Chapter 5
Analysis of Symbolic Trace

J. C. Huang
Department of Computer Science
University of Houston

The purpose of doing trace analysis

The significance of a correct test result, however, can be enhanced by analyzing the execution path to determine the condition under which it will be traversed, and the nature of computation to be performed in the process.

Symbolic trace

The symbolic trace of an execution path in a program is a list of statements and branch predicates that occur on the path.

For example, consider the C++ program listed below.

Program 5.1:

```
#include <iostream>
#include <string>
using namespace std
int atoi(string& s)
{
   int i, n, sign;
   i = 0;
   while (isspace(s[i]))
      i = i + 1;
   if (s[i] == '-')
      sign = -1;
   else
      sign = 1;
   if (s[i] == '+' || s[i] == '-')
      i = i + 1;
   n = 0;
   while (isdigit(s[i])) {
      n = 10 * n + (s[i] - '0');
      i = i + 1;
   }
   return sign * n;
}
```

Example

Potentially, this program can be executed along the path described by the following symbolic trace.

Trace 5.2:

```
i = 0;
/\ !(isspace(s[i]));
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i] == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

Another example

Consider another symbolic trace listed below that describes the path that iterates each loop one more time and redirecting the execution at the first “if” statement.

Trace 5.3:

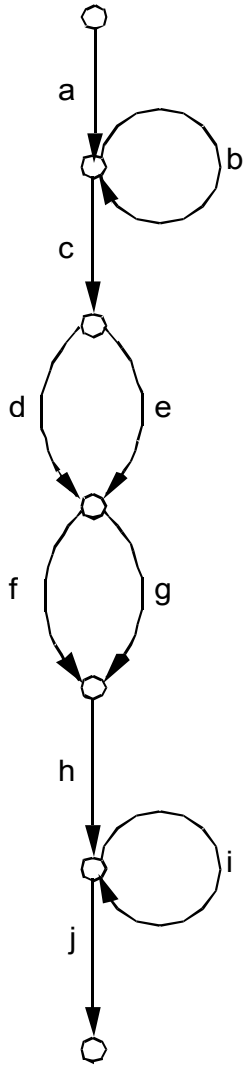
```
#include <iostream>
#include <string>
using namespace std
int atoi(string& s)
{
i = 0;
/\ (isspace(s[i]));
i = i + 1;
/\ !(isspace(s[i]));
/\ (s[i] == '-');
sign = -1;
/\ !(s[i] == '+' || s[i] == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
}
```

Program graph

To help envisaging the structure of an execution path, we will use a directed graph to represent the control-flow structure of a program. Each edge in the graph is associated with a pair of the form <∧C, S>, where C is the condition that must be true in order for that edge to be traversed, and S is a description of the computation to be performed when that edge is traversed.

Program graph (continued)

Program 5.1 can thus be represented by the program graph shown in Fig. 5.1, and Trace 5.2 can be represented by the program graph depicted in Fig. 5.2.



a: `i=0;`

b: `/\isspace(s[i]); i=i+1;`

c: `/\!(isspace(s[i]));`

d: `/\s[i]==' -'; sign= -1;`

e: `/\!(s[i]==' -'); sign=1;`

f: `/\s[i]=='+'||s[i]==' -'; i=i+1;`

g: `/\!(s[i]=='+' || s[i]==' -');`

h: `n=0;`

i: `/\isdigit(s[i]); n=10*n+s[i] -'0'; i=i+1;`

j: `/\!(isdigit(s[i])); ctoi=sign*n;`

Figure 5.1 The program graph of Program 5.1.

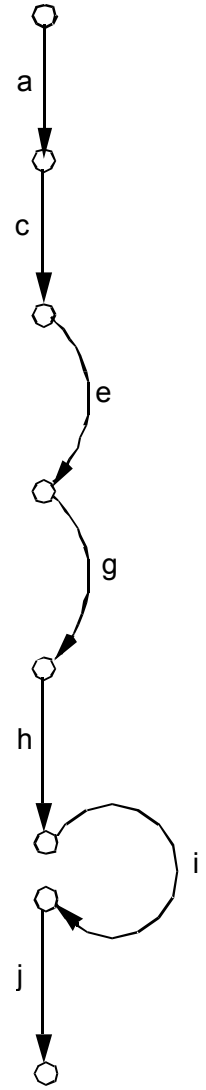


Figure 5.2 An execution path in Program 5.1.

The semantics of </\C, S>

It is the basic element of a symbolic trace.

If C is true, </\C, S> and "if C then S" will do the same.

If C is false, </\C, S> will give no information whatsoever. The statement “if C then S”, however, will maintain status quo.

The semantics of </\C, S> (continued)

If C is false, the branch associated with </\C, S> will not be traversed.

If C is false, the meaning of </\C, S> becomes undefined.

State constraint

We shall call /\C a constraint instead of a path or branch predicate. We shall use it as a shorthand notation for the restrictive clause:

The program state at this point must satisfy predicate C, or else the program becomes undefined.

State constraint (continued)

By program state here we mean the aggregate of values assumed by all variables involved. Since this clause constrains the states assumable by the program, it is called a *state constraint*, or a *constraint* for short, and is denoted by $/\backslash C$.

State constraint (continued)

State constraints are designed to be inserted into a program to create another program. For instance, given a program of the form S_1; S_2, a new program can be created by inserting the constraint /\C between the two statement to form a new program: S_1; /\C; S_2.

State constraint (continued)

This new program is said to be created from the original by constraining the program states to C prior to execution of S_2. Intuitively, this new program is a subprogram of the original because its definition is that of the original program restricted to C. Within that restriction this new program performs the same computation as the original.

State constraint (continued)

A state constraint is a semantic modifier. The meaning of a program modified by a state constraint can be formally defined in terms of Dijkstra's *weakest precondition* as follows. Let S be a programming construct and C be a predicate, then for any postcondition R,

Axiom 5.4: wp(/\C;S, R) ≡ C ∧ wp(S, R).

Logical-equivalence relation

Definition 5.5: Program S_1 is said to be *equivalent* to S_2 if $wp(S_1, R) \equiv wp(S_2, R)$ for any postcondition R. This relation is denoted by $S_1 \Leftrightarrow S_2$.

Subprogram relation

Definition 5.6: Program S_2 is said to be a *subprogram* of program S_1 if $wp(S_2, R) \supset wp(S_1, R)$ for any postcondition R. This relation is denoted by $S_1 \Rightarrow S_2$.

A symbolic trace is a subprogram

Thus, by Definition 5.6, program $S_1;/\backslash C;S_2$ is a subprogram of program $S_1;S_2$.

In general, a symbolic trace of a program constitutes a subprogram of the original as defined by Definition 5.6.

Two trivial relations

(1) /\T; S ⇔ S for any S

(2) /\F; S ⇔ /\F; S' for any S and S'.

In words, it would not change anything by constraining a program with a predicate that is always true, and any two programs are logically equivalent if each is constrained by a predicate that is always false.

Rules for moving constraints

A state constraint not only directly constrains the program state at the point where it is placed, but also indirectly at other points upstream and downstream in control flow as well.

What follows are the rules that allow us to move a constraint up or down without changing anything.

The scope of a constraint

The *scope* of a state constraint, which is defined to be the range of control flow within which the constraint has an effect, may or may not span the entire program. A state constraint will have no effect beyond a statement that undefines, or assigns a constant value to, the variables involved.

Moving a constraint upstream

Theorem 5.7: $S;/\backslash R \Leftrightarrow /\backslash wp(S, R);S$.

Eample: simplification of Trace 5.2

```
Trace 5.2:
i = 0;
/\ !(isspace(s[i]));
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i]
   == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
i = 0;
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i]
   == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
i = 0;
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i]
   == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
   == '-');
i = 0;
sign = 1;
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
       == '-');
i = 0;
sign = 1;
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
   == '-');
/\ (isdigit(s[0]));
i = 0;
sign = 1;
n = 0;
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
      == '-');
/\ (isdigit(s[0]));
i = 0;
sign = 1;
n = 0;
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
   == '-');
/\ (isdigit(s[0]));
i = 0;
sign = 1;
n = 0;
n = 10 * n + (s[i] - '0');
/\ !(isdigit(s[i+1]));
i = i + 1;
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
== '-');
/\ (isdigit(s[0]));
i = 0;
sign = 1;
n = 0;
n = 10 * n + (s[i] - '0');
/\ !(isdigit(s[i+1]));
i = i + 1;
return sign * n;
```

⇔

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0]
   == '-');
/\ (isdigit(s[0]));
/\ !(isdigit(s[1]));
i = 0;
sign = 1;
n = 0;
n = 10 * n + (s[i] - '0');
i = i + 1;
return sign * n;
```

Backward substitution

The basic step of source-code transformation made possible by virtue of Theorem 5.7 is also known as *backward substitution*. It is so called because the theorem is usually applied to an assignment statement of the form "x := e" and wp(x:=e, R) is computed by substituting e to each and every occurrence of x in R.

Application to program testing

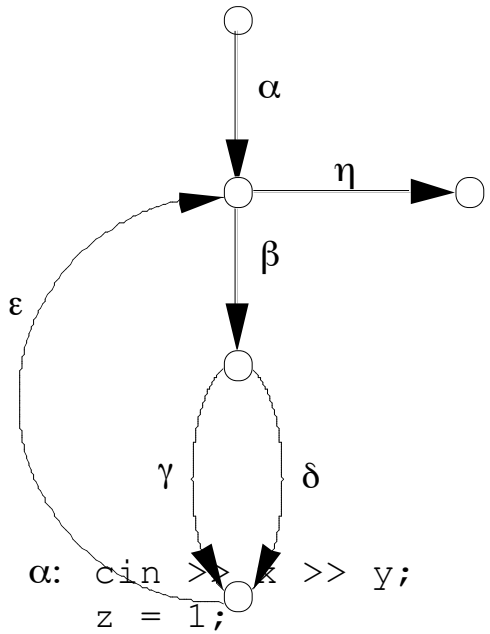
In attempt to find an appropriate test case, it is usually only apparent that a certain condition must be true at some point in the control flow so that a certain branch will be traversed when the control reaches that point, but it is often difficult to tell what has to be true at the input to make that condition true at that point. We can find the answer systematically by performing backward substitution repeatedly along the symbolic trace.

Another important application

In other occasions it may become necessary to determine if a given (syntactic) path in the program (graph) is feasible, and if so, what inputs would cause that path to be traversed. We could find the answer systematically by constructing the symbolic trace of the path, and then performing backward substitution repeatedly until all path constraints are located on the top of the trace. Any input that satisfies all the constraints on the top simultaneously will cause the trace to be traversed

An example

```
Program 2.2:
    main()
    {
        int x, y, z;
        cin >> x >> y;
        z = 1;
        while (y != 0) {
            if (y % 2 == 1)
                z = z * x;
            y = y / 2;
            x = x * x;
        }
    cout << z << endl;
    }
```



where α:
```
cin >> x >> y;
z = 1;
```
β:
```
/\ y != 0;
```
γ:
```
/\ !(y % 2 == 1);
```
δ:
```
/\ y % 2 == 1;
z = z * x;
```
ε:
```
y = y / 2;
x = x * x;
```
η:
```
/\ !(y != 0);
cout << z << endl;
```

Figure 5.3 The program graph of Program 2.2 .

A few potential execution paths

By studying the path segments in Fig. 5.3 that have to be exercised during a du-path test, we see that the loop in Program 2.2 may have to be iterated zero, one, or two times to traverse the paths described by $\alpha\eta$, $\alpha\beta\delta\varepsilon\eta$, $\alpha\beta\gamma\varepsilon\eta$, $\alpha\beta\delta\varepsilon\beta\delta\varepsilon\eta$, $\alpha\beta\delta\varepsilon\beta\gamma\varepsilon\eta$, $\alpha\beta\gamma\varepsilon\beta\delta\varepsilon\eta$, and $\alpha\beta\delta\varepsilon\beta\gamma\varepsilon\eta$.

The shortest execution path

αη:
```
cin >> x >> y;
z = 1;
/\ !(y != 0);
cout << z << endl;
```

⇔
```
cin >> x >> y;
/\ y == 0;
z = 1;
cout << z << endl
```

can be traversed by letting y == 0.

Another example

abδeh:
```
cin >> x >> y;
z = 1;
/\ y != 0;
/\ y % 2 == 1;
z = z * x;
y = y / 2;
x = x * x;
/\ !(y != 0);
cout << z << endl;
```

⇔

```
cin >> x >> y;
/\ y != 0;
/\ y % 2 == 1;
/\ (y / 2 == 0);
z = 1;
z = z * x;
y = y / 2;
x = x * x;
cout << z << endl;
```

This path can be traversed by letting y == 1 because that's what is required to make all the path predicates (listed below) true.

```
/\ y != 0;
/\ y % 2 == 1;
/\ (y / 2 == 0);
```

Yet another example

αβγεη:

```
cin >> x >> y;
z = 1;
/\ y != 0;
/\ !(y % 2 == 1);
y = y / 2;
x = x * x;
/\ !(y != 0);
cout << z << endl;
```

The simplified trace

⇔

```
cin >> x >> y;
/\ y != 0;
/\ !(y % 2 == 1);
/\ (y / 2 == 0);
z = 1;
y = y / 2;
x = x * x;
cout << z << endl;
```

The path predicates

This path cannot be used as a candidate path in selecting test cases because its path predicates (listed below) cannot be satisfied simultaneously by any y.

```
/\ y != 0;
/\ !(y % 2 == 1);
/\ (y / 2 == 0);
```

Simplifying concatenation of constraints

It can be shown, as the direct consequence of Axiom 5.4, that concatenation means logical conjunction semantically. Formally,

Corollary 5.8: $/\backslash C_1;/\backslash C_2;S \Leftrightarrow /\backslash C_1 \wedge C_2;S$.

Moving and simplifying statements

Corollary 5.11:

$$x := E_1;\ x := E_2 \ \Leftrightarrow\ x := (E_2)_{E_1 \to x}$$

Here $(E_2)_{E_1 \to x}$ denotes an expression obtained by substituting x with E_1 for every occurrence of x in E_2.

Corollary 5.12: If x_2 does not occur in E_1 then

$x_1 := E_1;\ x_2 := E_2 \Leftrightarrow x_2 := (E_2)_{E_1 \rightarrow x_1};\ x_1 := E_1$.

Redundant constraint

Definition 5.13: A statement in a program is said to be *redundant* if its sole purpose is to define the value of a data structure, and this particular value is not used anywhere in the program.

Statement simplification

Corollary 5.14: If $x_1 := E_1; x_2 := E_2$ is a sequence of two assignment statements such that, by interchanging these two statements, $x_1 := E_1$ becomes redundant, then

$$(x_1 := E_1; x_2 := E_2) \Leftrightarrow x_2 := (E_2)_{E1 \rightarrow x1}$$

Example

```
/\   (isdigit(s[0])) &&
        !(isdigit(s[1]));
i = 0;
n = s[i] - '0';
i = i + 1;
sign = 1;
return sign * n;
```

⇔

```
/\ (isdigit(s[0])) &&
        !(isdigit(s[1]));
i = 0;
n = s[i] - '0';
i = i + 1;
return 1 * n;
sign = 1;
```

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
i = 0;
n = s[i] - '0';
i = i + 1;
return 1 * n;
sign = 1;

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
i = 0;
n = s[i] - '0';
i = i + 1;
return n;

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
i = 0;
n = s[i] - '0';
i = i + 1;
return n;

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
i = 0;
i = i + 1;
return n;

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
i = 0;
i = i + 1;
return n;

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
i = 0 + 1;
i = 0;
return n;

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
i = 0 + 1;
i = 0;
return n;

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
i = 1;
return n;

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
i = 1;
return n;

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
return n;

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
n = s[0] - '0';
return n;

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
return s[0] - '0';
n = s[0] - '0';

Example (continued)

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
return s[0] - '0';
n = s[0] - '0';

⇔

/ \ (isdigit(s[0])) &&
!(isdigit(s[1]));
return s[0] - '0';

From this analysis we can definitely conclude, without further testing, that function `atoi` will work correctly if the input is a single digit.

Exceptions

```
r = a % b;
a = b;
b = r;
r = a % b;
a = b;
b = r;
```

⇔

$a = a \% b;$
$b = b \% a;$
$r = b;$

```
This is an example
trace that cannot
be simplified by
the present method,
although it can be
simplified by other
methods.
```

Other simplification methods

Corollaries 5.11, 5.12, and 5.14 together provide a method of symbolic trace simplification based on the syntax of the trace.

Simplification can also be done based on the semantics instead of the syntax. One such method is illustrated below.

Example

Consider the following sequence of assignment statements in C++:

```
r = a % b;
a = b;
b = r;
r = a % b;
a = b;
b = r;
```

Example (continued)

The three corollaries just mentioned are not applicable. Yet this sequence can be simplified to the one shown below.

```
a = a % b;
b = b % a;
r = b;
```

Can you devise rules to accomplish this?

Supporting tools

We can build two software tools: an instrumentor and a trace analyzer.

The instrumentor inserts necessary software instruments into a program to generate symbolic trace automatically. The trace analyzer helps the user to rewrite a given symbolic trace into another.

Supporting tools (continued)

We discuss the instrumentor in Chapter 7.

The functional design of a trace analyzer is given below.

Trace analyzer

We can build an analyzer to mimic the way we analyze a symbolic trace manually.

The analyzer needs to provide the basic functional capabilities of a modern interactive screen-oriented text editor: displaying a chunk of trace being analyzed, scrolling it up and down, cutting and pasting, searching and replacing, undoing and repeating an operation, etc.

Trace analyzer (continued)

The analyzer should be able to move a constraint upstream per Theorem 5.7 and Corollary 5.8, and to move an assignment statement downstream per Corollaries 5.11, 5.12, and 5.14.

The analyzer should have the capacity to simplify a predicate or a computational expression.

Trace analyzer (continued)

The operation of expression transformation and simplification cannot be completely automated. They have to be carried out interactively. The effectiveness of a specific tool implementation is therefore largely dependent on how its user interface is designed.

Trace analyzer (continued)

In the following we use an illustrative example to show how such an interface can be designed. The interface scheme has been proved to be effective through experimentation.

The tool is to analyze Trace 5.2 step by step as depicted below.

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
i = 0;
/\ !(isspace(s[i]));
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i] == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
i = ◊ 0;
/\ !(isspace(s[i]));
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i] == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
   i =0;
◊  /\ !(isspace(s[i]));
   /\ !(s[i] == '-');
   sign = 1;
   /\ !(s[i] == '+' || s[i] == '-');
   n = 0;
   /\ (isdigit(s[i]));
   n = 10 * n + (s[i] - '0');
   i = i + 1;
   /\ !(isdigit(s[i]));
   return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

◊ /\ !(isspace(s[0]));
i =0;
/\ !(s[i] == '-');
sign = 1;
/\ !(s[i] == '+' || s[i] == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\ !(isdigit(s[i]));
return sign * n;

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0] == '-');
/\ (isdigit(s[0]));
/\ !(isdigit(s[1]));
i = 0;
sign = 1;
n = 0;
n = 10 * n + (s[i] - '0');
i = i + 1;
return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
   /\ !(isspace(s[0]));
   /\ !(s[0] == '-');
   /\ !(s[0] == '+' || s[0] == '-');
   /\ (isdigit(s[0]));
   /\ !(isdigit(s[1]));
   i = 0;
   sign = 1;
◊  n = 0;
   n = 10 * n + (s[i] - '0');
   i = i + 1;
   return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
   /\ !(isspace(s[0]));
   /\ !(s[0] == '-');
   /\ !(s[0] == '+' || s[0] == '-');
   /\ (isdigit(s[0]));
   /\ !(isdigit(s[1]));
   i = 0;
   sign = 1;
◊  n = 10 * 0 + (s[i] - '0');
   i = i + 1;
   return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
   /\ !(isspace(s[0]));
   /\ !(s[0] == '-');
   /\ !(s[0] == '+' || s[0] == '-');
   /\ (isdigit(s[0]));
   /\ !(isdigit(s[1]));
   i = 0;
   sign = 1;
◊  n = s[i] - '0';
   i = i + 1;
   return sign * n;
```

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
/\ !(isspace(s[0]));
/\ !(s[0] == '-');
/\ !(s[0] == '+' || s[0] == '-');
/\ (isdigit(s[0]));
/\ !(isdigit(s[1]));
return s[0] - '0';
```

Trace analyzer (continued)

Up to this point, all operations performed can be carried out with relative ease. The information needed to do so can be obtained by parsing the symbolic trace and performing a data-flow analysis for all the variables involved.

Trace analyzer (continued)

Simplification of frequently encountered expressions, such as reducing x+0 to x, x×1 to x, or x+1+1+1 to x+3, can be done by using a rule-based rewriting system.

Trace analyzer (continued)

Simplification of a concatenation of constraints is to be done based on Corollary 5.8, which says /\$C_1$;/\$C_2$;S is logically equivalent to /\$C_1$ ∧ C_2 ;S. It is also known that if C_1 implies (⊃) C_2 then C_1 ∧ C_2 can be reduced to C_1. The problem is to find pairs of constraints that can be simplified by using this fact.

Trace analyzer (continued)

A simple-minded approach to simplification of a conjunction of n predicates $C_1 \wedge C_2 \wedge ... \wedge C_n$ is to use a mechanical theorem prover to prove that $C_i \supset C_j$ for all $1 \leq i, j \leq n$ and $i \neq j$. If successful, it means that C_i implies C_j, and therefore C_j can be discarded. Otherwise C_j remains.

Trace analyzer (continued)

If the analyzer has this theorem proving capability, then all we need to do is to select all the constraints on the top of the trace and click the simplify button. The analyzer should display the following as the result.

Trace analyzer (continued)

C-up S-down Simplify Validate Undo

```
/\ (isdigit(s[0]));
/\ !(isdigit(s[1]));
return s[0] - '0';
```