Program termination · Lecture 2

Berkeley · Spring ’09

Byron Cook

Review from the previous lecture

- Program termination = WF transition relation
- Proving WF can be reduced to finding a larger *ranking relation*
- Accurate transition relations often too hard to compute
  - Supporting invariants needed to establish termination
- Unions of WF-relations not WF, but transitive closure can be used to offset the problem

Review from the previous lecture

→ We can use variables with finite range to decompose termination proofs (*e.g.* the program counter)

→ Linear ranking function synthesis is decidable

- But linear ranking functions are often not enough…………

Review from the previous lecture

- Finding linear ranking functions (for relations with only linear updates and conditions) is decidable

- Not all WF linear relations have linear ranking functions, *e.g.*
  - $R \triangleq \mathsf{x} > 0 \wedge \mathsf{x}' = \mathsf{x} - \mathsf{y} \wedge \mathsf{y}' = \mathsf{y} + 1$
  - $R \triangleq \mathsf{x} \geq 0 \wedge \mathsf{x}' = -2\mathsf{x} + 10$
  - Ackermann's function
  - ....

Overview

- → Notes on a representation for programs
- → Checking termination arguments
- → Refining termination arguments
- → Induction
- → Termination analysis

Programs

$$[\![x := e]\!]_V \triangleq \{(s, t) \mid \forall v \in V - \{x\}.\ s(v) = t(v) \wedge t(x) = e\}$$

Programs

$$[\![x := e]\!]_V \triangleq \{(s,t) \mid \forall v \in V - \{x\}.\ s(v) = t(v) \wedge t(x) = e\}$$

$$[\![x := *]\!]_V \triangleq \{(s,t) \mid \forall v \in V - \{x\}.\ s(v) = t(v)\}$$

Programs

$$[\![x := e]\!]_V \triangleq \{(s,t) \mid \forall v \in V - \{x\}.\ s(v) = t(v) \wedge t(x) = e\}$$

$$[\![x := *]\!]_V \triangleq \{(s,t) \mid \forall v \in V - \{x\}.\ s(v) = t(v)\}$$

$$[\![\mathrm{assume}(e)]\!]_V \triangleq \{(s,t) \mid \forall v \in V.\ s(v) = t(v) \wedge e[s]\}$$

Programs

→ Programs are rooted cyclic graphs where edges are annotated with finite command sequences

→ A “cutpoint set” can be computed by uniquely numbering the nodes and marking each node that can transition to a node that’s lower in the order

Programs

- → The meaning of the program is a relation constructed from the graph and commands

- → A special variable (not used in the program) `pc` is used to track program location

- → Paths are sequences of `pc` valuations, traces are sequences of commands drawn from paths

Programs

The *relational meaning* of a finite path/trace segment is the relational composition of the meaning of each command.

$$[\![c_1; c_2; \ldots]\!]_{V,\rho} \quad \triangleq \quad [\![c_1]\!]_{V,\rho} \quad ; \quad [\![c_2]\!]_{V,\rho} \quad ; \quad \cdots$$

Programs

The *relational*
ment is the
of each comm

$[\![c_1; c_2; \ldots]\!]_{V,}$

$$(s,t) \text{ s.t. } \exists x_0, x_1, y_0. \bigwedge \left\{ \begin{array}{l} s(\mathsf{x}) = x_0 \\ s(\mathsf{y}) = y_0 \\ x_1 = x_0 + y_0 \\ x_1 > 0 \\ t(\mathsf{x}) = x_1 \\ t(\mathsf{y}) = y_0 \end{array} \right\}.$$

$$[\![\begin{array}{l} \mathsf{x} := \mathsf{x} + \mathsf{y}; \\ \textbf{assume}(\mathsf{x} > 0); \end{array}]\!]$$

Programs

The *relational*
ment is the
of each comm

$[\![c_1; c_2; \ldots]\!]_{V,}$

$$(s, t) \text{ s.t. } \exists x_0, x_1, y_0.$$

$$\bigwedge \left\{ \begin{array}{l} s(\mathsf{x}) = x_0 \\ s(\mathsf{y}) = y_0 \\ \boxed{\begin{array}{l} x_1 = x_0 + y_0 \\ x_1 > 0 \end{array}} \\ t(\mathsf{x}) = x_1 \\ t(\mathsf{y}) = y_0 \end{array} \right\}.$$

$$[\![\begin{array}{l} \mathsf{x} := \mathsf{x} + \mathsf{y}; \\ \textbf{assume}(\mathsf{x} > 0); \end{array}]\!]$$

Overview

- Notes on a representation for programs
- Checking termination arguments
- Refining termination arguments
- Induction
- Termination analysis

Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R\downarrow_{\ast I}^{+})\downarrow_{\mathsf{pc}=k} \subseteq Q \ \textit{iff} \ \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$

Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R{\downarrow}_{I}^{+}){\downarrow}_{\mathsf{pc}=k} \subseteq Q \;\; \textit{iff} \;\; \neg\underline{\textsc{Reachable}}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$

Tools like SLAM,
BLAST, IMPACT,
F-Soft, etc.

New program

Defining $\boxplus$ such that

$$\left(R{\downarrow_{\ast}}^{+}_{I}\right){\downarrow}_{\mathsf{pc}=k} \subseteq Q \;\; \textit{iff} \;\; \neg\textsc{Reachable}_{\boxed{\boxplus(\mathcal{P},k,Q)}}(\ell_{err})$$

Transformations to reachability supporting closure

Supports closure

Defining $\boxplus$ such that

$$(R{\downharpoonright_{\ast}}{}^{+}_{I})\downharpoonright_{\mathsf{pc}=k} \subseteq Q \;\; \textit{iff} \;\; \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$

Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$\underline{(R\downarrow_{*I}^{+})\downarrow_{\mathsf{pc}=k} \subseteq Q} \text{ iff } \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$

Using decomposition strategy we prove termination

Definition. Assume $[\![\mathcal{P}]\!] = (I, R, S)$. A reachability engine $\textsc{Reachable}_{\mathcal{P}}(k)$ returns **false** when

$$R^*(I) \cap [\![\mathsf{pc} = k]\!] = \emptyset$$

If $\textsc{Reachable}_{\mathcal{P}}(k) = \textbf{true}$, then a witness path from 0 to k is returned.

Transformations to reachability supporting closure

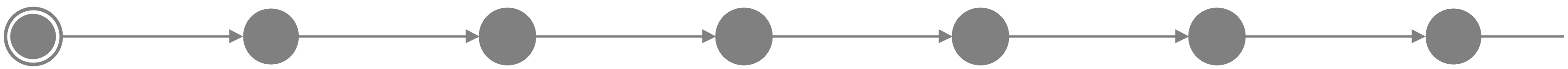
Defining $\boxplus$ such that

$$(R{\downarrow_{\ast}}^{+}_{I}){\downarrow}_{\mathsf{pc}=k} \subseteq Q \;\textit{iff}\; \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$

Transformations to reachability supporting closure

Defining $\boxplus$ such that

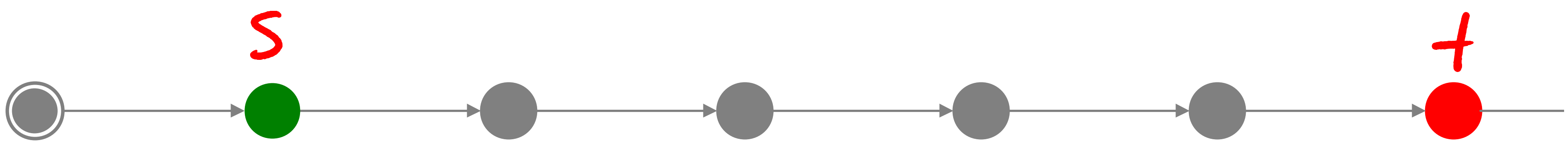
$$(R{\downarrow}_{*}{}^{+}_{I}){\downarrow}_{\mathsf{pc}=k} \subseteq Q \ \textit{iff} \ \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$



Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R{\downarrow}^{+}_{I}){\downarrow}_{\mathsf{pc}=k} \subseteq Q \ \textit{iff} \ \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$



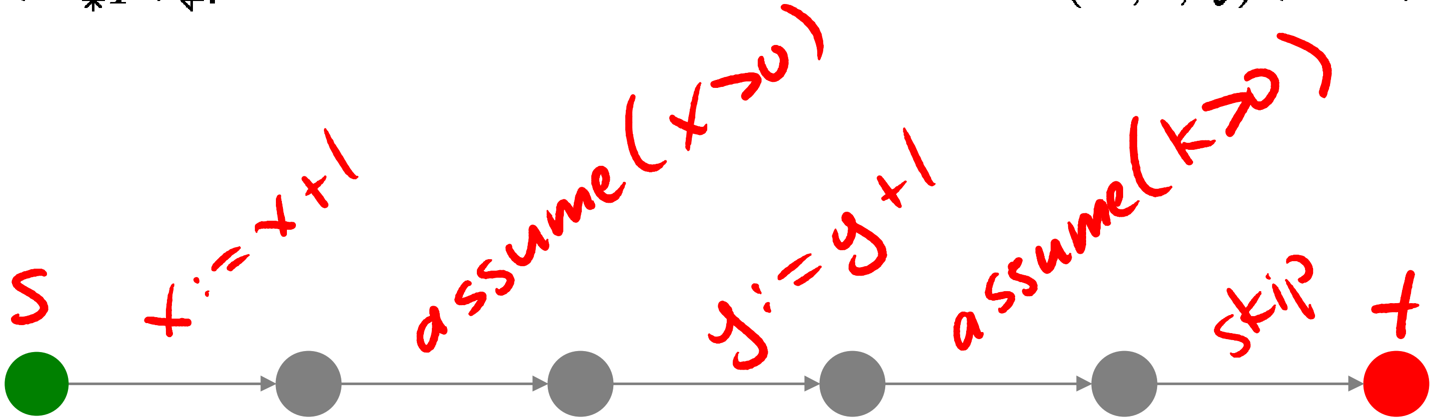
Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R{\downarrow}^{+}_{I})\downarrow_{\mathsf{pc}=k} \subseteq Q \;\; \textit{iff} \;\; \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$

Defining $\boxplus$ such that

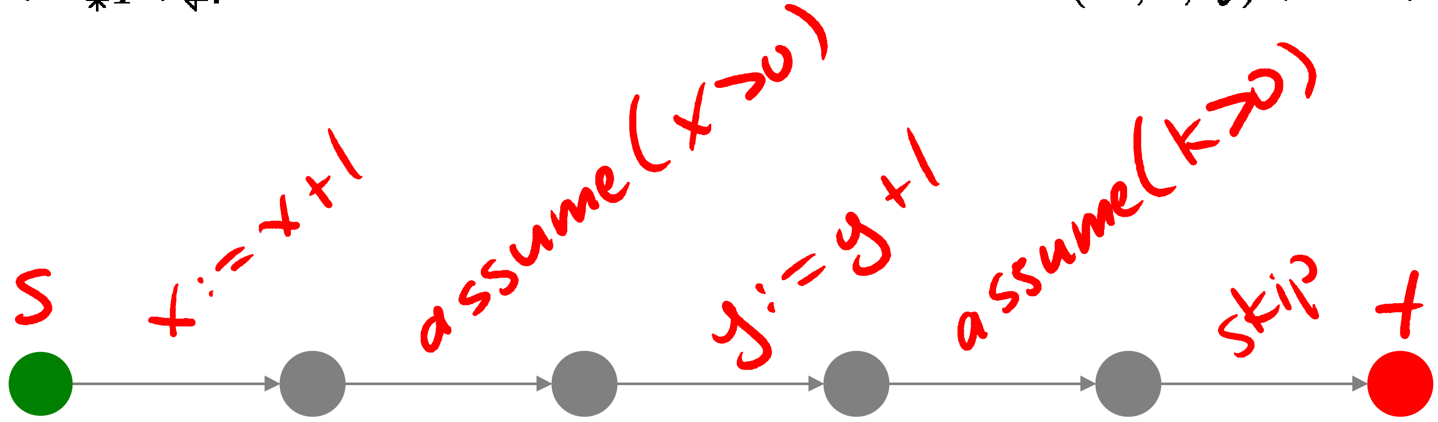
$$(R{\downarrow}^{+}_{*I})\downarrow_{\mathsf{pc}=k} \subseteq Q \ \textit{iff}\ \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$



Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R\big|_{*I}^{+})\big|_{\mathsf{pc}=k} \subseteq Q \;\; \textit{iff} \;\; \neg\textsc{Reachable}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$$



is $(s,t) \in Q$?

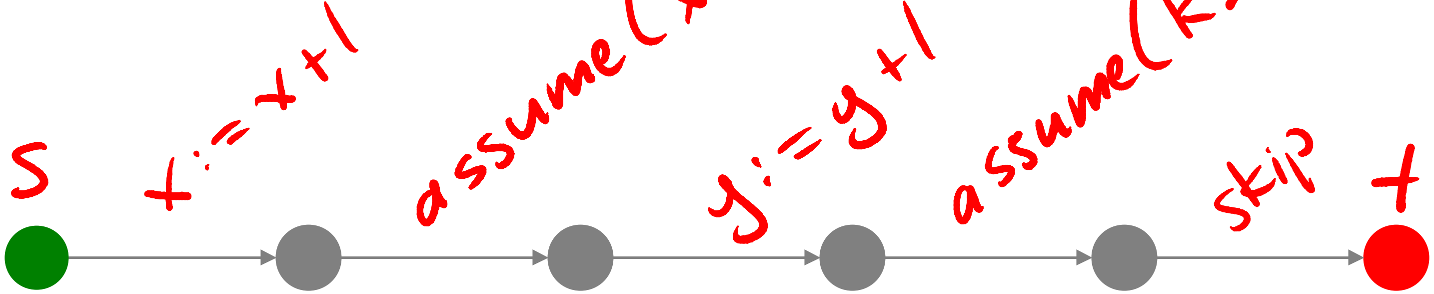
Transformations to reachability supporting closure

$[\![x:=x+1; \mathit{assume}(x>0); \ldots\ldots]\!] * R(t)$

$\subseteq \varphi$?

Def...

$(R|_{*I}^{+})\downarrow \ldots \sqsubseteq Q$ *iff* $\neg\text{REACHABLE}_{\boxplus(\mathcal{P},k,Q)}(\ell_{err})$



is $(s,t) \in \varphi$?

Transformations to reachability supporting closure

```
copied = 0;
    .
    .
    .
while(*) {
    k: loop body
}
```

```
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }
```

Transformations to reachability supporting closure

```
while(*) {
      k:  loop body
}
```

Transformations to reachability supporting closure

```
copied = 0;
    .
    .
    .
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }

  • loop body
}
```

Transformations to reachability supporting closure

assert(e)

$\triangleq$

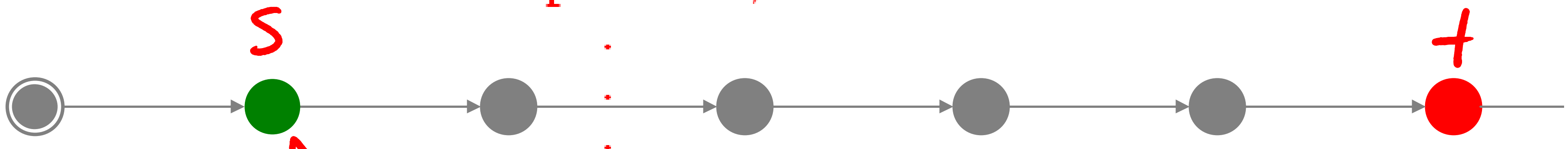
if (¬e) {

ℓ_{err} : skip;

}

```
copied = 0;
    .
    .
    .
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }
  • loop body
}
```

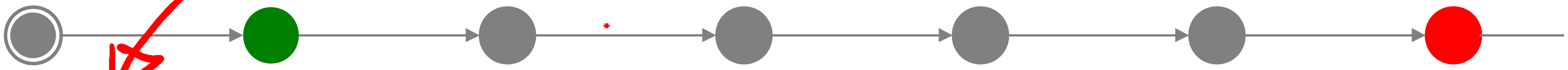
Transformations to reachability supporting closure

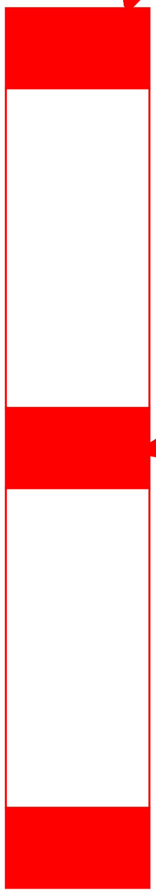



```
copied = 0;
      .
      .
      .

while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }

    loop body
}
```

Transformations to reachability supporting closure

```
copied = 0;
   .
   .
   .
```

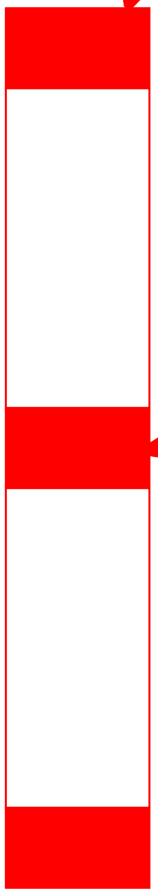




```
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }

    loop body
}
```

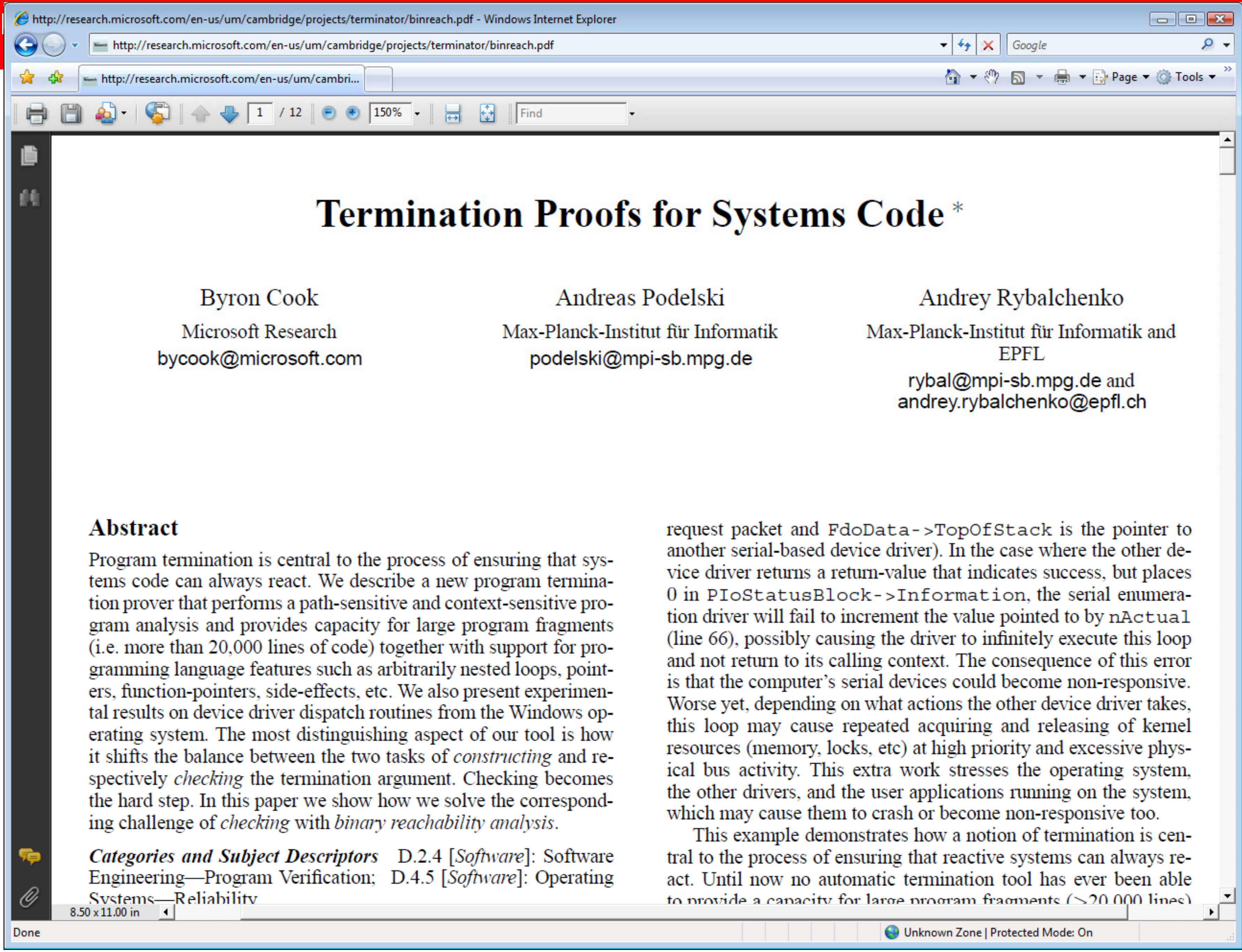
Transformations to reachability supporting closure

```
copied = 0;
.
.
.
while(*) {
    if (copied
        if (*) {
            old_x = x;
            old_y =
            .
            .
            copied = 1;
        }
    } else {
        assert(Q);
    }
    loop body
}
```



$$old\_x > x \wedge old\_x \geq 0$$

T



http://research.microsoft.com/en-us/um/cambridge/projects/terminator/binreach.pdf - Windows Internet Explorer

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
Max-Planck-Institut für Informatik and EPFL
rybal@mpi-sb.mpg.de and andrey.rybalchenko@epfl.ch

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

request packet and `FdoData->TopOfStack` is the pointer to another serial-based device driver). In the case where the other device driver returns a return-value that indicates success, but places 0 in `PIoStatusBlock->Information`, the serial enumeration driver will fail to increment the value pointed to by `nActual` (line 66), possibly causing the driver to infinitely execute this loop and not return to its calling context. The consequence of this error is that the computer's serial devices could become non-responsive. Worse yet, depending on what actions the other device driver takes, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc) at high priority and excessive physical bus activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now no automatic termination tool has ever been able to provide a capacity for large program fragments (>20,000 lines)

Isolation

Nesting of loops allows us to isolate pieces of the program

- → When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop
- → When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop

```
while(x<y) {
    k = nondet();
    while(k>0) {
        k--;
    }
    x++;
}
```

Isolation

Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop

```
while(x<y) {
     k = nondet();
     while(k>0) {
          k--;
     }
     x++;
}
```

Isolation

Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop

```
while(x<y) {
    k = nondet();
    while(k>0) {
        k--;
    }
    x++;
}
```

Comes for free

Isolation

Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop

```
while(x<y) {
   k = nondet();
   while(k>0) {
        k--;
   }
   x++;
}
```

Isolation

Nesting of loops allows us to isolate pieces of the program

- → When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

- → When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop

```
while(x<y) {
    k = nondet();
    while(k>0) {
        k--;
    }
    x++;
}
```

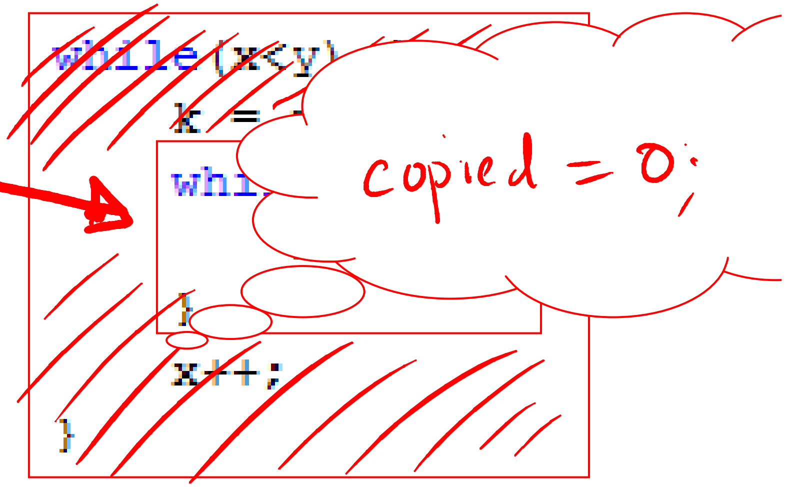
We don't support this yet......

Isolation

Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop



We don't support this yet......

Transformations to reachability supporting closure

```
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }

    loop body
}
```

copied = 0;

Transformations to reachability supporting closure

```
while(*) {
    if (copi...
}
```

Decomposition story now more complex........

copied = 0;

Transformations to reachability supporting closure

"ℓ is not visited infinitely often so long as the context is not entered infinitely often"

loop b

}

copied = 0;

Overview

- Notes on a representation for programs
- Checking termination arguments
- Refining termination arguments
- Induction
- Termination analysis

Rank function synthesis

Definition. $\textsc{Synthesis} : (S \leftrightarrow S) \to (S \to \mathbb{N})$ is a partial function such that $R \subseteq \trianglerighteq_{\textsc{Synthesis}(R)}$ when $\textsc{Synthesis}(R)$ is defined.

Refinement

$T := \emptyset$
while $\textsc{Reachable}_{\boxplus(\mathcal{P},\ell,T)}(\ell_{err})$ **do**
  let $\pi_s, \pi_c =$ lasso in $\boxplus(\mathcal{P}, \ell, T)$ from 0 to ℓ, and ℓ to ℓ_{err}
  let $\rho = \alpha(\llbracket\pi_c\rrbracket^*(\llbracket\pi_s\rrbracket))$
  if $\textsc{Synthesis}(\llbracket\pi_c\rrbracket\downarrow_\rho)$ returns ranking relation f **then**
    $T := T \cup \trianglerighteq_f$
  else
    report "potential counterexample found: π_s,π_c"
  fi
od
report "termination proved with argument T"

Refinement

$$\rho \supseteq [\![\pi_c]\!]^*([\![\pi_s]\!])$$

$T := \emptyset$

while REACHA

let π_s, π_c ...so in $\boxplus(\mathcal{P}, \iota, T)$ from 0 to ℓ, and ℓ to ℓ_{err}

let $\rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!]))$

if SYNTHESIS($[\![\pi_c]\!]\downarrow_\rho$) returns ranking relation f **then**

$T := T \cup \trianglerighteq_f$

else

report "potential counterexample found: π_s,π_c"

fi

od

report "termination proved with argument T"

Refinement

$$\rho \supseteq [\![\pi_c]\!]^*([\![\pi_s]\!])$$

$$[\![\pi_c]\!]\downarrow_{[\![\pi_s]\!]}^{*} \subseteq [\![\pi_c]\!]\downarrow_{\rho}$$

$T := \emptyset$

while REACHA…

  let π_s, π_c … in E…

  let $\rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!]))$

  if SYNTHESIS($[\![\pi_c]\!]\downarrow_\rho$) …

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c"

  fi

od

report "termination proved with argument T"

R

sas05.pdf - Adobe Reader

File Edit View Document Tools Window Help

1 / 16 200% Find

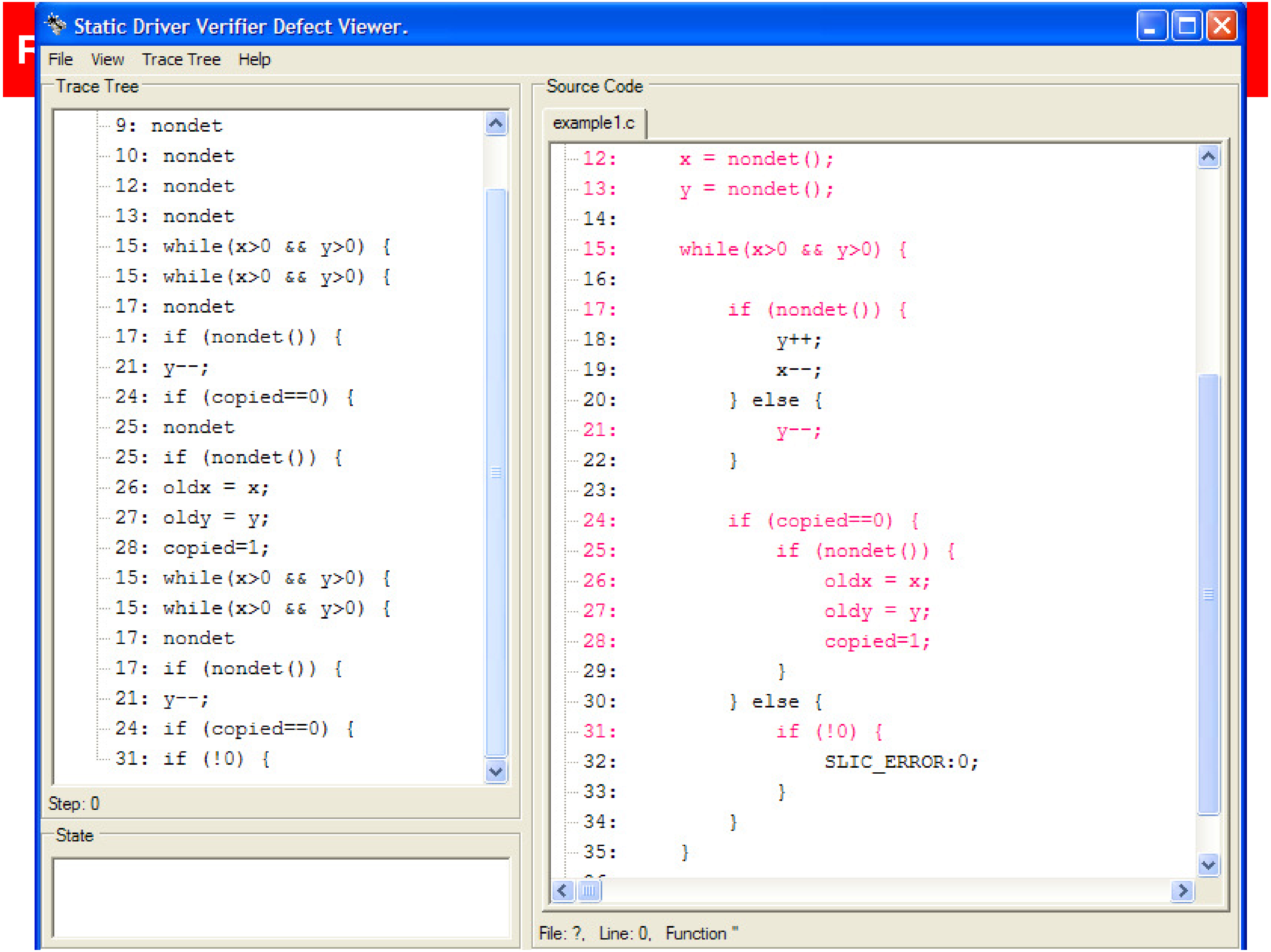
Abstraction Refinement for Termination

Byron Cook[1], Andreas Podelski[2], and Andrey Rybalchenko[2]

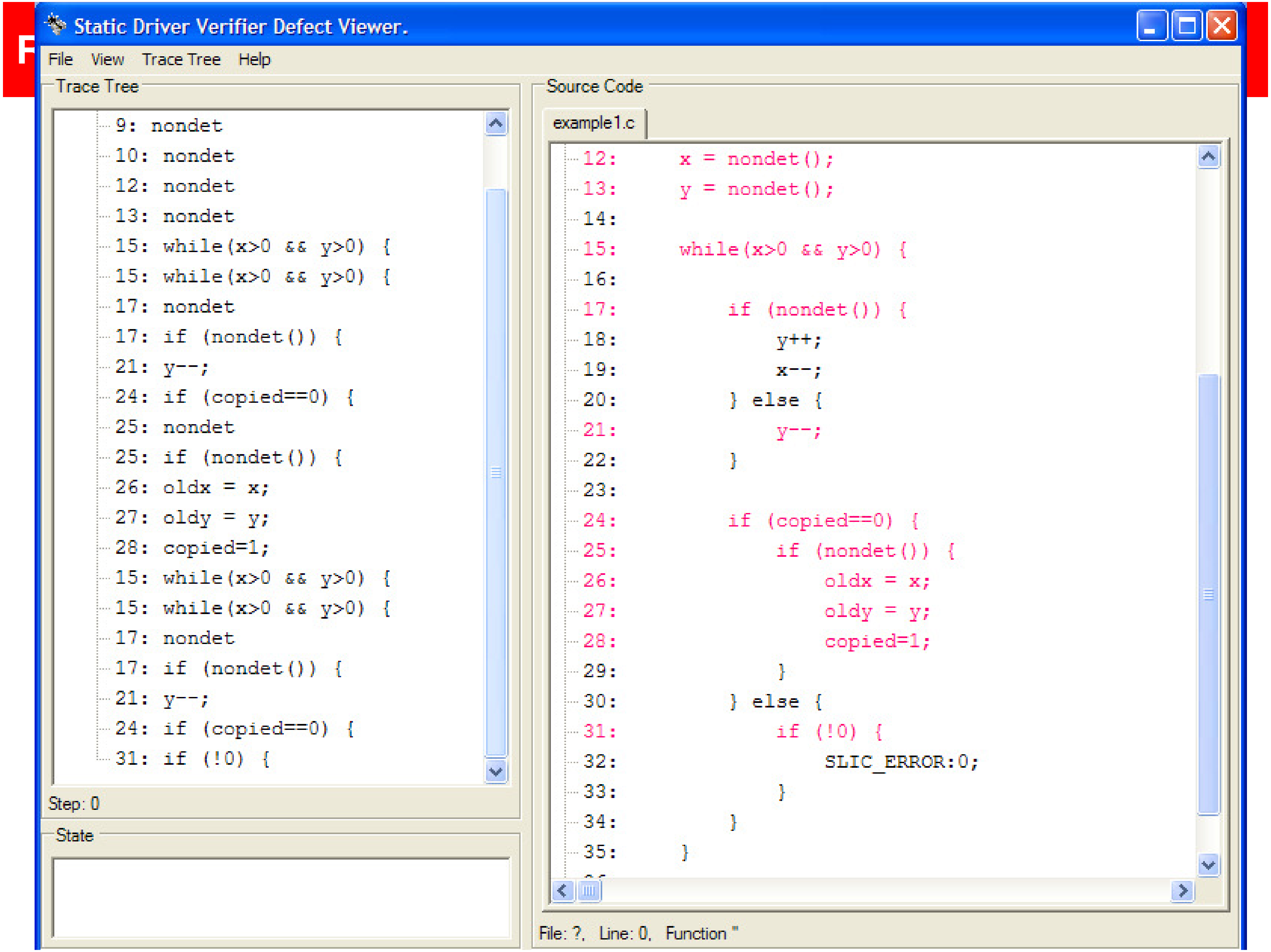
[1] Microsoft Research, Cambridge
[2] Max-Planck-Institut für Informatik, Saarbrücken

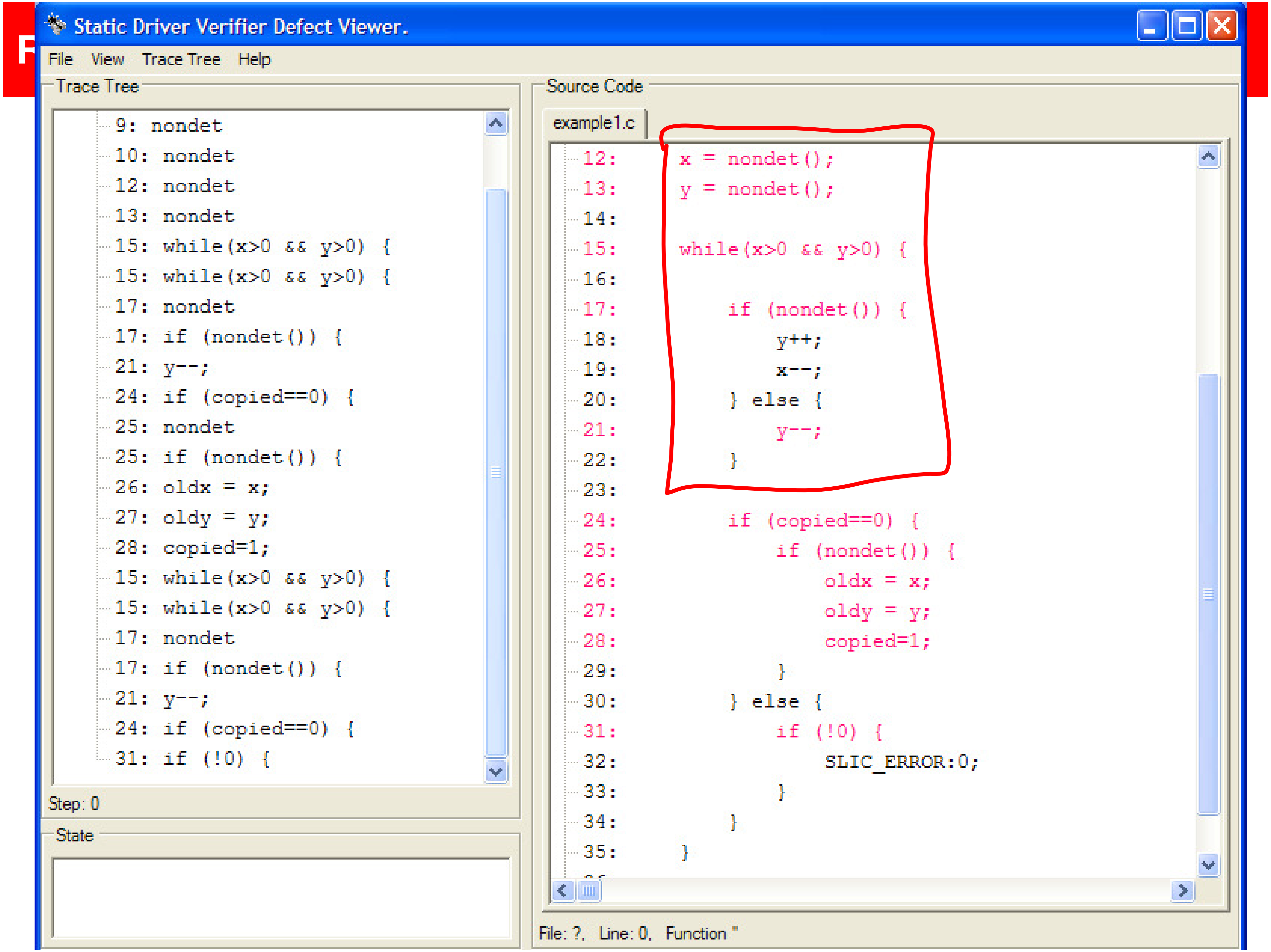
Abstract. Abstraction can often lead to spurious counterexamples. Counterexample-guided abstraction refinement is a method of strengthening abstractions based on the analysis of these spurious counterexamples. For invariance properties, a counterexample is a finite trace that violates the invariant; it is spurious if it is possible in the abstraction but not in the original system. When proving termination or other liveness properties of infinite-state systems, a useful notion of spurious counterexamples has remained an open problem. For this reason, no counterexample-guided abstraction refinement algorithm was known for termination. In this paper, we address this problem and present the first known automatic counterexample-guided abstraction refinement algorithm for termination proofs. We exploit recent results on transition invariants and transition predicate abstraction. We identify two reasons for spuriousness: abstractions that are too coarse, and candidate transition invariants that are too strong. Our counterexample-guided abstraction refinement algorithm successively weakens candidate transition invariants and re-

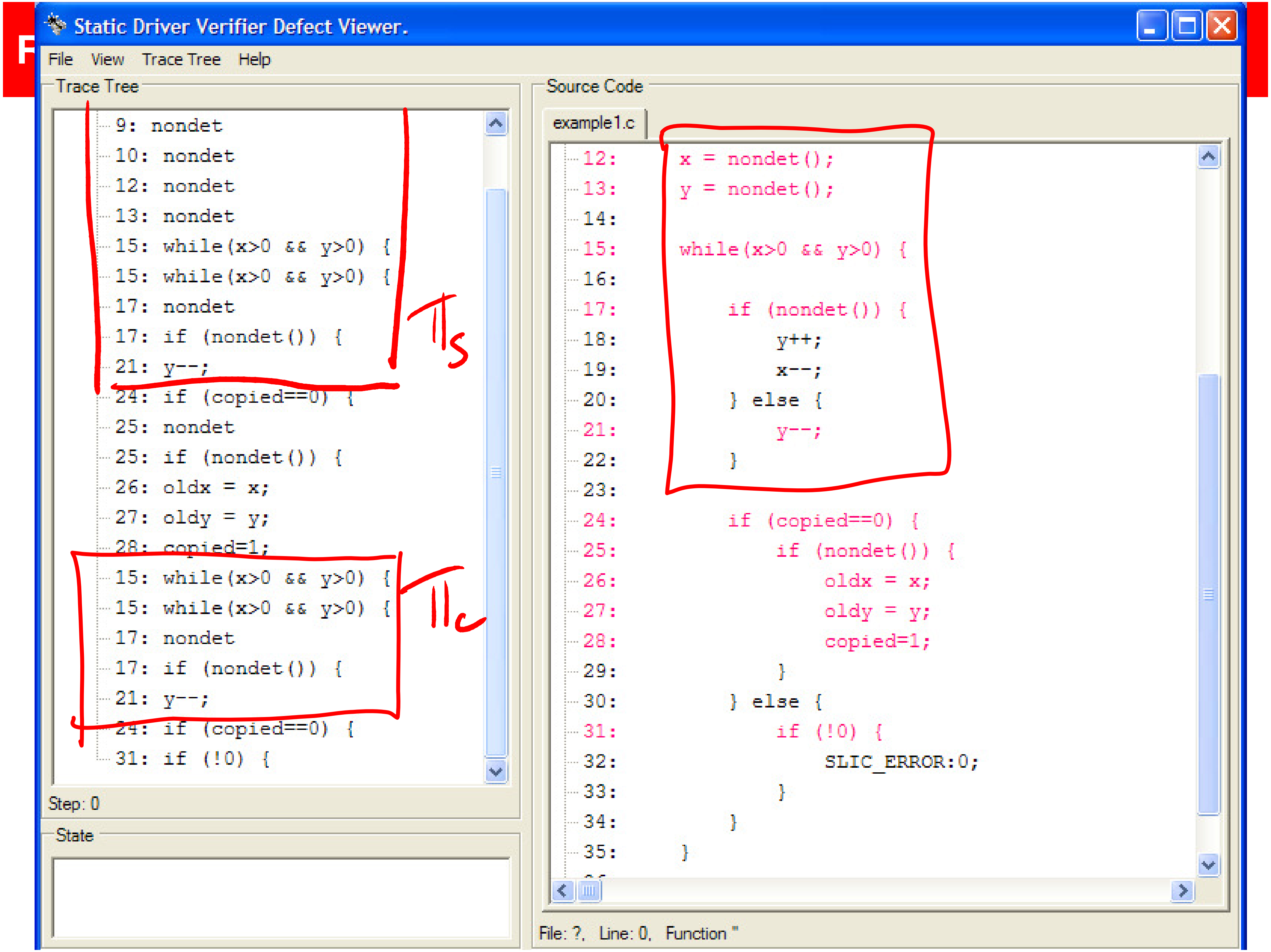
8.50 x 11.00 in

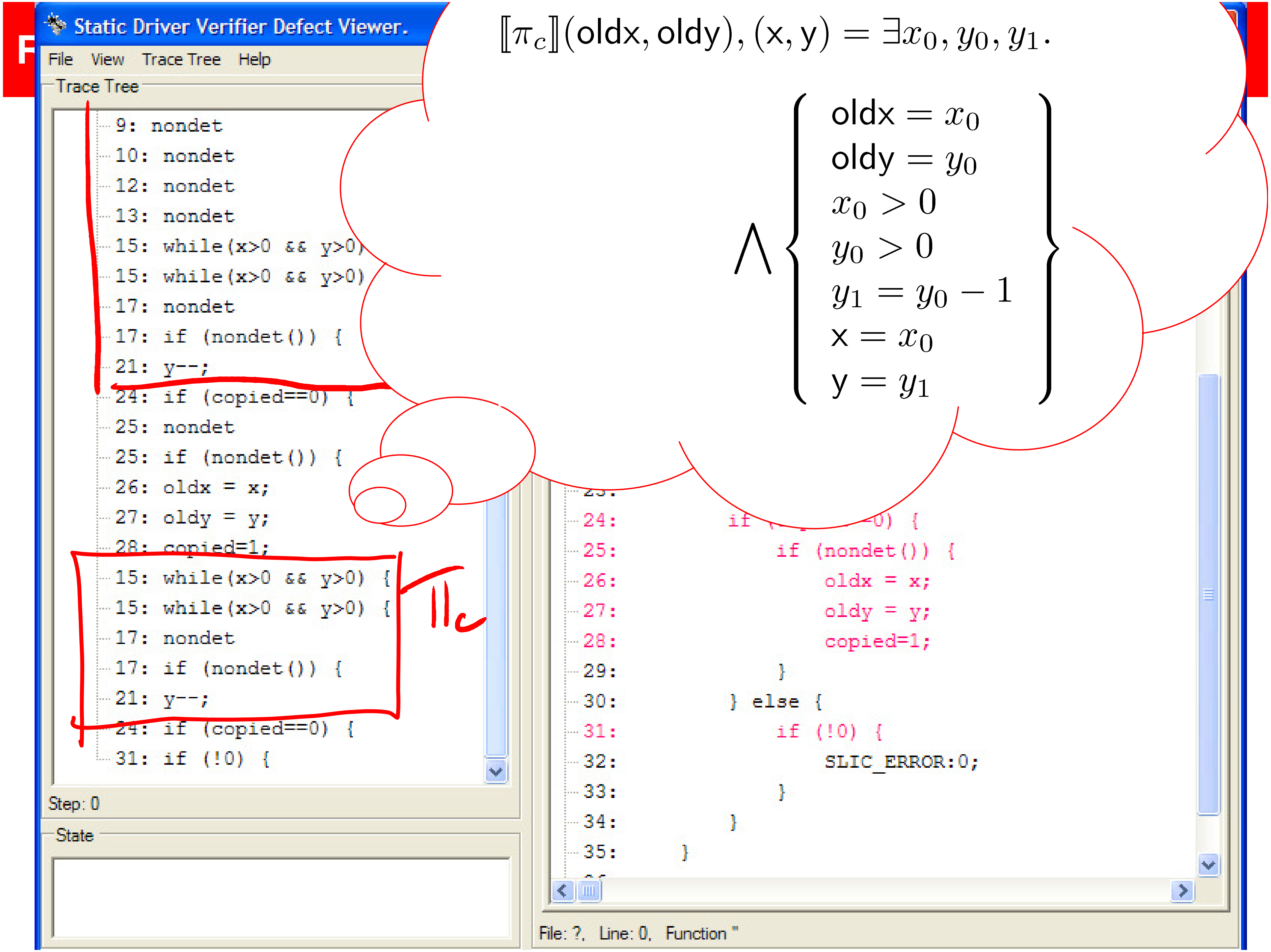

Static Driver Verifier Defect Viewer.
File
View
Trace Tree
Help
Trace Tree
9: nondet
10: nondet
12: nondet
13: nondet
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
25: nondet
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
31: if (!0) {
Step: 0
State
Source Code
example1.c
12: x = nondet();
13: y = nondet();
14:
15: while(x>0 && y>0) {
16:
17: if (nondet()) {
18: y++;
19: x--;
20: } else {
21: y--;
22: }
23:
24: if (copied==0) {
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
29: }
30: } else {
31: if (!0) {
32: SLIC_ERROR:0;
33: }
34: }
35: }
File: ?, Line: 0, Function ''

```
while(x>0 && y>0) {

    if (nondet()) {
        y++;
        x--;
    } else {
        y--;
    }
```


Static Driver Verifier Defect Viewer.
File View Trace Tree Help
Trace Tree
9: nondet
10: nondet
12: nondet
13: nondet
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
25: nondet
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
31: if (!0) {
Step: 0
State
Source Code
example1.c
12: x = nondet();
13: y = nondet();
14:
15: while(x>0 && y>0) {
16:
17: if (nondet()) {
18: y++;
19: x--;
20: } else {
21: y--;
22: }
23:
24: if (copied==0) {
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
29: }
30: } else {
31: if (!0) {
32: SLIC_ERROR:0;
33: }
34: }
35: }
File: ?, Line: 0, Function ''

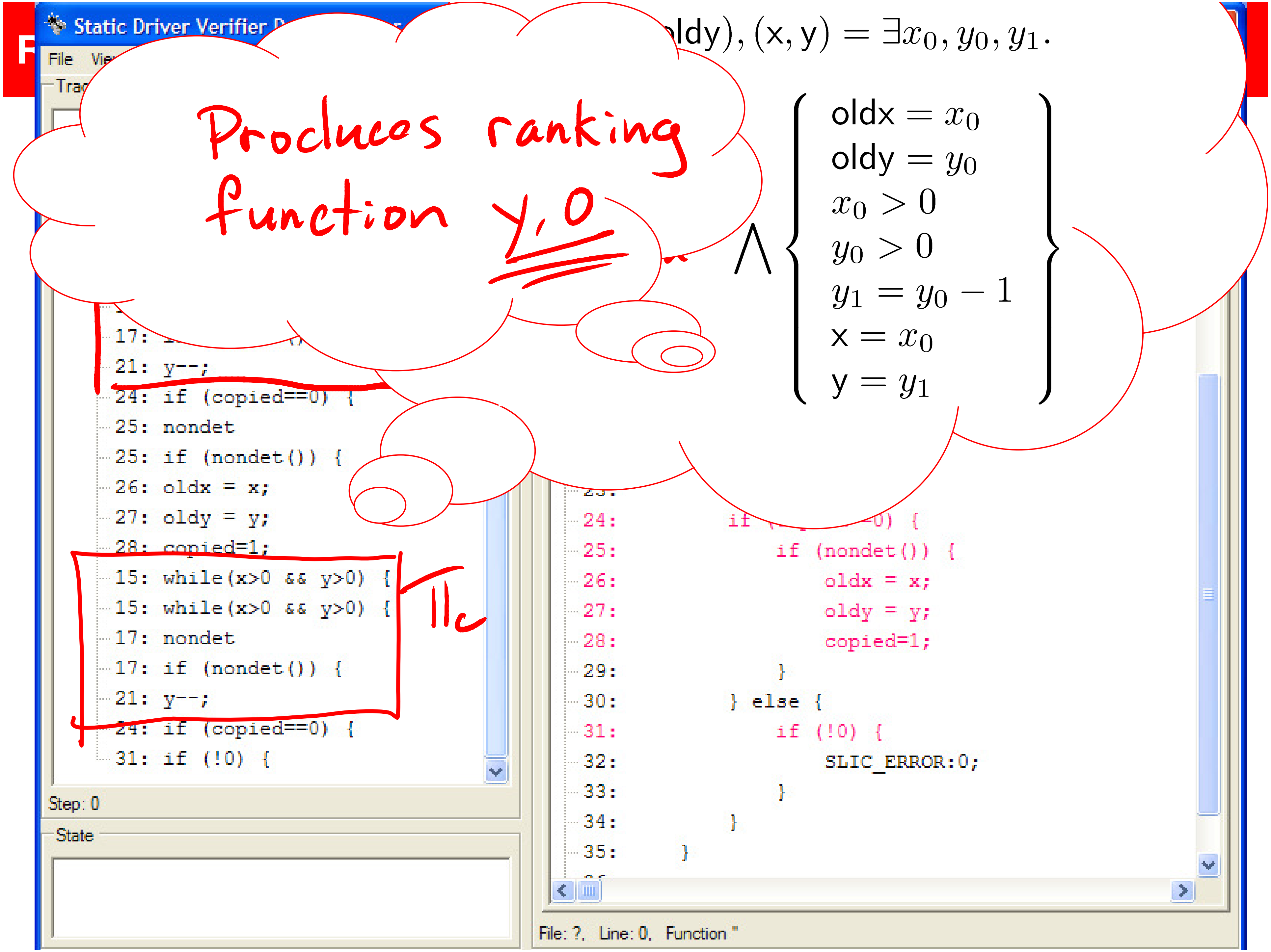

Static Driver Verifier Defect Viewer.
File View Trace Tree Help
Trace Tree
9: nondet
10: nondet
12: nondet
13: nondet
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
25: nondet
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
31: if (!0) {
Step: 0
State
Source Code
example1.c
12: x = nondet();
13: y = nondet();
14:
15: while(x>0 && y>0) {
16:
17: if (nondet()) {
18: y++;
19: x--;
20: } else {
21: y--;
22: }
23:
24: if (copied==0) {
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
29: }
30: } else {
31: if (!0) {
32: SLIC_ERROR:0;
33: }
34: }
35: }
File: ?, Line: 0, Function ''


Static Driver Verifier Defect Viewer.
File View Trace Tree Help
Trace Tree
9: nondet
10: nondet
12: nondet
13: nondet
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
25: nondet
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
15: while(x>0 && y>0) {
15: while(x>0 && y>0) {
17: nondet
17: if (nondet()) {
21: y--;
24: if (copied==0) {
31: if (!0) {
Π_s
Π_c
Step: 0
State
Source Code
example1.c
12: x = nondet();
13: y = nondet();
14:
15: while(x>0 && y>0) {
16:
17: if (nondet()) {
18: y++;
19: x--;
20: } else {
21: y--;
22: }
23:
24: if (copied==0) {
25: if (nondet()) {
26: oldx = x;
27: oldy = y;
28: copied=1;
29: }
30: } else {
31: if (!0) {
32: SLIC_ERROR:0;
33: }
34: }
35: }
File: ?, Line: 0, Function ''



$$[\![\pi_c]\!](\mathsf{oldx},\mathsf{oldy}),(\mathsf{x},\mathsf{y}) = \exists x_0, y_0, y_1.$$

$$\bigwedge \left\{ \begin{array}{l} \mathsf{oldx} = x_0 \\ \mathsf{oldy} = y_0 \\ x_0 > 0 \\ y_0 > 0 \\ y_1 = y_0 - 1 \\ \mathsf{x} = x_0 \\ \mathsf{y} = y_1 \end{array} \right\}$$



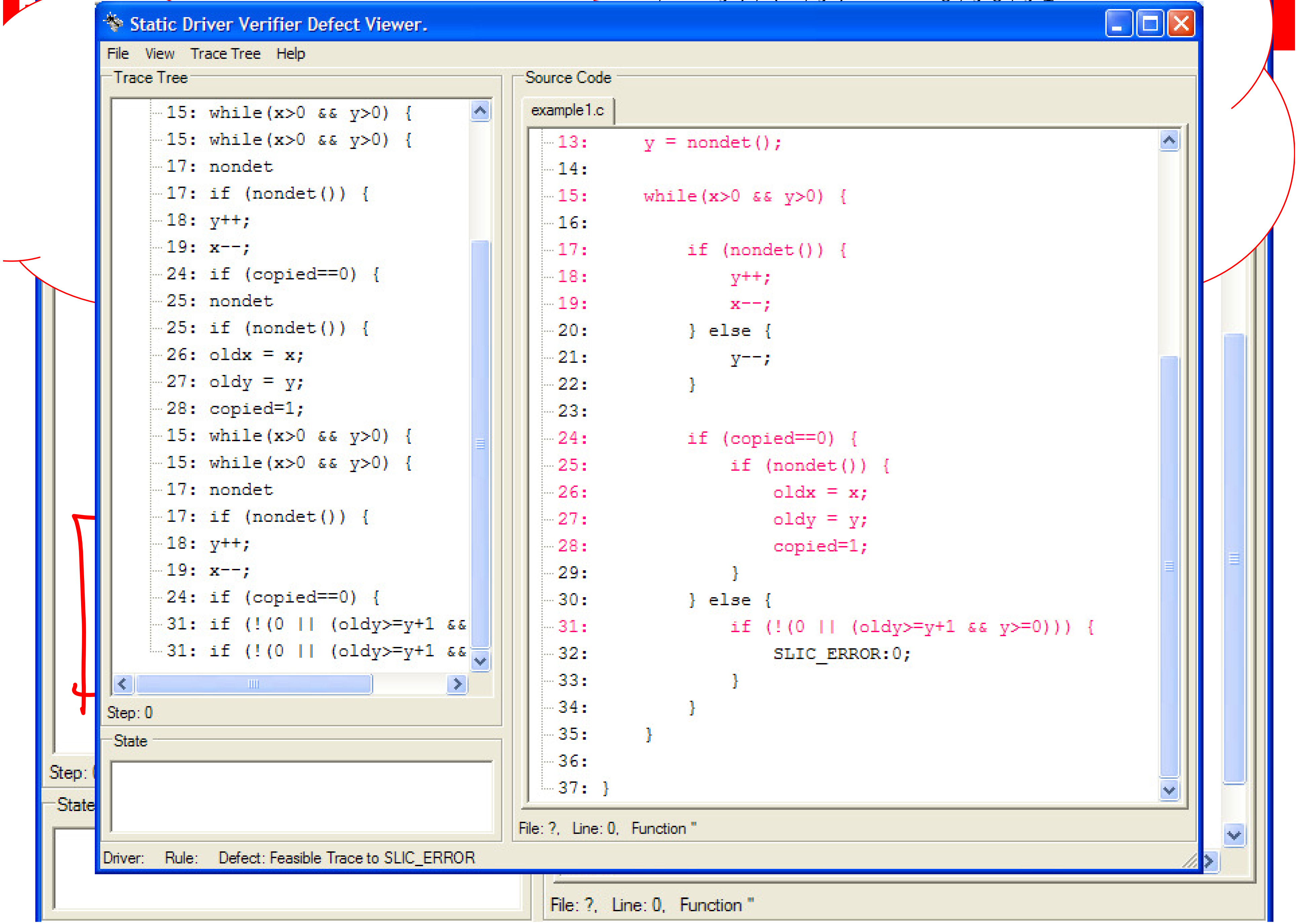
$$\ldots\text{ldy}), (\mathsf{x}, \mathsf{y}) = \exists x_0, y_0, y_1 .$$

$$\bigwedge \left\{ \begin{array}{l} \mathsf{oldx} = x_0 \\ \mathsf{oldy} = y_0 \\ x_0 > 0 \\ y_0 > 0 \\ y_1 = y_0 - 1 \\ \mathsf{x} = x_0 \\ \mathsf{y} = y_1 \end{array} \right\}$$

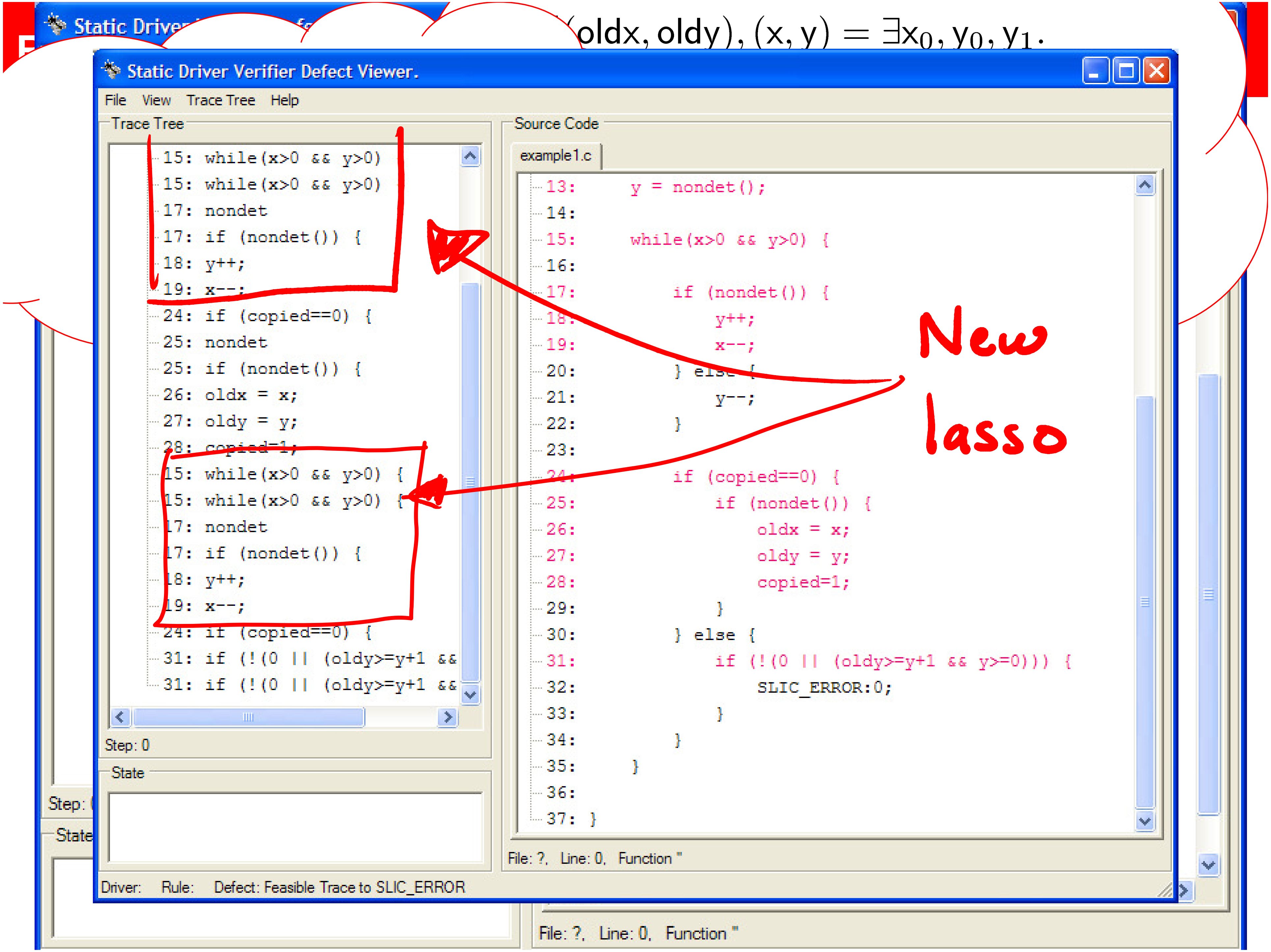
Produces ranking function y, 0

$\|_c$

$(oldx, oldy), (x, y) = \exists x_0, y_0, y_1.$



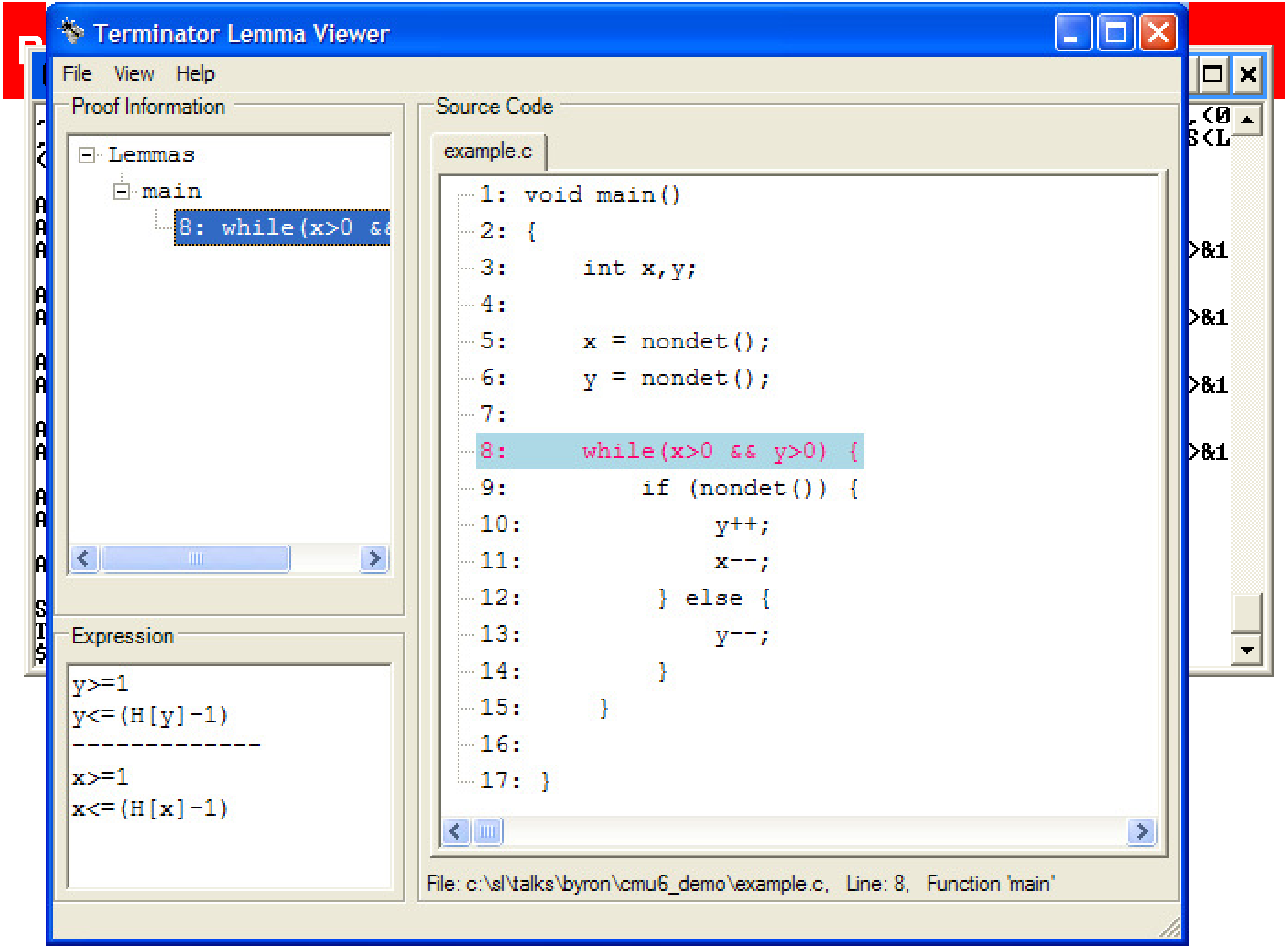
$(\text{oldx}, \text{oldy}), (x, y) = \exists x_0, y_0, y_1.$

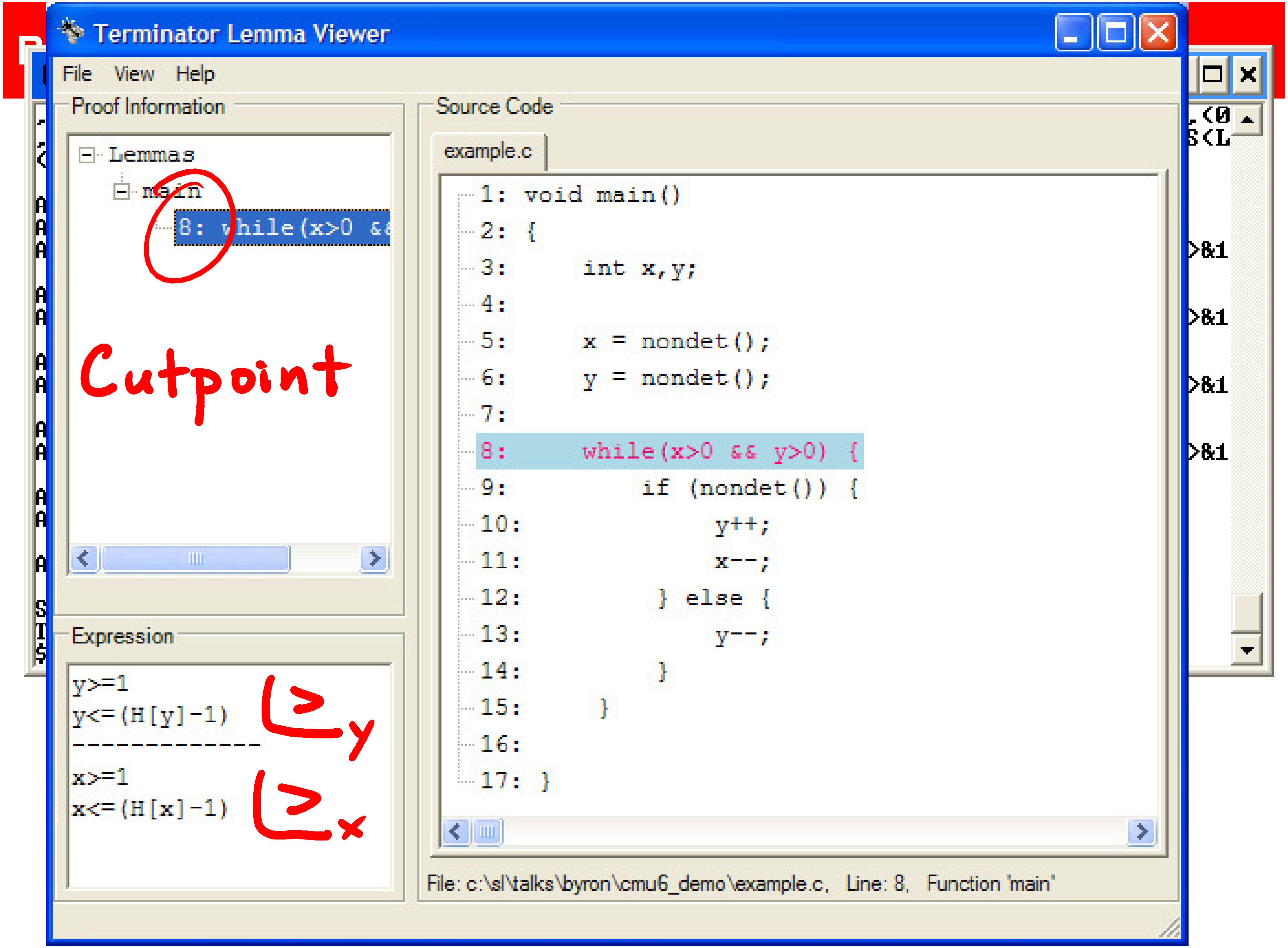


Refinement

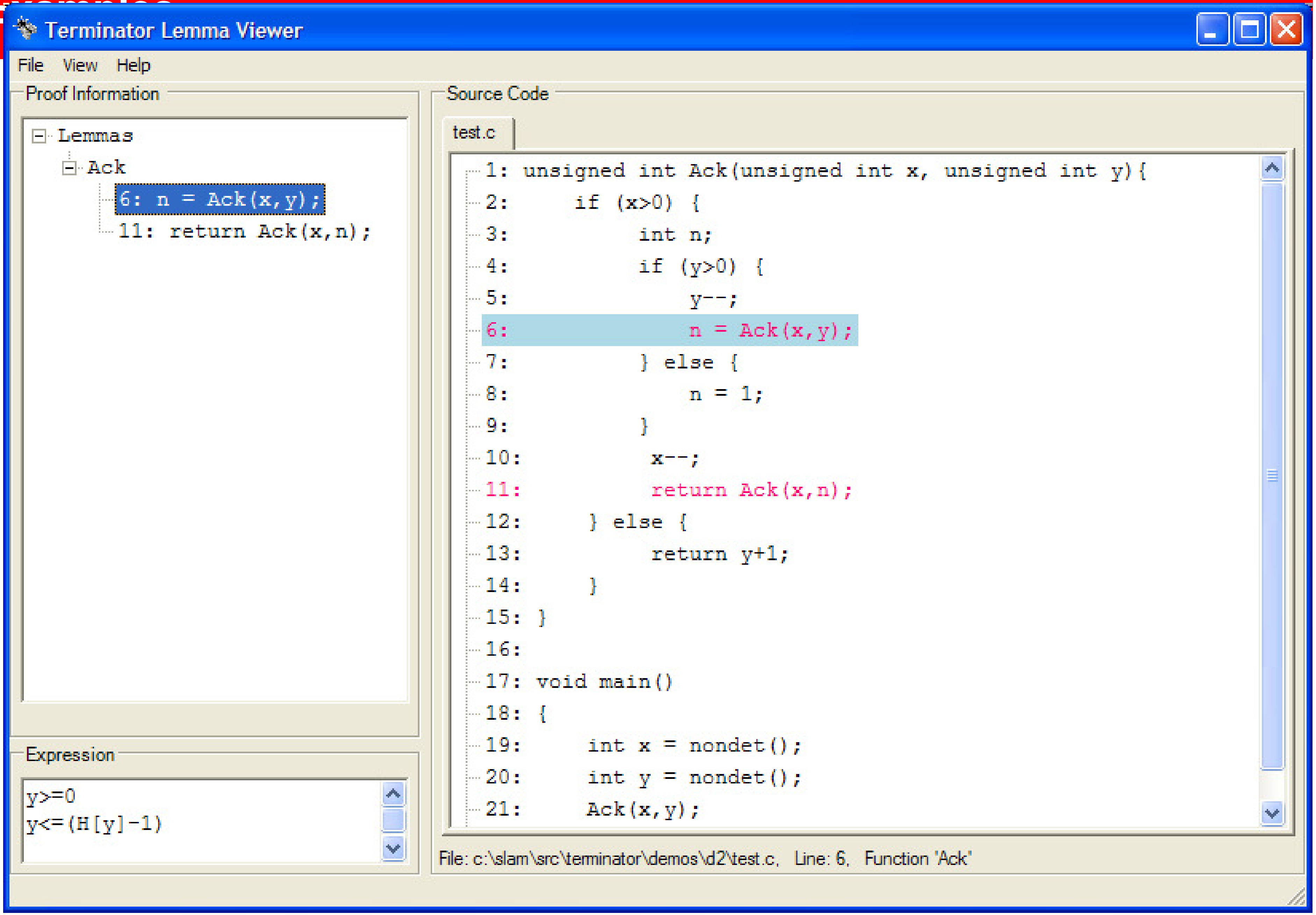
Cygwin

```
, acu 1, afu 1: G(t__y_1_1_2,0,0,1)==S(L(y,1,1,2),(0,29,2,0)), ((S(L(y,1,1,2),(0
,29,2,0))-1)-1)>0, L(y,1,1,2)==((S(L(y,1,1,2),(0,29,2,0))-1)-1), L(y,1,1,2)==S(L
(y,1,1,2),(0,29,2,0)), L(y,1,1,2)==(S(L(y,1,1,2),(0,29,2,0))-1)
 SLAM: iter-begin 9 ~~~~~~~~~~~~~
AR: no new preds at iter 9
AR: calling constrain at iter 9
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp  >bebop.out 2>&1

AR: bebop generated a path
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp  >bebop.out 2>&1

AR: bebop generated a path
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp  >bebop.out 2>&1

AR: bebop generated a path
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp  >bebop.out 2>&1

AR: bebop is raising completed
AR: watch_startup_end Completed
 SLAM: watch-startup-end 0, iter 9 ~~~~~~~~~~~
AR: saving preds
 Program [ example.c ] passed property
Saving termination lemmas to "witness.tt"
Time: 218.408
$
```


Terminator Lemma Viewer
File
View
Help
Proof Information
Lemmas
main
8: while(x>0 &&
Source Code
example.c
1: void main()
2: {
3: int x,y;
4:
5: x = nondet();
6: y = nondet();
7:
8: while(x>0 && y>0) {
9: if (nondet()) {
10: y++;
11: x--;
12: } else {
13: y--;
14: }
15: }
16:
17: }
Expression
y>=1
y<=(H[y]-1)
----
x>=1
x<=(H[x]-1)
File: c:\sl\talks\byron\cmu6_demo\example.c, Line: 8, Function 'main'

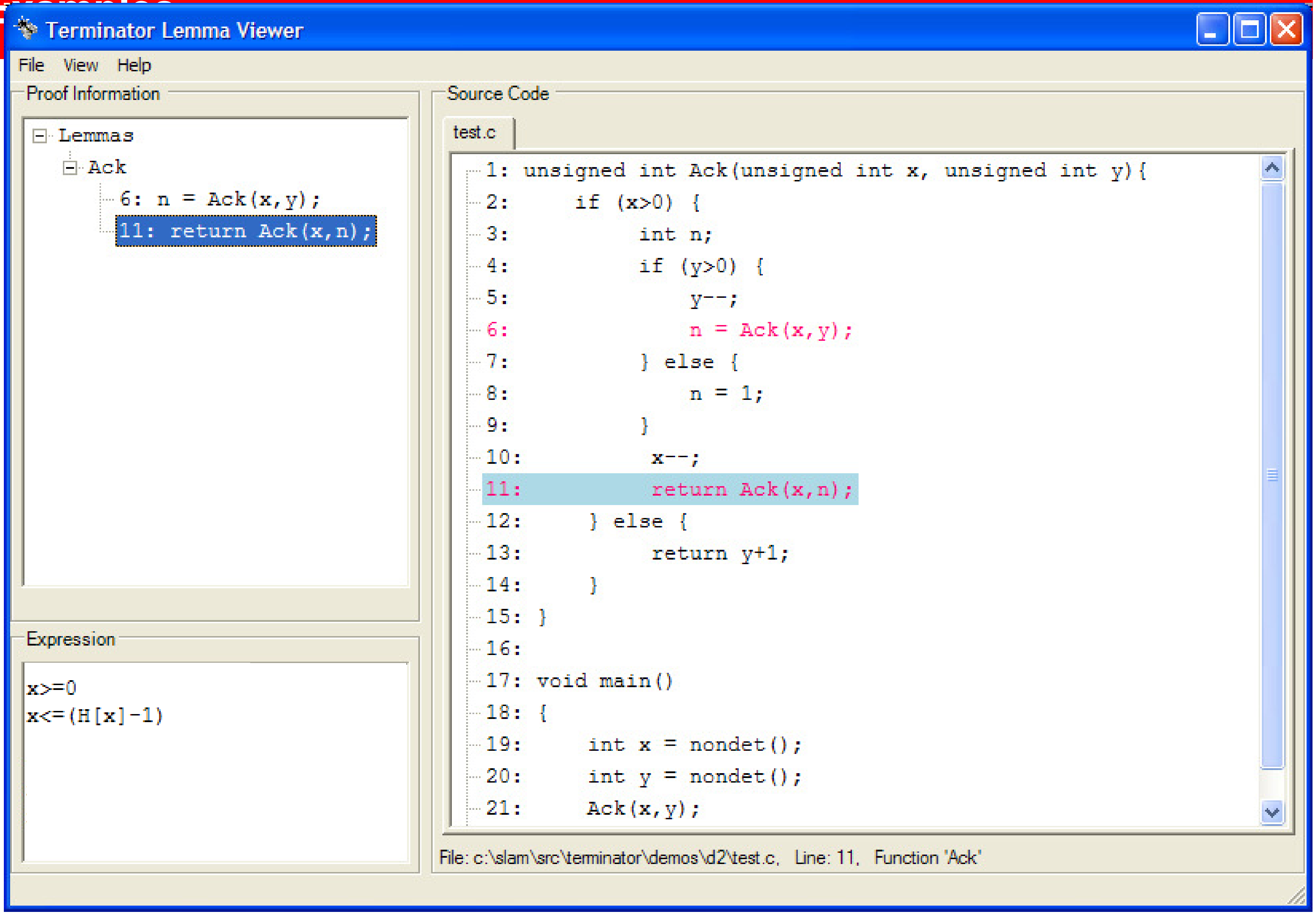

Terminator Lemma Viewer
File View Help
Proof Information
Lemmas
main
8: while(x>0 &&
Cutpoint
Source Code
example.c
1: void main()
2: {
3: int x,y;
4:
5: x = nondet();
6: y = nondet();
7:
8: while(x>0 && y>0) {
9: if (nondet()) {
10: y++;
11: x--;
12: } else {
13: y--;
14: }
15: }
16:
17: }
Expression
y>=1
y<=(H[y]-1)
----
x>=1
x<=(H[x]-1)
≥y
≥x
File: c:\sl\talks\byron\cmu6_demo\example.c, Line: 8, Function 'main'

Examples



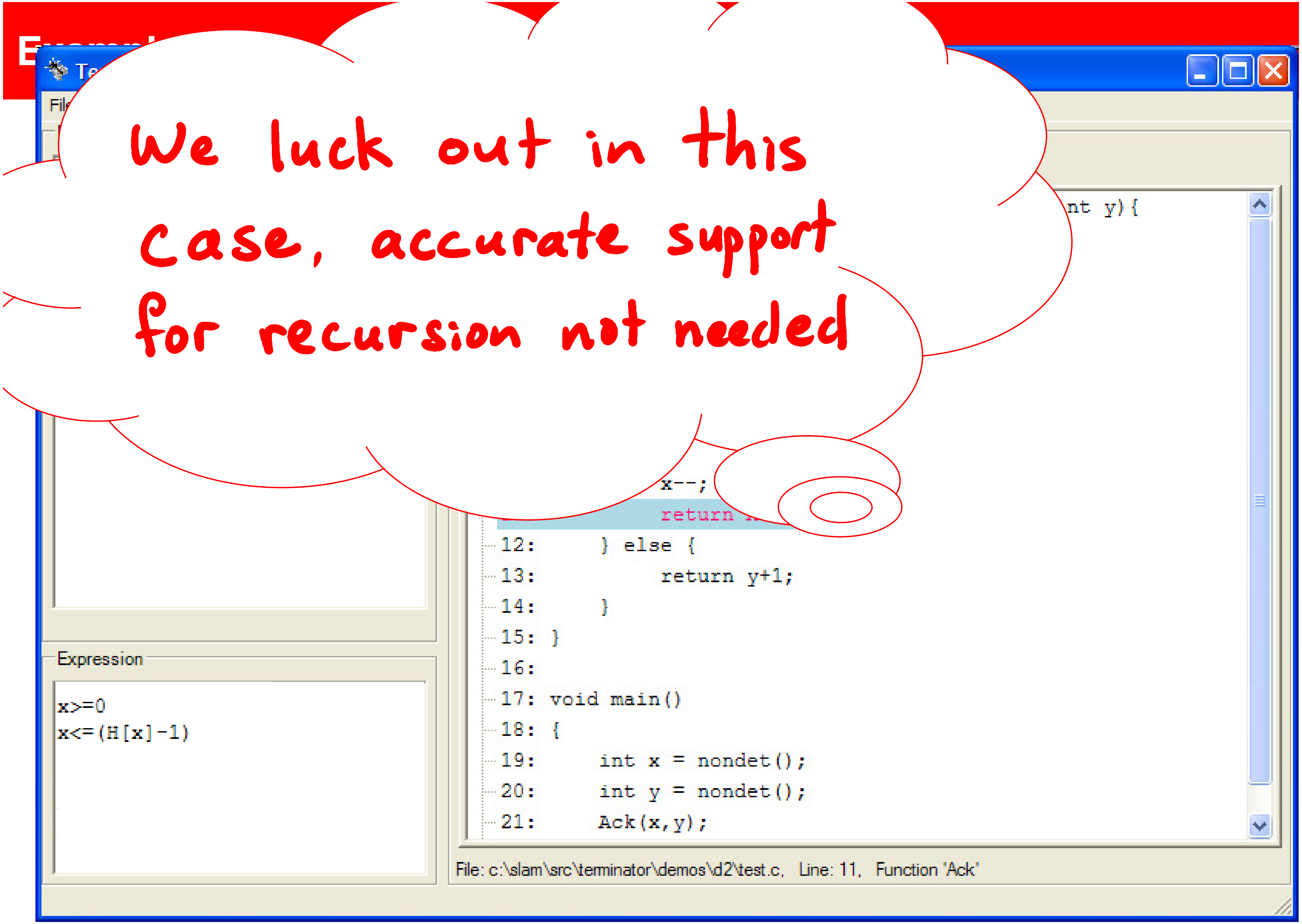
Terminator Lemma Viewer
File
View
Help
Proof Information
Lemmas
Ack
6: n = Ack(x,y);
11: return Ack(x,n);
Expression
y>=0
y<=(H[y]-1)
Source Code
test.c
1: unsigned int Ack(unsigned int x, unsigned int y){
2: if (x>0) {
3: int n;
4: if (y>0) {
5: y--;
6: n = Ack(x,y);
7: } else {
8: n = 1;
9: }
10: x--;
11: return Ack(x,n);
12: } else {
13: return y+1;
14: }
15: }
16:
17: void main()
18: {
19: int x = nondet();
20: int y = nondet();
21: Ack(x,y);
File: c:\slam\src\terminator\demos\d2\test.c, Line: 6, Function 'Ack'

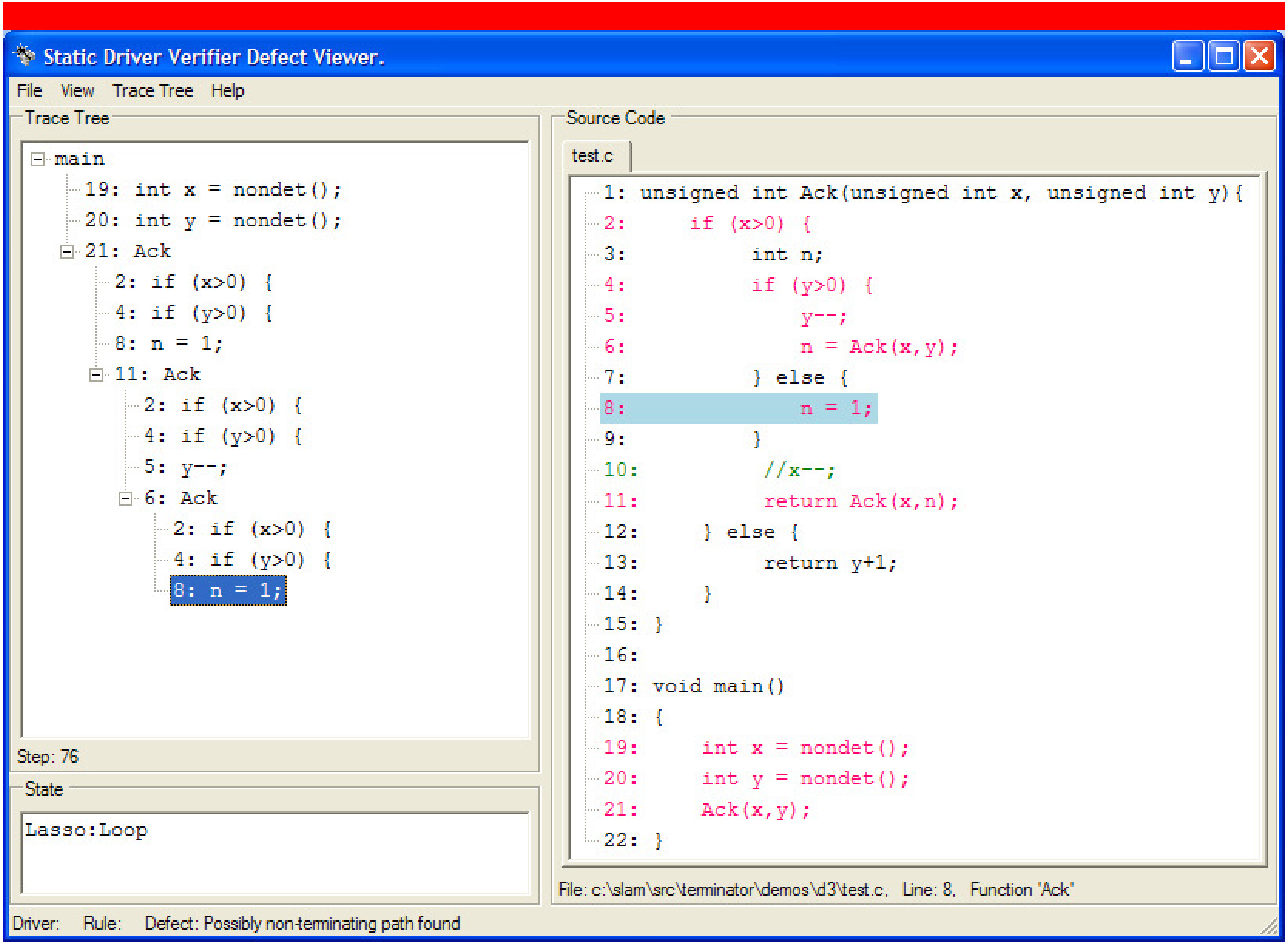
Examples



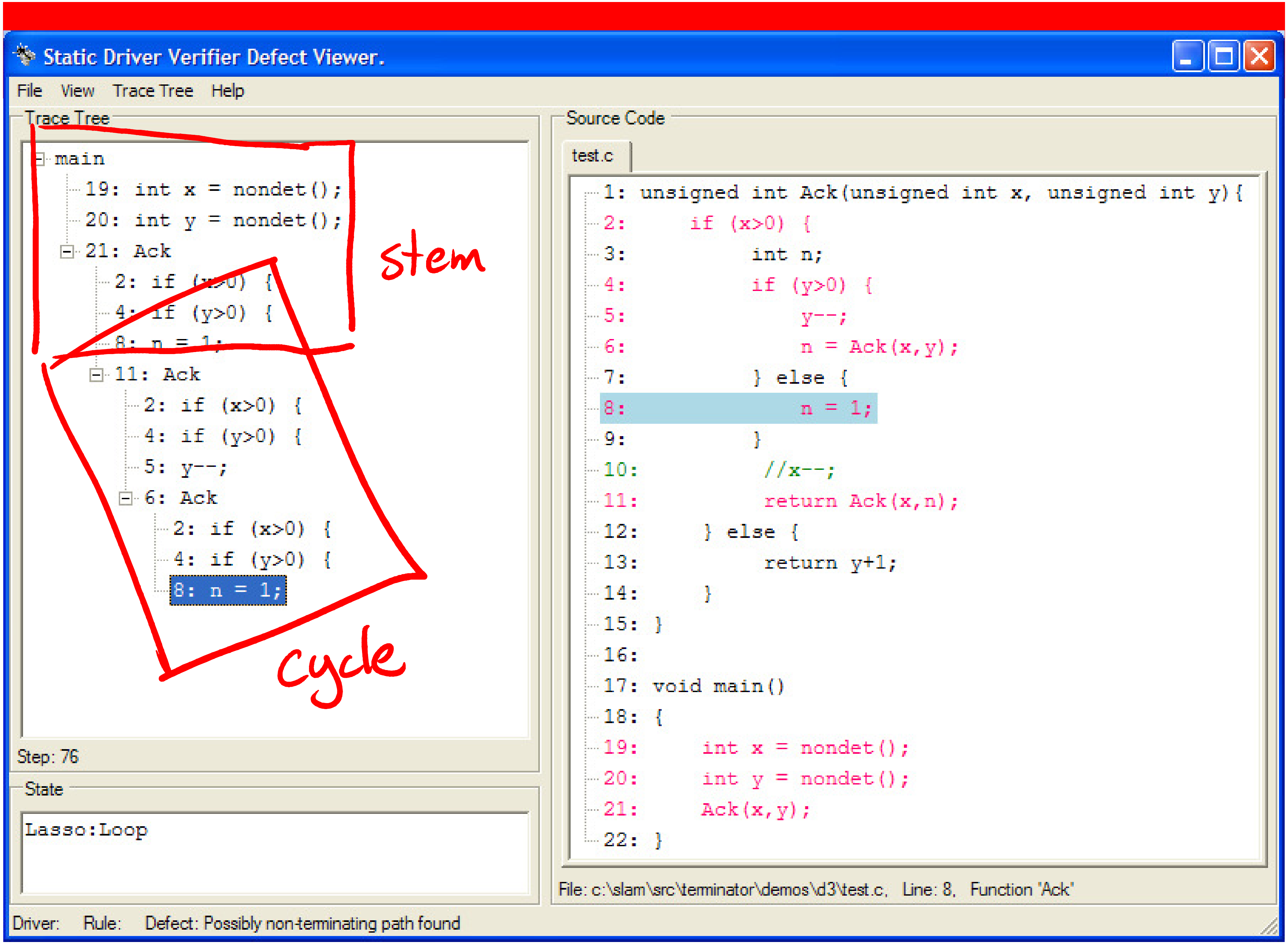
Terminator Lemma Viewer
File View Help
Proof Information
Lemmas
Ack
6: n = Ack(x,y);
11: return Ack(x,n);
Expression
x>=0
x<=(H[x]-1)
Source Code
test.c
1: unsigned int Ack(unsigned int x, unsigned int y){
2: if (x>0) {
3: int n;
4: if (y>0) {
5: y--;
6: n = Ack(x,y);
7: } else {
8: n = 1;
9: }
10: x--;
11: return Ack(x,n);
12: } else {
13: return y+1;
14: }
15: }
16:
17: void main()
18: {
19: int x = nondet();
20: int y = nondet();
21: Ack(x,y);
File: c:\slam\src\terminator\demos\d2\test.c, Line: 11, Function 'Ack'

We luck out in this case, accurate support for recursion not needed

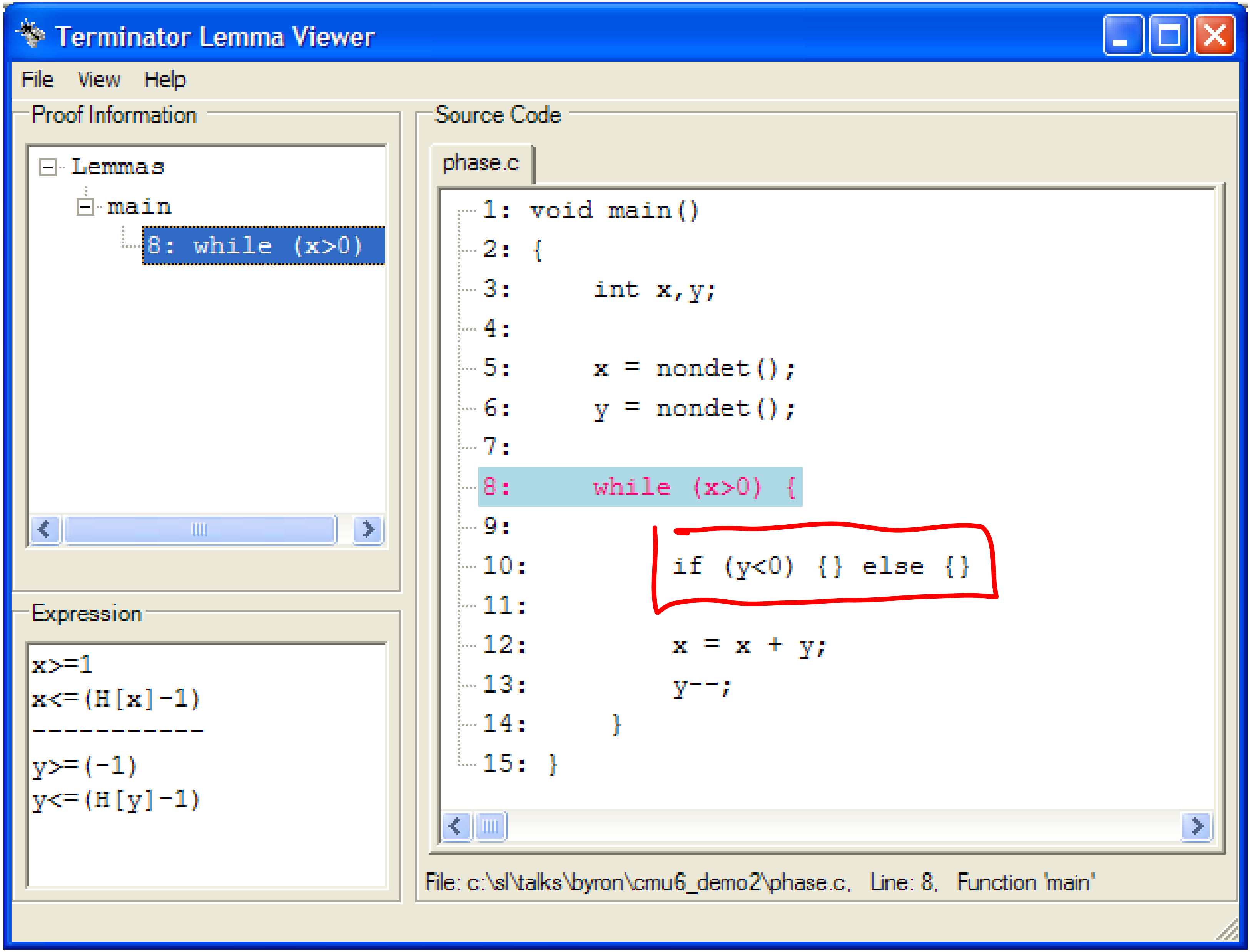




Static Driver Verifier Defect Viewer.
File
View
Trace Tree
Help
Trace Tree
main
19: int x = nondet();
20: int y = nondet();
21: Ack
2: if (x>0) {
4: if (y>0) {
8: n = 1;
11: Ack
2: if (x>0) {
4: if (y>0) {
5: y--;
6: Ack
2: if (x>0) {
4: if (y>0) {
8: n = 1;
Step: 76
State
Lasso:Loop
Source Code
test.c
1: unsigned int Ack(unsigned int x, unsigned int y){
2: if (x>0) {
3: int n;
4: if (y>0) {
5: y--;
6: n = Ack(x,y);
7: } else {
8: n = 1;
9: }
10: //x--;
11: return Ack(x,n);
12: } else {
13: return y+1;
14: }
15: }
16:
17: void main()
18: {
19: int x = nondet();
20: int y = nondet();
21: Ack(x,y);
22: }
File: c:\slam\src\terminator\demos\d3\test.c, Line: 8, Function 'Ack'
Driver:
Rule:
Defect: Possibly non-terminating path found

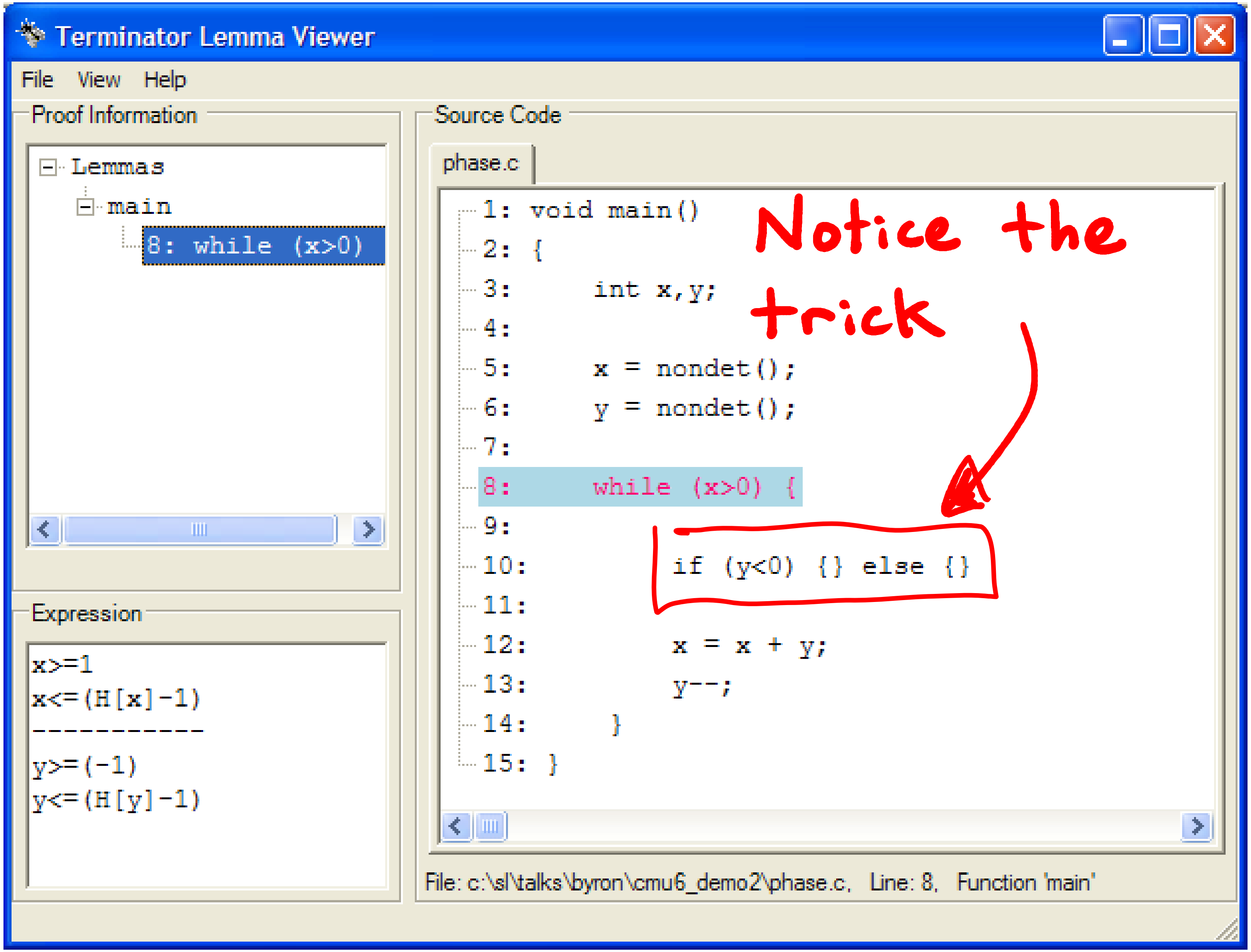

Static Driver Verifier Defect Viewer.
File View Trace Tree Help
Trace Tree
main
19: int x = nondet();
20: int y = nondet();
21: Ack
2: if (x>0) {
4: if (y>0) {
8: n = 1;
11: Ack
2: if (x>0) {
4: if (y>0) {
5: y--;
6: Ack
2: if (x>0) {
4: if (y>0) {
8: n = 1;
stem
cycle
Step: 76
State
Lasso:Loop
Source Code
test.c
1: unsigned int Ack(unsigned int x, unsigned int y){
2: if (x>0) {
3: int n;
4: if (y>0) {
5: y--;
6: n = Ack(x,y);
7: } else {
8: n = 1;
9: }
10: //x--;
11: return Ack(x,n);
12: } else {
13: return y+1;
14: }
15: }
16:
17: void main()
18: {
19: int x = nondet();
20: int y = nondet();
21: Ack(x,y);
22: }
File: c:\slam\src\terminator\demos\d3\test.c, Line: 8, Function 'Ack'
Driver: Rule: Defect: Possibly non-terminating path found

Examples



Terminator Lemma Viewer
File View Help
Proof Information
Lemmas
main
8: while (x>0)
Expression
x>=1
x<=(H[x]-1)
----
y>=(-1)
y<=(H[y]-1)
Source Code
phase.c
1: void main()
2: {
3: int x,y;
4:
5: x = nondet();
6: y = nondet();
7:
8: while (x>0) {
9:
10: if (y<0) {} else {}
11:
12: x = x + y;
13: y--;
14: }
15: }
File: c:\sl\talks\byron\cmu6_demo2\phase.c, Line: 8, Function 'main'

Examples



Terminator Lemma Viewer
File View Help
Proof Information
Lemmas
main
8: while (x>0)
Expression
x>=1
x<=(H[x]-1)
----
y>=(-1)
y<=(H[y]-1)
Source Code
phase.c
1: void main()
2: {
3: int x,y;
4:
5: x = nondet();
6: y = nondet();
7:
8: while (x>0) {
9:
10: if (y<0) {} else {}
11:
12: x = x + y;
13: y--;
14: }
15: }
File: c:\sl\talks\byron\cmu6_demo2\phase.c, Line: 8, Function 'main'
Notice the trick

Overview

- → Notes on a representation for programs
- → Checking termination arguments
- → Refining termination arguments
- → Induction
- → Termination analysis

Induction

$T := \emptyset$

while $\textsc{Reachable}(\boxplus(R, \ell, T), \ell_{err})$ **do**

  let π_s, π_c = lasso in $\boxplus(R, \ell, T)$ from 0 to ℓ, and ℓ to ℓ_{err}

  let $\rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!](\top)))$

  if $\textsc{Synthesis}([\![\pi_c]\!] \cap \rho \times \rho)$ returns ranking function f **then**

    $T := T \cup \trianglerighteq_f$

  else

    report “potential counterexample found: π_s,π_c

  fi

od

report “termination proved with argument T

Induction

$T := \emptyset$

while $\cancel{\text{REACHABLE}(\boxplus(R, \ell, T), \ell_{err})}$ **do**

  let π_s, π_c = lasso in $\boxplus(R, \ell, T)$ from 0 to ℓ, and ℓ to ℓ_{err}

  let $\rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!](\top)))$

  if $\text{SYNTHESIS}([\![\pi_c]\!] \cap \rho \times \rho)$ returns ranking function f **then**

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c

  fi

od

report "termination proved with argument T

Induction

Induction check

$T := \emptyset$

while $\cancel{\text{REACHABLE}(\boxplus(R, \ell, T), \ell_{err})}$ **do**

  let π_s, π_c = lasso in $\boxplus(R, \ell, T)$ from 0 to ℓ, and ℓ to ℓ_{err}

  let $\rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!](\top)))$

  if SYNTHESIS($[\![\pi_c]\!] \cap \rho \times \rho$) returns ranking function f **then**

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c

  fi

od

report "termination proved with argument T

Induction

Induction check

$T := \emptyset$

while ~~$\textsc{Reachable}(\boxplus(R, \ell, T), \ell_{err})$~~ **do**

  let $\pi_s, \pi_c =$ ~~lasso in $\boxplus(R, \ell, T)$ from 0 to ℓ, and~~ ℓ to ℓ_{err}

  let $\rho = \alpha(\llbracket \pi_c \rrbracket^* (\llbracket \pi_s \rrbracket (\top)))$

  if $\textsc{Synthesis}(\llbracket \pi_c \rrbracket \cap \rho \times \rho)$ returns ranking function f **then**

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c

  fi

od

report "termination proved with argument T

Induction

Induction check

$T := \emptyset$

while ~~$\textsc{Reachable}(\boxplus(R, \ell, T), \ell_{err})$~~ **do**

  let $\cancel{\pi_s}, \pi_c =$ ~~lasso in $\boxplus(R, \ell, T)$ from 0 to ℓ, and~~ ℓ to ℓ_{err}

  ~~**let** $\rho = \alpha(\llbracket \pi_c \rrbracket^*(\llbracket \pi_s \rrbracket(\top)))$~~

  if $\textsc{Synthesis}(\llbracket \pi_c \rrbracket \cap \rho \times \rho)$ returns ranking function f **then**

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c

  fi

od

report "termination proved with argument T

Induction

Induction check

$T := \emptyset$

while $\cancel{\textsc{Reachable}(\boxplus(R, \ell, T), \ell_{err}})$ **do**

  let $\cancel{\pi_s}, \pi_c = \cancel{\text{lasso in } \boxplus(R, \ell, T) \text{ from } 0 \text{ to } \ell, \text{ and}}\ \ell$ to ℓ_{err}

  $\cancel{\textbf{let } \rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!](\top)))}$

  if $\textsc{Synthesis}([\![\pi_c]\!] \cap \cancel{\rho \times} \rho)$ returns ranking function f **then**

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c

  fi

od

report "termination proved with argument T

Induction

Induction check

$T := \emptyset$

while $\cancel{\textsc{Reachable}(\boxplus(R, \ell, T), \ell_{err})}$ **do**

  let $\cancel{\pi_s}, \pi_c \cancel{= \text{lasso in } \boxplus(R, \ell, T) \text{ from } 0 \text{ to } \ell, \text{ and}}\ \ell$ to ℓ_{err}

  $\cancel{\textbf{let } \rho = \alpha(\llbracket \pi_c \rrbracket^*(\llbracket \pi_s \rrbracket(\top)))}$

  if $\textsc{Synthesis}(\llbracket \pi_c \rrbracket \cap \cancel{\rho \times \rho})$ returns ranking function f **then**

    $T := T \cup \trianglerighteq_f \wedge \trianglerighteq_\rho$ for all functions ρ which are preserved

  else

    report “potential counterexample found: $\cancel{\pi_s}, \pi_c$

  fi

od

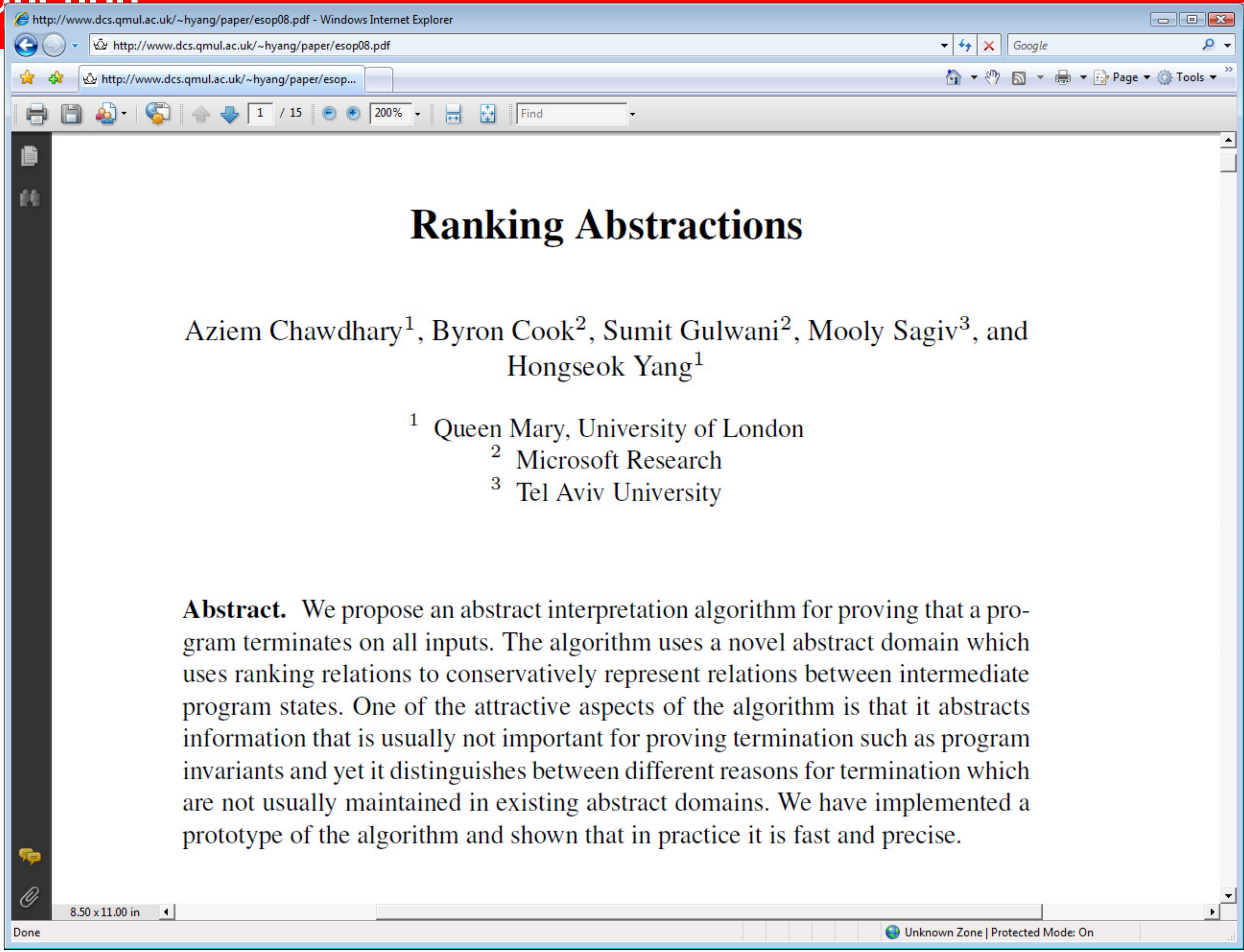
report “termination proved with argument T

Induction check

$T := \emptyset$
while $\cancel{\text{REACHABLE}(\boxplus(R, \ell, T), \ell_{err})}$ **do**
  let $\cancel{\pi_s}, \pi_c = \cancel{\text{lasso in } \boxplus(R, \ell, T) \text{ from } 0 \text{ to } \ell, \text{ and}}\ \ell$ to ℓ_{err}
  $\cancel{\textbf{let } \rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!](\top)))}$
  if SYNTHESIS($[\![\pi_c]\!] \cap \rho \times \rho$) returns ranking function f **then**
    $T := T \cup \trianglerighteq_f \cap [\trianglerighteq_f]$ for all functions f which are preserved
  else
    report "potential counterexample found: $\cancel{\pi_s}, \pi_c$
  fi
od
report "termination proved with argument T

Could use the invariant at location ℓ

Induction



Ranking Abstractions

Aziem Chawdhary[1], Byron Cook[2], Sumit Gulwani[2], Mooly Sagiv[3], and Hongseok Yang[1]

[1] Queen Mary, University of London
[2] Microsoft Research
[3] Tel Aviv University

Abstract. We propose an abstract interpretation algorithm for proving that a program terminates on all inputs. The algorithm uses a novel abstract domain which uses ranking relations to conservatively represent relations between intermediate program states. One of the attractive aspects of the algorithm is that it abstracts information that is usually not important for proving termination such as program invariants and yet it distinguishes between different reasons for termination which are not usually maintained in existing abstract domains. We have implemented a prototype of the algorithm and shown that in practice it is fast and precise.

The bad news

Bad news: lassos don't always lead to progress ....

The bad news

Bad news: lassos don't always lead to progress ....

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$

The bad news

Bad news: lassos don't always lead to progress ....

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n} \cup \trianglerighteq_{f_{n+1}} \cup \ldots ?$$

The bad news

Bad news: lassos don't always lead to progress ....

```
while(x>0) {
    y = x;
    while(y>0) {
        y = y - 1;
    }
    x = x + 1;
}
```

The bad news

Bad news: lassos don't always lead to progress ....

```
while(x>0) {
    y = x;
    while(y>0) {
        y = y - 1;
    }
    x = x + 1;
}
```

The bad news

Bad news: lassos don't always lead to progress ....

```
1    while(x>0) {
2        y = x;
3        while(y>0) {
4            y = y - 1;
5        }
6        x = x + 1;
7    }
```

1
2
3
4
3
4
3
6

The bad news

Bad news: lassos don't always lead to progress ....

```
1    while(x>0) {
2        y = x;
3        while(y>0) {
4            y = y - 1;
5        }
6        x = x + 1;
7    }
```

$$
\begin{array}{ll}
1: & x_0 > 0 \\
2: & y_1 = x_0 \\
3: & y_1 > 0 \\
4: & y_2 = y_1 - 1 \\
3: & y_2 > 0 \\
4: & y_3 = y_2 - 1 \\
3: & y_3 \leq 0 \\
6: & x_1 = x_0 + 1
\end{array}
$$

The bad news

Bad ... ways

lead ...

$y_2 = 1$

$y_3 = 0$

```
1    while(x>0) {
2         y = x;
3         while(y>0) {
4              y = y - 1;
5         }
6         x = x + 1;
7    }
```

$$
\begin{array}{ll}
1: & x_0 > 0 \\
2: & y_1 = x_0 \\
3: & y_1 > 0 \\
4: & y_2 = y_1 - 1 \\
3: & y_2 > 0 \\
4: & y_3 = y_2 - 1 \\
3: & y_3 \leq 0 \\
6: & x_1 = x_0 + 1
\end{array}
$$

The bad news

Bad ways

lead

$y_1 = 2$

$y_2 = 1$

$y_3 = 0$

```
1    while(x>0) {
2        y = x;
3        while(y>0) {
4            y = y - 1;
5        }
6        x = x + 1;
7    }
```

$$
\begin{aligned}
1 &: \quad x_0 > 0 \\
2 &: \quad y_1 = x_0 \\
3 &: \quad y_1 > 0 \\
4 &: \quad y_2 = y_1 - 1 \\
3 &: \quad y_2 > 0 \\
4 &: \quad y_3 = y_2 - 1 \\
3 &: \quad y_3 \leq 0 \\
6 &: \quad x_1 = x_0 + 1
\end{aligned}
$$

The bad news

Bad lways

lead

$y_1 = 2 \quad x_0 = 2$

$y_2 = 1$

$y_3 = 0$

```
1    while(x>0) {
2         y = x;
3         while(y>0) {
4              y = y - 1;
5         }
6         x = x + 1;
7    }
```

$$
\begin{array}{ll}
1: & x_0 > 0 \\
2: & y_1 = x_0 \\
3: & y_1 > 0 \\
4: & y_2 = y_1 - 1 \\
3: & y_2 > 0 \\
4: & y_3 = y_2 - 1 \\
3: & y_3 \le 0 \\
6: & x_1 = x_0 + 1
\end{array}
$$

The bad news

Bad … ways

lead …

$y_1 = 2$ $x_0 = 2$

$y_2 = 1$ $x_1 = 3$

$y_3 = 0$

```
1    while(x>0) {
2        y = x;
3        while(y>0) {
4            y = y - 1;
5        }
6        x = x + 1;
7    }
```

$$
\begin{array}{ll}
1: & x_0 > 0 \\
2: & y_1 = x_0 \\
3: & y_1 > 0 \\
4: & y_2 = y_1 - 1 \\
3: & y_2 > 0 \\
4: & y_3 = y_2 - 1 \\
3: & y_3 \leq 0 \\
6: & x_1 = x_0 + 1
\end{array}
$$

The bad news

Bad ... always

lead ...

$y_1 = 2 \quad x_0 = 2$

$y_2 = 1 \quad x_1 = 3$

$y_3 = 0$

```
1    while(x>0) {
2        y = x;
3        while(y>0) {
4            y = y - 1;
5        }
6        x = x + 1;
7  }
```

$$
\begin{array}{ll}
1: & x_0 > 0 \\
2: & y_1 = x_0 \\
3: & y_1 > 0 \\
4: & y_2 = y_1 - 1 \\
3: & y_2 > 0 \\
4: & y_3 = y_2 - 1 \\
3: & y_3 \leq 0 \\
6: & x_1 = x_0 + 1
\end{array}
$$

Thus: trivially well founded

More bad news

Standard symbolic model checkers dont really <u>like</u> the validity checking task

More bad news

Standard symbolic model checkers dont really <u>like</u> the validity checking task

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$

More bad news

Standard

dont

task

SLAM usually doesnt terminate unless modified significantly

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$

Overview

- → Notes on a representation for programs
- → Checking termination arguments
- → Refining termination arguments
- → Induction
- → Termination analysis

Transformations to reachability supporting closure

```
copied = 0;
    .
    .
    .
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        assert(Q);
    }

  • loop body
}
```

Transformations to reachability supporting closure

```
copied = 0;
    .
    .
    .

while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        [illegible]    skip;
    }

   • loop body
}
```

Transformations to reachability supporting closure

```
copied = 0;
      .
      .
      .
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        a[illegible]        skip;
    }
  • loop body
}
```

If the invariant here is disjunctively WF, then we have proved termination

Transformations to reachability supporting closure

```
copied = 0;
    .
    .
    .

while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                .
                .
            copied = 1;
        }
    } else {
        [illegible]        skip;
    }

  • loop body
}
```

Standard analysis techniques can over approximate these states

Transformations to reachability supporting closure

assume(inv_e)
old_x = x
old_y = y
goto

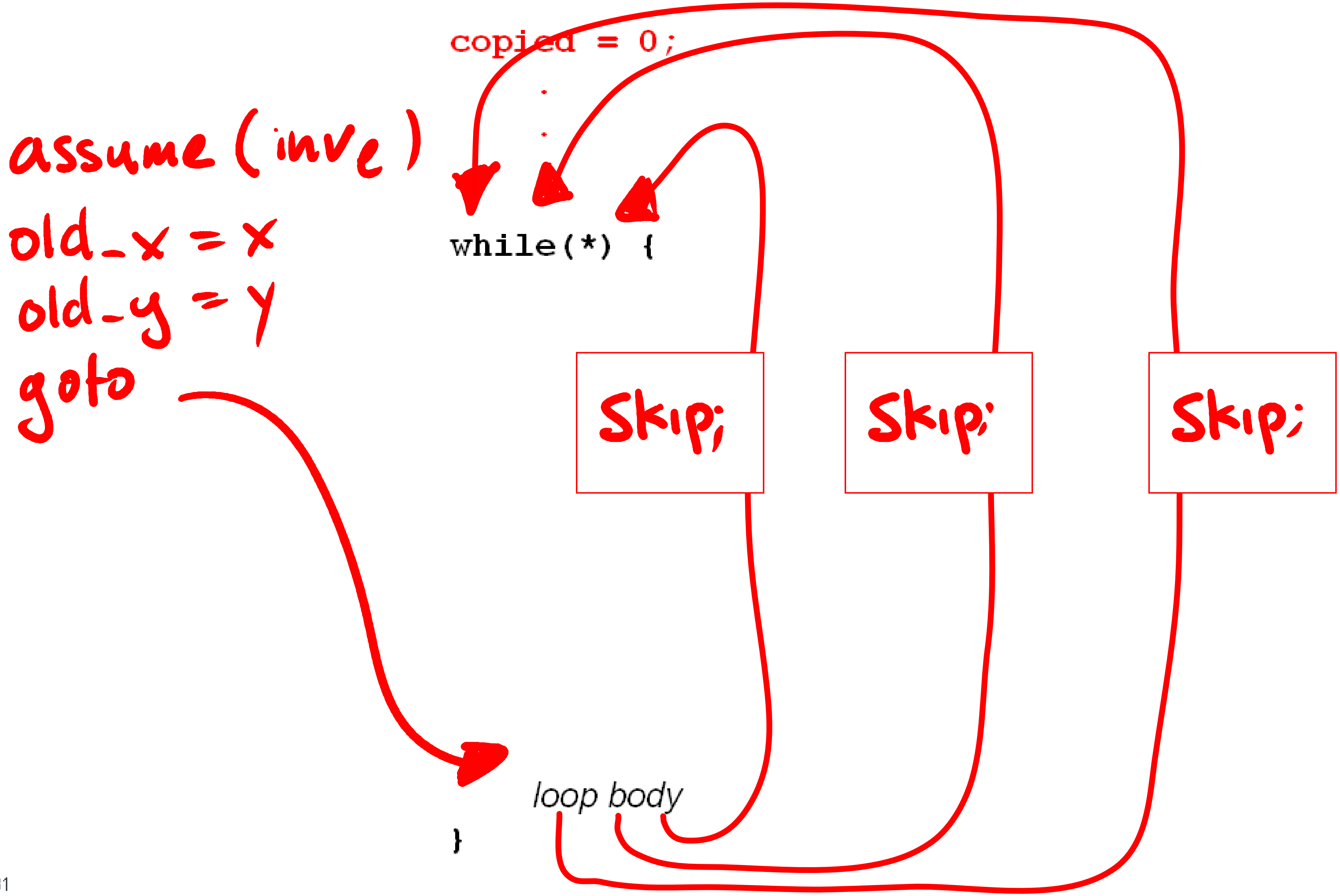
```
copied = 0;
      .
      .
      .
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
                 .
                 .
            copied = 1;
        }
    } else {
        [illegible]     skip;
    }
   . loop body
}
```

Transformations to reachability supporting closure

assume(inv_e)
old_x = x
old_y = y
goto

```
copied = 0;
    .
    .
while(*) {



        loop body
}
```

Transformations to reachability supporting closure



Transformations to … g cl…

old_x $\geq$ x+1
old_y $\geq$ y
x $\geq$ 0
y $\geq$ 0

old_x $\geq$ x+1
old_y $\geq$ y
x $\geq$ 0
y $\geq$ 0

old_x $\geq$ x
old_y $\geq$ y+1
x $\geq$ 0
y $\geq$ 0

Skip;

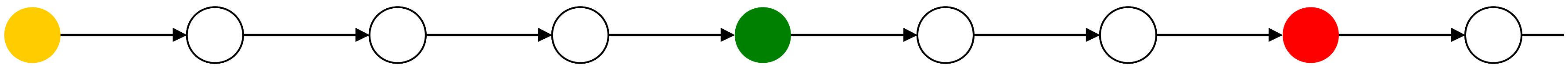
Skip;

Skip;

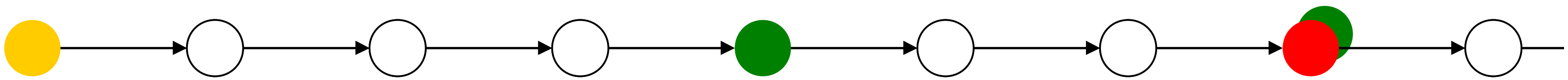
loop body

}

Variance analysis

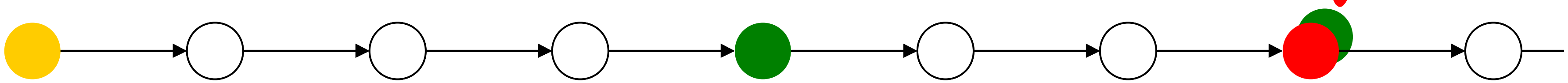


Variance analysis

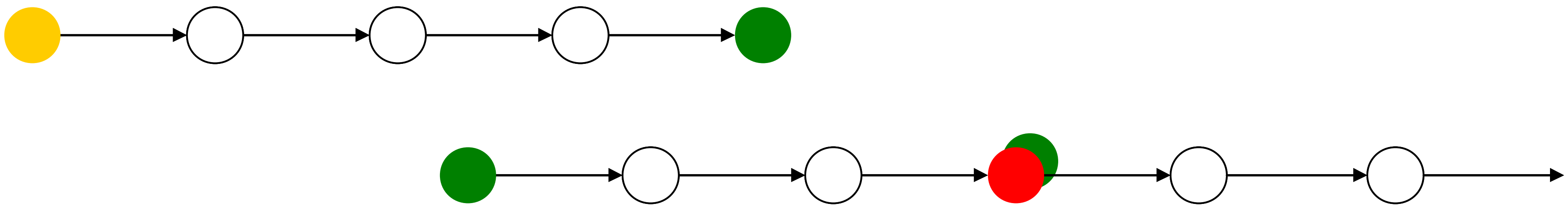


Variance analysis

Overapproximation
of relation encoded
in a state

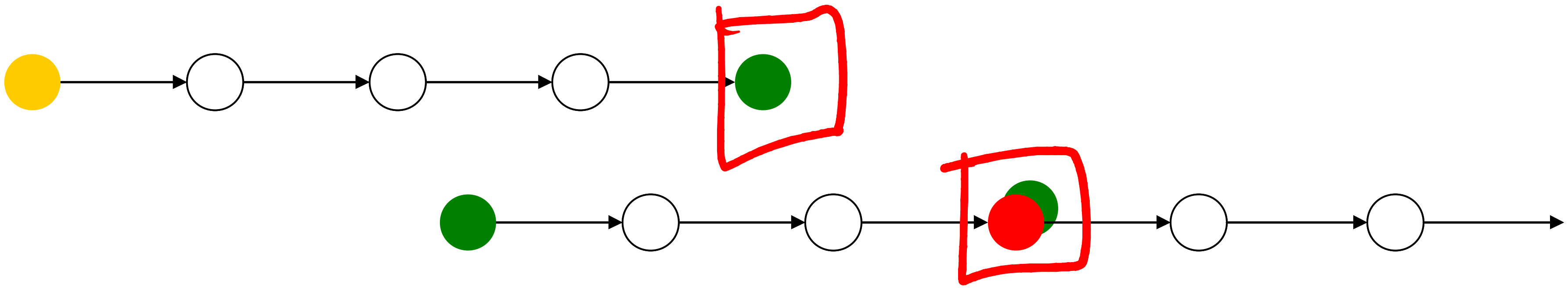


Variance analysis



Variance analysis

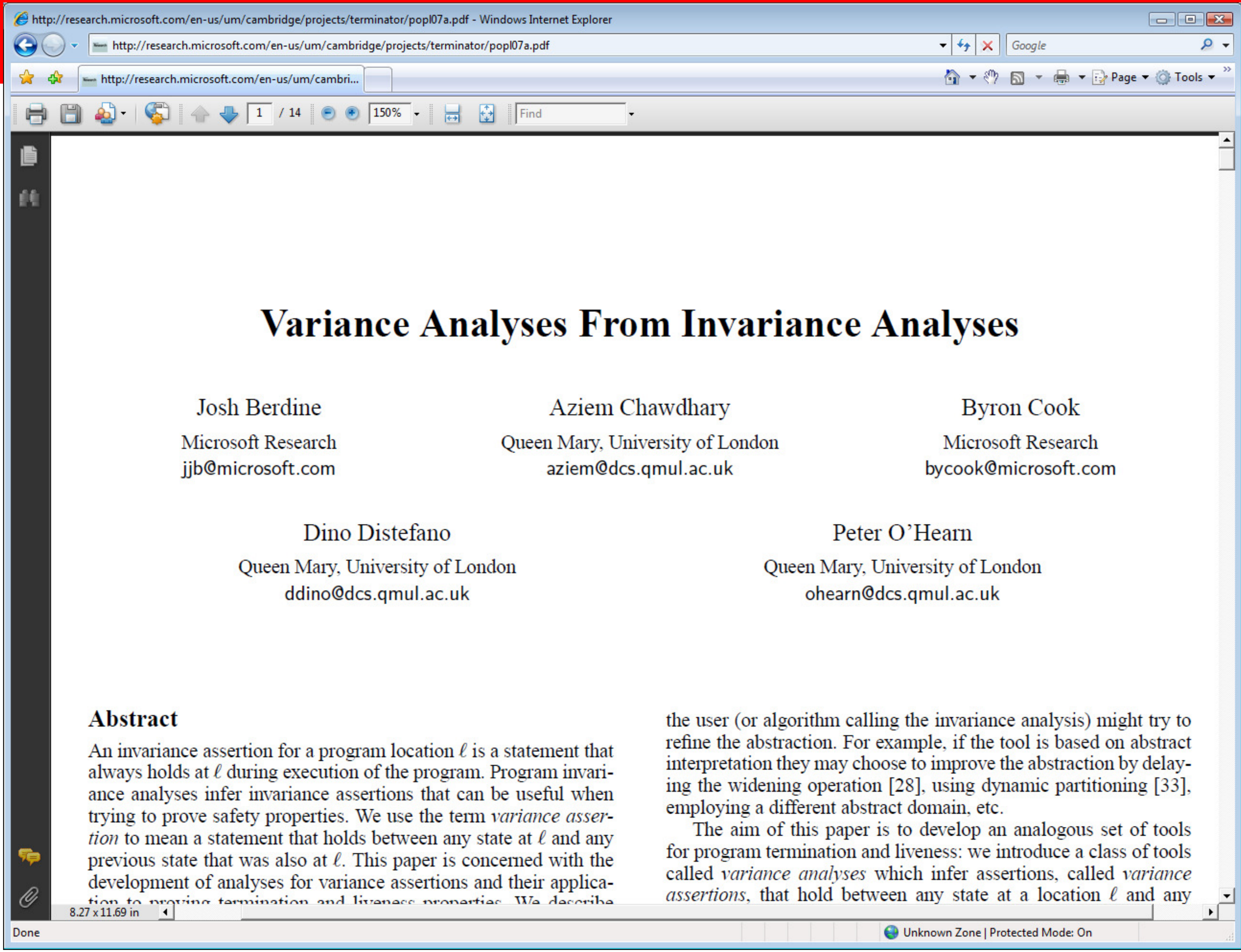
Use off-the-shelf abstract interpretation techniques to compute inclusion



Variance analysis

$R^{+}_{R(I)}$

$$\cancel{R^+_I} \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$



Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Dino Distefano
Queen Mary, University of London
ddino@dcs.qmul.ac.uk

Peter O'Hearn
Queen Mary, University of London
ohearn@dcs.qmul.ac.uk

Abstract

An invariance assertion for a program location ℓ is a statement that always holds at ℓ during execution of the program. Program invariance analyses infer invariance assertions that can be useful when trying to prove safety properties. We use the term *variance assertion* to mean a statement that holds between any state at ℓ and any previous state that was also at ℓ. This paper is concerned with the development of analyses for variance assertions and their application to proving termination and liveness properties. We describe

the user (or algorithm calling the invariance analysis) might try to refine the abstraction. For example, if the tool is based on abstract interpretation they may choose to improve the abstraction by delaying the widening operation [28], using dynamic partitioning [33], employing a different abstract domain, etc.

The aim of this paper is to develop an analogous set of tools for program termination and liveness: we introduce a class of tools called *variance analyses* which infer assertions, called *variance assertions*, that hold between any state at a location ℓ and any

```
VarianceAnalysis(P, L, I♯) {
    IAs := InvarianceAnalysis(P, I♯)
    foreach ℓ ∈ L {
        LTPreds[ℓ] := true
        O := Isolate(P, L, ℓ)
        foreach q ∈ IAs such that pc(q) = ℓ {
            VAs := InvarianceAnalysis(O, Step(O, {Seed(q)}))
            foreach r ∈ VAs {
                if pc(r) = ℓ ∧ ¬WellFounded(r) {
                    LTPreds[ℓ] := false
                }
            }
        }
    }
    return LTPreds
}
```

Variance analysis

- Strategy:
  - Use abstract interpretation techniques to compute (disjunctive) over-approximation
  - Check that the parts of the disjunction are well founded
- Advantages:
  - Can use existing abstract interpretation tools to compute overapproximation
  - Always terminates
  - Fast
- Disadvantages:
  - No counterexamples
  - Less accurate than refinement-based approach
  - Abstract domains (currently) not built for our application
    - Widening can be too aggressive
    - Redundant information kept

the smallest ordinal α such that π admits a ranking function with values $< \alpha$ is the *ranking height* of π. The following observation may be helpful in establishing termination.

LEMMA 1 (COVERING OBSERVATION). *Any transitive relation covered by finitely many well-founded relations is well-founded.*

In other words, if relations $U_1, \ldots, U_n$ are well-founded and $R \subseteq U_1 \cup \cdots \cup U_n$ is a transitive relation, then R is well-founded.

Apparently this observation was made independently a number of times, and each time it was related to the termination problem. As far as we know, the observation was made first by Alfons Geser in [1990, page 31]. A weaker form of the observation, in which the relations U_i are required to be transitive, had been proposed as a question on the web by Geser, and he informed us that he received proofs of it from Jean-Pierre Jouannaud, Werner Nutt, Franz Baader, George McNulty, Thomas Streicher, and Dieter Hofbauer; see [Lescanne, discussion list, items 38–42] for all but the last two of these. Both of our two referees pointed out that the observation was made independently in [Lee et al. 2001]. One of them wrote that "the covering observation lies at the heart of" [Lee et al. 2001] where it "is used implicitly in Theorem 4." The other referee pointed out that the covering observation was made independently in [Dershowitz et al. 2001] and in [Codish et al. 2003]; see [Bruynooghe et al.] in this connection. Recently the covering observation was rediscovered in [Podelski and Rybalchenko 2004] and was used for proving termination in [Podelski and Rybalchenko 2005; Cook et al. 2006; Berdine et al. 2007]. A stronger version of the covering observation, using a hypothesis that is weaker (but more complicated) than transitivity of R, was given in [Doornbos and Von Karger 1998].

The covering observation is proved by a straightforward application of the infinite version of Ramsey's theorem. The transitivity of R is essential here. If a, b are distinct elements then the relation $\{(a,b),(b,a)\}$ is covered by the well-founded relations $\{(a,b)\}$ and $\{(b,a)\}$ but is not well-founded.

Example 2. Let π_1 be the program

the smallest ordinal α such that π admits a ranking function with values $< \alpha$ is the *ranking height* of π. The following observation may be helpful in establishing termination.

Lemma 1 (Covering Observation). *Any transitive relation covered by finitely many well-founded relations is well-founded.*

In other words, if relations $U_1, \dots, U_n$ are well-founded and $R \subseteq U_1 \cup \cdots \cup U_n$ is a transitive relation, then R is well-founded.

Apparently this observation was made independently a number of times, and each time it was related to the termination problem. As far as we know, the observation was made first by Alfons Geser in [1990, page 31]. A weaker form of the observation, in which the relations U_i are required to be transitive, had been proposed as a question on the web by Geser, and he informed us that he received proofs of it from Jean-Pierre Jouannaud, Werner Nutt, Franz Baader, George McNulty, Thomas Streicher, and Dieter Hofbauer; see [Lescanne, discussion list, items 38–42] for all but the last two of these. Both of our two referees pointed out that the observation was made independently in [Lee et al. 2001]. One of them wrote that “the covering observation lies at the heart of” [Lee et al. 2001] where it “is used implicitly in Theorem 4.” The other referee pointed out that the covering observation was made independently in [Dershowitz et al. 2001] and in [Codish et al. 2003]; see [Bruynooghe et al.] in this connection. Recently the covering observation was rediscovered in [Podelski and Rybalchenko 2004] and was used for proving termination in [Podelski and Rybalchenko 2005; Cook et al. 2006; Berdine et al. 2007]. A stronger version of the covering observation, using a hypothesis that is weaker (but more complicated) than transitivity of R, was given in [Doornbos and Von Karger 1998].

The covering observation is proved by a straightforward application of the infinite version of Ramsey’s theorem. The transitivity of R is essential here. If a, b are distinct elements then the relation $\{(a, b), (b, a)\}$ is covered by the well-founded relations $\{(a, b)\}$ and $\{(b, a)\}$ but is not well-founded.

Example 2. Let π_1 be the program

http://research.microsoft.com/en-us/um/people/gurevich/Opera/178.pdf - Windows Internet Explorer

http://research.microsoft.com/en-us/um/people/gurevich/Opera/178.pdf

the smallest ordinal α such that π admits a ranking function with values $< \alpha$ is the *ranking height* of π. The following ... termination.

What about size-change?

..., had ... that he ... Franz Baader, George ... [Lescanne, discussion list, items 38–42] for all but the last two of ... two referees pointed out that the observation was made independently ... 2001]. One of them wrote that "the covering observation lies at the heart of" [Lee et al. 2001] where it "is used implicitly in Theorem 4." The other referee pointed out that the covering observation was made independently in [Dershowitz et al. 2001] and in [Codish et al. 2003]; see [Bruynooghe et al.] in this connection. Recently the covering observation was rediscovered in [Podelski and Rybalchenko 2004] and was used for proving termination in [Podelski and Rybalchenko 2005; Cook et al. 2006; Berdine et al. 2007]. A stronger version of the covering observation, using a hypothesis that is weaker (but more complicated) than transitivity of R, was given in [Doornbos and Von Karger 1998].

The covering observation is proved by a straightforward application of the infinite version of Ramsey's theorem. The transitivity of R is essential here. If a, b are distinct elements then the relation $\{(a, b), (b, a)\}$ is covered by the well-founded relations $\{(a, b)\}$ and $\{(b, a)\}$ but is not well-founded.

Example 2. Let π_1 be the program

Done Unknown Zone | Protected Mode: On

Termination proof rule

$$R^+ \subseteq \trianglerighteq_{\mathsf{a}} \cup \trianglerighteq_{\mathsf{b}} \cup \ldots \cup \trianglerighteq_{\mathsf{z}}$$

Termination proof rule

$$R^+ \subseteq \trianglerighteq_{\mathsf{a}} \cup \trianglerighteq_{\mathsf{b}} \cup \ldots \cup \trianglerighteq_{\mathsf{z}}$$

All program variables used

Termination proof rule

$$\cancel{R^+} \subseteq \trianglerighteq_a \cup \trianglerighteq_b \cup \ldots \cup \trianglerighteq_z$$

$\alpha(R)^+$

All program variables used