value ranges that variables may take within a program. The handling for loop iterations and conditional branches is explained in detail with example solutions. The idea we have presented can provide useful insight into the application of program verification and validation.
Future Research

Our project is work in progress, and many details remain to be considered. We currently have a portion of the algorithm implemented, where the ranges are limited to integer and floating point values. The future work would be extending the application to other data types, e.g. arrays.

Empirical benchmark results are not yet available. We plan to perform extensive experiments to evaluate the algorithm. Different sets of source code will be used as inputs, e.g. different complexity, number of variables, dependency relations, etc. We will collect the execution profilings, which reflect the accuracy/efficiency of our algorithm.

I will also look into the idea of dynamic analysis, or hybrid static/dynamic analysis, in order to address the “too conservative” issue our static solution may have.
References


Appendix A – Example 3 Program Code

// vehicle.h for traffic simulation
// includes class route for vehicle

#ifndef VEHICLE_H_
#define VEHICLE_H_

#include "simulation.h"

class route
{
    public:
        route():intersection_count(0), moving(reset), next(0){}
        route(int Aroute_count, direction Amoving, route *route_list);
        void add_route(int Aroute_count, direction Amoving);
        direction get_moving(void);
        int get_count(void);
        route *get_next(void);
    
    private:
        int intersection_count;
        direction moving;
        route *next;
};

// vehicle.h the vehicle class for autonomous vehicle

class vehicle
{
    public:
        vehicle( float Alength,
                  float Aweight,
                  float Aaccel,
                  float Adecel,
                  float Aposition,
                  float Aspeed,
                  float Aspeed_limit,
                  float Ax_base,
                  float Ay_base,
                  direction Amoving,
                  float Asignal_pos,
signal_state Asignal,
    vehicle *Anext_ptr);
void move(void);
vehicle *next(void);
void new_next(vehicle *a_next);
bool exiting(void);
void set_new_lane_data(float Aspeed_limit,
    float Ax_base,
    float Ay_base,
    direction Amoving,
    float Asignal_pos,
    signal_state Asignal);
void set_signal(signal_state Asignal);
void draw(bool erase_only=false);

private:
    int id;
    float length;
    float weight;
    float accel;
    float decell;
    float position;
    float position_last;
    float speed;
    float speed_limit;
    float time_last_move;
    float x_base;
    float y_base;
    direction moving;
    signal_state signal;
    float signal_pos;
    route *route_list;
    route *route_now;
    int route_count;
    vehicle *next_ptr;
}; // end vehicle.h

#endif // VEHICLE_H_
// vehicle.cc  for traffic simulation
//     includes route class for vehicle

#include "vehicle.h"
#include <cstdio>

route::route(int Aroute_count, direction Amoving, route *route_list)
{
    intersection_count = Aroute_count;
    moving = Amoving;
    next = route_list;
}

void route::add_route(int Aroute_count, direction Amoving)
{
    next = new route(intersection_count, moving, next);
    intersection_count = Aroute_count;
    moving = Amoving;
}

direction route::get_moving(void)
{
    return moving;
}

int route::get_count(void)
{
    return intersection_count;
}

route *route::get_next(void)
{
    return next;
}

// vehicle.cc

vehicle::vehicle(float Alength,
                 float Aweight,
                 float Accel,
float Adecell,
float Aposition,
float Aspeed,
float Aspeed_limit,
float Ax_base,
float Ay_base,
direction Amoving,
float Asignal_pos,
signal_state Asignal,
vehicle *Anext_ptr)
{
  id = unique_id();
  length = Alength;
  weight = Aweight;
  accel = Aaccel;
  decell = Adecell;
  position = Aposition;
  position_last = Aposition;
  speed = Aspeed;
  speedlimit = Aspeed_limit;
  time_last_move = (float)clock()/CLK_PER_SEC;
  x_base = Ax_base;
  y_base = Ay_base;
  moving = Amoving;
  signal = Asignal;
  signal_pos = Asignal_pos;
  next_ptr = Anext_ptr;
}

void vehicle::move(void)
{
  int i, size;
  float delta_time;
  float separation = 16.0/10.0; // 16 feet per 10 feet per second
  float speed_temp;
  float position_temp;
  float stopping_distance;
  
  delta_time = 1;
  size = 5;
  time_last_move = (float)clock()/CLK_PER_SEC;
  position_last = position;
  speed_temp = speed;
  if(position < signal_pos)
{ 
// attempt normal movement, accelerate to speed limit 
if (speed_temp <= speed_limit) // try to accelerate
{
    for (i=0; i<size; i++)
        speed_temp = speed_temp + accel*delta_time;
    if (speed_temp > speed_limit)
        speed_temp = speed_limit;
}
position_temp = position + speed_temp*delta_time;

// if signal is red, yellow or stop in range, must decelerate 
switch (signal)
{
    case red:
    case yellow:
    case stop:
        stopping_distance = 0.5 * speed_temp * speed_temp / decell;
        if (stopping_distance > signal_pos-position_temp)
        {
            if (speed > 0.0) // can not back up 
            {
                for(i=0; i<size; i++)
                    speed_temp = speed_temp - decell*delta_time;
                if (speed_temp < 0.0)
                    speed_temp = 0.0;
            }
            position_temp = position + speed_temp*delta_time;
        }
    } // end switch
    position = position_temp;
    speed = speed_temp;
}

draw();
}

} // end move

vehicle *vehicle::next(void)
{
    return next_ptr;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
void vehicle::new_next( vehicle *a_next )
{
    next_ptr = a_next;
}

bool vehicle::exiting( void )
{
    if ( this == 0 )
    {
        return false;
    }
    if ( position >= signal_pos && signal != red && signal != yellow )
    {
        if ( route_list != 0 )
        {
            route_count = route_count+1;
            if ( route_now->get_count() >= route_count )
            {
                moving = route_now->get_moving(); // go this way
                route_count = 0;
                route_now = route_now->get_next();
                if ( route_now->get_moving() == reset )
                {
                    route_now = route_list; // reset
                }
            }
            return true;
        }
        else
        {
            return false; // (position >= signal_pos);
        }
    }

void vehicle::set_new_lane_data(float Aspeed_limit,
                                    float Ax_base,
                                    float Ay_base,
                                    direction Amoving,
                                    float Asignal_pos,
173 signal_state Asignal) {
174     position_last = position;
175     draw(true); // erase id in old position
176     position = position-signal_pos; // retain increment beyond end
177     if(position < 0.0) position = 0.0;
178     speed_limit = Aspeed_limit;
179     x_base = Ax_base;
180     y_base = Ay_base;
181     moving = Amoving;
182     signal_pos = Asignal_pos;
183     signal = Asignal;
184 }
185
186 void vehicle::set_signal(signal_state Asignal) {
187     signal = Asignal;
188 }
189
190 void vehicle::draw(bool erase_only) {
191     float x_offset;
192     float y_offset;
193     char any_string[4];
194     sprintf(any_string,"%d",id);
195     get_offsets(moving, x_offset, y_offset);
196     WRITE_TEXT(x_base+x_offset*position_last/my_scale,
197                 y_base+y_offset*position_last/my_scale+0.5,
198                 any_string,
199                 CLEAR_OUTLINE);
200     if(erase_only) return;
201     WRITE_TEXT(x_base+x_offset*position/my_scale,
202                 y_base+y_offset*position/my_scale+0.5,
203                 any_string,
204                 OUTLINE);
205 } // end draw
206
207 // end vehicle.cc

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.