Proving that programs eventually do something good

Byron Cook

Collaborators

- Domagoj Babic,
- Josh Berdine,
- Aziem Chawdhary,
- Dino Distefano,
- Alexey Gotsman,
- Sumit Gulwani,
- Alan Hu,
- Samin Ishtiaq,
- Eric Koskinen,
- Tal Lev-Ami,
- Peter O'Hearn,
- Matthew Parkinson,
- Andreas Podelski,
- Zvonimir Rakameric,
- Andrey Rybalchenko,
- Mooly Sagiv,
- Moshe Vardi,
- Viktor Vafeiadis,
- Hongseok Yang,
- & the East London Massive.

Collaborators

- Domagoj Babic,
- Josh Berdine,
- Aziem Chawdhary,
- Dino Distefano,
- Alexey Gotsman,
- Sumit Gulwani,
- Alan Hu,
- Samin Ishtiaq,
- Eric Koskinen,
- Tal Lev-Ami,
- Peter O'Hearn,
- Matthew Parkinson,
- Andreas Podelski,
- Zvonimir Rakameric,
- Andrey Rybalchenko,
- Mooly Sagiv,
- Moshe Vardi,
- Viktor Vafeiadis,
- Hongseok Yang,
- & the East London Massive.

Formal verification

DOI:10.1145/1941487.1941509

In contrast to popular belief, proving termination is not always impossible.

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

THE PROGRAM TERMINATION problem, also known as the uniform halting problem, can be defined as follows:

Using only a finite amount of time, determine whether a given program will always finish running or could execute forever.

This problem rose to prominence before the invention of the modern computer, in the era of Hilbert's *Entscheidungsproblem*:[a] the challenge to formalize all of mathematics and use algorithmic means to determine the validity of all statements. In hopes of either solving Hilbert's challenge, or showing it impossible, logicians began to search for possible instances of undecidable problems. Turing's proof[38] of termination's undecidability is the most famous of those findings.[b]

The termination problem is structured as an infinite set of queries: to solve the problem we would need to invent a method capable of accurately answering either "terminates" or "doesn't terminate" when given any program drawn from this set. Turing's result tells us that any tool that attempts to solve this problem will fail to return a correct answer on at least one of the inputs. No number of extra processors nor terabytes of storage nor new sophisticated algorithms will lead to the development of a true oracle for program termination.

Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe we are always unable to prove termination, rather than more benign consequence that we are unable to always prove termination. Phrases like "but that's like the termination problem" are often used to end discussions that might otherwise have led to viable partial solutions for real but undecidable problems. While we cannot ignore termination's undecidability, if we develop a slightly modified problem statement we can build useful tools. In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. If the termination prover cannot prove or disprove termination, it should return "unknown."

Using only a finite amount of time, determine whether a given program will always finish running or could execute forever, or return the answer "unknown."

a In English: "decision problem."

b There is a minor controversy as to whether or not Turing proved the undecidability in[38]. Technically he did not, but termination's undecidability is an easy consequence of the result that is proved. A simple proof can be found in Strachey.[36]

» key insights

- For decades, the same method was used for proving termination. It has never been applied successfully to large programs.
- A deep theorem in mathematical logic, based on Ramsey's theorem, holds the key to a new method.
- The new method can scale to large programs because it allows for the modular construction of termination arguments.

Formal verification

review articles

DOI:10.1145/1941487.1941509

In contrast to popular belief, proving termination is not always impossible.

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

THE PROGRAM TERMINATION problem, also known as the uniform halting problem, can be defined as follows:

Using only a finite amount of time, determine whether a given program will always finish running or could execute forever.

This problem rose to prominence before the invention of the modern computer, in the era of Hilbert's *Entscheidungsproblem*:[a] the challenge to formalize all of mathematics and use algorithmic means to determine the validity of all statements. In hopes of either solving Hilbert's challenge, or showing it impossible, logicians began to search for possible instances of undecidable problems. Turing's proof[38] of termination's undecidability is the most famous of those findings.[b]

The termination problem is structured as an infinite set of queries: to solve the problem we would need to invent a method capable of accurately answering either "terminates" or "doesn't terminate" when given any program drawn from this set. Turing's result tells us that any tool that attempts to solve this problem will fail to return a correct answer on at least one of the inputs. No number of extra processors nor terabytes of storage nor new sophisticated algorithms will lead to the development of a true oracle for program termination.

Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe we are always unable to prove termination, rather than more benign consequence that we are unable to always prove termination. Phrases like "but that's like the termination problem" are often used to end discussions that might otherwise have led to viable partial solutions for real but undecidable problems. While we cannot ignore termination's undecidability, if we develop a slightly modified problem statement we can build useful tools. In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. If the termination prover cannot prove or disprove termination, it should return "unknown."

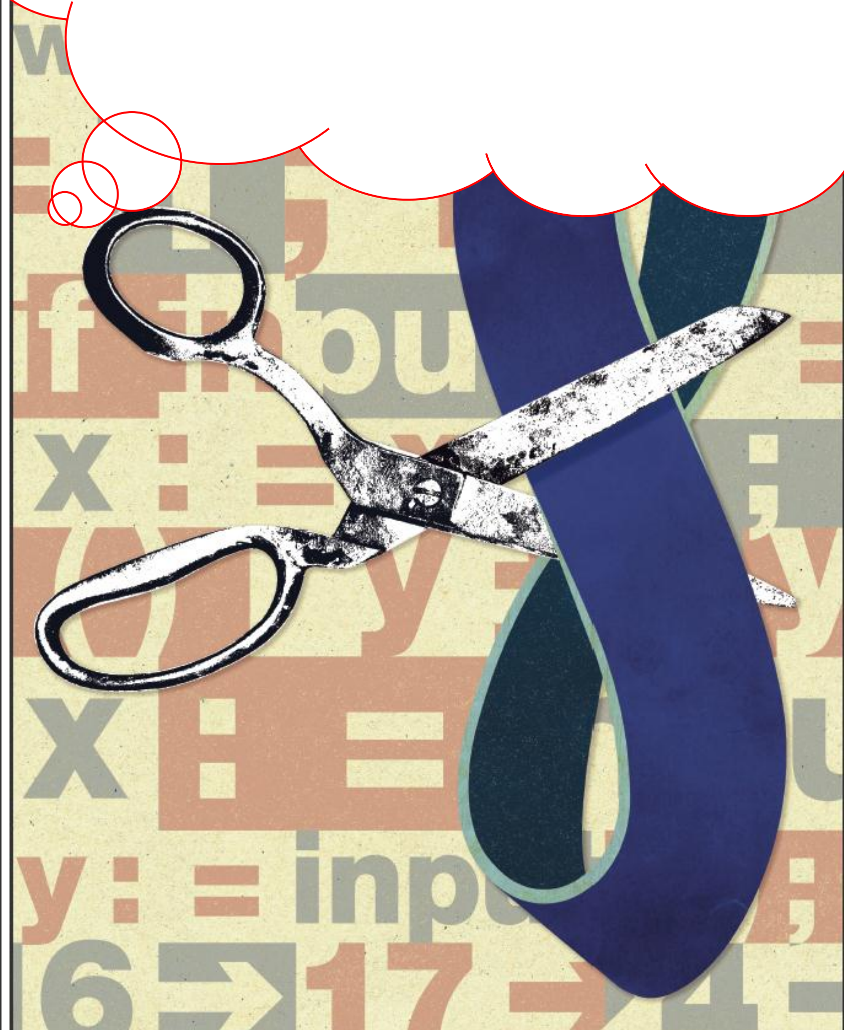
Using only a finite amount of time, determine whether a given program will always finish running or could execute forever, or return the answer "unknown."

» **key insights**

- **For decades, the same method was used for proving termination. It has never been applied successfully to large programs.**
- **A deep theorem in mathematical logic, based on Ramsey's theorem, holds the key to a new method.**
- **The new method can scale to large programs because it allows for the modular construction of termination arguments.**

a In English: "decision problem."

b There is a minor controversy as to whether or not Turing proved the undecidability in*. Technically he did not, but termination's undecidability is an easy consequence of the result that is proved. A simple proof can be found in Strachey.[36]



Automatic formal verification

- View artifact of interest as a mathematical system:
  - Software
  - Hardware
  - Biological system
  - *etc ……*

- Build tools that find proofs of correctness using mathematics and logic

- 100% testing coverage
  - Faster and more scalable than brute force
  - Allows for 100% coverage even for infinite-state systems

Example property

"The parallel port device driver's event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE"

Example property

"The parallel port device driver's event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE"

Exa

“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”

Exa

"The parallel port device driver's event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE"

SLAM

Example property

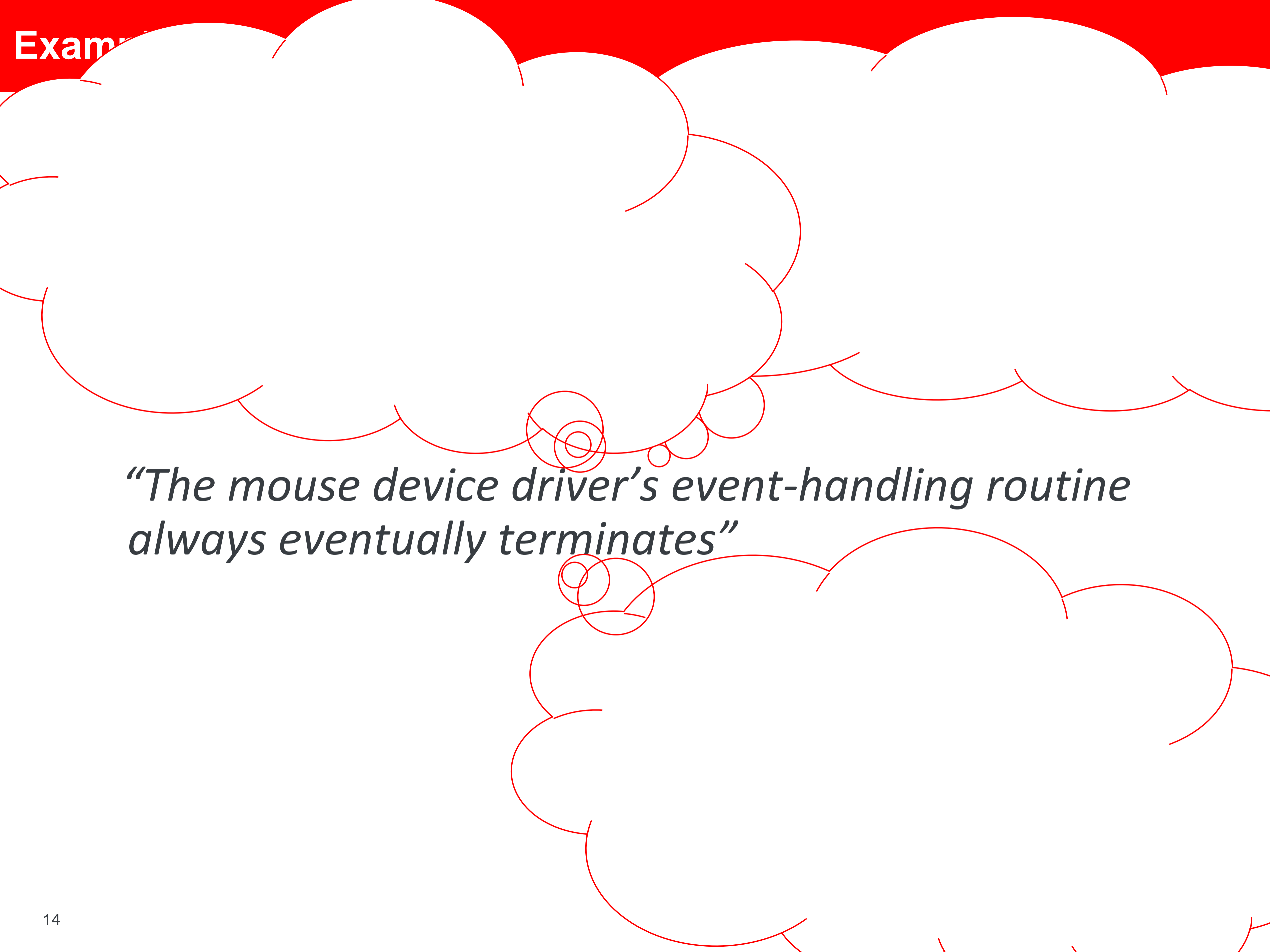
"The mouse device driver's event-handling routine always eventually terminates"

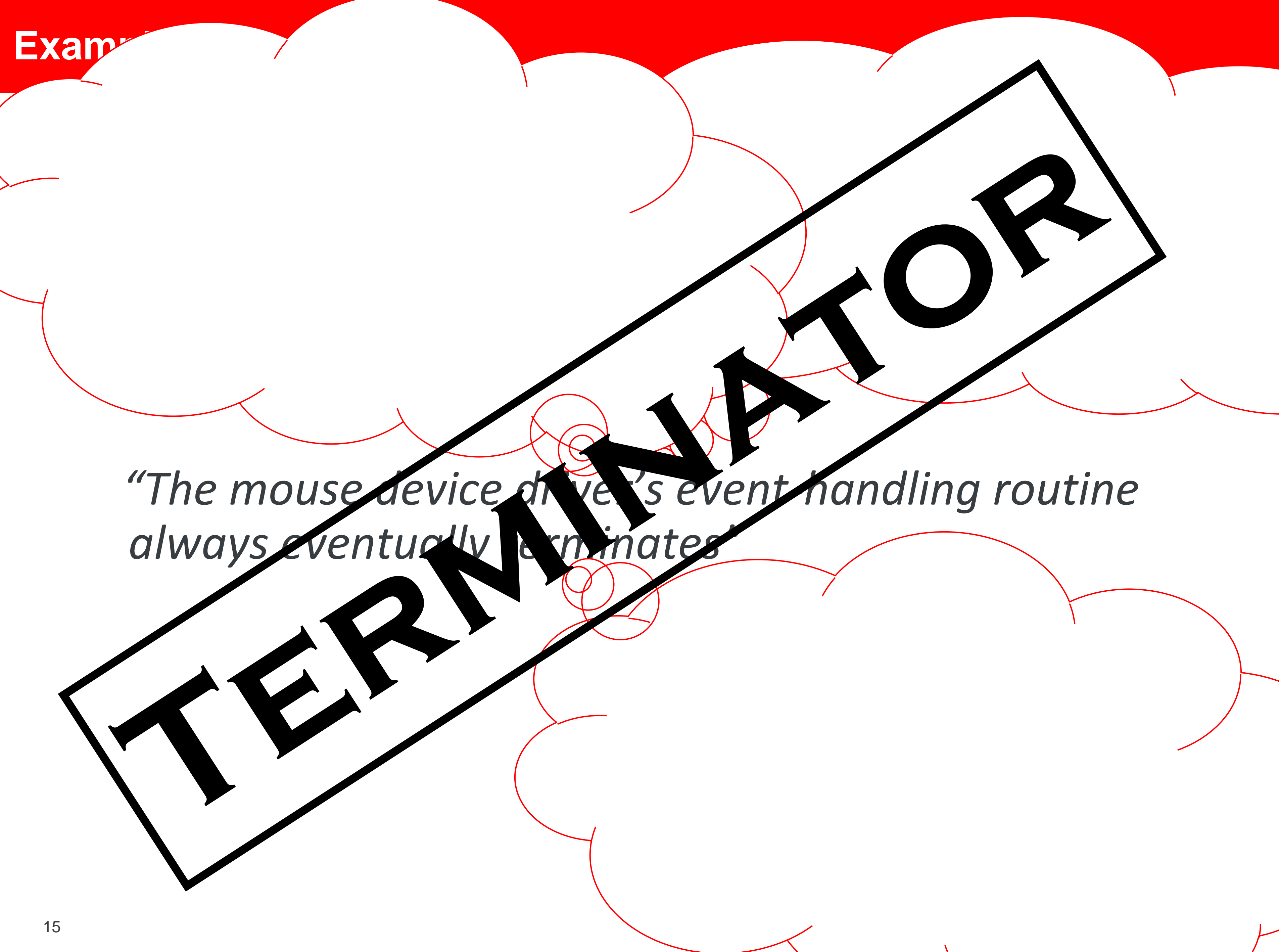
Example property

"The mouse device driver's event-handling routine always eventually terminates"

Example property

"The mouse device driver's event-handling routine always eventually terminates"


“The mouse device driver’s event-handling routine always eventually terminates”


Exam
“The mouse device driver’s event handling routine always eventually terminates”
TERMINATOR

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this IRP.
            // What we're interested in is not whether IoCancelIrp() was called
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this IRP.
            // What we're interested in is not whether IoCancelIrp() was called
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```c
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this IRP.
            // What we're interested in is not whether IoCancelIrp() was called
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Proving termination

- Traditional termination proving method originally proposed by the forefathers of computing

- *E.g.* Turing, “Checking a large routine”, 1949

> Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In

Proving termination

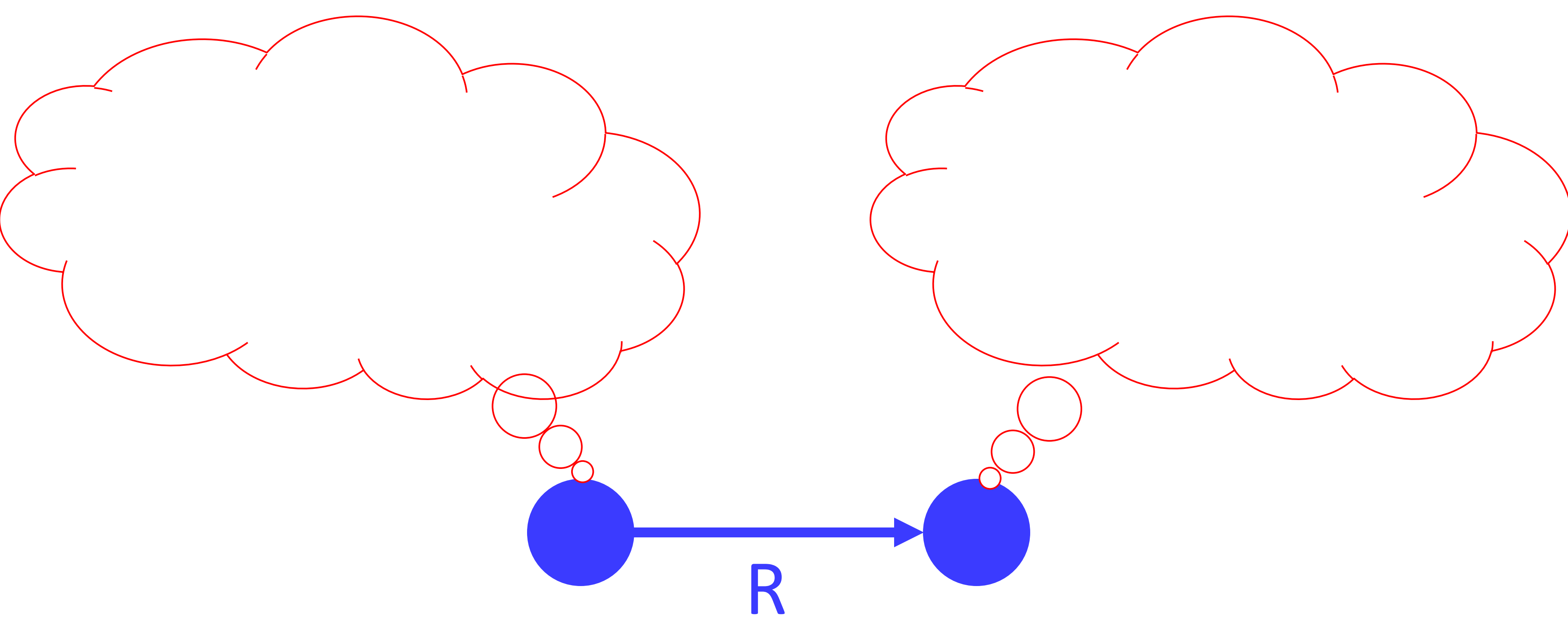
- Traditional termin… proving method originally propose… …f computing

- E.… ”, 1949

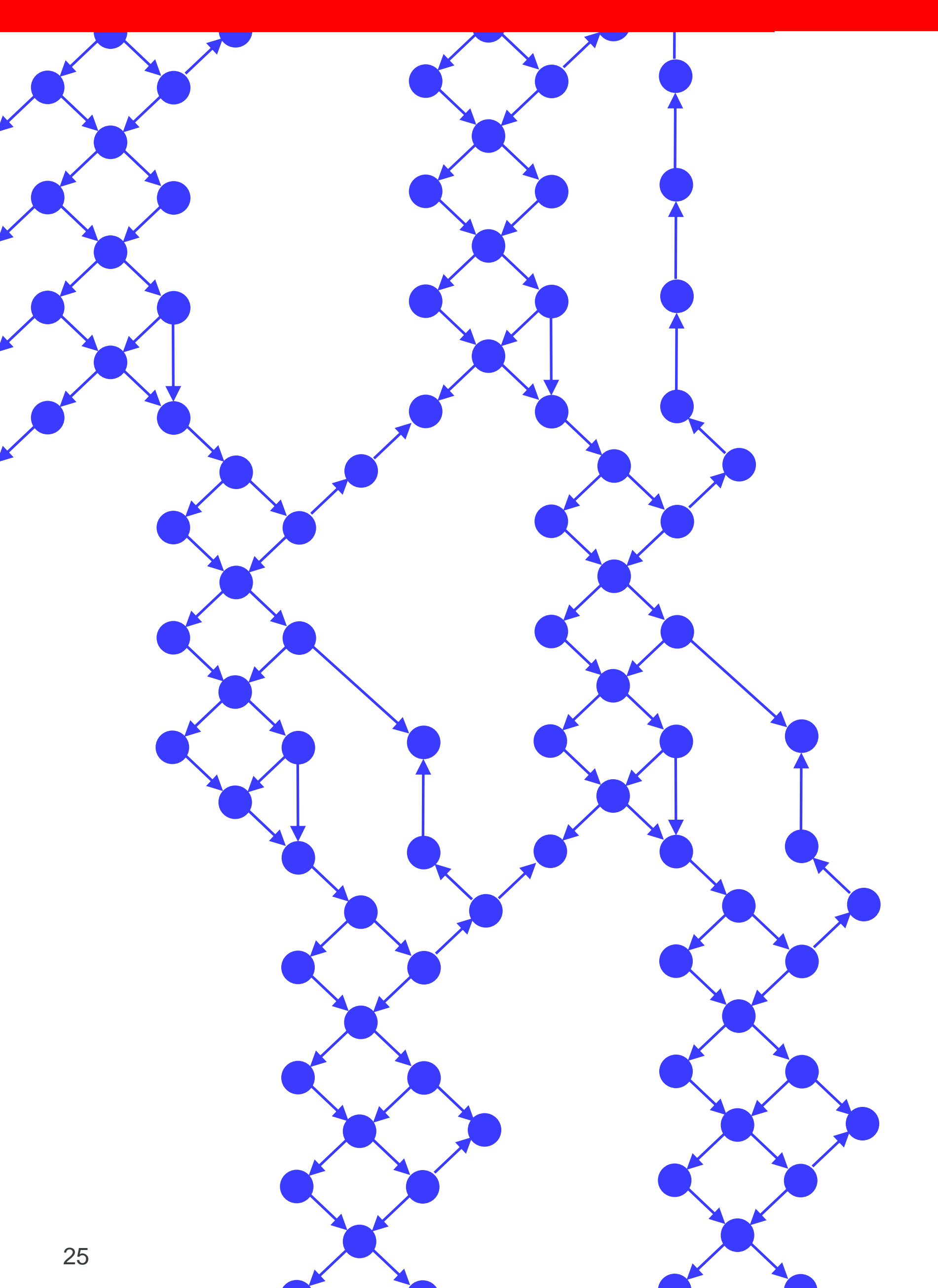
F… …process comes
to an e… …by the program-
mer giving a furt… …be verified. This
may take the form of a quantity which … …erted to decrease
continually and vanish when the machine … …s. To the pure
mathematician it is natural to give an ordinal number. In

Proving termination

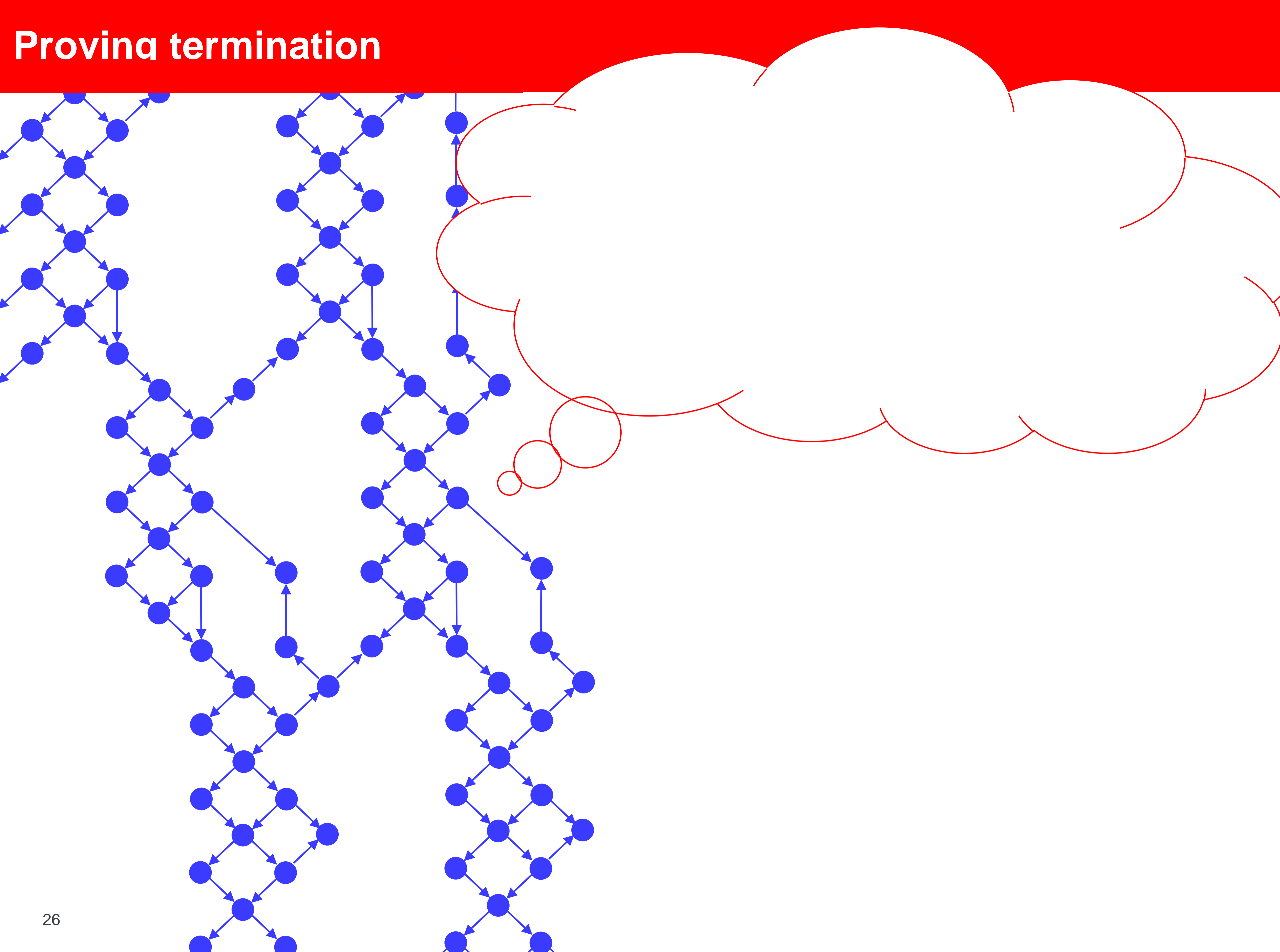
Proving termination



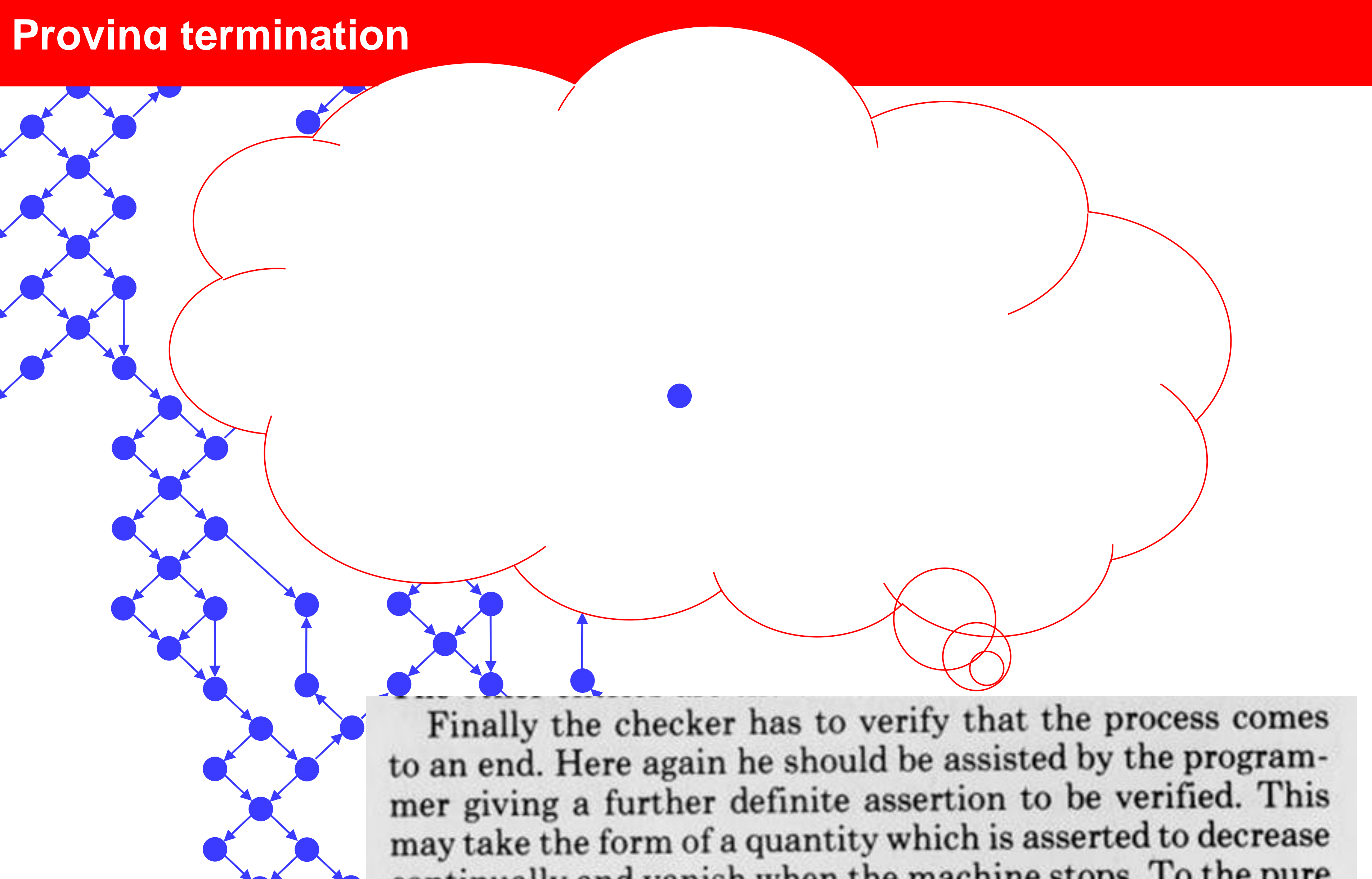
Proving termination



Proving termination

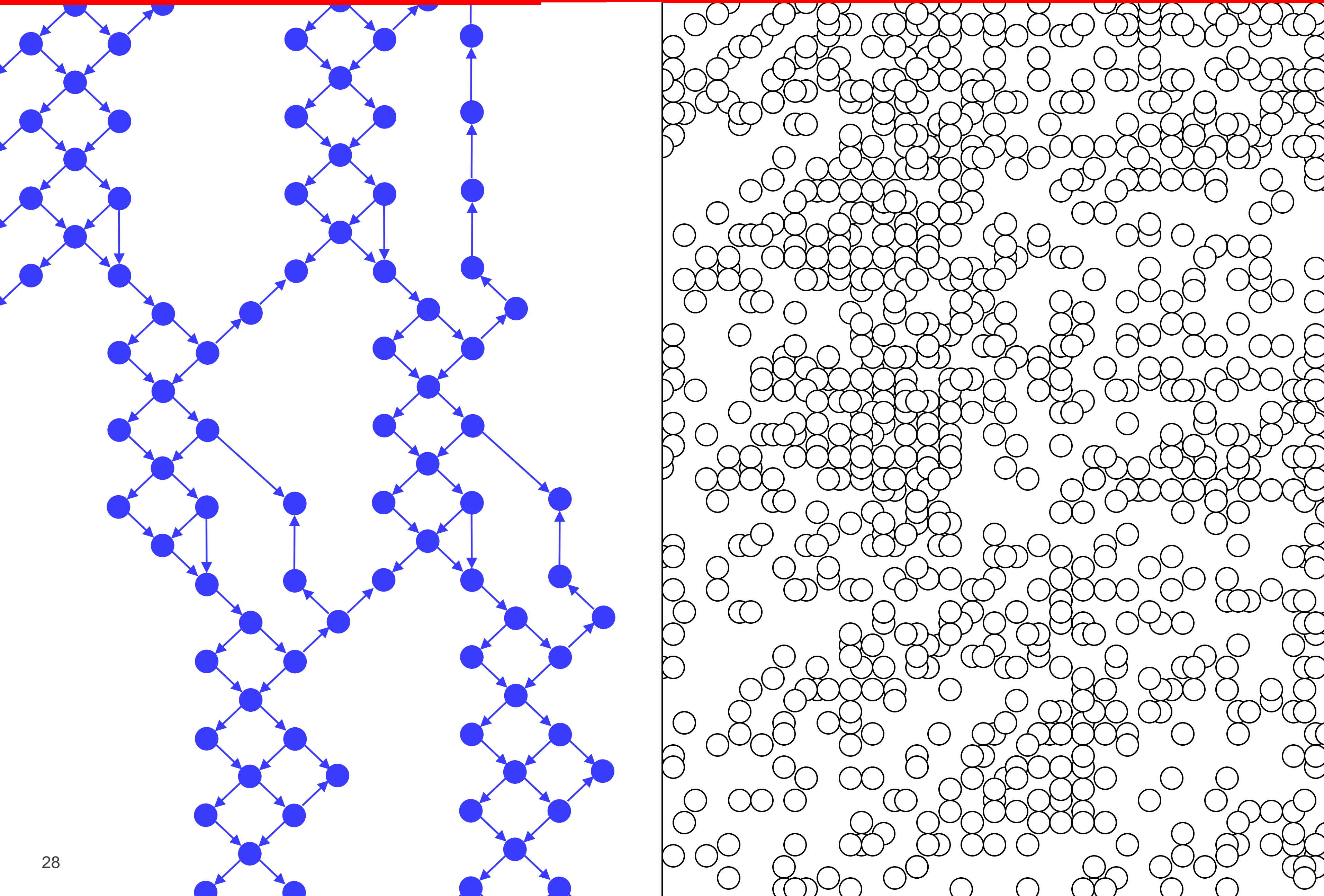


Proving termination

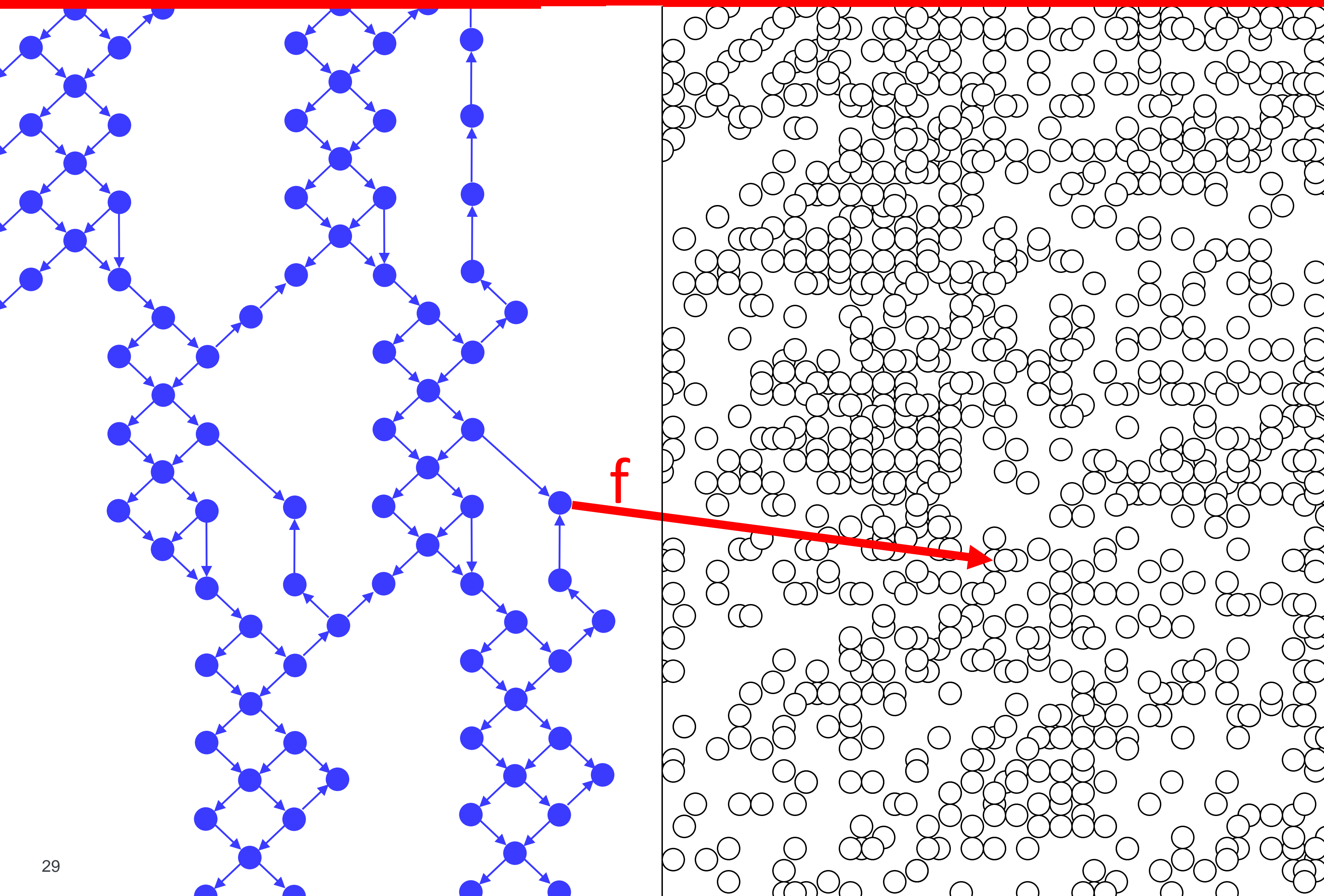


Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In

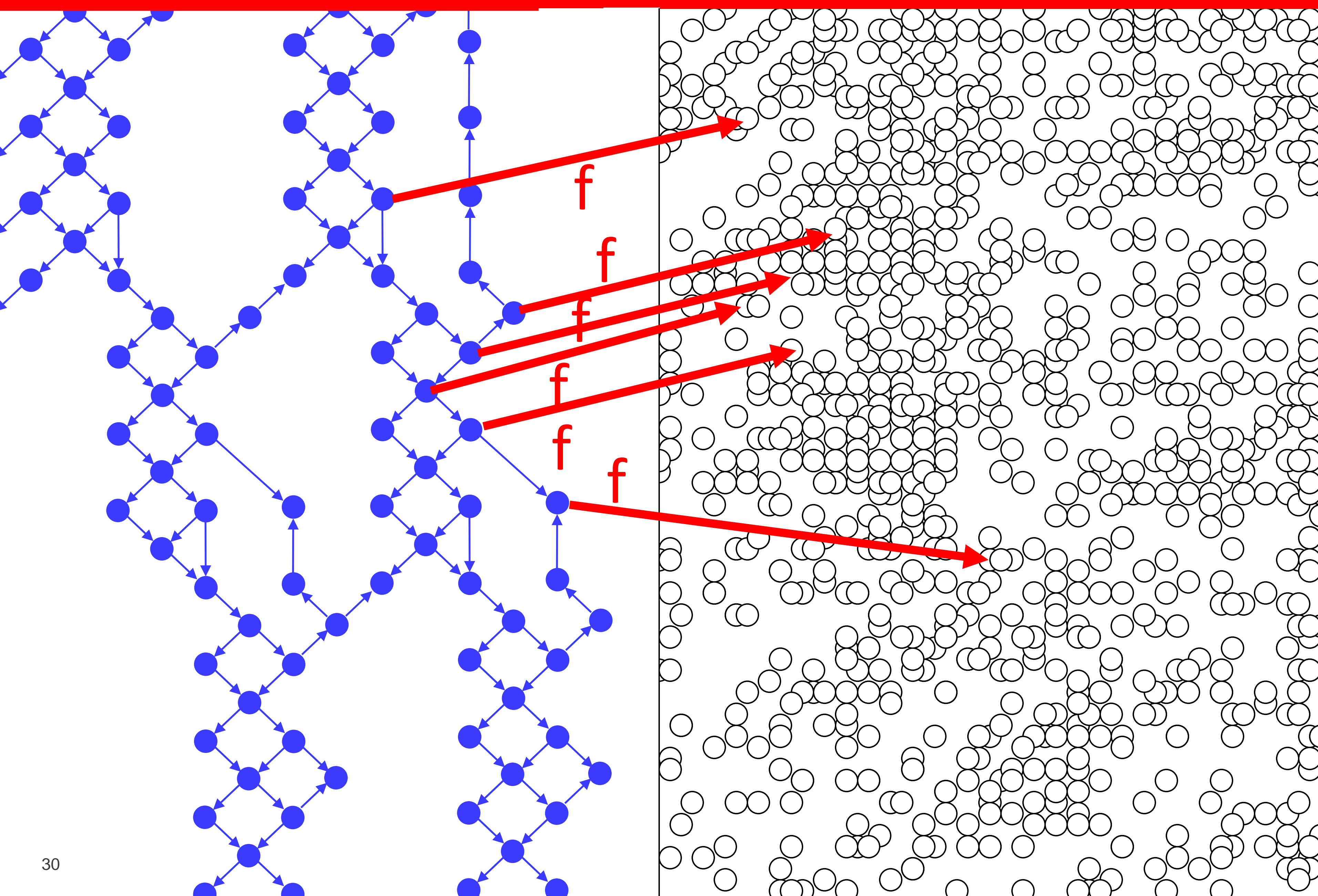
Proving termination



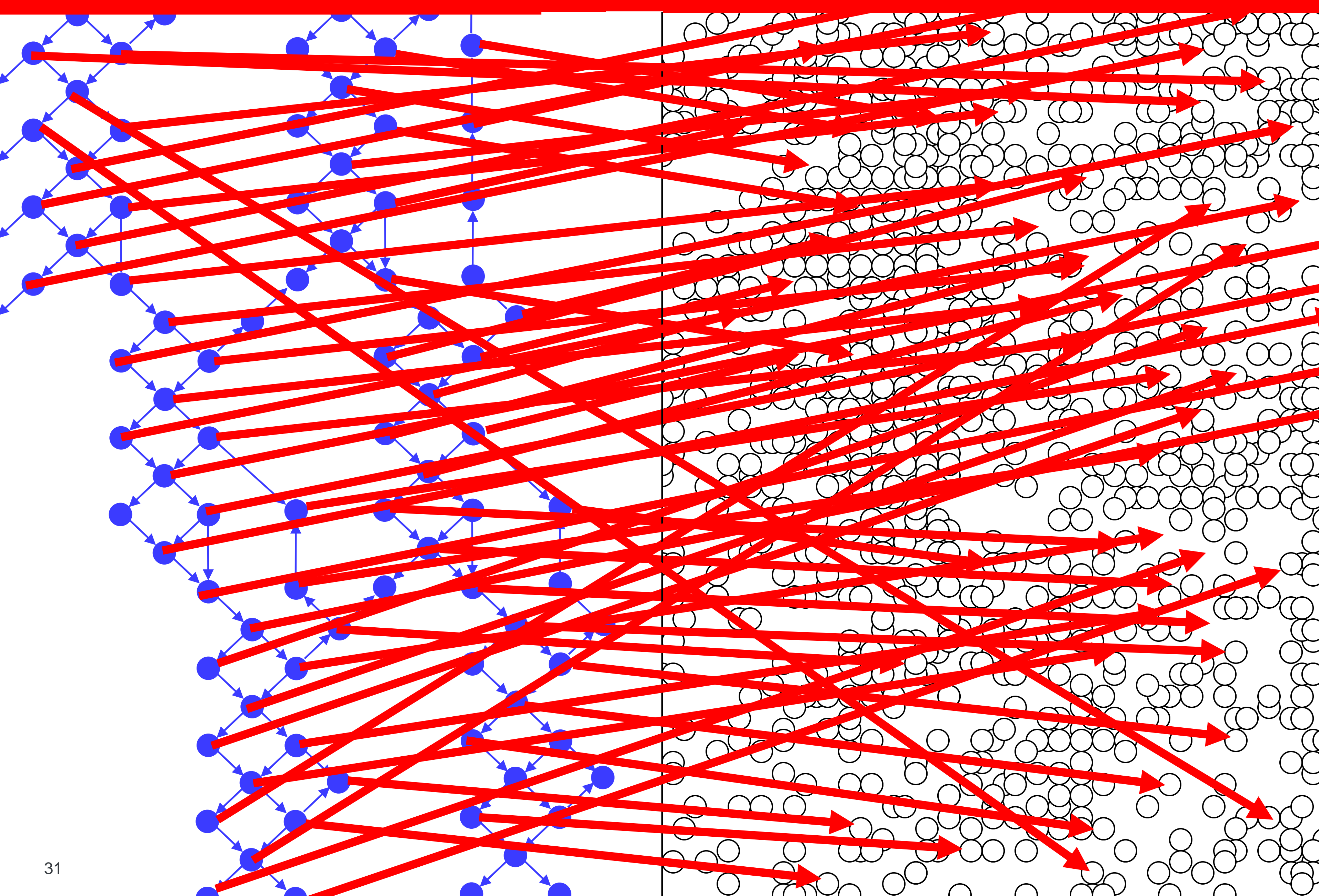
Proving termination



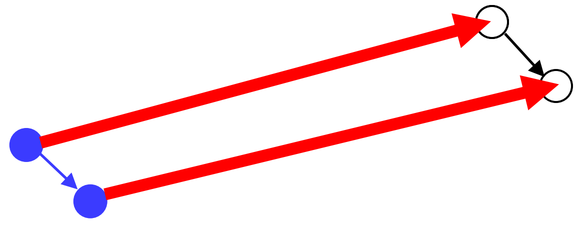
Proving termination



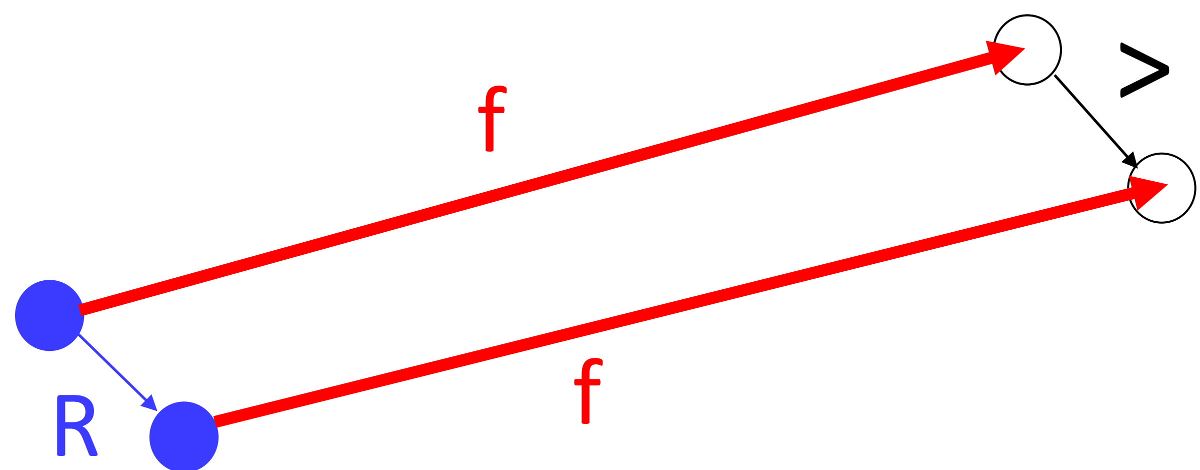
Proving termination

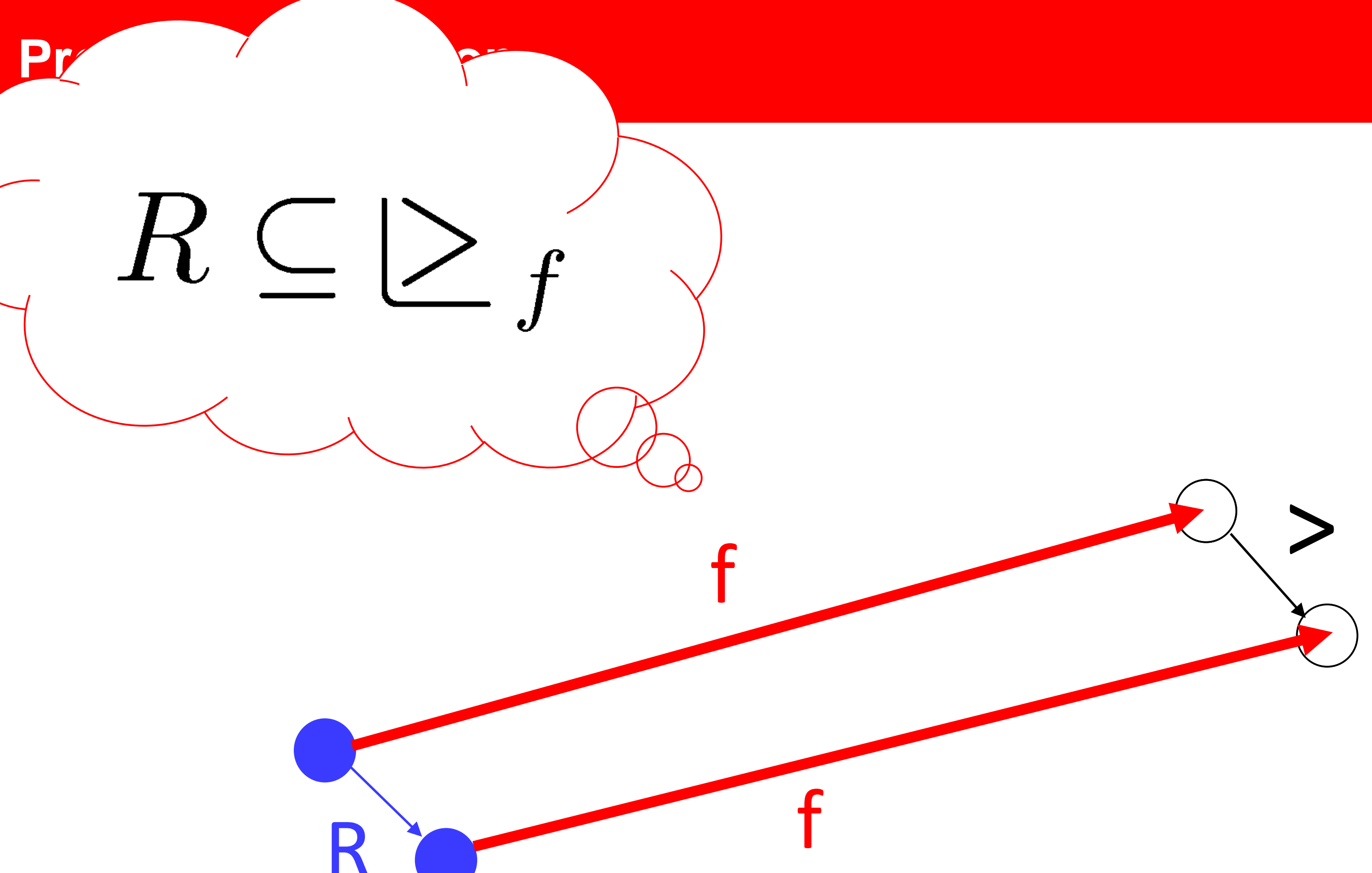


Proving termination



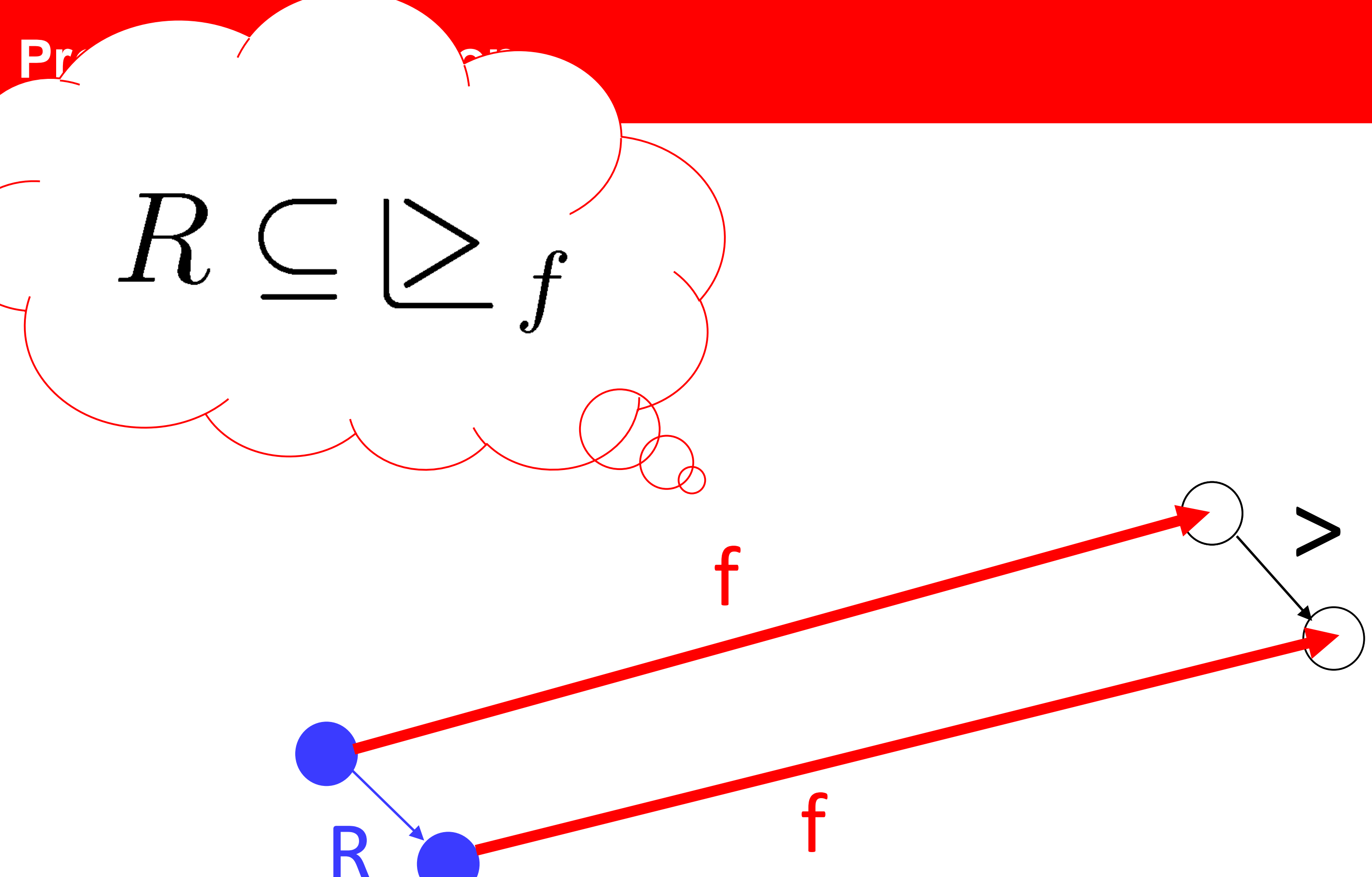
Proving termination

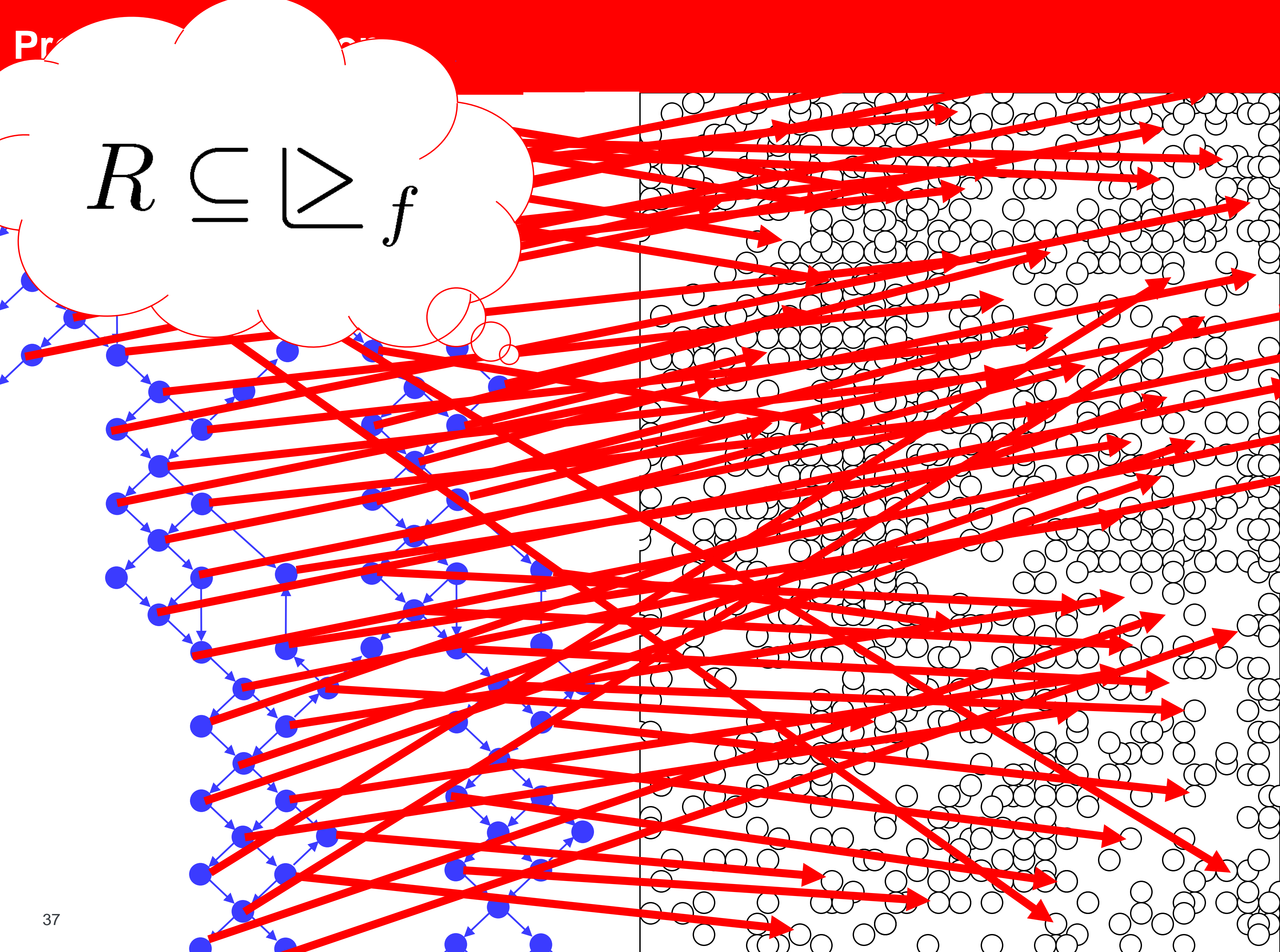



Pr
$R \subseteq \trianglerighteq_f$
f
>
f
R

Proving termination

$$\trianglerighteq_f \triangleq \{(s, t) \mid f(s) > f(t)\}$$


$R \subseteq \trianglerighteq_f$
f
>
f
R


$R \subseteq \trianglerighteq_f$

Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Outline

- Introduction
- Termination basics & history
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

Automating the search for proofs

- Transition relations must be computed

$$R = U \cap [(U^*(I) \times U^*(I))]$$

- Technically, computing $U^*(I)$ is undeciable, so we must find a sound over-approximation using available techniques:

$$U^*(I) \subseteq Q$$

- Q represents an infinite set of states, but has a compact expression

Automating the search for proofs

- Transition relations must be compute

$$R = U \cap [(U^*(I) \times U$$

- Technically, computing $U^*(I)$ is unde
  find a sound over-approximation usin
  techniques:

$$U^*(I) \subseteq Q$$

- Q represents an infinite set of states
  compact expression

Microsoft Development Environment [design] - mouclass.

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueu
         entry != &DeviceExtension->ReadQu
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IR
        stack = IoGetCurrentIrpStackLocati
        if (stack->FileObject == FileObjec
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancel

            //
            // IoCancelIrp() could have ju
            // What we're interested in is
            // (ie, nextIrp->Cancel is set
            // is about to call) our cance
            // of the test-and-set macro I
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not cal
                //
                return irp;
            }
            else {
                //
                // This IRP was just cance
                // be) called. The cancel
                // we drop the spinlock. S
                //
                // Also, the cancel routin
                // IRP's listEntry point t
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->
            }
        }
```

Automating the search for proofs

- Transition relations must be computed

$$R = U \cap [(U^*(I) \times U^*(I))]$$

- Technically, computing $U^*(I)$ is undeciable, so we must find a sound over-approximation using available techniques:

$$U^*(I) \subseteq Q$$

- Q represents an infinite set of states, but has a compact expression

→ Transitio... be computed

$$R = U \cap [(U^*(I) \times U^*(I))]$$

→ Technically, computing $U^*(I)$ is undeciable, so we must find a sound over-approximation using available techniques:

$$U^*(I) \subseteq Q$$

→ Q represents an infinite set of states, but has a compact expression

Automating the search for proofs

- We use an over-approximation of the transition relation

$$R' = U \cap [Q \times Q]$$

- Since $R \subseteq R'$, we can prove termination by showing

$$R' \subseteq \trianglerighteq_f$$

- Meaning: there might be unrealistic transitions that we have to worry about

R'

Automating the search for proofs

- In practice, its extremely hard to find the right overapproximation Q

- Luckily: recent breakthroughs in safety proving now make this possible.

- In fact: the checking the validity of a termination argument can be directly encoded as a safety property

- Tools like **SLAM** can be used to prove validity

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

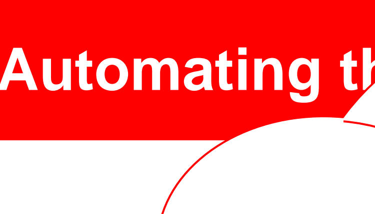
Automating th

$$R^+ \subseteq \unrhd_f \cup \unrhd_g \cup \unrhd_h$$

- Provi
  undeci

- Fin an f such that $R \subseteq \unrhd_f$ is even harder in practice (and undecidable in theory)

Automating th

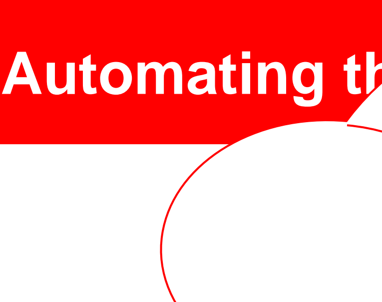


$$R^+ \subseteq \trianglerighteq_f \cup \trianglerighteq_g \cup \trianglerighteq_h$$

- Provi
  undeci

- Fin g an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

Automating th



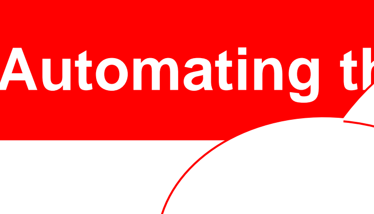
$$R^+ \subseteq \trianglerighteq_f \cup \trianglerighteq_g \cup \trianglerighteq_h$$

→ [

- Provi
undeci

- Fin an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

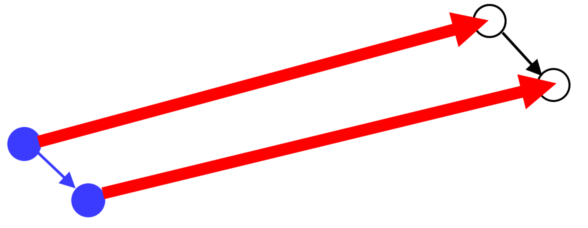
Automating th



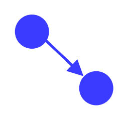
$$R^+ \subseteq \trianglerighteq_f \cup \trianglerighteq_g \cup \trianglerighteq_h$$

- Provi… undeci…
- Fin… an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

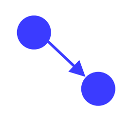
Modular termination arguments



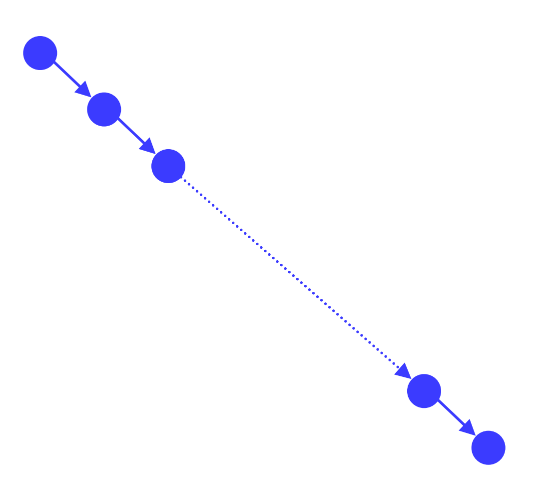
Modular termination arguments



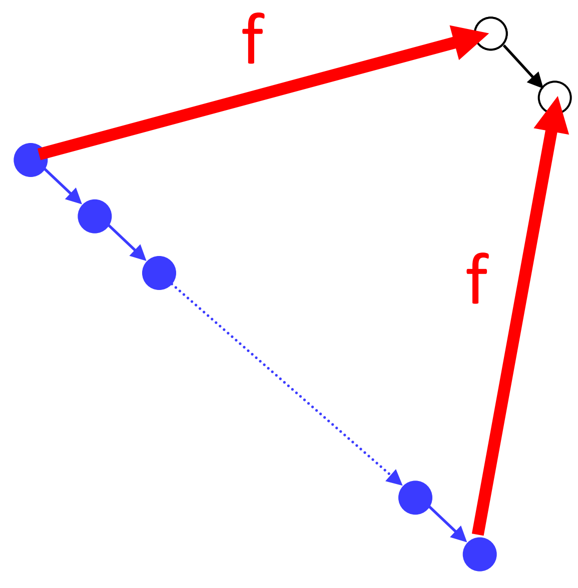
Modular termination arguments



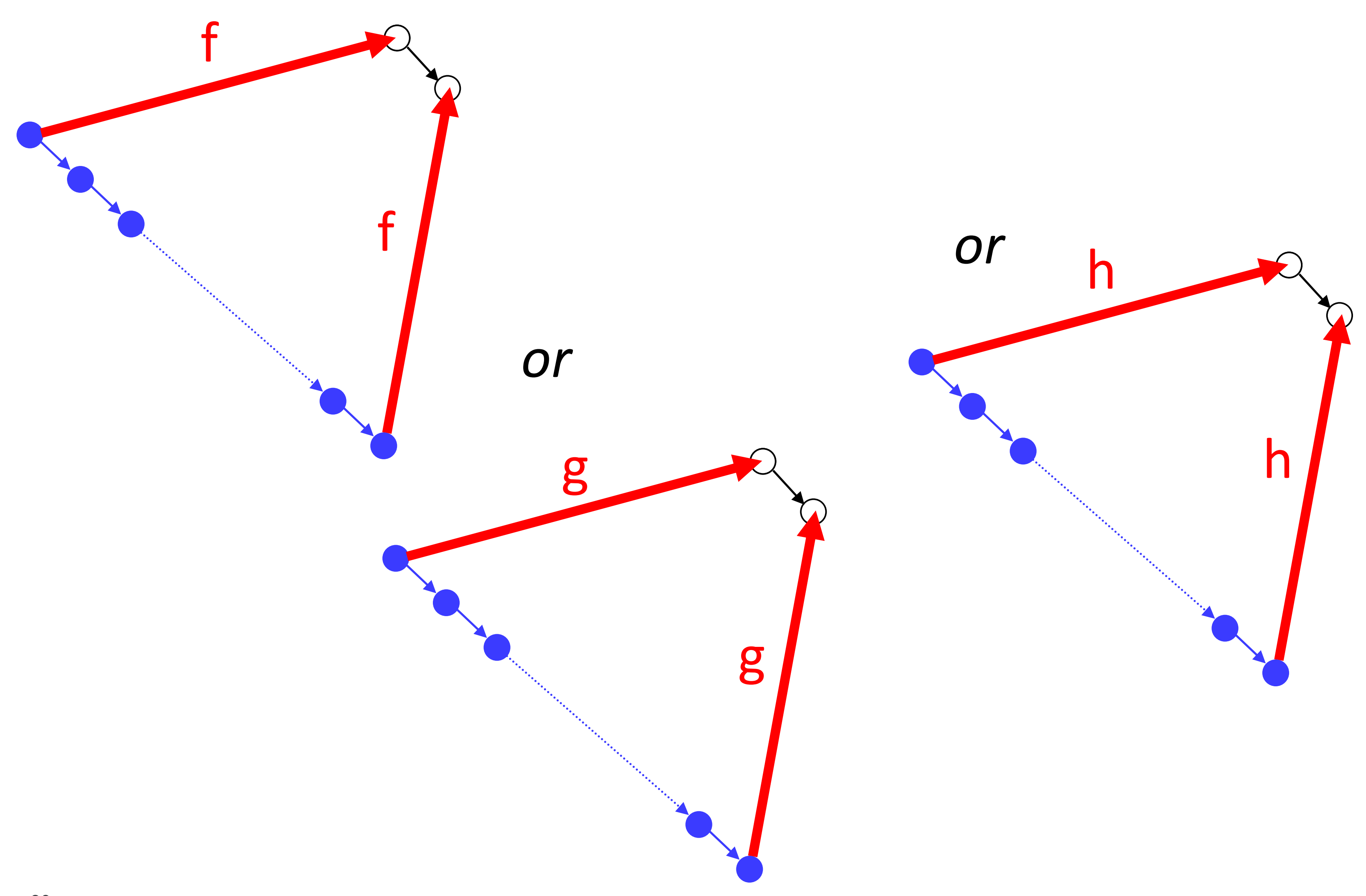
Modular termination arguments



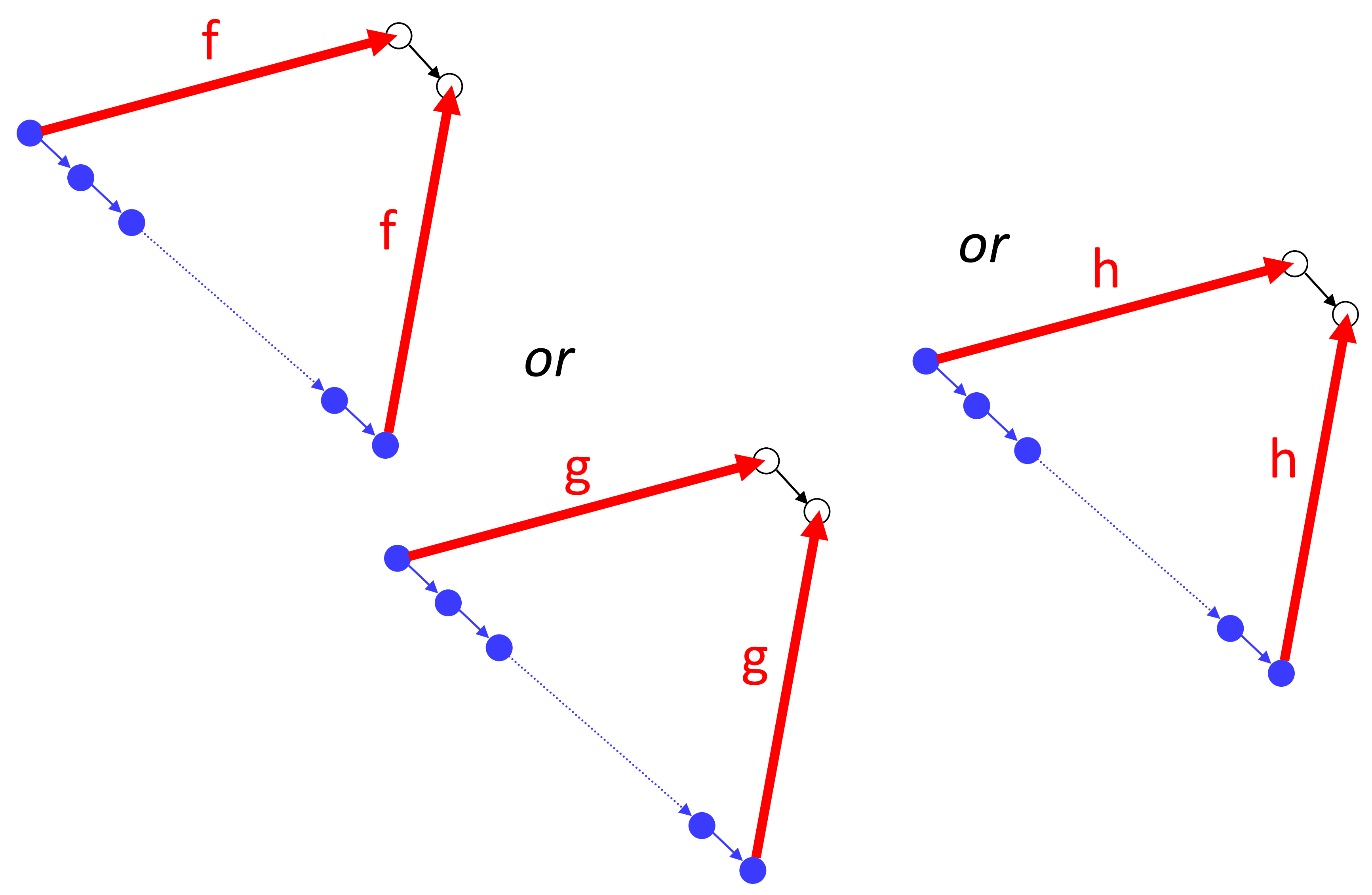
Modular termination arguments



Modular termination arguments



Modular termination arguments



Modular termination arguments

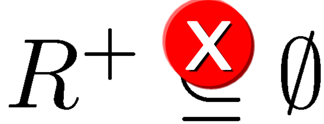
- Modularity gives us freedom when looking for valid arguments

- Strategy: refinement based on failed attempts
  - Start with empty termination argument
  - Check inclusion
  - If inclusion check succeeds, termination has been proved
  - If it fails, synthesize a new ranking function from a counterexample and add it in
  - Go to start

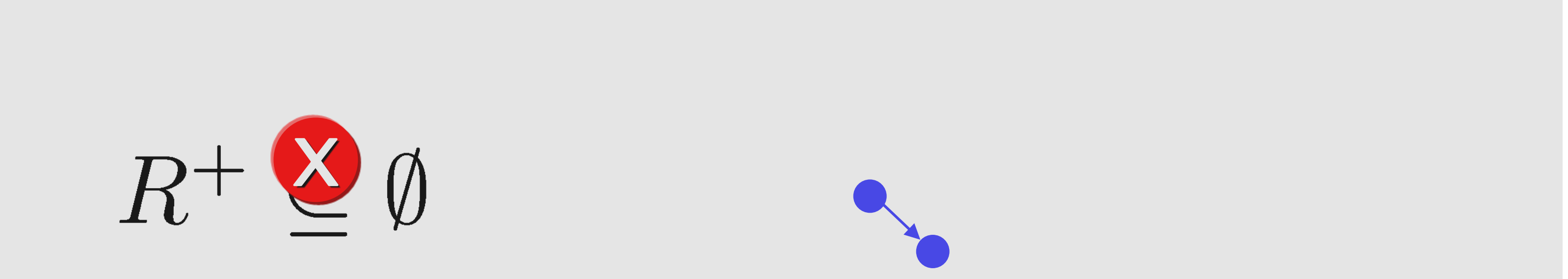
Modular termination arguments

$$R^{+} \subseteq \emptyset$$

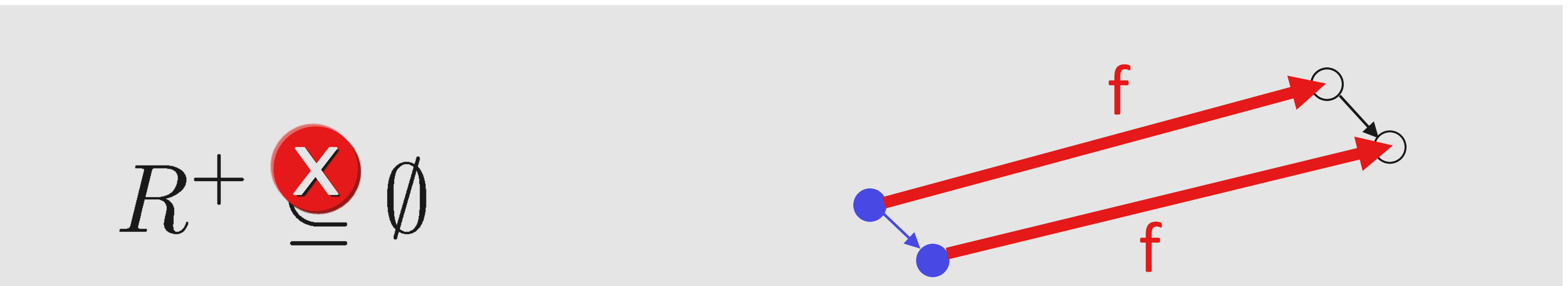
Modular termination arguments



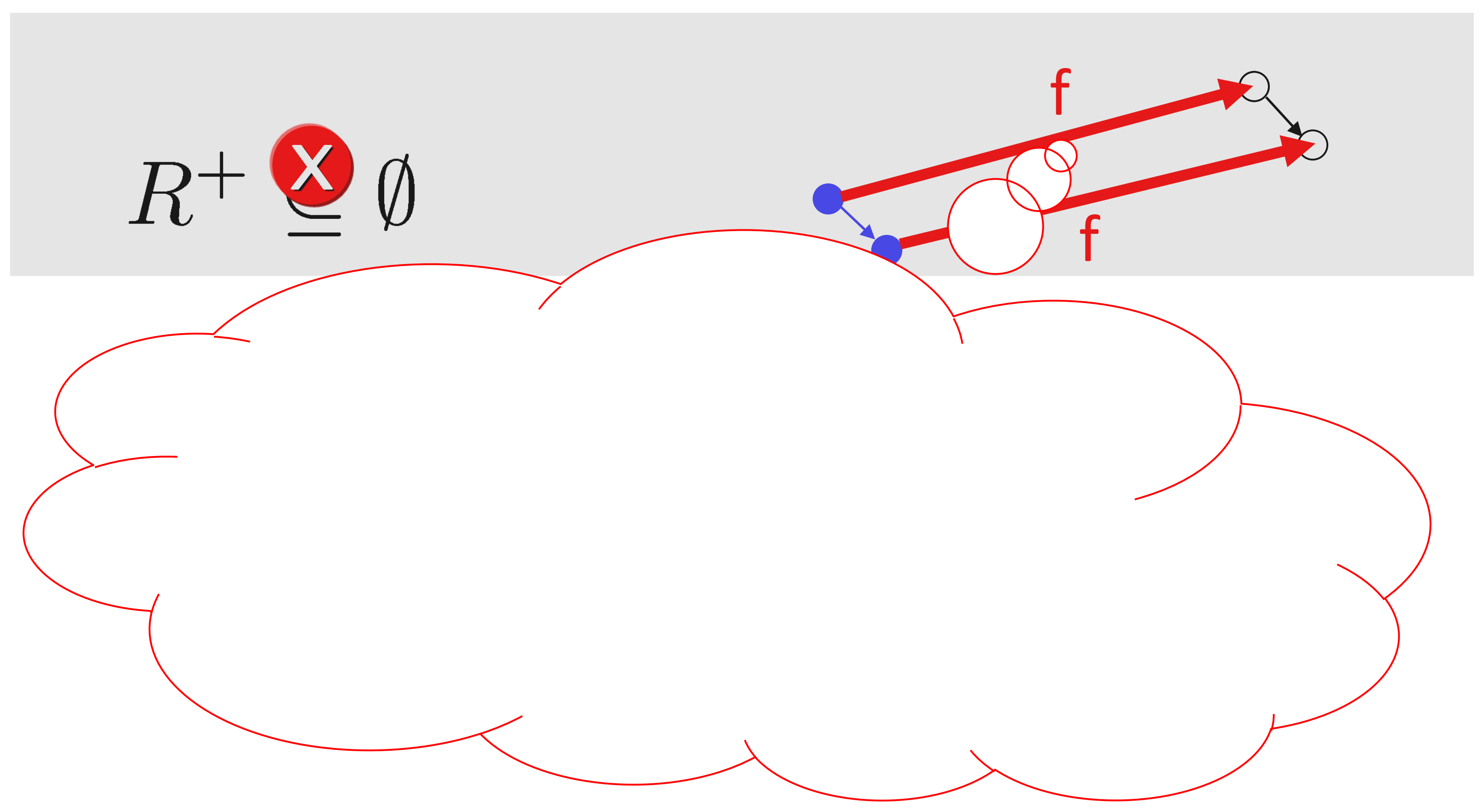
Modular termination arguments



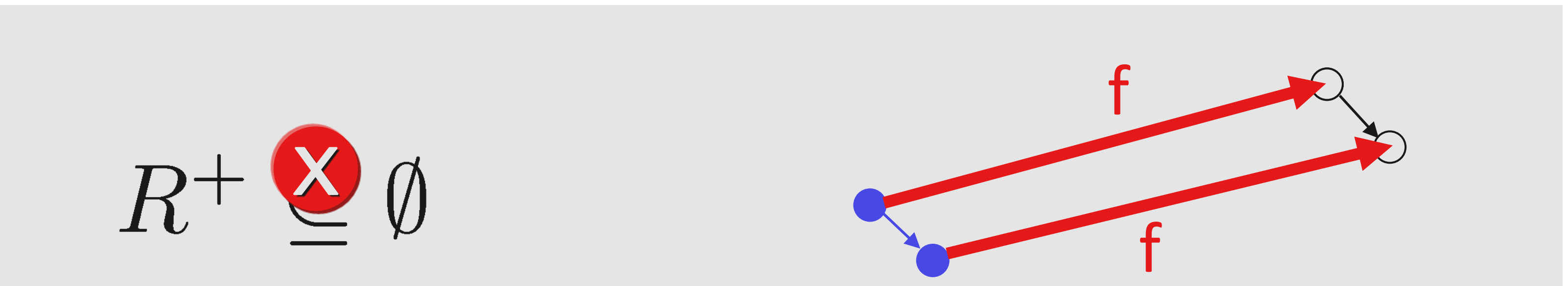
Modular termination arguments



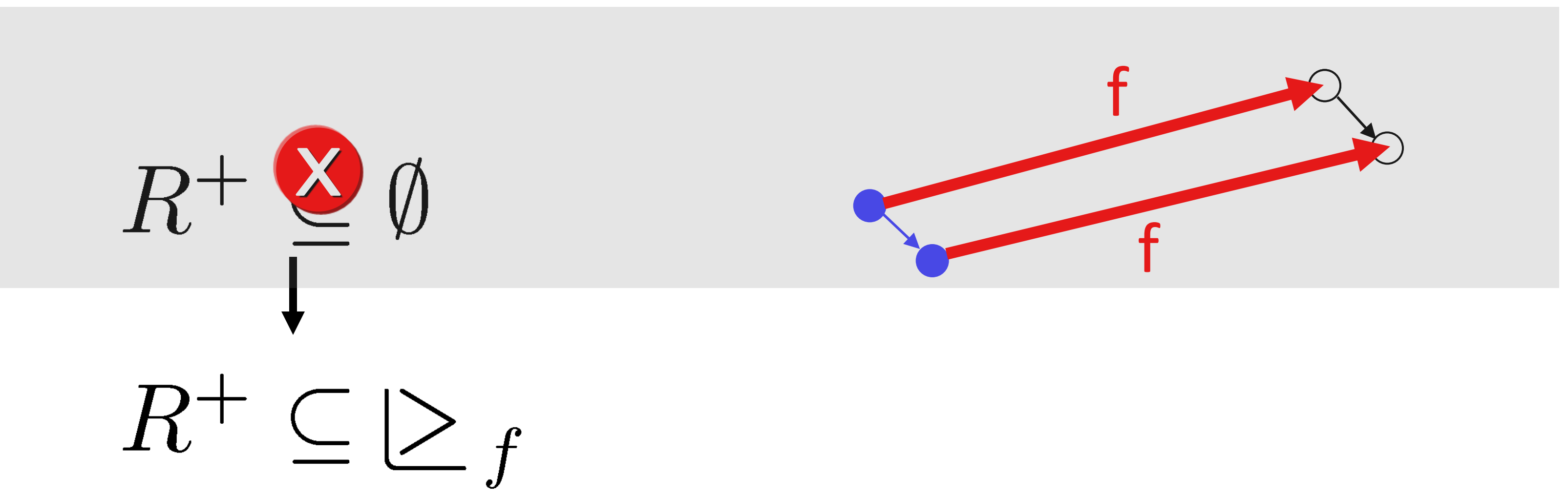
Modular termination arguments



Modular termination arguments



Modular termination arguments



Modular termination arguments

$$R^+ \not\subseteq \emptyset$$

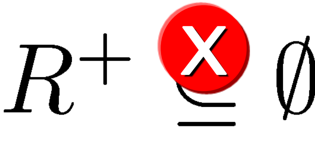
$$R^+ \subseteq \trianglerighteq_f$$

Modular termination arguments

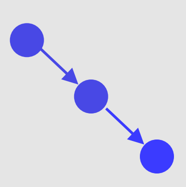
$$R^+ \subseteq \emptyset$$

$$R^+ \subseteq \trianglerighteq_f$$

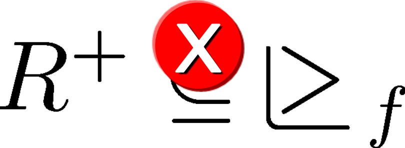
Modular termination arguments

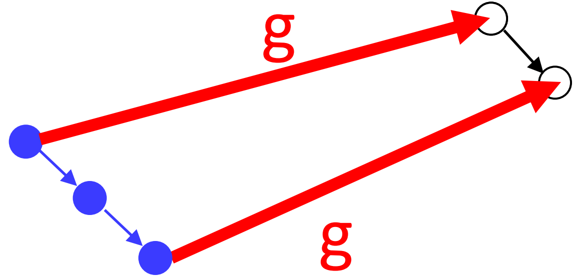


$$R^+ \subseteq \trianglerighteq_f$$

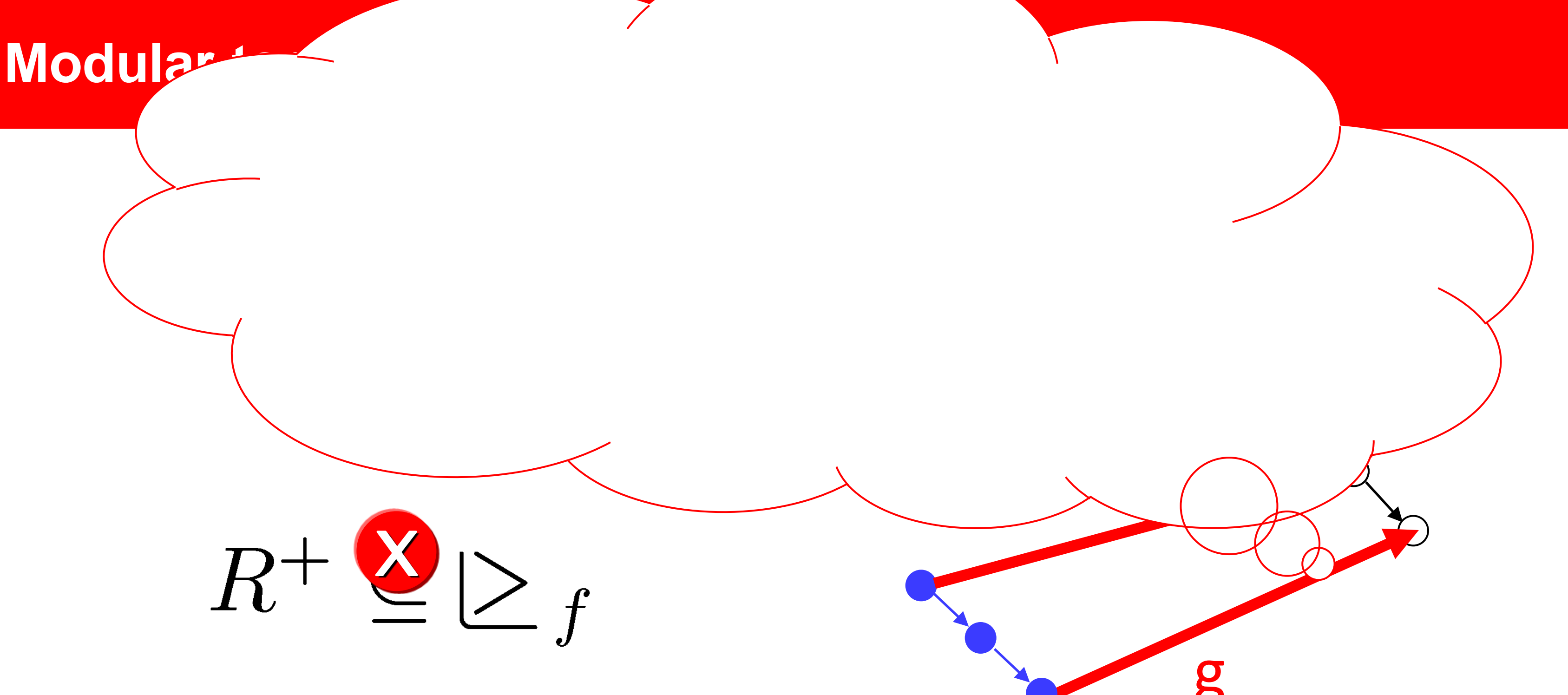


Modular termination arguments





Modular t



$$R^{+} \subseteq \sqsupseteq_{f}$$

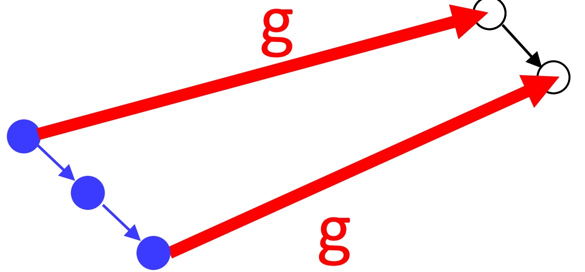
Modular termination arguments

$$R^+ \subseteq \trianglerighteq_f$$

(crossed out with an X mark)

$$\Downarrow$$

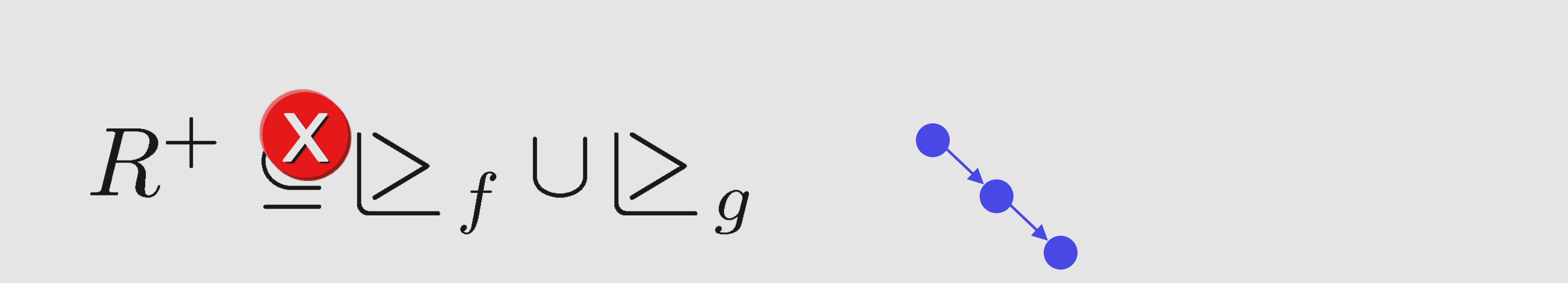
$$R^+ \subseteq \trianglerighteq_f \cup \trianglerighteq_g$$



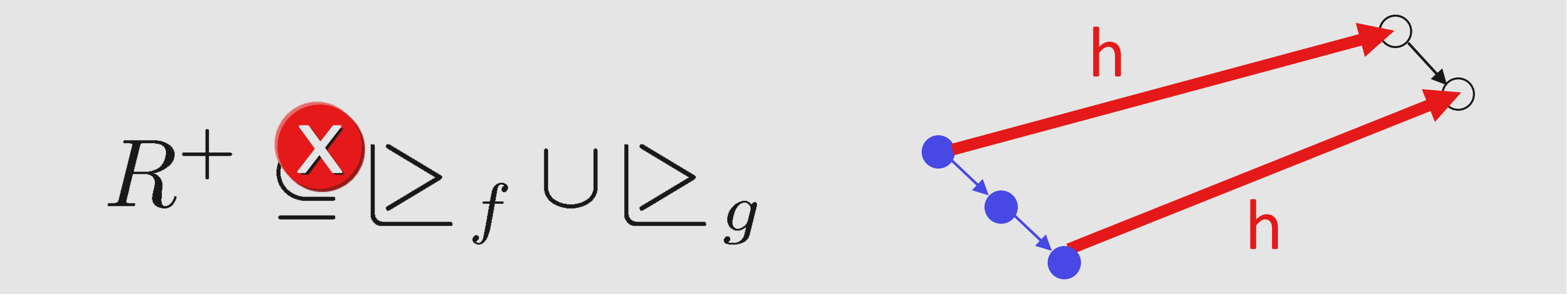
Modular termination arguments

$$R^+ \not\subseteq \trianglerighteq_f \cup \trianglerighteq_g$$

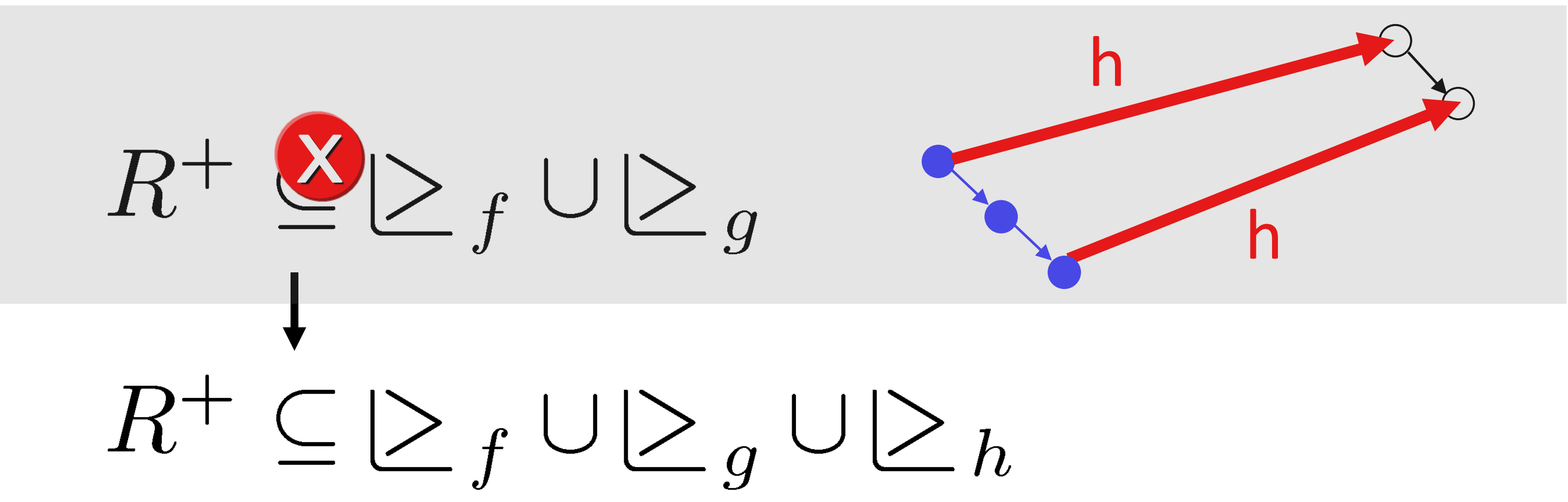
Modular termination arguments



Modular termination arguments



Modular termination arguments



Modular termination arguments

$$R^+ \subseteq \trianglerighteq_f \cup \trianglerighteq_g \cup \trianglerighteq_h$$

Modular termination arguments

$$R^+ \subseteq \trianglerighteq_f \cup \trianglerighteq_g \cup \trianglerighteq_h$$

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is even harder in practice (and undecidable in theory)

$$R^{+} \subseteq T_1 \cup T_2 \cup T_3$$

```
copied = 0;
    .
    .
    .

while(x<y) {
    x = f(x,y);
    g(&y,x);
}
```

```
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T1 || T2 || T3);
    }
```

```
copied = 0;
```

Terminator

$$R^+ \subseteq T_1 \cup T_2 \cup T_3$$

```
copied = 0;
        .
        .
        .

while(x<y) {
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T1 || T2 || T3);
    }

    x = f(x,y);
    g(&y,x);
}
```

$$R^+ \subseteq T_1 \cup T_2 \cup T_3$$

```
copied = 0;
     .
     .
     .

while(x<y) {
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T1 || T2 || T3);
    }

    x = f(x,y);
    g(&y,x);
}
```

Terminator

$$R^+ \subseteq T_1 \cup T_2 \cup T_3$$

```
copied = 0;
     .
     .
     .

while(x<y) {
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T1 || T2 || T3);
    }

    x = f(x,y);
    g(&y,x);
}
```

$$R^+ \subseteq T_1 \cup T_2 \cup T_3$$

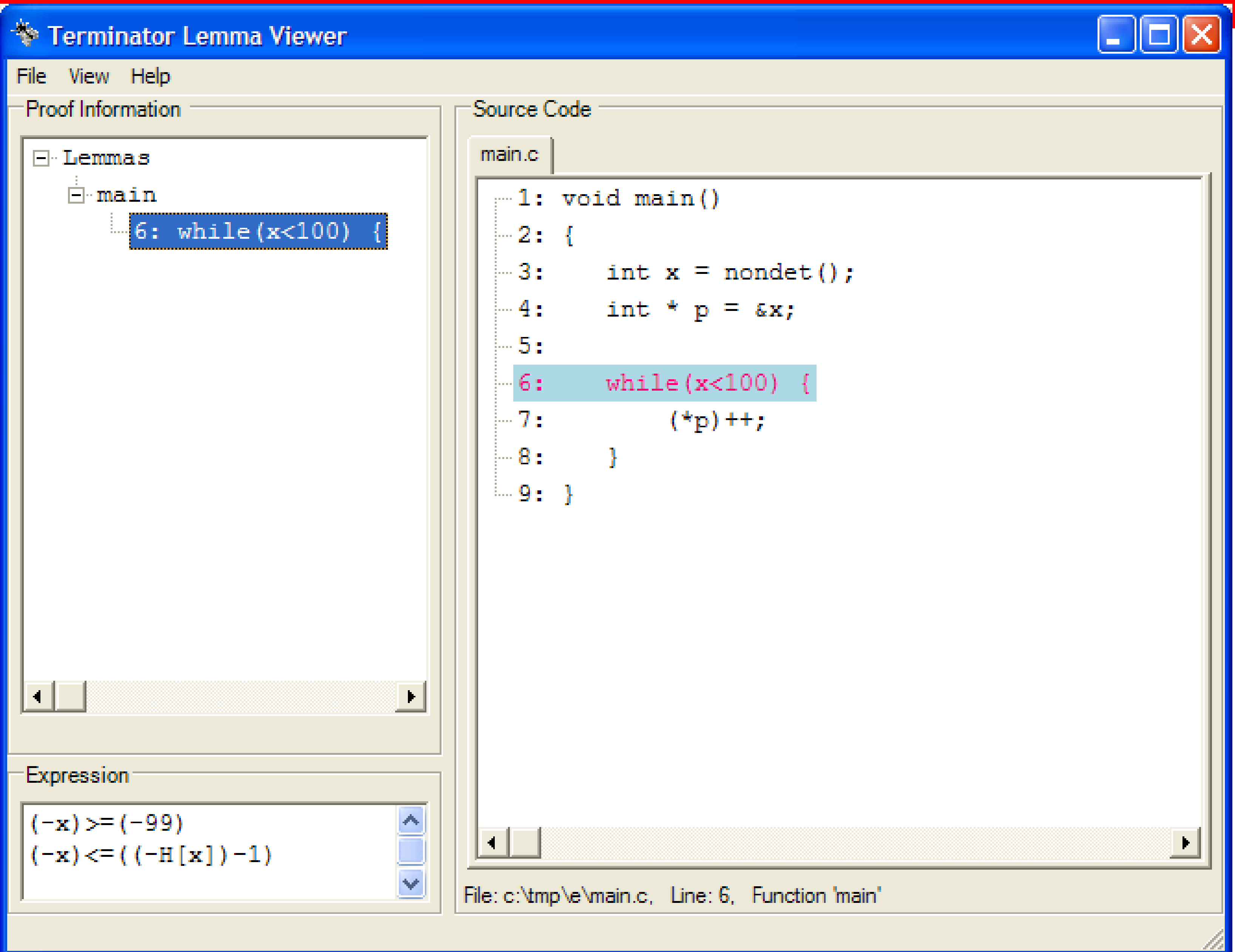
```
copied = 0;
    .
    .
    .

while(x<y) {
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T1 || T2 || T3);
    }

    x = f(x,y);
    g(&y,x);
}
```

Terminator

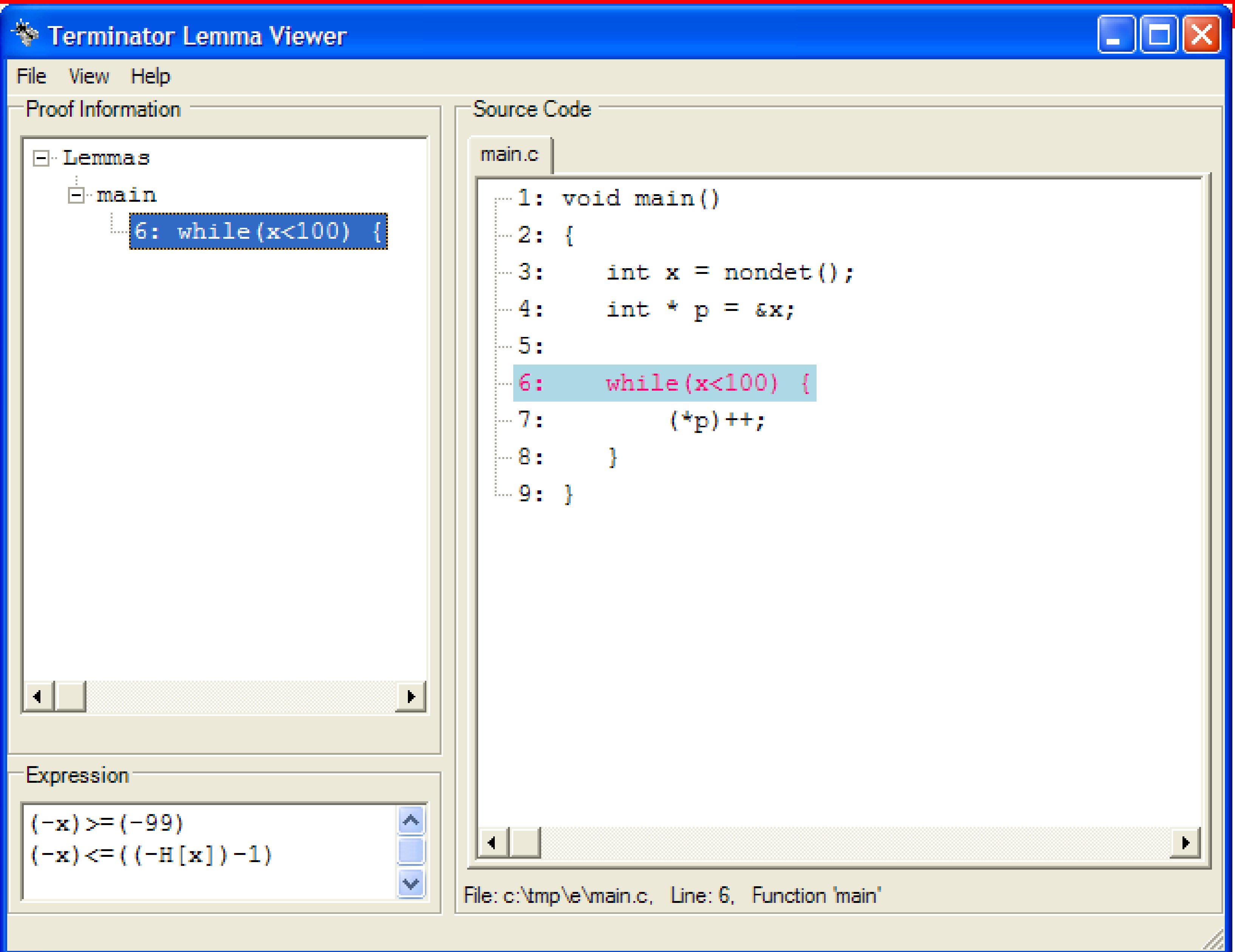
$$R^+ \subseteq T_1 \cup T_2 \cup T_3$$

```
copied = 0;
      .
      .
      .

while(x<y) {
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T1 || T2 || T3);
    }

    x = f(x,y);
    g(&y,x);
}
```

Examples



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'

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'

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

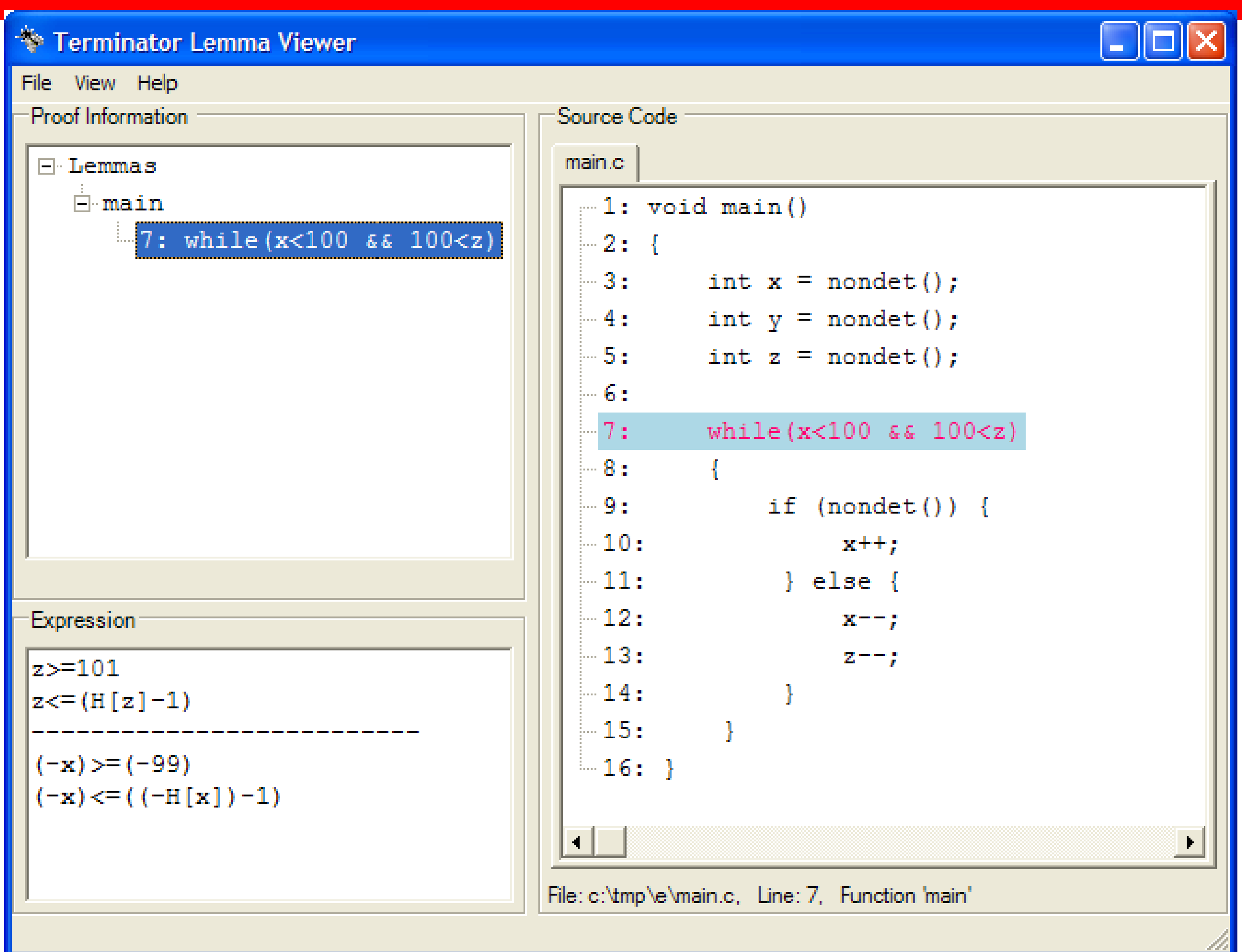
test.c

```
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: }
```

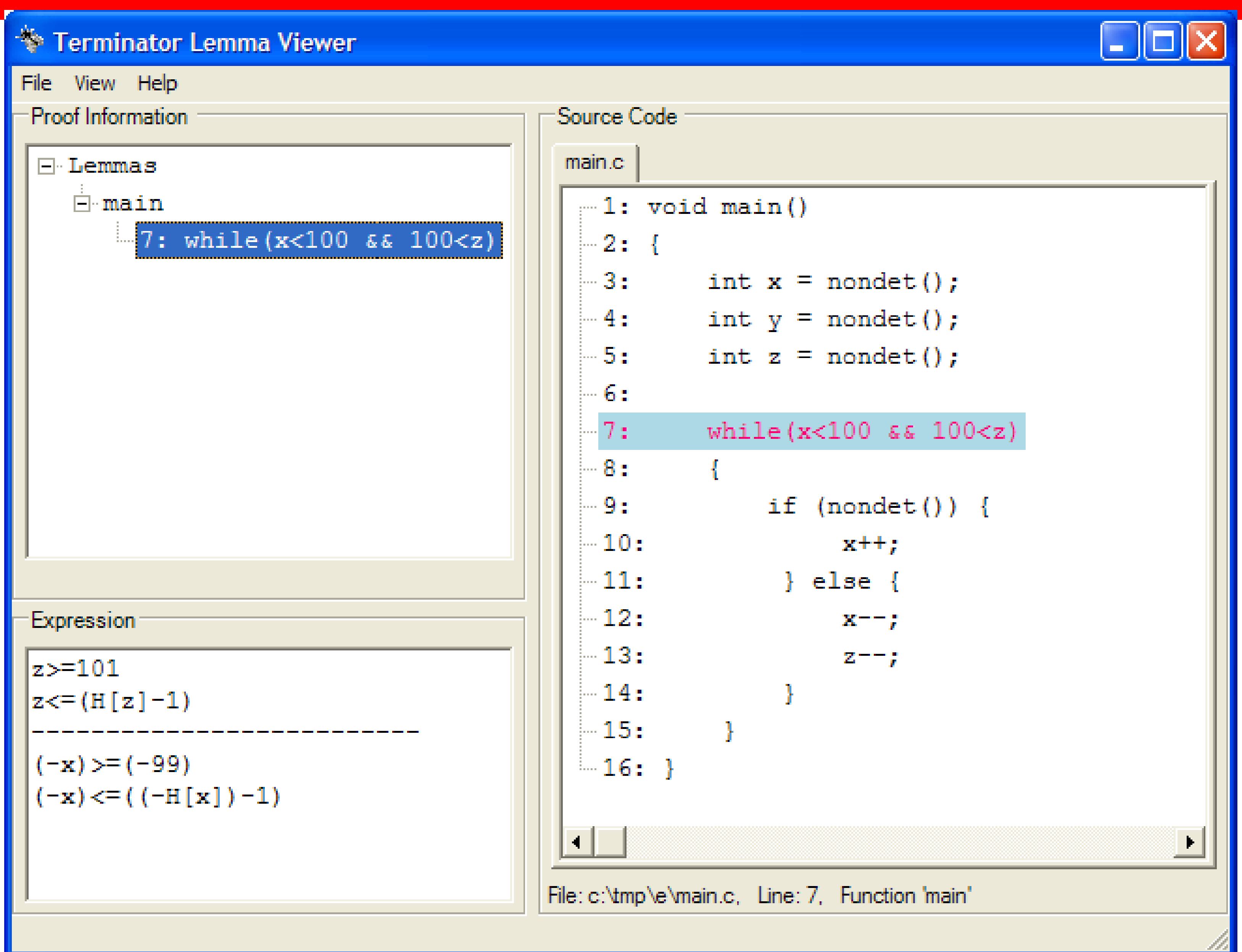
File: c:\slam\src\terminator\demos\d3\test.c, Line: 8, Function 'Ack'

Driver: Rule: Defect: Possibly non-terminating path found

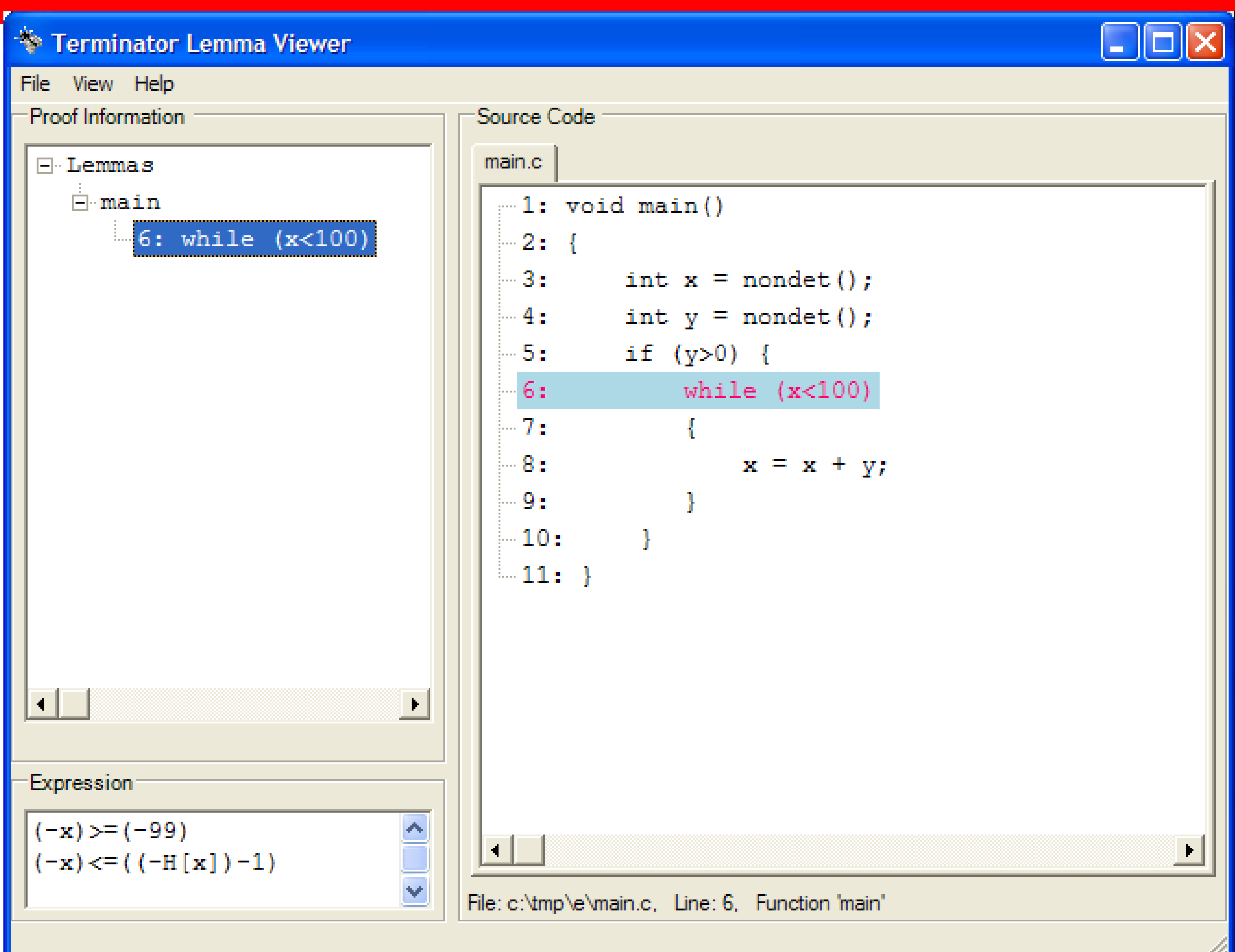
Examples



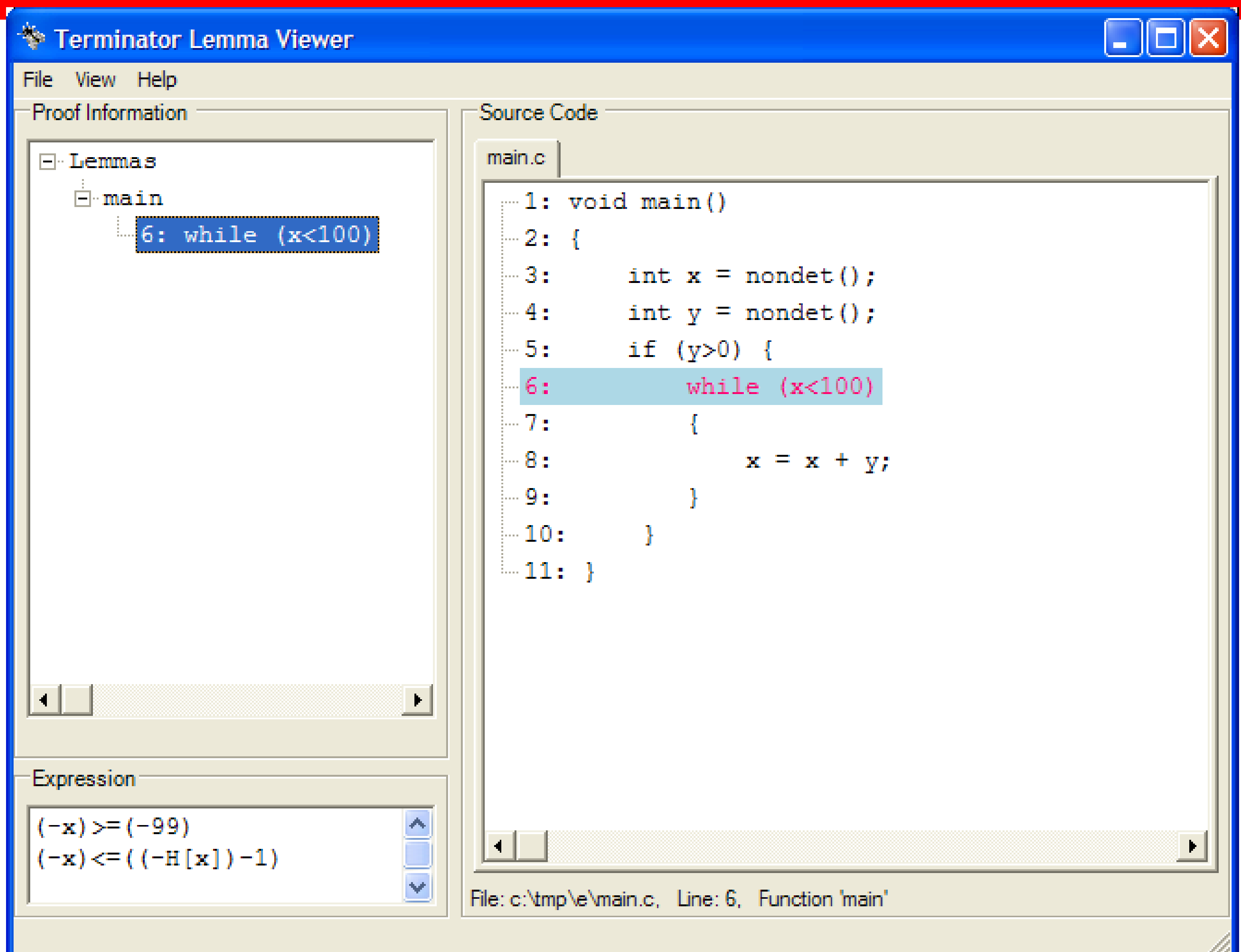
Examples



Examples



Examples



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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the 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) together with accurate support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. In this paper we describe such a tool, called TERMINATOR.

TERMINATOR's most distinguishing aspect, with respect to previous methods and tools for proving program termination, is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. The classical method is to construct an expression defining the *rank* of a state and then to check that its value decreases in every transition from a reachable state to a next one. The construction of the ranking function is the hard part and forms a task that needs to be applied to the whole program. The checking part is relatively easy. In our method, the task of constructing ranking functions is the relatively easy part; they are constructed on demand based on the examination of only a few selected paths through the program.

Furthermore, TERMINATOR is not required to construct only *one* correct termination argument but rather a set of guesses of possible arguments, some of which may be bad guesses. That is, this set need not be the exact set of the 'right' ranking functions but only a *superset*. We find the same monotonicity of the refinement of the termination argument as with iterative abstraction refinement for safety (the set of predicates need not be the exact set of 'right' predicates but only a superset).

Checking the termination argument is the hard part of our method. This is because the termination argument is now a set of ranking functions, not a single ranking function. With a single ranking function one must show that the rank decreases from the pre- to post-state after executing each single transition step. In our setting it is not sufficient to look at a single transition step. Instead, we must consider all *finite sequences of transitions*. We must show that, for every sequence, one of the ranking functions decreases

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

```
nondet();
nondet();
{
le (x<100)
x = x + y;
```

Function 'main'

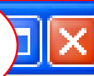
Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

A… M… andrey.…

TERMINATOR
2006



```
nondet();
nondet();
 {
le (x<100)

x = x + y;
```

Function 'main'

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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopOfStack` is … nter to another serial-based device driver). In the case whe… her device driver returns a return-value that indica… cce… t places 0 in `PIoStatusBlock->Informatio`… serial enumeration driver will fail to increment the val… 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) together with accurate support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. In this paper we describe such a tool, called TERMINATOR.

TERMINATOR's most distinguishing aspect, with respect to previous methods and tools for proving program termination, is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. The classical method is to construct an expression defining the *rank* of a state and then to check that its value decreases in every transition from a reachable state to a next one. The construction of the ranking function is the hard part and forms a task that needs to be applied to the whole program. The checking part is relatively easy. In our method, the task of constructing ranking functions is the relatively easy part; they are constructed on demand based on the examination of only a few selected paths through the program.

Furthermore, TERMINATOR is not required to construct only *one* correct termination argument but rather a set of guesses of possible arguments, some of which may be bad guesses. That is, this set need not be the exact set of the 'right' ranking functions but only a *superset*. We find the same monotonicity of the refinement of the termination argument as with iterative abstraction refinement for safety (the set of predicates need not be the exact set of 'right' predicates but only a superset).

Checking the termination argument is the hard part of our method. This is because the termination argument is now a set of ranking functions, not a single ranking function. With a single ranking function one must show that the rank decreases from the pre- to post-state after executing each single transition step. In our setting it is not sufficient to look at a single transition step. Instead, we must consider all *finite sequences of transitions*. We must show that, for every sequence, one of the ranking functions decreases

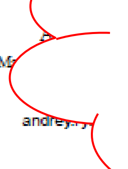
* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

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...

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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006...$5.00.

request packet and `FdoData->TopOfStack` is another serial-based device driver). In the case whe... vice driver returns a return-value that indicat... 0 in `PIoStatusBlock->Informatio`... seri... tion driver will fail to increment the val... pointed to by (line 66), possibly causing the driver to infinitely exec... and not return to its calling context. The consequence ... is that the computer's serial devices could become non-... Worse yet, depending on what actions the other device ... this loop may cause repeated acquiring and releasin... resources (memory, locks, etc) at high priority and exce... ical bus activity. This extra work stresses the operat... the other drivers, and the user applications running on ... which may cause them to crash or become non-respons...

This example demonstrates how a notion of termina... tral to the process of ensuring that reactive systems can... act. Until now no automatic termination tool has eve... to provide a capacity for large program fragments (>2... together with accurate support for programming langu... such as arbitrarily nested loops, pointers, function-poi... effects, etc. In this paper we describe such a tool, called... TOR.

TERMINATOR's most distinguishing aspect, with res... vious methods and tools for proving program terminati... shifts the balance between the two tasks of *constructing*... tively *checking* the termination argument. The classica... to construct an expression defining the *rank* of a state ... check that its value decreases in every transition from ... state to a next one. The construction of the ranking fun... hard part and forms a task that needs to be applied to... program. The checking part is relatively easy. In our ... task of constructing ranking functions is the relatively... they are constructed on demand based on the examina... a few selected paths through the program.

Furthermore, TERMINATOR is not required to co... *one* correct termination argument but rather a set of ... possible arguments, some of which may be bad guess... this set need not be the exact set of the 'right' ranking f... only a *superset*. We find the same monotonicity of the ... of the termination argument as with iterative abstraction ... for safety (the set of predicates need not be the exact s... predicates but only a superset).

Checking the termination argument is the hard ... method. This is because the termination argument is ... of ranking functions, not a single ranking function. W... ranking function one must show that the rank decreas... pre- to post-state after executing each single transition ... setting it is not sufficient to look at a single transition s... we must consider all *finite sequences of transitions*. We... that, for every sequence, one of the ranking function...

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
9	1294	0	5	6K	38
10	158	0	0	8K	21
11	13	0	0	2.5K	6
12	204	0	0	2.5K	16
13	257	1	1	7.5K	26
14	5	0	0	1K	2
15	141	0	1	6.5K	18
16	22	0	0	1.5K	2
17	800	1	6	4K	35
18	1503	1	0	6.5K	31
19	209	0	3	3K	28
20	4099	0	2	10K	63
21	1461	1	4	16K	56
22	114762	0	5	34K	65
23	158746	2	10	35K	75

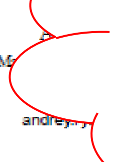
Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.

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...



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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopOfStack` is another serial-based device driver). In the case whe... vice driver returns a return-value that indicat... 0 in `PIoStatusBlock->Informatio`... seri... tion driver will fail to increment the val... pointed to by... (line 66), possibly causing the driver to infinitely execut... and not return to its calling context. The consequence ... is that the computer's serial devices could become non-... Worse yet, depending on what actions the other device ... this loop may cause repeated acquiring and releasin... resources (memory, locks, etc) at high priority and exce... ical bus activity. This extra work stresses the operat... the other drivers, and the user applications running on ... which may cause them to crash or become non-respons...

This example demonstrates how a notion of termina... tral to the process of ensuring that reactive systems can... act. Until now no automatic termination tool has eve... to provide a capacity for large program fragments (>2... together with accurate support for programming langu... such as arbitrarily nested loops, pointers, function-poi... effects, etc. In this paper we describe such a tool, called... TOR.

TERMINATOR's most distinguishing aspect, with res... vious methods and tools for proving program terminati... shifts the balance between the two tasks of *constructing*... tively *checking* the termination argument. The classica... to construct an expression defining the *rank* of a state... check that its value decreases in every transition from... state to a next one. The construction of the ranking fun... hard part and forms a task that needs to be applied to... program. The checking part is relatively easy. In our ... task of constructing ranking functions is the relatively... they are constructed on demand based on the examinat... a few selected paths through the program.

Furthermore, TERMINATOR is not required to con... *one* correct termination argument but rather a set of... possible arguments, some of which may be bad guess... this set need not be the exact set of the 'right' ranking fu... only a *superset*. We find the same monotonicity of the... of the termination argument as with iterative abstraction... for safety (the set of predicates need not be the exact s... predicates but only a superset).

Checking the termination argument is the hard ... method. This is because the termination argument is... of ranking functions, not a single ranking function. W... ranking function one must show that the rank decreas... pre- to post-state after executing each single transition... setting it is not sufficient to look at a single transition st... we must consider all *finite sequences of transitions*. We... that, for every sequence, one of the ranking function...

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

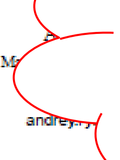
Copyright © 2006 ACM 1-59593-320-4/06/0006... $5.00.

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
9	1294	0	5	6K	38
10	158	0	0	8K	21
11	13	0	0	2.5K	6
12	204	0	0	2.5K	16
13	257	1	1	7.5K	26
14	5	0	0	1K	2
15	141	0	1	6.5K	18
16	22	0	0	1.5K	2
17	800	1	6	4K	35
18	1503	1	0	6.5K	31
19	209	0	3	3K	28
20	4099	0	2	10K	63
21	1461	1	4	16K	56
22	114762	0	5	34K	65
23	158746	2	10	35K	75

X X X X X X X

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.

Termination Proofs for Systems Code *



Byron Cook
Microsoft Research
bycook@microsoft.com

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

A
M
andrey.

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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006... $5.00.

request packet and `FdoData->TopOfStack` is another serial-based device driver). In the case whe vice driver returns a return-value that indicat 0 in `PIoStatusBlock->Informatio` seri tion driver will fail to increment the val pointed to by (line 66), possibly causing the driver to infinitely execu and not return to its calling context. The consequence is that the computer's serial devices could become non- Worse yet, depending on what actions the other device this loop may cause repeated acquiring and releasin resources (memory, locks, etc) at high priority and exce ical bus activity. This extra work stresses the operat the other drivers, and the user applications running on which may cause them to crash or become non-respons

This example demonstrates how a notion of terminat tral to the process of ensuring that reactive systems ca act. Until now no automatic termination tool has eve to provide a capacity for large program fragments (>2 together with accurate support for programming langu such as arbitrarily nested loops, pointers, function-poi effects, etc. In this paper we describe such a tool, called TOR.

TERMINATOR's most distinguishing aspect, with res vious methods and tools for proving program terminati shifts the balance between the two tasks of *constructing* tively *checking* the termination argument. The classica to construct an expression defining the *rank* of a state check that its value decreases in every transition from state to a next one. The construction of the ranking fu hard part and forms a task that needs to be applied to program. The checking part is relatively easy. In our task of constructing ranking functions is the relatively they are constructed on demand based on the examina a few selected paths through the program.

Furthermore, TERMINATOR is not required to co *one* correct termination argument but rather a set of possible arguments, some of which may be bad guess this set need not be the exact set of the 'right' ranking f only a *superset*. We find the same monotonicity of the of the termination argument as with iterative abstraction for safety (the set of predicates need not be the exact s predicates but only a superset).

Checking the termination argument is the hard method. This is because the termination argument is of ranking functions, not a single ranking function. W ranking function one must show that the rank decreas pre- to post-state after executing each single transition setting it is not sufficient to look at a single transition s we must consider all *finite sequences of transitions*. We that, for every sequence, one of the ranking function

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
9	1294	0	5	6K	38
10	158	0	0	8K	21
11	13	0	0	2.5K	6
12	204	0	0	2.5K	16
13	257	1	1	7.5K	26
14	5	0	0	1K	2
15	141	0	1	6.5K	18
16	22	0	0	1.5K	2
17	800	1	6	4K	35
18	1503	1	0	6.5K	31
19	209	0	3	3K	28
20	4099	0	2	10K	63
21	1461	1	4	16K	56
22	114762	0	5	34K	65
23	158746	2	10	35K	75

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.

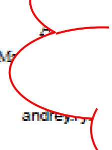
Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

A
M

andrey.r



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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

request packet and `FdoData->TopOfStack` is another serial-based device driver). In the case whe vice driver returns a return-value that indicat 0 in `PIoStatusBlock->Informatio` tion driver will fail to increment the val (line 66), possibly causing the driver to infinitely execu and not return to its calling context. The consequence is that the computer's serial devices could become non- Worse yet, depending on what actions the other device this loop may cause repeated acquiring and releasin resources (memory, locks, etc) at high priority and exce ical bus activity. This extra work stresses the operat the other drivers, and the user applications running on which may cause them to crash or become non-respons

This example demonstrates how a notion of termina tral to the process of ensuring that reactive systems car act. Until now no automatic termination tool has eve to provide a capacity for large program fragments (>2 together with accurate support for programming langu such as arbitrarily nested loops, pointers, function-poi effects, etc. In this paper we describe such a tool, called TOR.

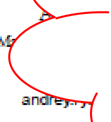
TERMINATOR's most distinguishing aspect, with res vious methods and tools for proving program terminati shifts the balance between the two tasks of *constructing* tively *checking* the termination argument. The classica to construct an expression defining the *rank* of a state check that its value decreases in every transition from state to a next one. The construction of the ranking fun hard part and forms a task that needs to be applied to program. The checking part is relatively easy. In our task of constructing ranking functions is the relatively they are constructed on demand based on the examinat a few selected paths through the program.

Furthermore, TERMINATOR is not required to con *one* correct termination argument but rather a set of possible arguments, some of which may be bad guess this set need not be the exact set of the 'right' ranking fu only a *superset*. We find the same monotonicity of the of the termination argument as with iterative abstraction for safety (the set of predicates need not be the exact s predicates but only a superset).

Checking the termination argument is the hard method. This is because the termination argument is of ranking functions, not a single ranking function. W ranking function one must show that the rank decreas pre- to post-state after executing each single transition setting it is not sufficient to look at a single transition s we must consider all *finite sequences of transitions*. We that, for every sequence, one of the ranking function

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
9	1294	0	5	6K	38
10	158	0	0	8K	21
11	13	0	0	2.5K	6
12	204	0	0	2.5K	16
13	257	1	1	7.5K	26
14	5	0	0	1K	2
15	141	0	1	6.5K	18
16	22	0	0	1.5K	2
17	800	1	6	4K	35
18	1503	1	0	6.5K	31
19	209	0	3	3K	28
20	4099	0	2	10K	63
21	1461	1	4	16K	56
22	114762	0	5	34K	65
23	158746	2	10	35K	75

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.



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...

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*

Categories and Subject D
Engineering—Program Ve
Systems—Reliability

General Terms Reliabili

Keywords Program term
cation, formal verification

1. Introduction

Reactive systems (e.g. oper
database engines, etc) are
ponents that we expect w
functions unexpectedly do
to non-responsive systems
ample, must eventually ret
in Figure 1 which is calle
the Windows serial enumer
serial-based device drivers
kernel routine IoCallDr

* The second and third autho
search Foundation (DFG) as
Research Center "Automatic
tems" (SFB/TR 14 AVACS),
tion and Research (BMBF) in
grant 01 IS C38.

Permission to make digital or har
classroom use is granted without
for profit or commercial advantag
on the first page. To copy otherw
to lists, requires prior specific per
PLDI'06 June 11–14, 2006, Ot
Copyright © 2006 ACM 1-59593

request packet and FdoData->TopOfStack is
another serial-based device driver). In the case whe
vice driver returns a return-value that indicat... succe
0 in PIoStatusBlock->Informatio... seria
tion driver will fail to increment the val... pointed to by
(line 66), possibly causing the driver to infinitely execu
and not return to its calling context. The consequence
is that the computer's serial devices could become non-
Worse yet, depending on what actions the other device d
this loop may cause repeated acquiring and releasin
resources (memory, locks, etc) at high priority and exce
ical bus activity. This extra work stresses the operat
the other drivers, and the user applications running on
which may cause them to crash or become non-respons

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
	1294	0	5	6K	38
	158	0	0	8K	21
	13	0	0	2.5K	6
	204	0	0	2.5K	16
	257	1	1	7.5K	26
	5	0	0	1K	2
	141	0	1	6.5K	18
	22	0	0	1.5K	2
	300	1	6	4K	35
	1503	1	0	6.5K	31
	209	0	3	3K	28
	4099	0	2	10K	63
	1461	1	4	16K	56
	114762	0	5	34K	65
	158746	2	10	35K	75

Results of experiments using an integration of TERMI-
n the Windows Static Driver Verifier[21] product (SDV)
dard 23 Windows OS device drivers used to test SDV.
e driver exports from 5 to 10 dispatch routines, all of
be proved terminating.

?

SOFTWARE

Send in the Terminator

A MICROSOFT TOOL LOOKS FOR PROGRAMS THAT FREEZE UP BY GARY STIX

Alan Turing, the mathematician who was among the founders of computer science, showed in 1936 that it is impossible to devise an algorithm to prove that any given program will always run to completion. The essence of his argument was that such an algorithm can always trip up if it analyzes itself and finds that it is unable to stop. "It leads to a logical paradox," remarks David Schmidt, professor of computer science at Kansas State University. On a pragmatic level, the inability to "terminate," as it is called in computerese, is familiar to any user of the Windows operating system who has clicked a mouse button and then stared indefinitely at the hourglass icon indicating that the program is looping endlessly through the same lines of code.

The current version of Microsoft's operating system, known as XP, is more stable than previous ones. But manufacturers of printers, MP3 players and other devices still write faulty "driver" software that lets the peripheral interact with the operating system. So XP users have not lost familiarity with frozen hourglasses. The research arm of Microsoft has tried recently to address the long-simmering frustration by focusing on tools to check drivers for the absence of bugs.

Microsoft Research has yet to contradict Turing, but it has started presenting papers at conferences on a tool called Terminator that tries to prove that a driver will finish what it is doing. Computer scientists had never succeeded until now in constructing a practical automated verifier for termination of large programs because of the ghost of Turing, asserts Byron Cook, a theoretical computer scientist at Microsoft Research's laboratory in Cambridge, England, who led the project. "Turing proved that the problem was undecidable, and in some sense, that scared people off," he says.

Blending several previous techniques for automated program analysis, Terminator creates a finite representation of the infinite number of states that a driver could occupy while executing a program. It then attempts to derive a logical argument that shows that the software will finish its task. It does this by combining multiple "ranking functions," which measure how far a device driver has progressed through the loops in a program, sequences of instructions that rerun until a specified condition is met. Terminator begins with an initial, rather weak argument that it refines repeatedly based on information learned from previous failed attempts at creating a proof (a sufficiently strong argument). The procedure may consume hours on a powerful computer until, if everything goes according to plan, a proof emerges that shows that no execution pathway in the driver will cause the dreaded hourglassing.

Terminator, which has been operating for only nine months and has yet to be distributed to outside developers of Windows device drivers, has turned up a few termination bugs in drivers for the soon-to-be-released Vista version of Windows while trying to come up with a proof. Cook predicts that Terminator may eventually find proofs for 99.9 percent of commercial programs that finish executing. (Of course, some programs are designed to run forever.) Turing, however, can still rest in peace. "There will always be an input to Terminator that you can't prove will terminate," Cook says. "But if you can make Terminator work for any program in the real world, then it doesn't really matter."

Patrick Cousot of the École Normale Supérieure in Paris, a pioneer in mathematical program analysis, notes that Terminator should work for a limited set of well-defined applications. "I doubt, for example, that Terminator is able to handle mathematically hard termination problems"—those for floating-point numbers or programs that run at the same time. Cook does not disagree, saying that he plans to develop termination proof methods for such programs. Finding a way to ensure that more complex programs do not freeze is such a difficult challenge, however, that Cook thinks it could consume the rest of his career.

ALAN TURING created a mathematical proof that explains the uncertainty of any computer program ever completing a task.

COMPUTER ENTOMOPHOBIA

Worldwide, software bugs cost billions of dollars in losses every year, which explains a trend among companies for automated program verification. In 2005 Microsoft released an automated bug-catching program, Static Driver Verifier, that checks the source code for device drivers against a mathematical model to determine whether it deviates from its expected behavior.

Static verifiers look for programming errors that cause a program to stop its execution. A device driver, for instance, should never interact with program B before it has done so with program A, or it will simply cease operation. Terminator, Microsoft's latest tool, looks for mistakes that may lead a program to continue running forever in an endless loop, thereby preventing it from finishing the job at hand.

COPYRIGHT 2006 SCIENTIFIC AMERICAN, INC.

Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
12	0	1	1K	3
8	0	0	1K	8
410	0	1	8K	26
1475	0	1	7.5K	24
123292	1	11	5.5K	50
196	1	3	5K	29
4174	0	0	8K	23
210	0	11	5K	27
1294	0	5	6K	38
158	0	0	8K	21
13	0	0	2.5K	6
204	0	0	2.5K	16
257	1	1	7.5K	26
5	0	0	1K	2
141	0	1	6.5K	18
22	0	0	1.5K	2
300	1	6	4K	35
1503	1	0	6.5K	31
209	0	3	3K	28
4099	0	2	10K	63
1461	1	4	16K	56
114762	0	5	34K	65
158746	2	10	35K	75

Results of experiments using an integration of TERMI-
h the Windows Static Driver Verifier[21] product (SDV)
dard 23 Windows OS device drivers used to test SDV.
e driver exports from 5 to 10 dispatch routines, all of
be proved terminating.

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Andre
Max-Planck-I

rybal@r
andrey.ry

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

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

request packet and `FdoData->TopO`
another serial-based device driver). In t
vice driver returns a return-value that i
0 in `PIoStatusBlock->Informa`
tion driver will fail to increment the val
(line 66), possibly causing the driver to
and not return to its calling context. Th
is that the computer's serial devices cou
Worse yet, depending on what actions t
this loop may cause repeated acquirin
resources (memory, locks, etc) at high p
ical bus activity. This extra work stre
the other drivers, and the user applicati
which may cause them to crash or becc

This example demonstrates how a n
tral to the process of ensuring that react
act. Until now no automatic terminati
to provide a capacity for large program
together with accurate support for prog
such as arbitrarily nested loops, pointe
effects, etc. In this paper we describe su
TOR.

TERMINATOR's most distinguishing
vious methods and tools for proving pro
shifts the balance between the two tasks
tively *checking* the termination argume
to construct an expression defining the
check that its value decreases in every
state to a next one. The construction of
hard part and forms a task that needs
program. The checking part is relativel
task of constructing ranking functions
they are constructed on demand based
a few selected paths through the progra

Furthermore, TERMINATOR is not
one correct termination argument but
possible arguments, some of which ma
this set need not be the exact set of the
only a *superset*. We find the same mon
of the termination argument as with iter
for safety (the set of predicates need no
predicates but only a superset).

Checking the termination argumen
method. This is because the terminati
of ranking functions, not a single ranki
ranking function one must show that t
pre- to post-state after executing each s
setting it is not sufficient to look at a si
we must consider all *finite sequences of*
that, for every sequence, one of the ra

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

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

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a Windows device driver. Does this loop guarantee termination? It's *supposed to*: failure of this loop to terminate would have catastrophic effects on the stability and responsiveness of the computer. Why would it be a problem if this loop didn't terminate? First of all, the device that this code is managing would cease to function. Secondly, due to the fact that this code executes at kernel-level priority, non-termination would cause it to starve other threads running on the system. Note that we cannot simply kill the thread, as it can be holding kernel locks and modifying kernel-level data-structures—forcibly killing the thread would leave the operating system in an inconsistent state. Furthermore, if the loop hangs, the machine might not actually crash.[3] Instead, the thread will likely just hang until the user resets the machine. This means that the bug cannot be diagnosed using post-crash analysis tools.

This example highlights the importance of termination in systems level code: in order to improve the responsiveness and stability of the operating system it is vital that we can automatically check the termination of loops like this one. In this case, in order to prove the termination of the loop, we need to show the following conditions:

1. `DeviceExtension->ReadQueue.Flink` is a pointer to a circular list of elements (via the `Flink` field).

[3] Although hanging kernel-threads can trigger other bugs within the operating system.

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Andre
Max-Planck-I

rybal@r
andrey.ry

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 a method of constructing program variance analyses from invariance analyses. If we change the underlying invariance analysis, we get a different variance analysis. We describe several applications of the method, including variance analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variance analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today's state-of-the-art termination provers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Program Analysis, Liveness, Termination

1. Introduction

An *invariance analysis* takes in a program as input and infers a set of possibly disjunctive invariance assertions (*a.k.a.*, invariants) that is indexed by program locations. Each location ℓ in the program has an invariant that always holds during any execution at ℓ. These invariants can serve many purposes. They might be used directly to prove safety properties of programs. Or they might be used indirectly, for example, to aid the construction of abstract transition relations during symbolic software model checking [29]. If a desired safety property is not directly provable from a given invariant, 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 previous state that was also at location ℓ. Note that a single variance assertion may itself be a disjunction. We present a generic method of constructing variance analyses from invariance analyses. For each invariance analysis, we can construct what we call its *induced variance analysis*.

This paper also introduces a condition on variance assertions called the *local termination predicate*. In this work, we show how the variance assertions inferred during our analysis can be used to establish local termination predicates. If this predicate can be established for each variance assertion inferred for a program, whole program termination has been proved; the correctness of this step relies on a result from [37] on *disjunctively well-founded overapproximations*. Analogously to invariance analysis, even if the induced variance analysis fails to prove whole program termination, it can still produce useful information. If the predicate can be established only for some subset of the variance assertions, this induces a different liveness property that holds of the program. Moreover, the information inferred can be used by other termination provers based on disjunctive well-foundedness, such as TERMINATOR [14]. If the underlying invariance analysis is based on abstract interpretation, the user or algorithm could use the same abstraction refinement techniques that are available for invariance analyses.

In this paper we illustrate the utility of our approach with three induced variance analyses. We construct a variance analysis for arithmetic programs based on the Octagon abstract domain [34]. The invariance analysis used as input to our algorithm is composed of a standard analysis based on Octagon, and a post-analysis phase that recovers some disjunctive information. This gives rise to a fast and yet surprisingly accurate termination prover. We similarly construct an induced variance analysis based on the domain of Polyhedra [23]. Finally, we show that an induced variance analysis based on the separation domain [24] is an improvement on a termination prover that was recently described in the literature [3]. These three

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

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

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a Windows device driver. Does this loop guarantee termination? It's *supposed to*: failure of this loop to terminate would have catastrophic effects on the stability and responsiveness of the computer. Why would it be a problem if this loop didn't terminate? First of all, the device that this code is managing would cease to function. Secondly, due to the fact that this code executes at kernel-level priority, non-termination would cause it to starve other threads running on the system. Note that we cannot simply kill the thread, as it can be holding kernel locks and modifying kernel-level data-structures—forcibly killing the thread would leave the operating system in an inconsistent state. Furthermore, if the loop hangs, the machine might not actually crash.[3] Instead, the thread will likely just hang until the user resets the machine. This means that the bug cannot be diagnosed using post-crash analysis tools.

This example highlights the importance of termination in systems level code: in order to improve the responsiveness and stability of the operating system it is vital that we can automatically check the termination of loops like this one. In this case, in order to prove the termination of the loop, we need to show the following conditions:

1. `DeviceExtension->ReadQueue.Flink` is a pointer to a circular list of elements (via the `Flink` field).

[3] Although hanging kernel-threads can trigger other bugs within the operating system.

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Andre
Max-Planck-I
rybal@r
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

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

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 a method of constructing program variance analyses from invariance analyses. If we change the underlying invariance analysis, we get a different variance analysis. We describe several applications of the method, including variance analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variance analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today's state-of-the-art termination provers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Program Analysis, Liveness, Termination

1. Introduction

An *invariance analysis* takes in a program as input and infers a set of possibly disjunctive invariance assertions (*a.k.a.*, invariants) that is indexed by program locations. Each location ℓ in the program has an invariant that always holds during any execution at ℓ. These invariants can serve many purposes. They might be used directly to prove safety properties of programs. Or they might be used indirectly, for example, to aid the construction of abstract transition relations during symbolic software model checking [29]. If a desired safety property is not directly provable from a given invariant,

the user (or algorithm calling the invariance analysis) mi refine the abstraction. For example, if the tool is based o interpretation they may choose to improve the abstraction ing the widening operation [28], using dynamic partition employing a different abstract domain, etc.

The aim of this paper is to develop an analogous se for program termination and liveness: we introduce a clas called *variance analyses* which infer assertions, called *assertions*, that hold between any state at a location ℓ previous state that was also at location ℓ. Note that a single assertion may itself be a disjunction. We present a generi of constructing variance analyses from invariance anal each invariance analysis, we can construct what we call it *variance analysis*.

This paper also introduces a condition on variance a called the *local termination predicate*. In this work, we s the variance assertions inferred during our analysis can b establish local termination predicates. If this predicate c tablished for each variance assertion inferred for a progra program termination has been proved; the correctness of relies on a result from [37] on *disjunctively well-found approximations*. Analogously to invariance analysis, even duced variance analysis fails to prove whole program ter it can still produce useful information. If the predicate can lished only for some subset of the variance assertions, thi a different liveness property that holds of the program. M the information inferred can be used by other termination based on disjunctive well-foundedness, such as TERMINAT If the underlying invariance analysis is based on abstract tation, the user or algorithm could use the same abstracti ment techniques that are available for invariance analyses

In this paper we illustrate the utility of our approach w induced variance analyses. We construct a variance ana arithmetic programs based on the Octagon abstract dom The invariance analysis used as input to our algorithm is c of a standard analysis based on Octagon, and a post-analy that recovers some disjunctive information. This gives ris and yet surprisingly accurate termination prover. We simil struct an induced variance analysis based on the domain o dra [23]. Finally, we show that an induced variance analy on the separation domain [24] is an improvement on a te prover that was recently described in the literature [3]. Th

Permission to make digital or hard copies of all or part of this work for personal or

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Andrey Rybalchenko
EPFL and MPI-Saarbrücken
rybal@mpi-sb.mpg.de

Moshe Y. Vardi
Rice University
vardi@cs.rice.edu

Abstract

In recent years we have seen great progress made in the area of automatic source-level static analysis tools. However, most of today's program verification tools are limited to properties that guarantee the absence of bad events (*safety properties*). Until now no formal software analysis tool has provided fully automatic support for proving properties that ensure that good events eventually happen (*liveness properties*). In this paper we present such a tool, which handles liveness properties of large systems written in C. Liveness properties are described in an extension of the specification language used in the SDV system. We have used the tool to automatically prove critical liveness properties of Windows device drivers and found several previously unknown liveness bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Liveness, Termination

1. Introduction

As computer systems become ubiquitous, expectations of system dependability are rising. To address the need for improved software quality, practitioners are now beginning to use static analysis and automatic formal verification tools. However, most of software verification tools are currently limited to *safety properties* [2, 3] (see Section 5 for discussion). No software analysis tool offers fully automatic scalable support for the remaining set of properties: *liveness properties*.

Windows kernel APIs that acquire resources and APIs that release resources. For example:

> *A device driver should never call* `KeReleaseSpinlock` *unless it has already called* `KeAcquireSpinlock`.

This is a safety property for the reason that any counterexample to the property will be a finite execution through the device driver code. We can think of safety properties as guaranteeing that specified bad events will not happen (*i.e.* calling `KeReleaseSpinlock` before calling `KeAcquireSpinlock`). Note that SDV cannot check the equally important related liveness property:

> *If a driver calls* `KeAcquireSpinlock` *then it must eventually make a call to* `KeReleaseSpinlock`.

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace in which `KeAcquireSpinlock` is called but it is not followed by a call to `KeReleaseSpinlock`. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will eventually happen (*i.e.* that `KeReleaseSpinlock` will eventually be called in the case that a call to `KeAcquireSpinlock` occurs).

Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: "`f(); g(); h();`". It is easy to prove that the function `f` is always called before `h`: in this case we need only to look at the structure of the control-flow graph. It is much harder to prove that `h` is eventually called after `f`: we first have to prove the termination of `g`. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Andre
Max-Planck-I
rybal@r
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

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

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

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Zvonimir Rakamarić
Department of Computer Science, University of British Columbia
{babic,ajh,zrakamar}@cs.ubc.ca

Byron Cook
Microsoft Research
bycook@microsoft.com

Abstract

We describe a simple and efficient algorithm for proving the termination of a class of loops with nonlinear assignments to variables. The method is based on divergence testing for each variable in the cone-of-influence of the loop's termination condition. The analysis allows us to automatically prove the termination of loops that cannot be handled using previous techniques. The paper closes with experimental results using short examples drawn from industrial code.

1 Introduction

From the very beginnings of the formal analysis of software [12, 14], the task of formally verifying the correctness of a program has been decomposed into the tasks of proving correctness *if* the program terminates, and separately proving termination. Deciding termination, in general, is obviously undecidable, but thanks to considerable research progress over the years (e.g., [9, 20, 5, 23, 3, 6, 13, 4, 16, 18, 21, 8, 7]), a variety of techniques and heuristics can now automatically prove termination of many loops that occur in practice.

```
while (x < y) {
  x = pow(x,3) - 2*pow(x,2) - x + 2;
}
```

This paper outlines a new proof procedure for cases of this sort. Using combination techniques described in [1] and [2], our intention for this proposed procedure is to be combined with the existing termination analysis techniques—making future termination provers a little less temperamental.

The proposed technique is based on divergence testing: the transition system of each program variable is independently examined for divergence to plus- or minus-infinity. The approach is limited to loops containing only polynomial update expressions with finite degree, allowing highly efficient computation of certain regions that guarantee divergence. Like all automated termination provers, the technique can't handle all loops. However, it is very fast, it is sound, and it can prove termination in cases that previously could not be handled or could be handled only by a much more expensive analysis. Our hope is that, in practice, this restricted analysis (and some extensions) will handle the termination of the majority of loops in which a nonlinear analysis is required. In our investigations, we have found that this simple type of nonlinear loop appears in industrial numerical computations and nonlinear digital fil-

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Andrey Rybalchenko
EPFL and MPI-Saarbrücken
rybal@mpi-sb.mpg.de

Moshe Y. Vardi
Rice University
vardi@cs.rice.edu

Abstract

In recent years we have seen great progress made in the area of automatic source-level static analysis tools. However, most of today's program verification tools are limited to properties that guarantee the absence of bad events (*safety properties*). Until now no formal software analysis tool has provided fully automatic support for proving properties that ensure that good events eventually happen (*liveness properties*). In this paper we present such a tool, which handles liveness properties of large systems written in C. Liveness properties are described in an extension of the specification language used in the SDV system. We have used the tool to automatically prove critical liveness properties of Windows device drivers and found several previously unknown liveness bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Liveness, Termination

1. Introduction

As computer systems become ubiquitous, expectations of system dependability are rising. To address the need for improved software quality, practitioners are now beginning to use static analysis and automatic formal verification tools. However, most of software verification tools are currently limited to *safety properties* [2, 3] (see Section 5 for discussion). No software analysis tool offers fully automatic scalable support for the remaining set of properties: *liveness properties*.

Windows kernel APIs that acquire resources and APIs that release resources. For example:

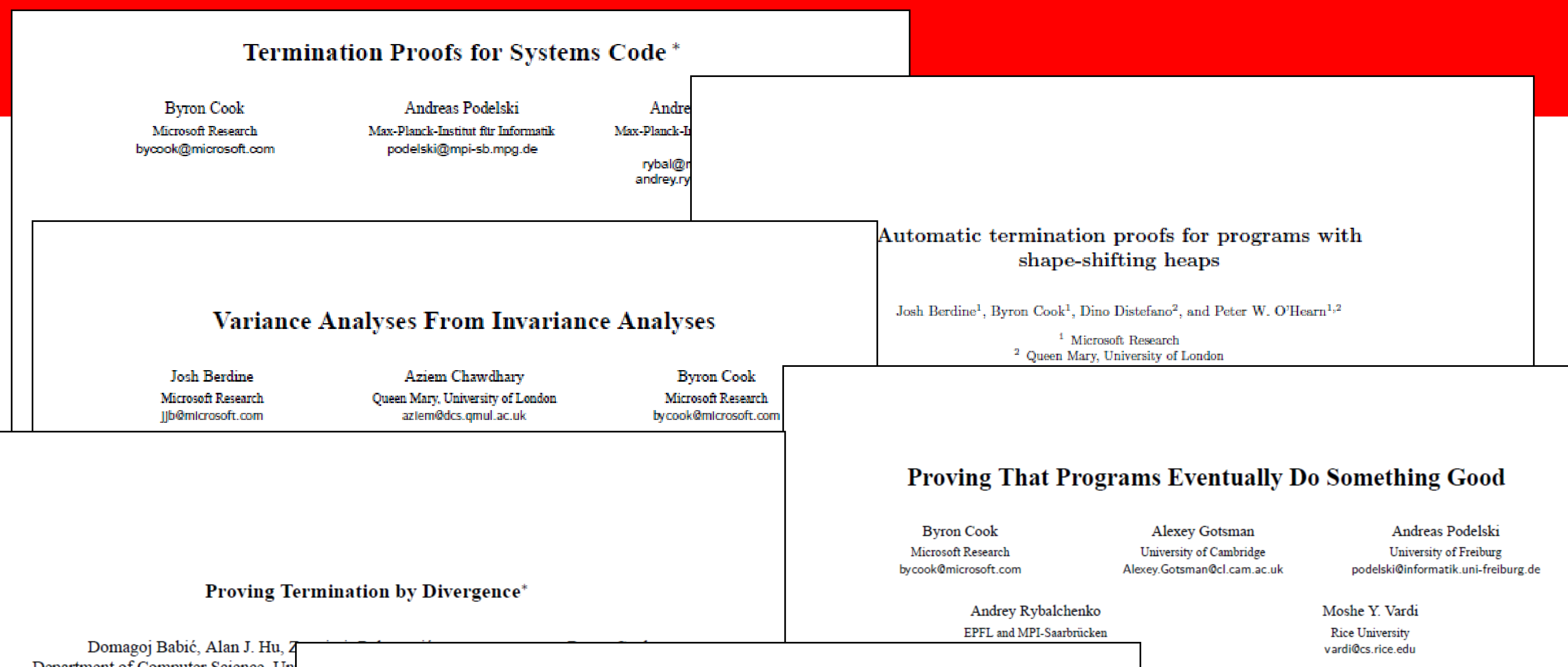
> *A device driver should never call* KeReleaseSpinlock *unless it has already called* KeAcquireSpinlock.

This is a safety property for the reason that any counterexample to the property will be a finite execution through the device driver code. We can think of safety properties as guaranteeing that specified bad events will not happen (*i.e.* calling KeReleaseSpinlock before calling KeAcquireSpinlock). Note that SDV cannot check the equally important related liveness property:

> *If a driver calls* KeAcquireSpinlock *then it must eventually make a call to* KeReleaseSpinlock.

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace in which KeAcquireSpinlock is called but it is not followed by a call to KeReleaseSpinlock. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will eventually happen (*i.e.* that KeReleaseSpinlock will eventually be called in the case that a call to KeAcquireSpinlock occurs).

Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: "f(); g(); h();". It is easy to prove that the function f is always called before h: in this case we need only to look at the structure of the control-flow graph. It is much harder to prove that h is eventually called after f: we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:



Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

Abstract

Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Examples of such cases can be found in operating systems, web servers, e-mail clients, etc. Unfortunately, no known automatic program termination prover supports a practical method of proving the termination of threads. In this paper we describe such a procedure. The procedure's scalability is achieved through the use of environment models that abstract away the surrounding threads. The procedure's accuracy is due to a novel method of incrementally constructing environment abstractions. Our method finds the conditions that a thread requires of its environment in order to establish termination by looking at the conditions necessary to prove that certain paths through the thread represent well-founded relations if executed *in isolation of the other threads*. The paper gives a description of experimental results using an implementation of our procedure on Windows device drivers, and a description of a previously unknown bug found with the tool.


```
KeAcquireSpinLock(&Ext->SpinLock, &irql);

do {
  irp = DequeueReadByFileObject(Ext, FileObject);
  if (irp) {
    irp->IoStatus.Status = STATUS_CANCELLED;
    irp->IoStatus.Information = 0;

    InsertTailList (&listHead,LinkPtr(irp));
  }
} while (irp != NULL);

KeReleaseSpinLock(&Ext->SpinLock, irql);
```

Figure 1. Code fragment from a keyboard device driver whose termination partially depends on the correct behavior of other threads from the driver.

ple, is a demonstration of this problem. This loop, which comes

Abstract

We describe a simple and efficient algorithm fo the termination of a class of loops with nonlinea ments to variables. The method is based on diverg ing for each variable in the cone-of-influence of t termination condition. The analysis allows us to cally prove the termination of loops that cannot be using previous techniques. The paper closes wit mental results using short examples drawn from i code.

1 Introduction

From the very beginnings of the formal analysi ware [12, 14], the task of formally verifying the co of a program has been decomposed into the tasks ing correctness *if* the program terminates, and s proving termination. Deciding termination, in g obviously undecidable, but thanks to considerable progress over the years (e.g., [9, 20, 5, 23, 3, 6, 1 18, 21, 8, 7]), a variety of techniques and heuristics automatically prove termination of many loops that practice.

Windows kernel APIs that acquire resources and APIs that release resources. For example:

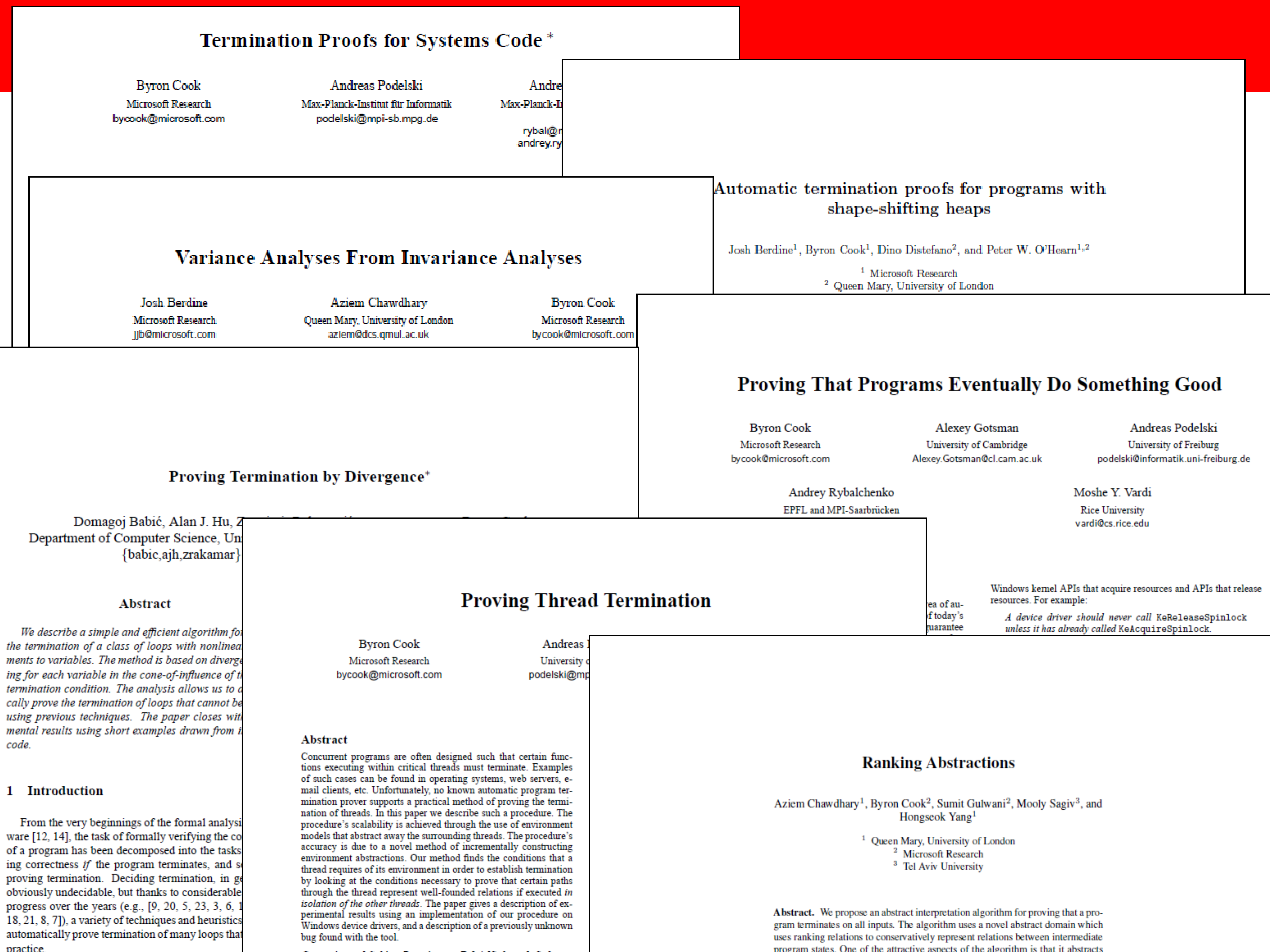
> *A device driver should never call* KeReleaseSpinlock *unless it has already called* KeAcquireSpinlock.

This is a safety property for the reason that any counterexample to the property will be a finite execution through the device driver code. We can think of safety properties as guaranteeing that specified bad events will not happen (*i.e.* calling KeReleaseSpinlock before calling KeAcquireSpinlock). Note that SDV cannot check the equally important related liveness property:

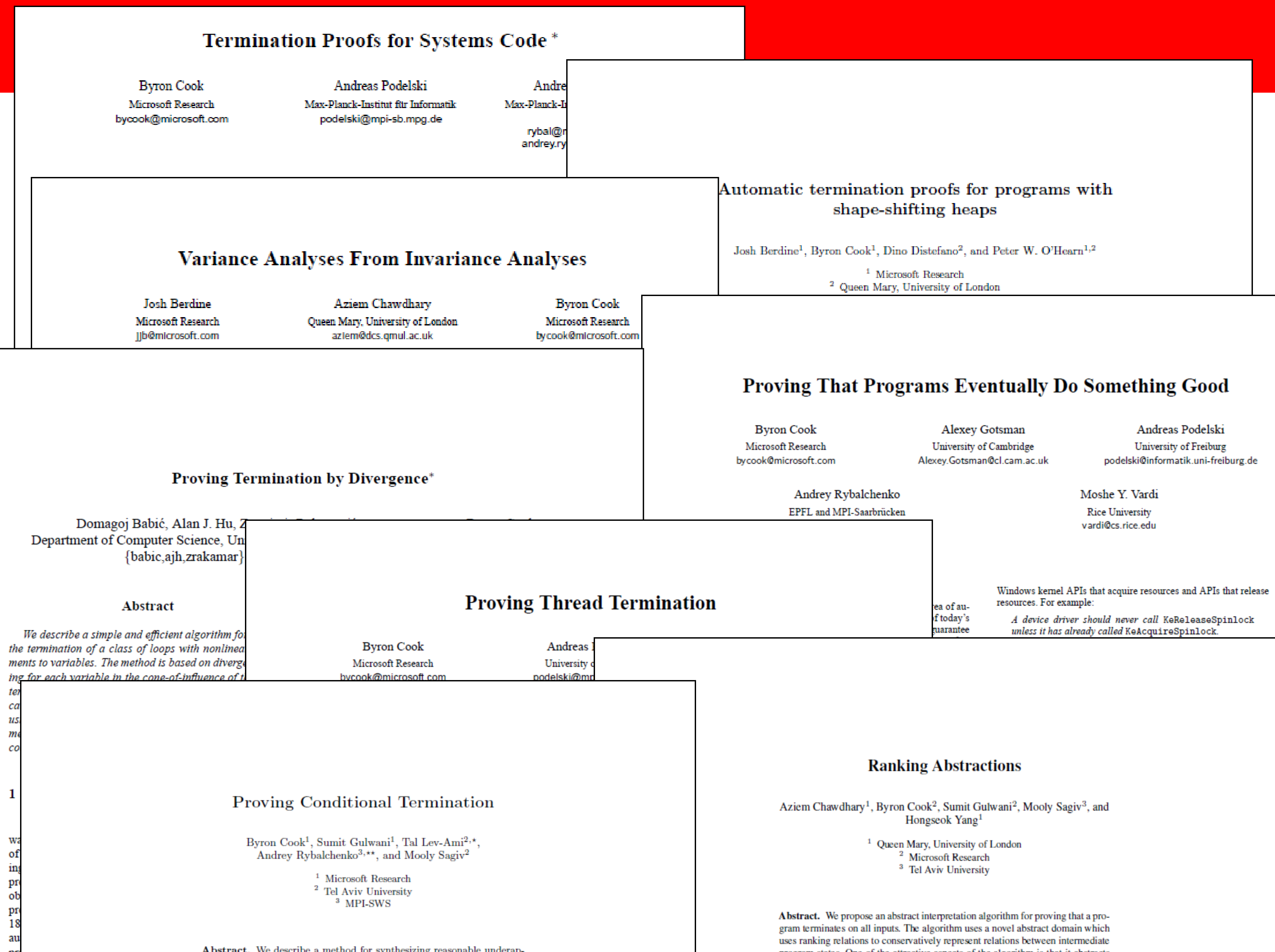
> *If a driver calls* KeAcquireSpinlock *then it must eventually make a call to* KeReleaseSpinlock.

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace in which KeAcquireSpinlock is called but it is not followed by a call to KeReleaseSpinlock. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will eventually happen (*i.e.* that KeReleaseSpinlock will eventually be called in the case that a call to KeAcquireSpinlock occurs).

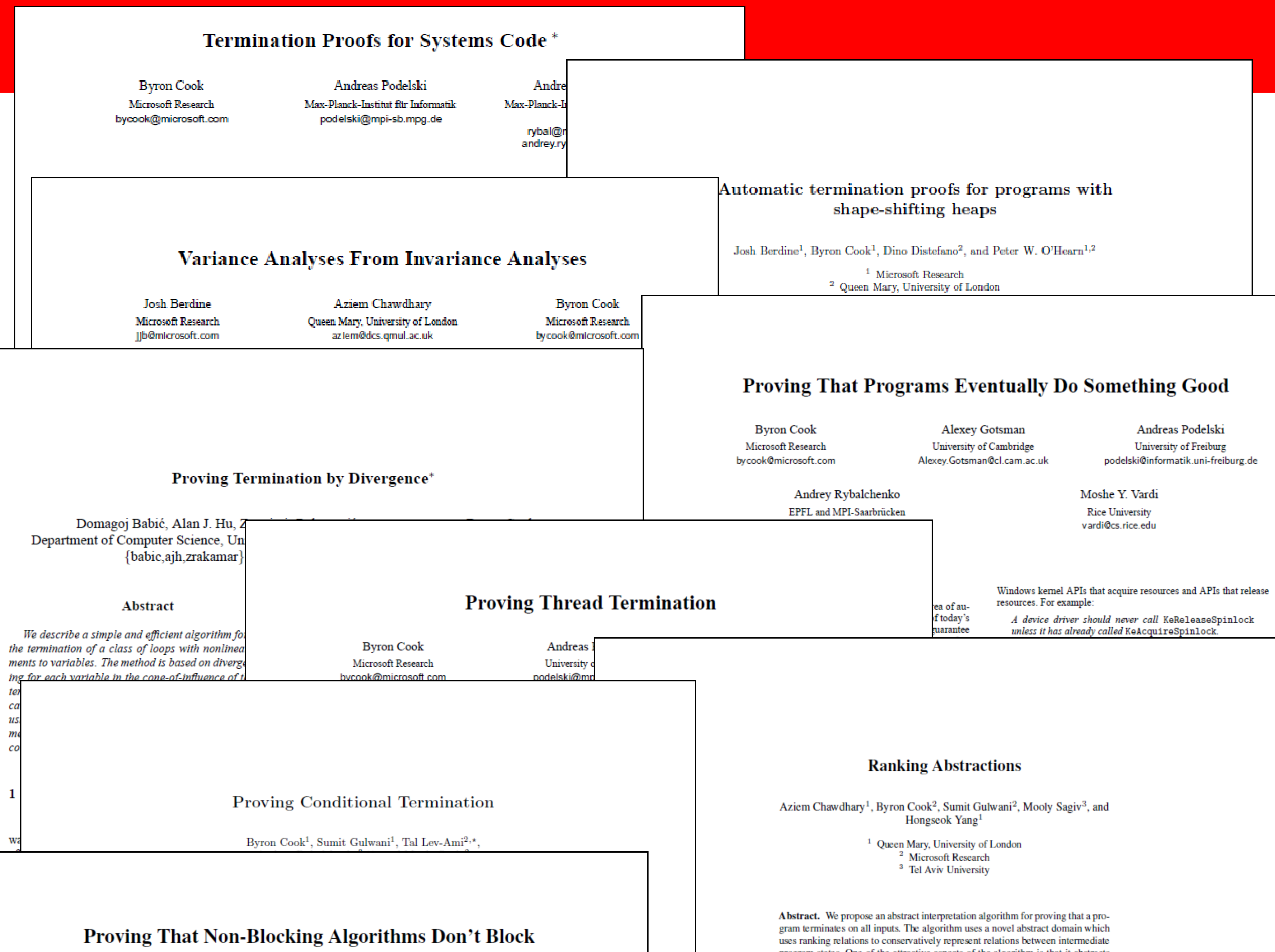
Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: "f(); g(); h();". It is easy to prove that the function f is always called before h: in this case we need only to look at the structure of the control-flow graph. It is much harder to prove that h is eventually called after f: we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:



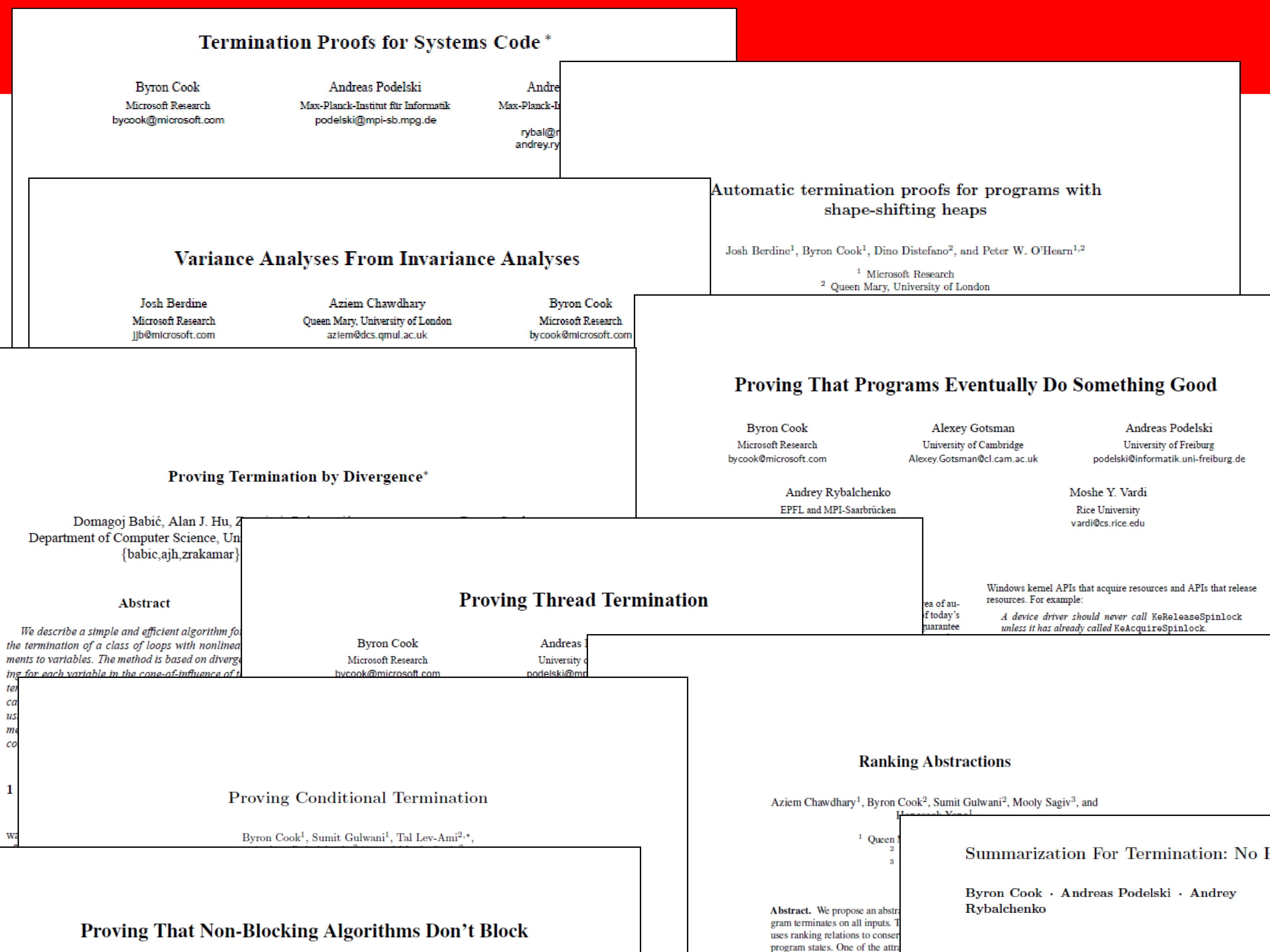
Termination Proofs for Systems Code *
Byron Cook
Microsoft Research
bycook@microsoft.com
Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de
Automatic termination proofs for programs with shape-shifting heaps
Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]
[1] Microsoft Research
[2] Queen Mary, University of London
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
Proving That Programs Eventually Do Something Good
Byron Cook
Microsoft Research
bycook@microsoft.com
Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk
Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de
Andrey Rybalchenko
EPFL and MPI-Saarbrücken
Moshe Y. Vardi
Rice University
vardi@cs.rice.edu
Windows kernel APIs that acquire resources and APIs that release resources. For example:
A device driver should never call KeReleaseSpinlock unless it has already called KeAcquireSpinlock.
Proving Termination by Divergence*
Abstract
1 Introduction
Proving Thread Termination
Byron Cook
Microsoft Research
bycook@microsoft.com
Abstract
Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Examples of such cases can be found in operating systems, web servers, e-mail clients, etc. Unfortunately, no known automatic program termination prover supports a practical method of proving the termination of threads. In this paper we describe such a procedure. The procedure's scalability is achieved through the use of environment models that abstract away the surrounding threads. The procedure's accuracy is due to a novel method of incrementally constructing environment abstractions. Our method finds the conditions that a thread requires of its environment in order to establish termination by looking at the conditions necessary to prove that certain paths through the thread represent well-founded relations if executed in isolation of the other threads. The paper gives a description of experimental results using an implementation of our procedure on Windows device drivers, and a description of a previously unknown bug found with the tool.
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



Termination Proofs for Systems Code*
Byron Cook
Microsoft Research
bycook@microsoft.com
Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de
Automatic termination proofs for programs with shape-shifting heaps
Josh Berdine1, Byron Cook1, Dino Distefano2, and Peter W. O'Hearn1,2
1 Microsoft Research
2 Queen Mary, University of London
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
Proving That Programs Eventually Do Something Good
Byron Cook
Microsoft Research
bycook@microsoft.com
Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk
Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de
Andrey Rybalchenko
EPFL and MPI-Saarbrücken
Moshe Y. Vardi
Rice University
vardi@cs.rice.edu
Windows kernel APIs that acquire resources and APIs that release resources. For example:
A device driver should never call KeReleaseSpinlock unless it has already called KeAcquireSpinlock.
Proving Termination by Divergence*
Abstract
Proving Thread Termination
Byron Cook
Microsoft Research
Proving Conditional Termination
Byron Cook1, Sumit Gulwani1, Tal Lev-Ami2,*, Andrey Rybalchenko3,**, and Mooly Sagiv2
1 Microsoft Research
2 Tel Aviv University
3 MPI-SWS
Abstract. We describe a method for synthesizing reasonable underap-
Ranking Abstractions
Aziem Chawdhary1, Byron Cook2, Sumit Gulwani2, Mooly Sagiv3, and Hongseok Yang1
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



Termination Proofs for Systems Code*
Byron Cook
Microsoft Research
bycook@microsoft.com
Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de
Automatic termination proofs for programs with shape-shifting heaps
Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]
[1] Microsoft Research
[2] Queen Mary, University of London
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
Proving That Programs Eventually Do Something Good
Byron Cook
Microsoft Research
bycook@microsoft.com
Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk
Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de
Andrey Rybalchenko
EPFL and MPI-Saarbrücken
Moshe Y. Vardi
Rice University
vardi@cs.rice.edu
Windows kernel APIs that acquire resources and APIs that release resources. For example:
A device driver should never call KeReleaseSpinlock unless it has already called KeAcquireSpinlock.
Proving Termination by Divergence*
Abstract
Proving Thread Termination
Byron Cook
Microsoft Research
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
Proving Conditional Termination
Proving That Non-Blocking Algorithms Don't Block



Termination Proofs for Systems Code *
Byron Cook
Microsoft Research
bycook@microsoft.com
Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de
Automatic termination proofs for programs with shape-shifting heaps
Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]
[1] Microsoft Research
[2] Queen Mary, University of London
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
Proving That Programs Eventually Do Something Good
Byron Cook
Microsoft Research
bycook@microsoft.com
Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk
Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de
Andrey Rybalchenko
EPFL and MPI-Saarbrücken
Moshe Y. Vardi
Rice University
vardi@cs.rice.edu
Windows kernel APIs that acquire resources and APIs that release resources. For example:
A device driver should never call KeReleaseSpinlock unless it has already called KeAcquireSpinlock.
Proving Termination by Divergence*
Abstract
Proving Thread Termination
Byron Cook
Microsoft Research
Ranking Abstractions
Aziem Chawdhary[1], Byron Cook[2], Sumit Gulwani[2], Mooly Sagiv[3], and
Proving Conditional Termination
Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,*],
Summarization For Termination: No R
Byron Cook · Andreas Podelski · Andrey Rybalchenko
Proving That Non-Blocking Algorithms Don't Block

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Andre
Max-Planck-I

Temporal property verification as a program analysis task

Byron Cook[1], Eric Koskinen[2], and Moshe Vardi[3]

[1] Microsoft Research and Queen Mary University of London
[2] University of Cambridge
[3] Rice University

Abstract. We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (*e.g.* backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (*e.g.* [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (*e.g.* [3, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential non-recursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL).

Dep

We des
the termi
ments to
ing for ea
ter
ca
us
me
co

1

wa

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

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

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Andrey Rybalchenko
EPFL and MPI-Saarbrücken

Moshe Y. Vardi
Rice University
vardi@cs.rice.edu

tion

ea of au-
f today's
uarantee

Windows kernel APIs that acquire resources and APIs that release resources. For example:

A device driver should never call `KeReleaseSpinlock` *unless it has already called* `KeAcquireSpinlock`.

Ranking Abstractions

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

[1] Queen

Abstract. We propose an abstr
gram terminates on all inputs. T
uses ranking relations to conser
program states. One of the attra

Summarization For Termination: No R

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Proving That Non-Blocking Algorithms Don't Block

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Andre
Max-Planck-I

Temporal property verification as a program analysis task

Byron Cook[1], Eric Koskinen[2], and Moshe Vardi[3]

[1] Microsoft Research and Queen Mary University of London
[2] University of Cambridge
[3] Rice University

Abstract. We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (*e.g.* backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (*e.g.* [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (*e.g.* [3, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential non-recursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL).

Making Prophecies with Decision Predicates

Byron Cook
Microsoft Research &
Queen Mary, University of London
bycook@microsoft.com

Eric Koskinen
University of Cambridge
ejk39@cam.ac.uk

Abstract

We describe a new algorithm for proving temporal properties expressed in LTL of infinite-state programs. Our approach takes advantage of the fact that LTL properties can often be proved more efficiently using techniques usually associated with the branching-time logic CTL than they can with native LTL algorithms. The caveat is that, in certain instances, nondeterminism in the system's transition relation can cause CTL methods to report counterexamples that are spurious with respect to the original LTL formula. To address this problem we describe an algorithm that, as it attempts to apply CTL proof methods, finds and then removes problematic nondeterminism via an analysis on the potentially spurious counterexamples. Problematic nondeterminism is characterized using *decision predicates*, and removed using a partial, symbolic determinization procedure which introduces new prophecy variables to predict the future outcome of these choices. We demonstrate—using examples taken from the PostgreSQL database server, Apache web server, and Windows OS kernel—that our method can yield enormous performance improvements in comparison to known tools, allowing us to automatically prove properties of programs where we could not prove them before.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; Correctness proofs; Reliability; D.4.5 [*Operating Systems*]: Reliability—Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Verification, Theory, Reliability

Keywords Linear temporal logic, formal verification, termination, program analysis, model checking

1. Introduction

The common wisdom amongst users and developers of tools that prove temporal properties of systems is that the linear specification logic LTL [33] is more intuitive than CTL [10], but that properties expressed in the universal fragment of CTL (∀CTL) without fairness constraints are often easier to prove than their LTL cousins [3, 32, 44][1]. Properties expressed in CTL without fairness can be proved in a purely syntax-directed manner using state-based reasoning techniques, whereas LTL requires deeper reasoning about whole sets of traces and the subtle relationships between families of them.

In this paper we aim to make an LTL prover for infinite-state programs with performance closer to what one would expect from a CTL prover. We use the observation that ∀CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don't always fit together: there are cases when, due to some instances of nondeterminism in the transition system, ∀CTL alone is not powerful enough to prove an LTL property.

In these cases our LTL prover works around the problem using something we call *decision predicates*, which are used to characterize and treat such instances of nondeterminism. A decision predicate is represented as a pair of first-order logic formulae (a, b), where the formula a defines the decision predicate's presupposition (*i.e. when* the decision is made), and b characterizes the binary choice made when this presupposition holds. Any transition from state s to state s' in the system that meets the constraint $a(s) \wedge b(s')$ is distinguished by the decision predicate (a, b) from $a(s) \wedge \neg b(s')$.

We use decision predicates as the basis of a partial symbolic determinization procedure: for each predicate we introduce a new prophecy variable [3] to predict the future outcome of the decision. After partially determinizing with respect to these prophecy variables, we find that CTL proof methods succeed, thus allowing us to prove LTL properties with CTL proof techniques in cases where this strategy would have previously failed. To synthesize the decision predicates we employ a form of symbolic execution on spurious ∀CTL counterexamples together with an application of Farkas' lemma [23].

With our new approach we can automatically prove properties of infinite-state programs in minutes or seconds which were intractable using existing tools. Examples include code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Limitations. In practice, the applicability and performance of our technique is dependent on the heuristic used to choose new decision predicates when given an abstract representation of a specific point in a spurious counterexample. The predicate synthesis mechanism implemented in our tool is applicable primarily to infinite-state programs over arithmetic variables with commands that only contain linear arithmetic. However, no matter which predicate selection mechanism is used, our predicate-based determinization strategy is sound. Thus, unsound approximations to predicate synthesis could potentially be used in instances where the systems considered do not meet the constraints given above. Our technique is also based

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$10.00

[1] Abadi and Lamport [3] make this point using the terminology of "refinement mappings" and "trace equivalence" instead of phrasing it in the context of temporal logics.

Dep

We des the termi ments to ing for ec te ca us me co

1

wa

d

ki
rg
reiburg.de

s that release

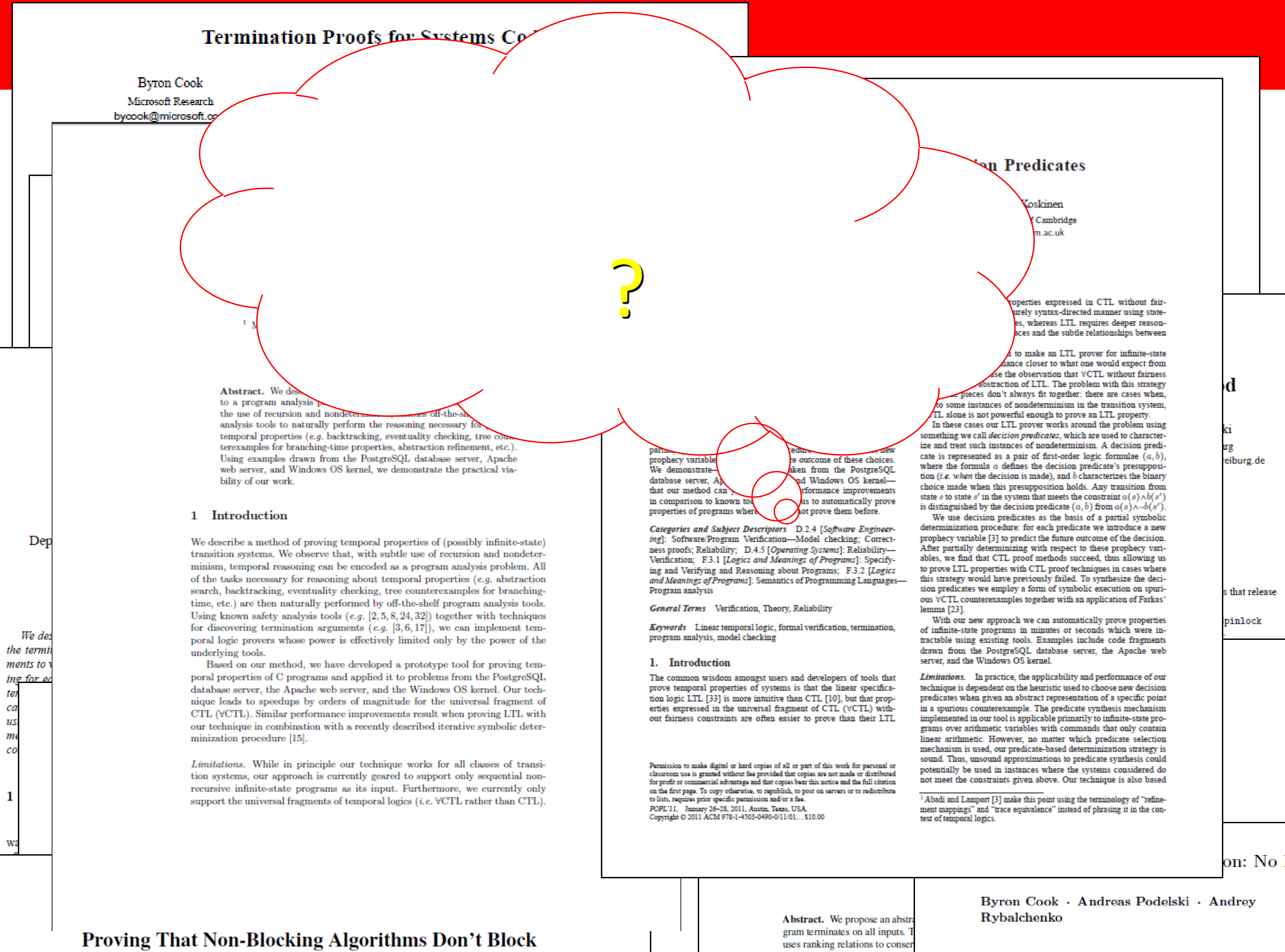
pinlock

on: No R

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Abstract. We propose an abstr gram terminates on all inputs. T uses ranking relations to conser program states. One of the attra

Proving That Non-Blocking Algorithms Don't Block



Termination Proofs for Systems Co

Byron Cook
Microsoft Research
bycook@microsoft.co

?

Abstract. We des to a program analysis the use of recursion and nondetermini analysis tools to naturally perform the reasoning necessary fo temporal properties (e.g. backtracking, eventuality checking, tree cou terexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g. abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g. [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (e.g. [3, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential non-recursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (i.e. ∀CTL rather than CTL).

on Predicates

Koskinen
f Cambridge
m.ac.uk

roperties expressed in CTL without fairurely syntax-directed manner using statees, whereas LTL requires deeper reasonaces and the subtle relationships between

to make an LTL prover for infinite-state nance closer to what one would expect from se the observation that ∀CTL without fairness abstraction of LTL. The problem with this strategy pieces don't always fit together: there are cases when, to some instances of nondeterminism in the transition system, TL alone is not powerful enough to prove an LTL property.

In these cases our LTL prover works around the problem using something we call *decision predicates*, which are used to characterize and treat such instances of nondeterminism. A decision predicate is represented as a pair of first-order logic formulae (a, b), where the formula a defines the decision predicate's presupposition (*i.e. when* the decision is made), and b characterizes the binary choice made when this presupposition holds. Any transition from state s to state s' in the system that meets the constraint $a(s) \wedge b(s')$ is distinguished by the decision predicate (a, b) from $a(s) \wedge \neg b(s')$.

We use decision predicates as the basis of a partial symbolic determinization procedure: for each predicate we introduce a new prophecy variable [3] to predict the future outcome of the decision. After partially determinizing with respect to these prophecy variables, we find that CTL proof methods succeed, thus allowing us to prove LTL properties with CTL proof techniques in cases where this strategy would have previously failed. To synthesize the decision predicates we employ a form of symbolic execution on spurious ∀CTL counterexamples together with an application of Farkas' lemma [23].

With our new approach we can automatically prove properties of infinite-state programs in minutes or seconds which were intractable using existing tools. Examples include code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Limitations. In practice, the applicability and performance of our technique is dependent on the heuristic used to choose new decision predicates when given an abstract representation of a specific point in a spurious counterexample. The predicate synthesis mechanism implemented in our tool is applicable primarily to infinite-state programs over arithmetic variables with commands that only contain linear arithmetic. However, no matter which predicate selection mechanism is used, our predicate-based determinization strategy is sound. Thus, unsound approximations to predicate synthesis could potentially be used in instances where the systems considered do not meet the constraints given above. Our technique is also based

[1] Abadi and Lamport [3] make this point using the terminology of "refinement mappings" and "trace equivalence" instead of phrasing it in the context of temporal logics.

prophecy variable e outcome of these choices. We demonstrate aken from the PostgreSQL database server, Ap nd Windows OS kernel— that our method can rformance improvements in comparison to known to us to automatically prove properties of programs where ot prove them before.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; Correctness proofs; Reliability; D.4.5 [*Operating Systems*]: Reliability—Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Verification, Theory, Reliability

Keywords Linear temporal logic, formal verification, termination, program analysis, model checking

1. Introduction

The common wisdom amongst users and developers of tools that prove temporal properties of systems is that the linear specification logic LTL [33] is more intuitive than CTL [10], but that properties expressed in the universal fragment of CTL (∀CTL) without fairness constraints are often easier to prove than their LTL

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01…$10.00

on: No R

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Abstract. We propose an abstr gram terminates on all inputs. T uses ranking relations to conser program states. One of the attra

Proving That Non-Blocking Algorithms Don't Block

Termination Proofs for Systems Co

Byron Cook
Microsoft Research
bycook@microsoft.co

Proving stabilization for biological systems

Byron Cook[1,2], Jasmin Fisher[1], Elzbieta Krepska[1,3], and Nir Piterman[4]

[1] Microsoft Research
[2] Queen Mary, University of London
[3] VU University Amsterdam
[4] Imperial College London

Abstract. We describe an efficient procedure for proving stabilization of biological systems modeled as qualitative networks. For scalability, our procedure uses modular proof techniques, where state-space exploration is applied only locally to small pieces of the system rather than the entire system as a whole. Our procedure exploits the observation that, in practice, the form of modular proofs required can be restricted to a very limited set. Using our new procedure, we have solved a number of challenging published examples, including a 3-D model of the mammalian epidermis, a model of metabolic networks operating in type-2 diabetes, and a model of fate determination of vulval precursor cells in the *C. elegans* worm. Our results show many orders of magnitude speedup in cases where previous stabilization proving techniques were known to succeed, and new results in cases where tools had previously failed.

1 Introduction

Biologists are increasingly turning to techniques from computer science in their quest to understand and predict the behavior of complex biological systems [2–4]. In particular, the application of formal verification tools to models of biological processes is gaining impetus among biologists. In some cases known formal verification techniques work well (*e.g.* [5–7]). Unfortunately in other cases—such as proving stabilization [8]—we find that existing abstractions and heuristics are not effective.

In this paper we address the open challenge to find scalable algorithms for proving stabilization of biological systems. In computer science terms, we are trying to prove a liveness property similar to termination of large parallel systems. The sizes of these systems forces us to use some form of modular reasoning. Unfortunately, because stabilization is a liveness property, we must be careful when using the more powerful cyclic modular proof rules (*e.g.* [9,10]), as they are formally only sound in the context of safety [11]. Furthermore, we find that the complex temporal interactions between the modules are crucial to the stabilization of the system as a whole; meaning that we cannot use scalable techniques that simply abstract away the interactions altogether.

In this paper we show that in practice non-circular modular proofs can be found using local liveness lemmas of a limited form:

$$[FG(p_1) \wedge \ldots \wedge FG(p_n)] \Rightarrow FG(q)$$

...n Predicates

on: No R...

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Proving That Non-Blocking Algorithms Don't Block

Misunderstanding the halting problem

- Automatic searches for proofs of program termination don’t make for exciting demos

- Termination bugs found from failed proof attempts are usually more entertaining

Microsoft Development Environment [design] - mouclass.c [Read Only]

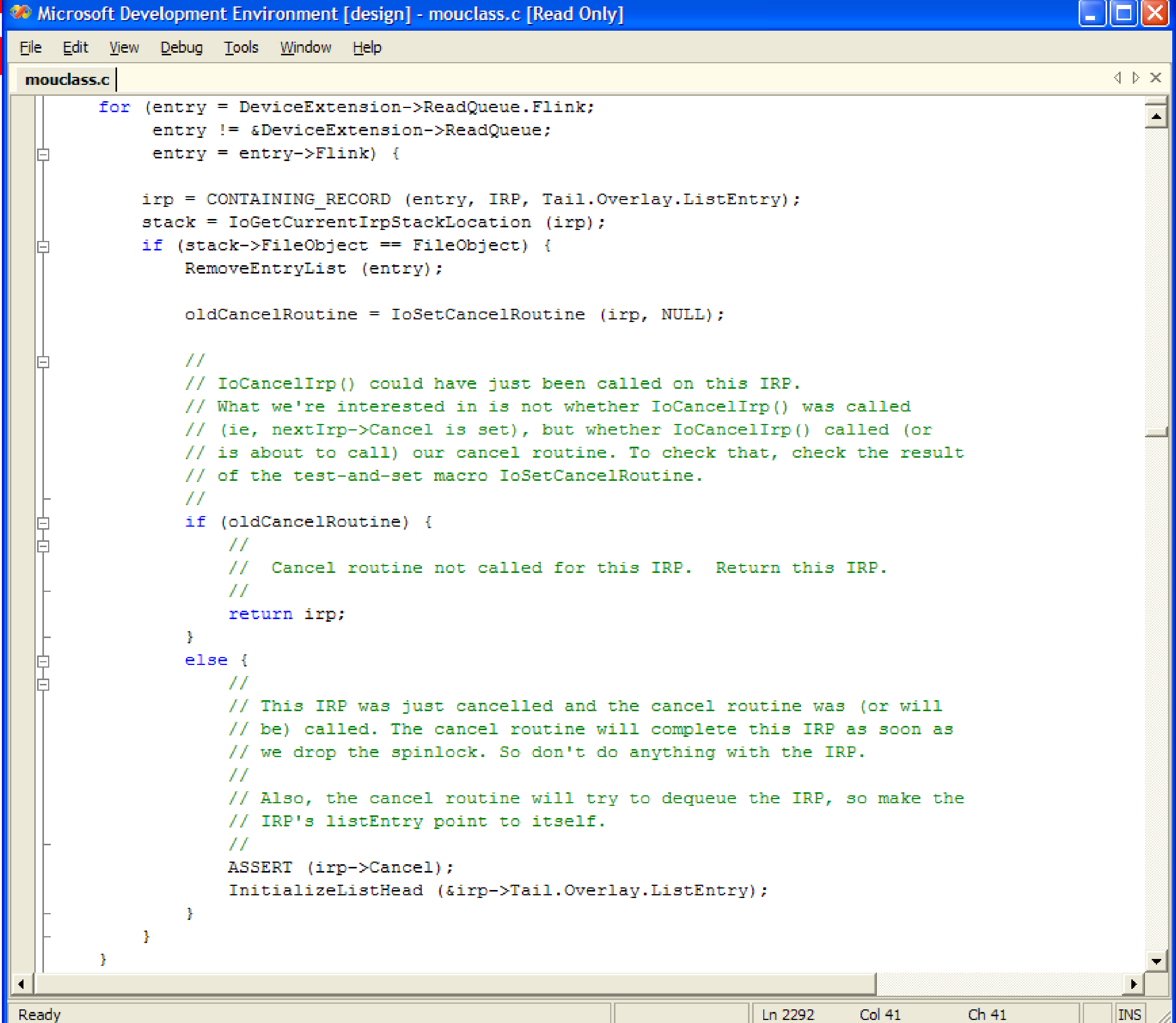
File Edit View Debug Tools Window Help

mouclass.c

```c
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this IRP.
            // What we're interested in is not whether IoCancelIrp() was called
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu




```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this IRP.
            // What we're interested in is not whether IoCancelIrp() was called
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```c
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

File Edit View Debug Tools Window Help

mouclass.c

```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Ready | Ln 2292 | Col 41 | Ch 41 | INS

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

RE: Question about mouclass driver - Message (Plain Text)

File Edit View Insert Format Tools Actions Help

Reply | Reply to All | Forward

You replied on 1/26/2006 12:53 AM.

From: Adrian Oney   Sent: Sat 12/10/2005 3:52 AM
To: Byron Cook
Cc:
Subject: RE: Question about mouclass driver

Still solving the halting problem I see. If you ever find spare time, you might also want to give a go at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

This, is indeed fucked. The for loop should be scrapped so that the else clause can read the next entry before whacking it.

Note also that *two* processors will be wedgied, not just one: the cancel routine will wait until the lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the O(n*m) condition created by the invocation by MouseClassCleanupQueue, where n is the number of non-FO matching objects in the beginning of the queue and m is the number of matching ones. DOS attack anyone?

- A

-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

RE: Question about mouclass driver - Message (Plain Text)

File Edit View Insert Format Tools Actions Help

Reply | Reply to All | Forward

You replied on 1/26/2006 12:53 AM.

From: Adrian Oney   Sent: Sat 12/10/2005 3:52 AM
To: Byron Cook
Cc:
Subject: RE: Question about mouclass driver

Still solving the halting problem I see. If you ever find spare time, you might also want to give a go at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

This, is indeed fucked. The for loop should be scrapped so that the else clause can read the next entry before whacking it.

Note also that *two* processors will be wedgied, not just one: the cancel routine will wait until the lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the O(n*m) condition created by the invocation by MouseClassCleanupQueue, where n is the number of non-FO matching objects in the beginning of the queue and m is the number of matching ones. DOS attack anyone?

- A

-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver

Misu

Microsoft Development Environment [design] - mouclass.c [Read Only]

RE: Question about mouclass driver - Message (Plain Text)

File Edit View Insert Format Tools Actions Help

Reply | Reply to All | Forward

You replied on 1/26/2006 12:53 AM.

From: Adrian Oney
To: Byron Cook
Cc:
Subject: RE: Question about mouclass driver

Sent: Sat 12/10/2005 3:52 AM

Still solving the halting problem I see. If you ever find spare time, you might also want to give a go at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

This, is indeed fucked. The for loop should be scrapped so that the else clause can read the next entry before whacking it.

Note also that *two* processors will be wedgied, not just one: the cancel routine will wait until the lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the O(n*m) condition created by the invocation by MouseClassCleanupQueue, where n is the number of non-FO matching objects in the beginning of the queue and m is the number of matching ones. DOS attack anyone?

- A

-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver

Outline

- Introduction
- Termination basics & history
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Outline

- Introduction
- Termination basics & history
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Future work

- → Previous wisdom: proving termination for industrial systems code is impossible

- → Now people are beginning to think that it's effectively "solved".

- → Much left to do, including
  - Complex data structures (safety)
  - Infinite-state systems w/ bit vectors (safety)
  - Binaries (safety)
  - Non-linear systems (liveness and safety)
  - Better support for concurrent programs
  - Modern programming features (*e.g.* closures)
  - Finding preconditions to termination
  - Scalability, performance, precision

Future work

- Termination proving is at the heart of many undecidable problems (*e.g.* Wang's tiling problem)
- Modern termination proving techniques could potentially be used to building working tools
- Challenge: "black-box" solutions to undecidable problems die in the most unpredictable ways

Conclusion

→ Conventional wisdom about termination overturned
- Undecidable does not mean we cannot soundly approximate a solution

→ TERMINATOR shows that automatic termination proving is not hopeless for industrial systems code

→ Current state-of-the-art solutions based on
- Abstraction search for safety property verification (*e.g.* SLAM)
- Farkas-based linear rank function synthesis
- Ramsey-based modular termination arguments
- Separation Logic based data structure analysis

For more information

→ http://research.microsoft.com/TERMINATOR

- Research papers
- Recorded technical lectures
- Contact details

→ CACM review article

review articles

DOI:10.1145/1941487.1941509

In contrast to popular belief, proving termination is not always impossible.

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

THE PROGRAM TERMINATION problem, also known as the uniform halting problem, can be defined as follows:

Using only a finite amount of time, determine whether a given program will always finish running or could execute forever.

This problem rose to prominence before the invention of the modern computer, in the era of Hilbert's *Entscheidungsproblem*:[a] the challenge to formalize all of mathematics and use algorithmic means to determine the validity of all statements. In hopes of either solving Hilbert's challenge, or showing it impossible, logicians began to search for possible instances of undecidable problems. Turing's proof[38] of termination's undecidability is the most famous of those findings.[b]

The termination problem is structured as an infinite set of queries: to solve the problem we would need to invent a method capable of accurately answering either "terminates" or "doesn't terminate" when given any program drawn from this set. Turing's result tells us that any tool that attempts to solve this problem will fail to return a correct answer on at least one of the inputs. No number of extra processors nor terabytes of storage nor new sophisticated algorithms will lead to the development of a true oracle for program termination.

Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe we are always unable to prove termination, rather than more benign consequence that we are unable to always prove termination. Phrases like "but that's like the termination problem" are often used to end discussions that might otherwise have led to viable partial solutions for real but undecidable problems. While we cannot ignore termination's undecidability, if we develop a slightly modified problem statement we can build useful tools. In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. If the termination prover cannot prove or disprove termination, it should return "unknown."

Using only a finite amount of time, determine whether a given program will always finish running or could execute forever, or return the answer "unknown."

» key insights

- For decades, the same method was used for proving termination. It has never been applied successfully to large programs.
- A deep theorem in mathematical logic, based on Ramsey's theorem, holds the key to a new method.
- The new method can scale to large programs because it allows for the modular construction of termination arguments.

a In English: "decision problem."
b There is a minor controversy as to whether or not Turing proved the undecidability in[*]. Technically he did not, but termination's undecidability is an easy consequence of the result that is proved. A simple proof can be found in Strachey.[*]