Program termination · Lecture 4

Berkeley · Spring ’09

Byron Cook

Outline

- → Fair termination
- → Data structures
- → Concurrency
- → Conclusion

Proving liveness properties

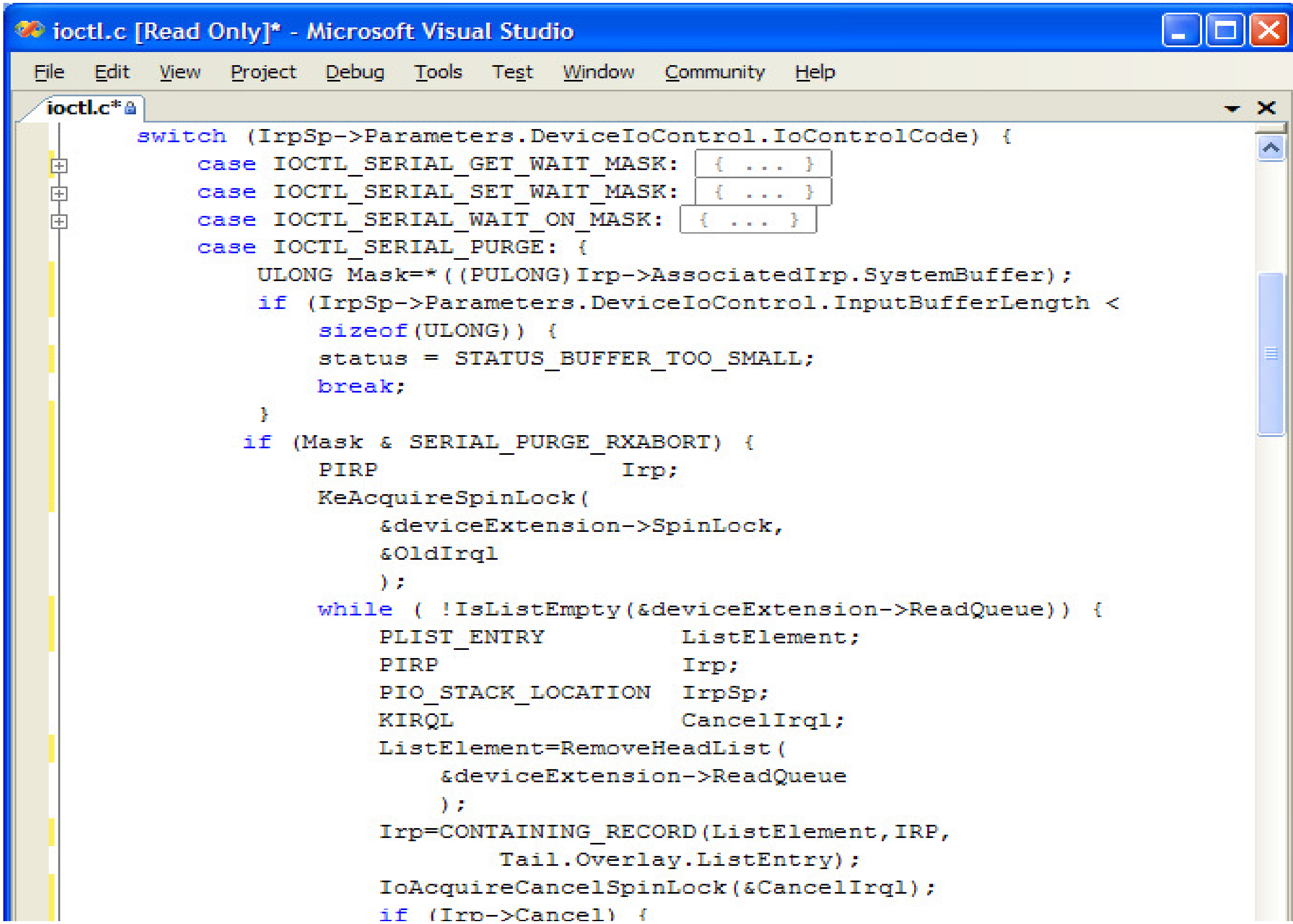
- Only pure termination considered (until now)

- Liveness vs. safety
  - Safety properties always have a finite counterexample
    - Example: “*Every Release() is proceeded by Acquire()*”
  - Liveness properties may have only infinite counterexamples
    - Example: “*Every Acquire() is followed by Release()*”

- Termination is the most basic liveness property
  - Other liveness properties are like termination with certain counterexamples removed

Proving liveness properties

- Only pure termination considered (until now)



ble

e()”

xamples

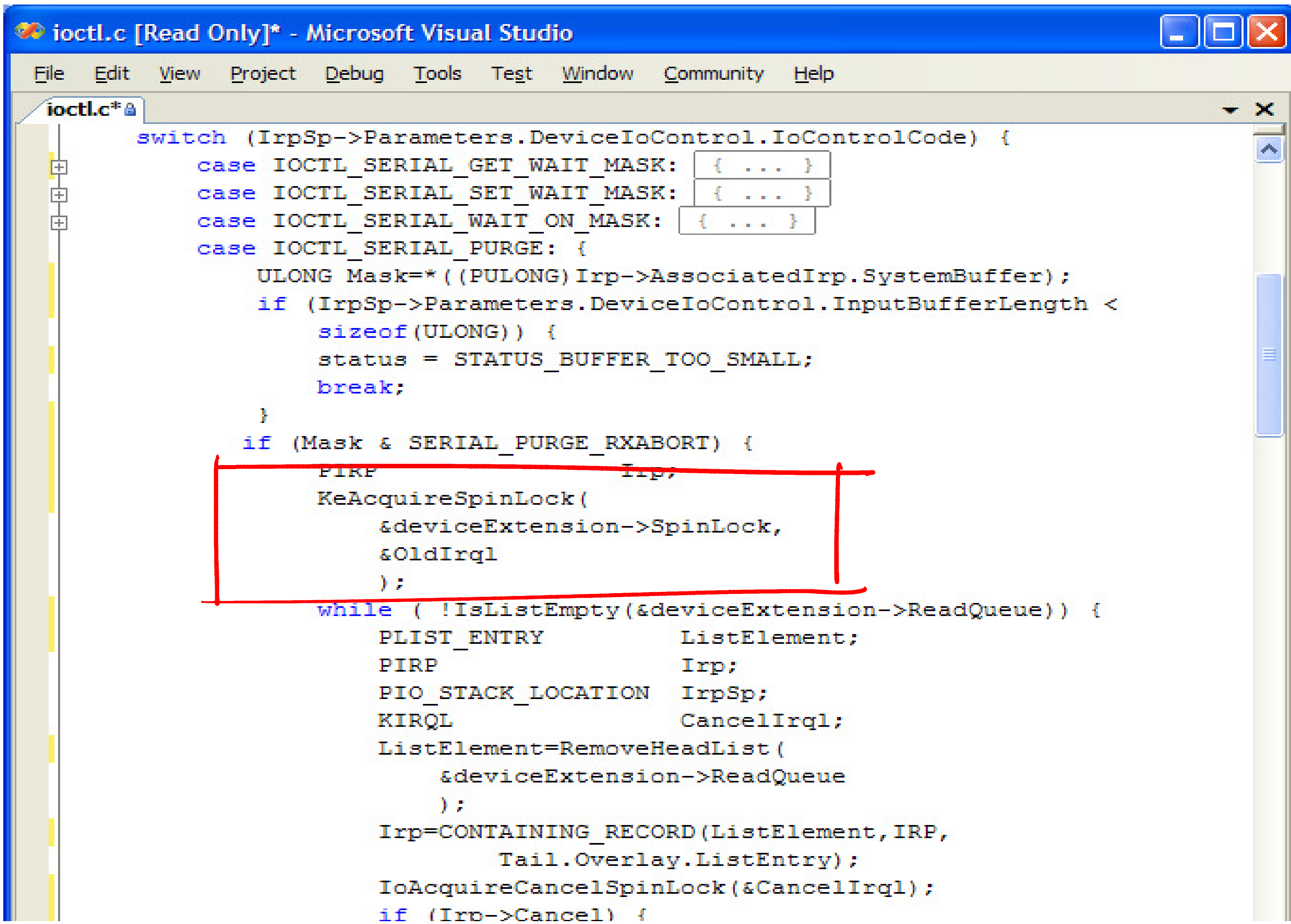
”

ty

certain

Proving liveness properties

→ Only pure termination considered (until now)



Proving liveness prope…

→ Only pure t…

Acquire ⟹ ◇ Release is a termination-like property

ioctl.c [Read Only]* - M…

File Edit View Project

ioctl.c*

```
switch (IrpSp->…
    case IOCTL_…
    case IOCTL_…
    case IOCTL_…
    case IOCTL_…
        ULONG Ma…
        if (IrpSp->…
            sizeof(ULONG…
            status = STATU…
            break;
        }
    if (Mask & SERIAL_PURGE_…
            PIRP
            KeAcquireSpinLock(
                &deviceExtension->SpinLock,
                &OldIrql
            );
            while ( !IsListEmpty(&deviceExtension->ReadQueue)) {
                PLIST_ENTRY             ListElement;
                PIRP                    Irp;
                PIO_STACK_LOCATION      IrpSp;
                KIRQL                   CancelIrql;
                ListElement=RemoveHeadList(
                    &deviceExtension->ReadQueue
                );
                Irp=CONTAINING_RECORD(ListElement,IRP,
                        Tail.Overlay.ListEntry);
                IoAcquireCancelSpinLock(&CancelIrql);
                if (Irp->Cancel) {
```

…amples

…”

…ty

…ertain

Specifying liveness properties

- Automata on finite/infinite words
  - Good for programmers/testers, as they look like programs
  - Difficult to compose, reason about
  - Usually more expressive
  - More common in industrial applications
    - Examples: PSL, SLIC, ForSpec, ……….

- Temporal logics
  - Difficult for programmers/testers
  - Easy to compose using logical operators

Example: SLIC

- SLIC: SLAM's specification language
  - Automata over finite words of program counters (safety properties)
  - Infinite words not considered
- Function entry / exit
  - Automata triggers limited to those of function entry / exit
- Failing words marked with calls to "error()" in transfer functions:

```
IoCallDriver.entry {
      if ($2->Tail.Overlay.Sl->MajorFunction==IRP_MJ_POWER) {
          error();
      }
}
```

- Implemented as a transformation to reachability

Example: SLIC

Microsoft Research Cambridge

```
void AcquireLock()
{
    ................
}

    ..................
}

void ReleaseLock()
{
    ................
}

    ..................
}

void main()
{
    ................
```

Are these reachable?

```
state { int l = 0; }

AcquireLock.entry
{
    if (l==1) {
        error();
    } else {
        l=1;
    }
}

ReleaseLock.entry
{
    if (l==0) {
        error();
    } else {
        l=0;
    }
}
```

Example: SLIC

```
void AcquireLock()
{
      …………
}

void ReleaseLock()
{
      …………
}

void main()
{
      …………
```

```
state { int l = 0; }

AcquireLock.entry
{
    if (l==1) {
        error();
    } else {
        l=1;
    }
}

ReleaseLock.entry
{
    if (l==0) {
        error();
    } else {
        l=0;
    }
}
```

Example: SLIC

```
void AcquireLock()
{
      ……………
}

void ReleaseLock()
{
      ……………
}

void main()
{
      ……………
```

```
int l = 0;

void AcquireLock()
{
    if (l==1) {
        error();
    } else {
        l=1;
    }
    ……………
}

void ReleaseLock()
{
    if (l==0) {
        error();
    } else {
        l=0;
    }
    ……………
}

void main()
{
    ……………
```

```
state { int l = 0; }

AcquireLock.entry
{
    if (l==1) {
        error();
    } else {
        l=1;
    }
}

ReleaseLock.entry
{
    if (l==0) {
        error();
    } else {
        l=0;
    }
}
```

Example: SLIC

```
int l = 0;

void AcquireLock()
{
    if (l==1) {
        error();
    } else {
        l=1;
    }
    ……………
}


void ReleaseLock()
{
    if (l==0) {
        error();
    } else {
        l=0;
    }
    ……………
}

void main()
{
    …………
```

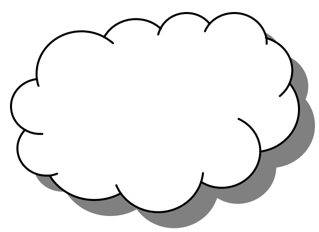
Are these reachable?

Extending SLIC with liveness properties

- To extend SLIC with support for liveness we
  - Change acceptance condition to consider infinite traces
  - Add fairness constraints
    - Unfair infinite traces are not accepted
    - Fair infinite traces are accepted
- Fair termination
  - Weak fairness = Buchi acceptance conditions = “justice”
  - Strong fairness = Streett/Rabin acceptance conditions = “compassion”

Proving liveness properties

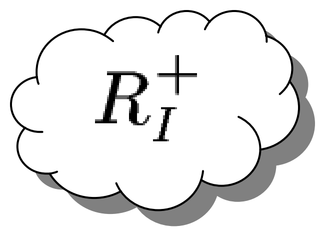
- Fairness constraints remove classes of counterexamples from consideration
  - The program doesn't terminate, but terminates if certain paths are ignored
  - Fairness constraints describe those traces



Proving liveness properties

→ Fairness constraints remove classes of counterexamples from consideration

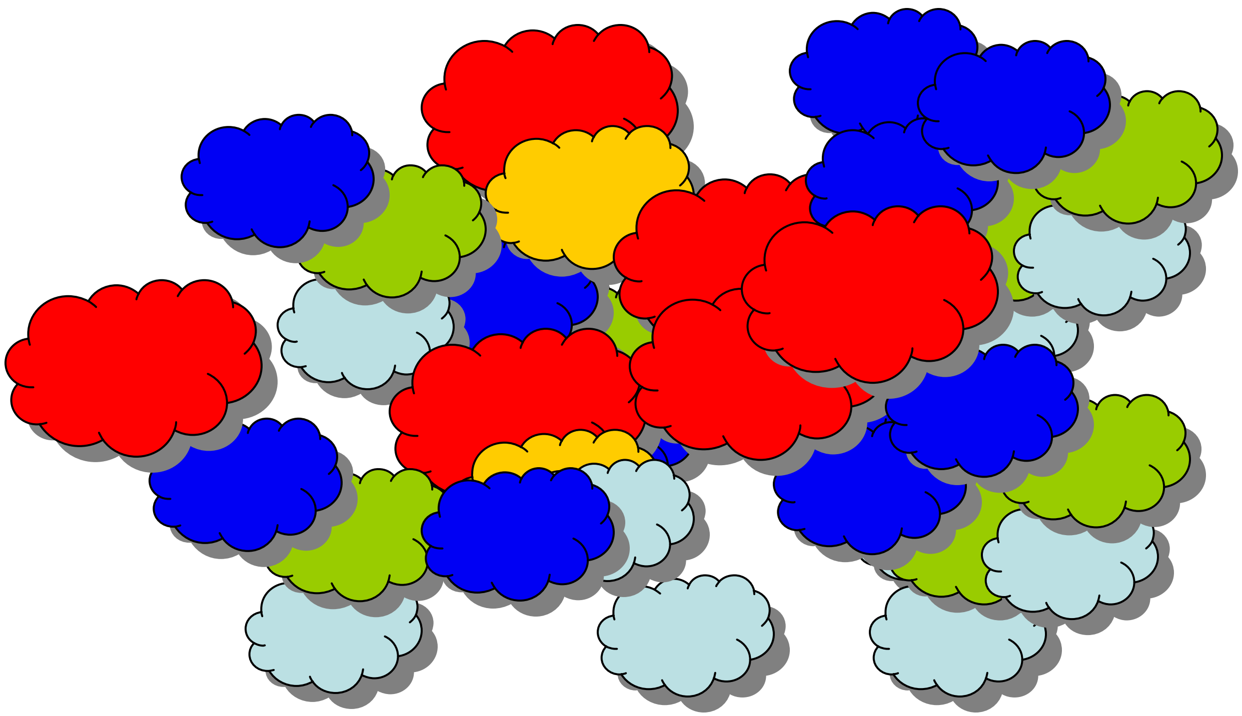
- The program doesn't terminate, but terminates if certain paths are ignored
- Fairness constraints describe those traces



Proving liveness properties

→ Fairness constraints remove classes of counterexamples from consideration

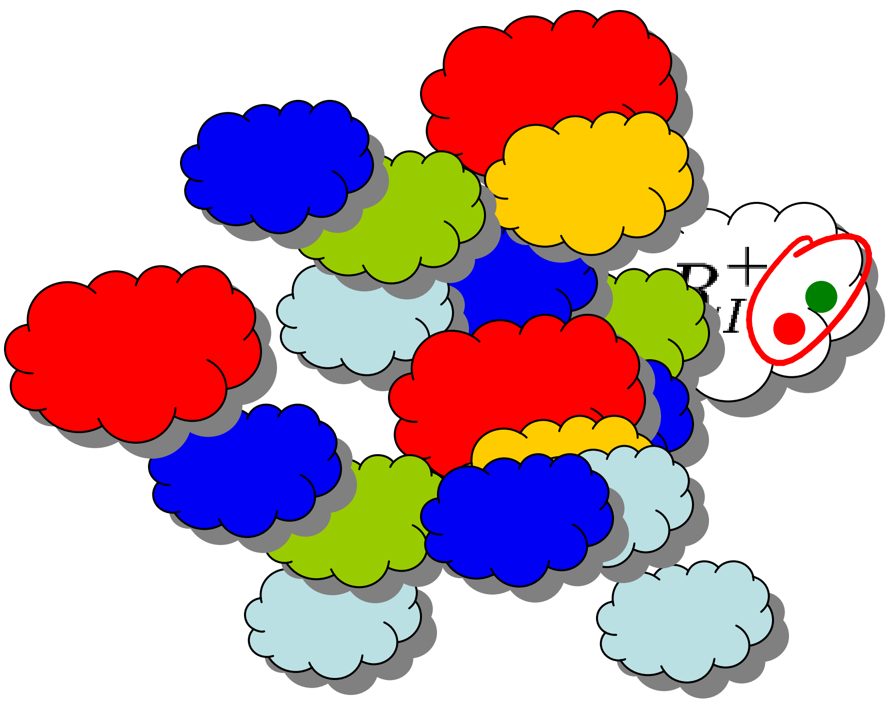
- The program doesn't terminate, but terminates if certain paths are ignored
- Fairness constraints describe those traces



Proving liveness properties

→ Fairness constraints remove classes of counterexamples from consideration

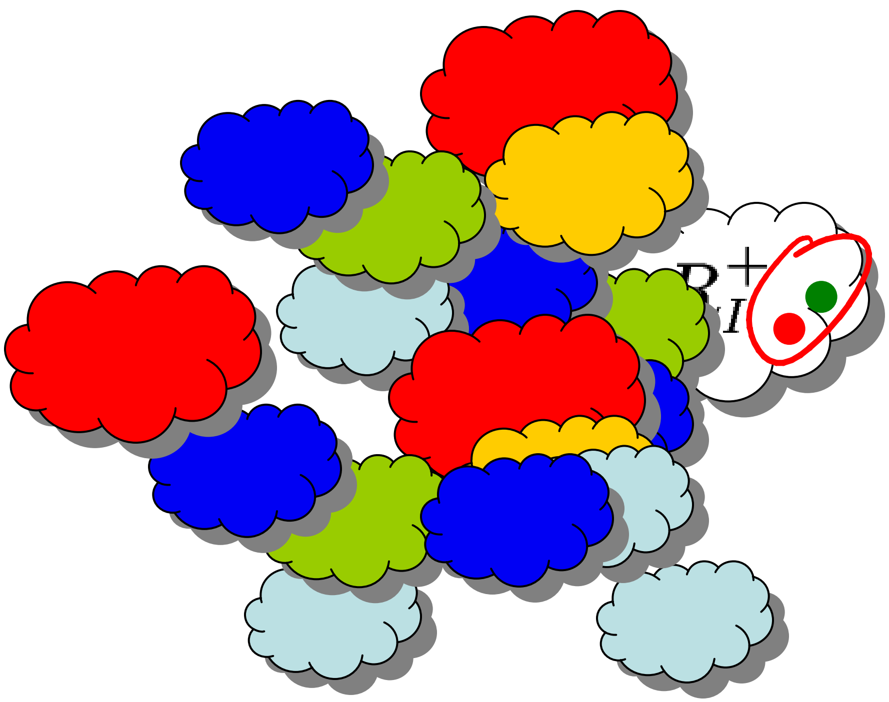
- The program doesn't terminate, but terminates if certain paths are ignored
- Fairness constraints describe those traces



Proving liveness properties

→ Fairness constraints remove classes of counterexamples from consideration

- The program doesn't terminate, but terminates if certain paths are ignored
- Fairness constraints describe those traces



Is this a real **and** **fair** counterexample?

Strong fairness

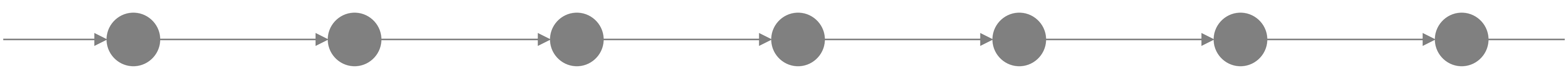
→ Fair and unfair traces:

- Fairness constraints are pairs of sets of program states $S, T \subseteq \mathcal{S}$
- A $\mathcal{P}$-trace, τ, is *fair* if either only finite S-states occur in τ or else an infinite number T states occur in τ.

Strong fairness

→ Fair and unfair traces:

- Fairness constraints are pairs of sets of program states $S, T \subseteq \mathcal{S}$
- A $\mathcal{P}$-trace, τ, is *fair* if either only finite S-states occur in τ or else an infinite number T states occur in τ.



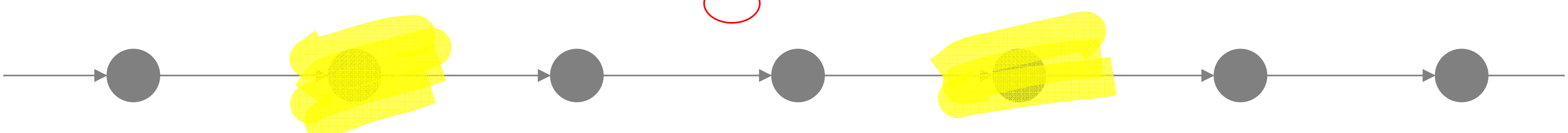
Stro

```
state {}

fairness {
    // First Boolean expression: succeeds
    // on every return from IoCreateDevice
    IoCreateDevice.exit { 1 }

    // Second Boolean expression: succeeds
    // if IoCreateDevice returns
    // something other than
    // STATUS_OBJ_NAME_COLLISION
    IoCreateDevice.exit {
        $return != STATUS_OBJ_NAME_COLLISION
    }
}
```

... program states $S, T \subseteq$

... only finite S-states occur in τ

... nber T states occur in τ.



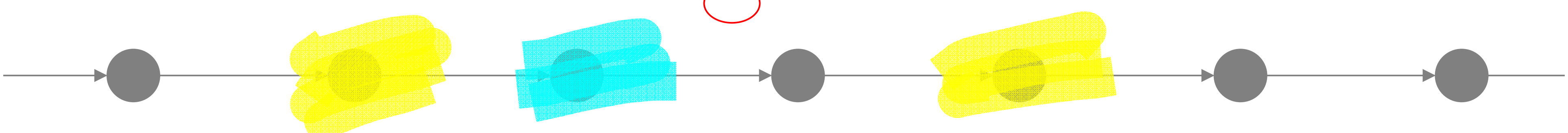
Stro

```
state {}

fairness {
    // First Boolean expression: succeeds
    // on every return from IoCreateDevice
    IoCreateDevice.exit { 1 }

    // Second Boolean expression: succeeds
    // if IoCreateDevice returns
    // something other than
    // STATUS_OBJ_NAME_COLLISION
    IoCreateDevice.exit {
        $return != STATUS_OBJ_NAME_COLLISION
    }
}
```

… program states $S, T \subseteq$

… only finite S-states occur in τ

… nber T states occur in τ.



Stro

S

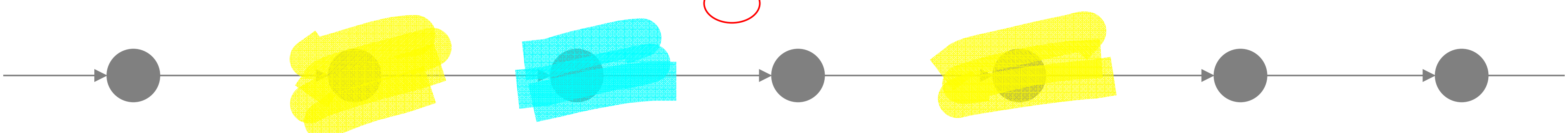
```
state {}

fairness {
    // First Boolean expression: succeeds
    // on every return from IoCreateDevice
    IoCreateDevice.exit { 1 }

    // Second Boolean expression: succeeds
    // if IoCreateDevice returns
    // something other than
    // STATUS_OBJ_NAME_COLLISION
    IoCreateDevice.exit {
        $return != STATUS_OBJ_NAME_COLLISION
    }
}
```

T

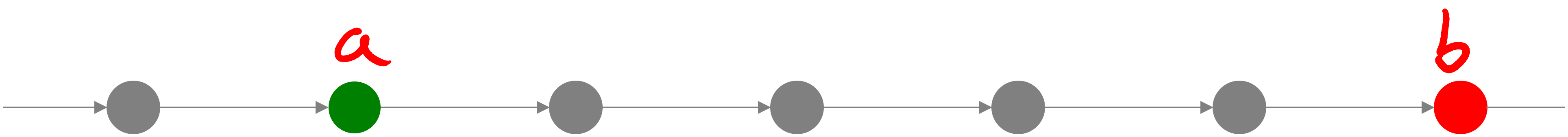
program states $S, T \subseteq$

only finite S-states occur in τ

nber T states occur in τ.



Eliminating unfair paths

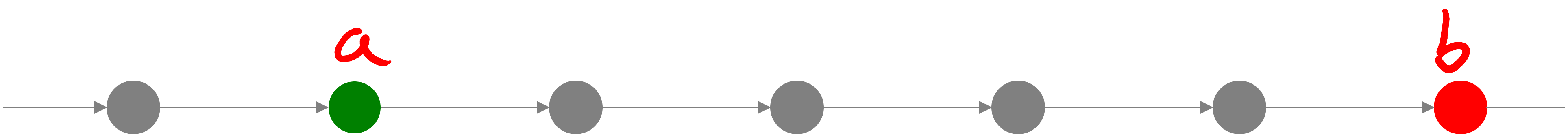


Eliminating unfair paths

$$R^+ \subseteq \phi$$

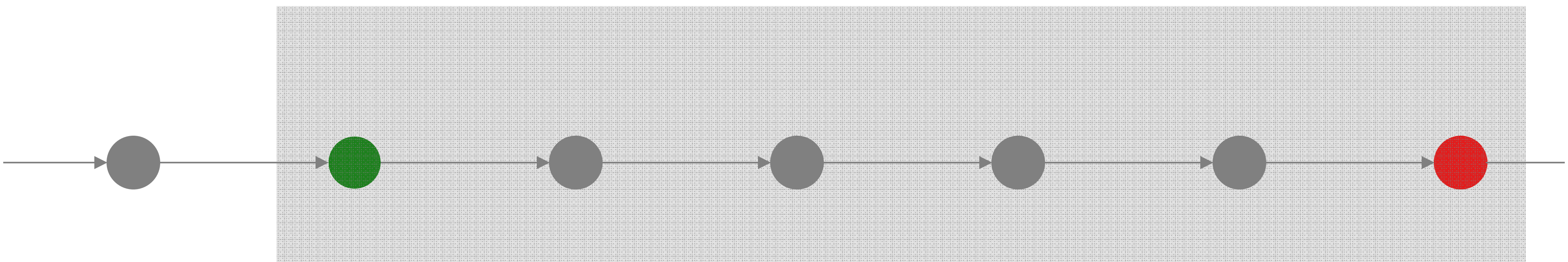
$$\equiv$$

$$\forall (a,b) \in R^+ . \ (a,b) \in \phi$$



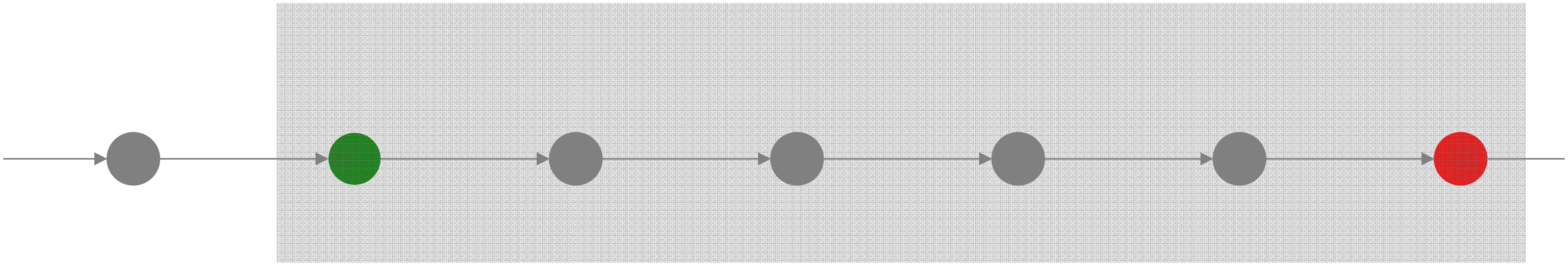
Eliminating unfair paths

$$
\begin{array}{lcl}
\textsc{Fair}(S,T) \triangleq \{(q_0, q_{k-1}) & | & \exists (q,k) \in \textsc{TraceSegs}(P) \\
 & \wedge & (\exists i.\ q_i \in S) \Rightarrow (\exists j.\ q_j \in T) \\
\} & &
\end{array}
$$



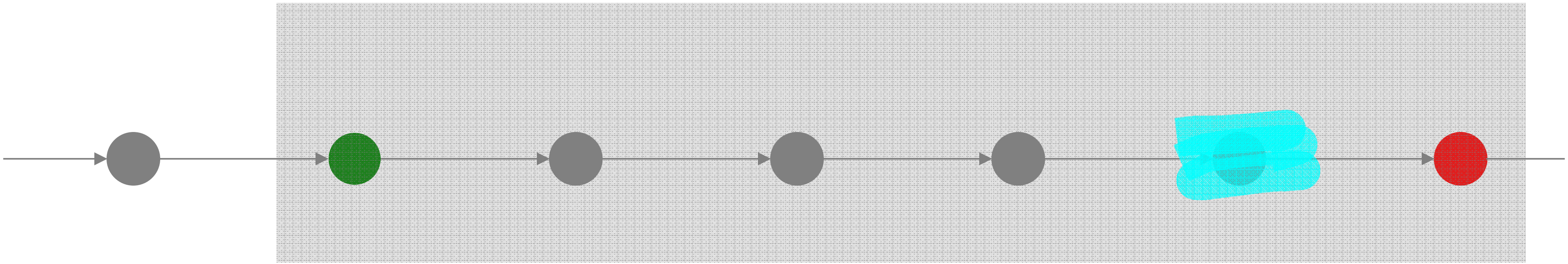
Eliminating unfair paths

Is this an "unfair path"?



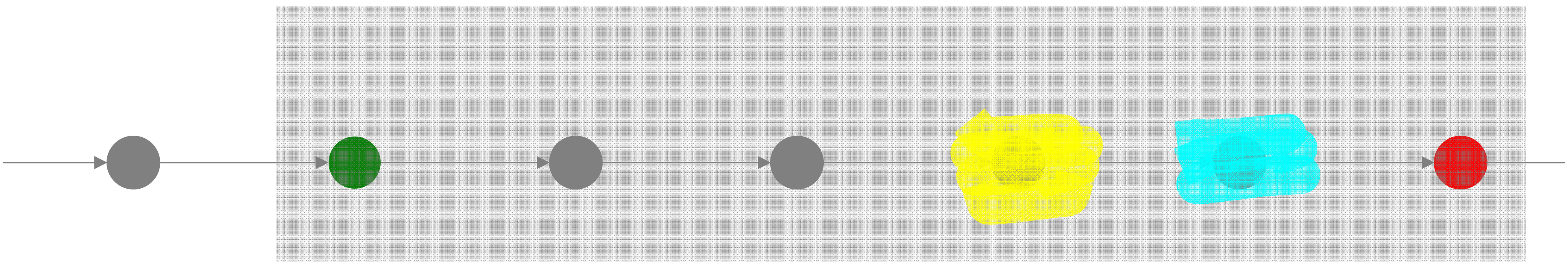
Eliminating unfair paths

Is this an "unfair path"?



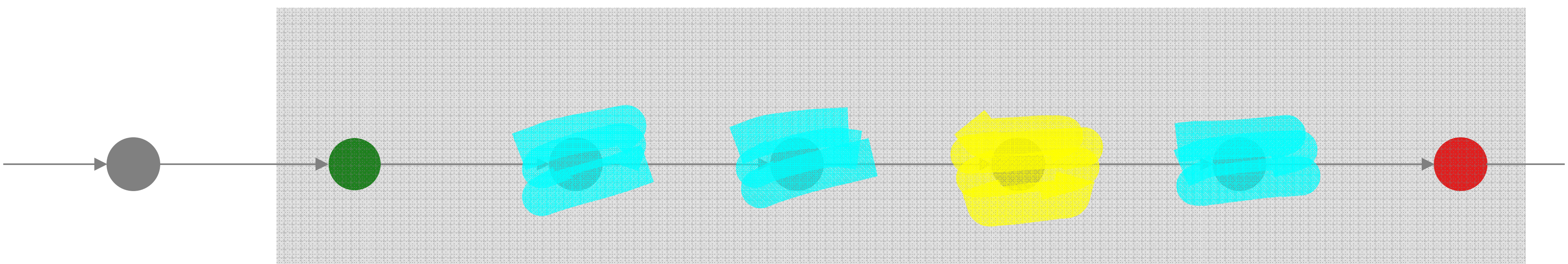
Eliminating unfair paths

Is this an "unfair path"?



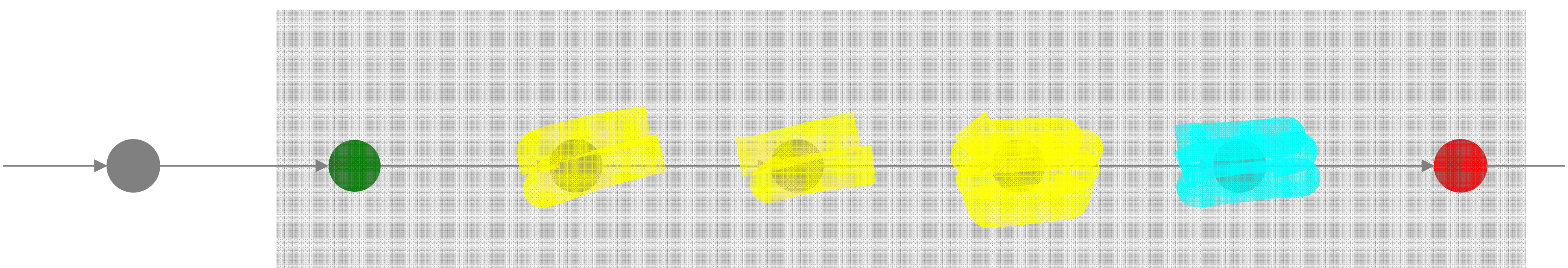
Eliminating unfair paths

Is this an "unfair path"?



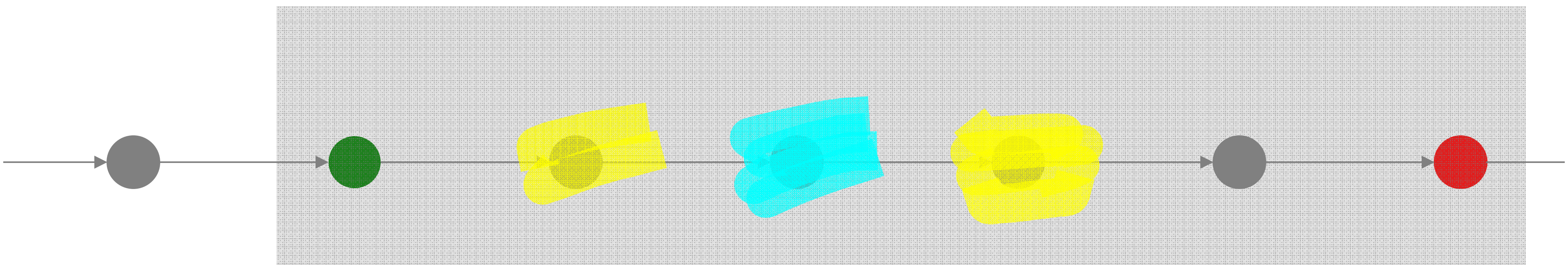
Eliminating unfair paths

Is this an "unfair path"?



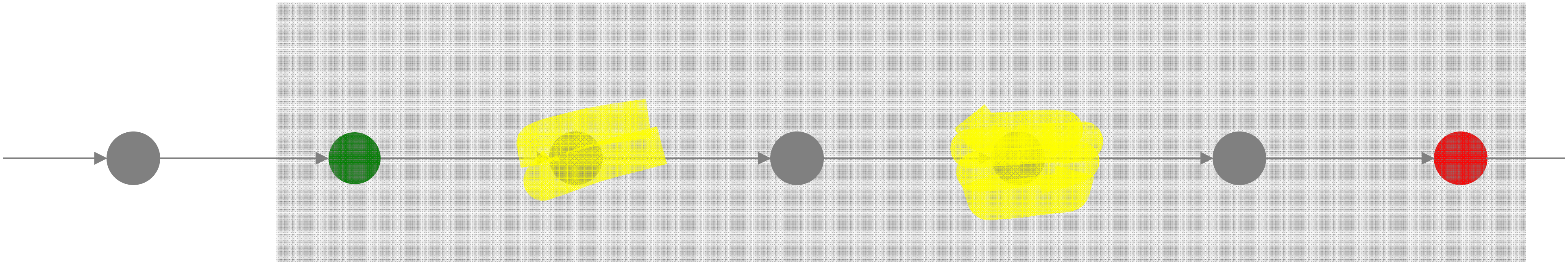
Eliminating unfair paths

Is this an "unfair path"?



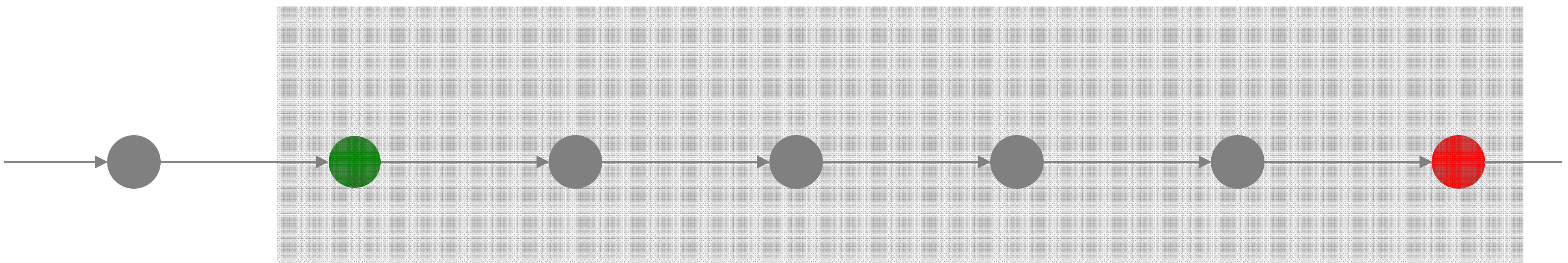
Eliminating unfair paths

Is this an "unfair path"?



Eliminating unfair paths

$$
\begin{array}{rcl}
\textsc{Fair}(S,T) \triangleq \{(q_0, q_{k-1}) & \mid & \exists (q,k) \in \textsc{TraceSegs}(P) \\
& \wedge & (\exists i.\ q_i \in S) \Rightarrow (\exists j.\ q_j \in T) \\
\} & &
\end{array}
$$



Eliminating unfair paths

$$
\begin{array}{rcl}
\textsc{Fair}(S,T) \triangleq \{(q_0, q_{k-1}) & \mid & \exists (q,k) \in \textsc{TraceSegs}(P) \\
& \wedge & (\exists i.\ q_i \in S) \Rightarrow (\exists j.\ q_j \in T) \\
\} & &
\end{array}
$$

Observation. R is well-founded with respect to fairness constraint (S,T) iff $R^+ \cap \textsc{Fair}(S,T) \subseteq Q$ and Q is disjunctively well-founded.

Eliminating unfair paths

Handwaving. $\mathcal{P} \models \Phi$ iff

$$\{(S_1, T_1), \ldots (S_n, T_n)\} = \textsc{Compile}(\neg\Phi)$$

and $\mathcal{P}$ terminates with respect to fairness constraints $(S_1, T_1), \ldots (S_n, T_n)$.

FA

Observation. R is well-founded with respect to fairness constraint (S, T) iff $R^+ \cap \textsc{Fair}(S, T) \subseteq Q$ and Q is disjunctively well-founded.

Eliminating unfair paths

- Strategy: variables help to track unfair vs. fair paths

- Unfair paths lead trimmed out with use of **assume** or with constraints that make them well founded

- Termination proof is performed over the new program
  - Reachability-based approach: introduce extra variables into the translation
  - Invariance analysis: need only consider case where starting state is any reachable head of a fair path
  - Induction-based approach: …………………?

Expressing fairness

```
fairness {
  any { 1 }
  any { q==PENDING }
}
```

```
void f()
{
  AcquireLock();
        .
        .
        .
        .
        .
        .
        .
        .
        .
        .
        .
  ReleaseLock();
        .
        .
        .
        .
        .
}       .
        .
}void main()
{
void main()
{
    ...............
```

```
AcquireLock.entry
{
  if (s==NONE) {
    if (nondet()) {
      s=PENDING;
    }
  }
}


ReleaseLock.entry
{
  if (s==PENDING) {
    assume(false);
  }
}


main.entry {
  s=NONE;
}
```

Expressing fairness

```
fairness {
  any { 1 }
  any { q==PENDING }
}
```

```
void f()
{
  AcquireLock();
  if (s==NONE) {
    if (nondet()) {
      s=PENDING;
    }
  }
        .
        .
        .
        .
        .
  ReleaseLock();
  if (s==PENDING) {
    assume(false);
  }
        .
        .
}

void main()
{
  s=NONE;
```

Only paths s.t.
p==PENDING
forever are fair

Infinite loops
are added to
exit points

Expressing fairness

```
fairness {
  any { 1 }
  any { q==PENDING }
}
```

```
void f()
{
  AcquireLock();
  if (s==NONE) {
    if (nondet()) {
      s=PENDING;
    }
  }
          .
          .
          .
          .
          .
  ReleaseLock();
  if (s==PENDING) {
    assume(false);
  }
          .
          .
}

void main()
{
  s=NONE;
```

Note that this can only happen once

Only paths s.t. p==PENDING forever are fair

Infinite loops are added to exit points

Expressing fairness

```
fairness {
  any { 1 }
  any { s==PENDING }
}
```

```
void f()
{
  AcquireLock();
  if (s==NONE) {
    if (nondet()) {
      s=PENDING;
    }
  }
          .
          .
          .
          .
          .
  ReleaseLock();
  if (s==PENDING) {
    assume(false);
  }
          .
          .
}

void main()
{
  s=NONE;
```

Weak fairness

Liveness property library

```
void set() {
    if (q == NONE) {
        if (nondet()) {
            q = PENDING;
        }
    }
}

void unset() {
    if (q == PENDING) {
        assert(false);
    }
}

main.entry {
    q = NONE;
}

main.exit {
    if (q == PENDING) {
        error();
    }
}

fairness {
    any { 1 }
    any { q == PENDING }
}
```

```
state { int irql = -1; }

KeRaiseIrql.entry {
    if (irql == -1) {
        irql = KeGetCurrentIrql();
        set();
    }
}

KeLowerIrql.entry {
    if ($1 == irql && irql > -1) {
        unset();
    }
    irql = -1;
}
```

Liveness property library

```
void set() {
    if (q == NONE) {
        if (nondet()) {
            q = PENDING;
        }
    }
}

void unset() {
    if (q == PENDING) {
        assert(false);
    }
}

main.entry {
    q = NONE;
}

main.exit {
    if (q == PENDING) {
        error();
    }
}

fairness {
    any { 1 }
    any { q == PENDING }
}
```

```
state { int irql = -1; }

KeRaiseIrql.entry {
    if (irql == -1) {
        irql = KeGetCurrentIrql();
        set();
    }
}

KeLowerIrql.entry {
    if ($1 == irql && irql > -1) {
        unset();
    }
    irql = -1;
}
```

Liveness property from Windows OS kernel

Liveness property library

```
void set() {
    if (q == NONE) {
        if (nondet()) {
            q = PENDING;
        }
    }
}

void unset() {
    if (q == PENDING) {
        assert(false);
    }
}

main.entry {
    q = NONE;
}

main.exit {
    if (q == PENDING) {
        error();
    }
}

fairness {
    any { 1 }
    any { q == PENDING }
}
```

```
state { int irql = -1; }

KeRaiseIrql.entry {
    if (irql == -1) {
        irql = KeGetCurrentIrql();
        set();
    }
}

KeLowerIrql.entry {
    if ($1 == irql && irql > -1) {
        unset();
    }
    irql = -1;
}
```

Liveness property from Windows OS kernel

Transformation to reachability for fair termination

$T := \emptyset$

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

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

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

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

    $T := T \cup \trianglerighteq_f$

  else

    report "potential counterexample found: π_s,π_c

  fi

od

report "termination proved with argument T

Tr… for fair termination

Initialization:

```
• inS = 0;
• inT = 0;
• set = 0;
```

```
while(……) {
    body
}
```

```
if (*) {
    assume(set==1);
    assume(!inS || inT);
    assert(M);
} else if (*) {
    set = 1;
    inS = 0;
    inT = 0;
    `x = x;
    `y = y;
       .
       .
}
```

Add the following at each command in the program:

```
• if (S) inS=1;
• if (T) inT=1;
```

Tra

Predicates, interpolants, etc are computed on demand.

Thus: fairness does not add much overhead

```
        ...);
   ...sume(!inS || inT);
   ...rt(M);
   ... (*) {
   ... 1;
       inS = 0;
       inT = 0;
       `x = x;
       `y = y;
          .
          .
    }
  body
}
```

http://research.microsoft.com/en-us/um/cambridge/projects/terminator/popl07b.pdf - Windows Internet Explorer

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 Engineer-*

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

Outline

- Fair termination
- Data structures
- Concurrency
- Conclusion

ioctl.c [Read Only]* - Microsoft Visual Studio

File Edit View Project Debug Tools Test Window Community Help

ioctl.c*

```c
switch (IrpSp->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_SERIAL_GET_WAIT_MASK: { ... }
    case IOCTL_SERIAL_SET_WAIT_MASK: { ... }
    case IOCTL_SERIAL_WAIT_ON_MASK: { ... }
    case IOCTL_SERIAL_PURGE: {
        ULONG Mask=*((PULONG)Irp->AssociatedIrp.SystemBuffer);
        if (IrpSp->Parameters.DeviceIoControl.InputBufferLength <
            sizeof(ULONG)) {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }
       if (Mask & SERIAL_PURGE_RXABORT) {
            PIRP                Irp;
            KeAcquireSpinLock(
                &deviceExtension->SpinLock,
                &OldIrql
                );
            while ( !IsListEmpty(&deviceExtension->ReadQueue)) {
                PLIST_ENTRY         ListElement;
                PIRP                Irp;
                PIO_STACK_LOCATION  IrpSp;
                KIRQL               CancelIrql;
                ListElement=RemoveHeadList(
                    &deviceExtension->ReadQueue
                    );
                Irp=CONTAINING_RECORD(ListElement,IRP,
                        Tail.Overlay.ListEntry);
                IoAcquireCancelSpinLock(&CancelIrql);
                if (Irp->Cancel) {
                    /* ... */
                    Irp->IoStatus.Information=STATUS_CANCELLED;
                    IoReleaseCancelSpinLock(CancelIrql);
                    continue;
                }
                IoSetCancelRoutine( Irp, NULL);
                IoReleaseCancelSpinLock(CancelIrql);
                KeReleaseSpinLock( &deviceExtension->SpinLock, OldIrql);
                Irp->IoStatus.Information=0;
                RemoveReferenceAndCompleteRequest(
                    deviceExtension->DeviceObject, Irp, STATUS_CANCELLED);
                KeAcquireSpinLock( &deviceExtension->SpinLock, &OldIrql);
            }
```

Termination of programs with heap

- Where to get the termination argument?
  - Over changing delta in the heap shapes?
  - Over values stored in heap?
- Current approaches:
  - Finding abstractions of heap shapes expressed in arithmetic
  - New variables introduced track sizes of data structures
  - Proving termination over abstractions using arithmetic techniques
- Approach used here:
  - Perform separation logic based shape analysis
  - If memory safety proved, then we produce abstraction
  - Arithmetic techniques to prove termination of abstraction

Separation logic based shape analysis

Microsoft Research Cambridge

- Shape analysis: abstract interpretation for programs with heap
  - Goal: to prove memory safety
  - To prove memory safety you need to know A LOT about the shape of memory
  - Thus, we get other properties about the heap-shapes constructed during execution
  - Example: “at line 35 x is a pointer to a well-formed cyclic doubly-linked list”

Separation logic based shape analysis

→ Separation logic

- Classical logic (quantifiers, conjunction, etc)
- Extension:
  - emp : "The heaplet is empty"
  - $x \mapsto \mathsf{f} : y, \mathsf{d} : 5$: "The heaplet has exactly one cell x, holding a record with field f=y and field d=5."
  - $A * B$: "The heaplet can be divided so A is true of exactly one partition, and B is true of the other"
  - Induction definitions

Separation logic based shape an

$$\begin{aligned} \mathbf{ls}(x,y) &\triangleq \exists z.\ x \mapsto \mathsf{next} : z * \mathbf{ls}(z,y) \\ &\vee\ \ x = y \end{aligned}$$

- Sepa
  - Clas
  - Extension:
    - emp : "The heaplet is empty"
    - $x \mapsto \mathsf{f} : y, \mathsf{d} : 5$: "The heaplet has exactly one cell x, holding a record with field f=y and field d=5."
    - $A * B$: "The heaplet can be divided so A is true of exactly one partition, and B is true of the other"
    - Induction definitions

- Sepa
  - Clas
  - Extension:

$$\begin{aligned} \mathbf{ls}(x, y) \quad &\triangleq \quad \exists z.\ x \mapsto \mathsf{next} : z * \mathbf{ls}(z, y) \\ &\vee \quad x = y \end{aligned}$$

$$\begin{aligned} \mathbf{ls}(k, x, y) \quad &\triangleq \quad k \geq 1 \wedge \exists z.\ x \mapsto \mathsf{next} : z * \mathbf{ls}(k-1, z, y) \\ &\vee \quad k = 0 \wedge x = y \end{aligned}$$

Separation logic based shape analysis

- Cyclic lists?
  - $\exists y.\mathbf{ls}(x, y) * \mathbf{ls}(y, x)$
- Acyclic lists?
  - $\exists y.\mathbf{ls}(x, 0)$
- "Pan handle lists"?
  - $\exists y, z.\mathbf{ls}(x, y) * \mathbf{ls}(y, z) * \mathbf{ls}(z, y)$

Separation logic based shape analysis

- Double linked lists? ✓
- Sorted lists? ✓
- Lists of lists? ✓
- Lists with back edges to head nodes? ✓
- Trees? Balanced trees? ✓
- Skiplists? ✓
- DAGs? BDDs? ✗

Microsoft Research Cambridge

→ Separation logic based shape analysis:

- Sets of *-conjuncted formulae represent abstract heaps at program locations
  - *e.g.* $\ell : \mathbf{ls}(k, \mathsf{x}, 0)$ "The program's heap when executing the command at location ℓ consists only of an acyclic list pointed to by x"
- Forward symbolic simulation, *e.g.* $\{\mathbf{ls}(k, \mathsf{x}, y) \wedge k \geq 1\}$

$$\mathsf{x}{:=}(*\mathsf{x}).\mathsf{next}$$

$$\{\mathbf{ls}(k, \mathsf{x}, y) \wedge k \geq 0\}$$

Separation logic based shape analysis:

- Use of abstraction to improve *the chance* of analysis-termination, *e.g.*

$$\alpha(\exists y.\mathbf{ls}(x,y) * \mathbf{ls}(y,z)) = \mathbf{ls}(x,z)$$
$$\alpha(x \mapsto \textsf{next} : y) = \exists k.\mathbf{ls}(k,x,y) \wedge k \geq 1$$

- Summaries for procedures, and “Frame Rule”:
  - if $\{A\}p(x)\{B\}$, then forall H, $\{A*H\}p(x)\{B*H\}$

Separation logic based shape analysis

$$\mathsf{x} := 0$$

$$\begin{array}{l}
\mathbf{while} * \mathbf{do} \\
\quad \mathsf{y} := \mathbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\mathbf{done}
\end{array}$$

$$\begin{array}{l}
\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\mathbf{done}
\end{array}$$

Separation logic based shape analysis

$\mathbf{emp}$

$$
\begin{array}{l}
\mathsf{x} := 0 \\
\\
\textbf{while} * \textbf{do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done} \\
\\
\textbf{while } \mathsf{x} \neq 0 \textbf{ do} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\textbf{done}
\end{array}
$$

Separation logic based shape analysis

$\mathbf{emp}$

$\mathsf{x} := 0$

$\mathbf{emp} \wedge \mathsf{x} = 0$

$\mathbf{while} \ * \ \mathbf{do}$

$\quad \mathsf{y} := \mathbf{malloc}()$

$\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x}$

$\quad \mathsf{x} := \mathsf{y}$

$\mathbf{done}$

$\mathbf{while} \ \mathsf{x} \neq 0 \ \mathbf{do}$

$\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next}$

$\mathbf{done}$

Separation logic based shape analysis

$$\mathbf{emp}$$

$$x := 0$$

$$\mathbf{emp} \wedge x = 0$$

$$\mathbf{while} * \mathbf{do}$$

$$\quad y := \mathbf{malloc}()$$

$$\mathbf{emp} * y \mapsto \mathsf{next}:\_ \wedge x = 0$$

$$\quad (*y).\mathsf{next} := x$$

$$\quad x := y$$

$$\mathbf{done}$$

$$\mathbf{while}\ x \neq 0\ \mathbf{do}$$

$$\quad x := (*x).\mathsf{next}$$

$$\mathbf{done}$$

Separation logic based shape analysis

Microsoft Research Cambridge

$\mathbf{emp}$

$\mathsf{x} := 0$

$\mathbf{emp} \wedge \mathsf{x} = 0$

$\mathbf{while} * \mathbf{do}$

$\quad \mathsf{y} := \mathbf{malloc}()$

$\mathbf{emp} * \mathsf{y} \mapsto \mathsf{next} :\_ \wedge \mathsf{x} = 0$

$\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x}$

$\mathbf{emp} * \mathsf{y} \mapsto \mathsf{next} : 0 \wedge \mathsf{x} = 0$

$\quad \mathsf{x} := \mathsf{y}$

$\mathbf{done}$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$

$\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next}$

$\mathbf{done}$

Separation logic based shape analysis

Microsoft Research Cambridge

$$\mathbf{emp}$$

$$\mathsf{x} := 0$$

$$\mathbf{emp} \wedge \mathsf{x} = 0$$

$$\begin{array}{l}
\mathbf{while} * \mathbf{do} \\
\quad \mathsf{y} := \mathbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\mathbf{done}
\end{array}$$

$$\mathbf{emp} * \mathsf{y} \mapsto \mathsf{next}:\_ \wedge \mathsf{x} = 0$$

$$\mathbf{emp} * \mathsf{y} \mapsto \mathsf{next}:0 \wedge \mathsf{x} = 0$$

$$\mathbf{emp} * \mathsf{x}, \mathsf{y} \mapsto \mathsf{next}:0$$

$$\begin{array}{l}
\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\mathbf{done}
\end{array}$$

Separation logic based shape analysis

Microsoft Research Cambridge

$$\mathbf{emp}$$

$$\mathsf{x} := 0$$

$$\mathbf{emp} \wedge \mathsf{x} = 0$$

$$\mathbf{while} * \mathbf{do}$$

$$\mathsf{y} := \mathbf{malloc}()$$

$$\mathbf{emp} * \mathsf{y} \mapsto \mathsf{next}: \_ \wedge \mathsf{x} = 0$$

$$(*\mathsf{y}).\mathsf{next} := \mathsf{x}$$

$$\mathbf{emp} * \mathsf{y} \mapsto \mathsf{next}: 0 \wedge \mathsf{x} = 0$$

$$\mathsf{x} := \mathsf{y}$$

$$\mathbf{done}$$

$$\mathbf{emp} * \mathsf{x}, \mathsf{y} \mapsto \mathsf{next}: 0$$

cleanup

$$\mathsf{x}, \mathsf{y} \mapsto \mathsf{next}: 0$$

$$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$$

$$\mathsf{x} := (*\mathsf{x}).\mathsf{next}$$

$$\mathbf{done}$$

Separation logic based shape analysis

$$\mathbf{emp}$$

$$x := 0$$

$$\mathbf{emp} \wedge x = 0$$

$$\mathbf{while} * \mathbf{do}$$

$$\quad y := \mathbf{malloc}()$$

$$\mathbf{emp} * y \mapsto \mathsf{next} : \_ \wedge x = 0$$

$$\quad (*y).\mathsf{next} := x$$

$$\mathbf{emp} * y \mapsto \mathsf{next} : 0 \wedge x = 0$$

$$\quad x := y$$

$$\mathbf{done}$$

$$\mathbf{emp} * x, y \mapsto \mathsf{next} : 0$$

cleanup

$$x, y \mapsto \mathsf{next} : 0$$

α

$$\mathbf{ls}(x, 0) \wedge x = y$$

$$\mathbf{while}\ x \neq 0\ \mathbf{do}$$

$$\quad x := (*x).\mathsf{next}$$

$$\mathbf{done}$$

Separation logic based shape analysis

$x := 0$

$\mathbf{while} \ast \mathbf{do}$

$\quad y := \mathbf{malloc}()$

$\quad (\ast y).\mathsf{next} := x$

$\quad x := y$

$\mathbf{done}$

$\mathbf{while}\ x \neq 0\ \mathbf{do}$

$\quad x := (\ast x).\mathsf{next}$

$\mathbf{done}$

$\mathbf{emp} \wedge x = 0$

$\mathbf{ls}(x, 0) \wedge x = y$

Separation logic based shape analysis

$$\mathbf{emp} \wedge \mathsf{x} = 0 \;\vee\; \mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}$$

$$
\begin{array}{l}
\mathsf{x} := 0 \\
\\
\mathbf{while} * \mathbf{do} \\
\quad \mathsf{y} := \mathbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\mathbf{done} \\
\\
\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\mathbf{done}
\end{array}
$$

Separation logic based shape analysis

$$\mathsf{x} := 0$$

$$\mathbf{while} * \mathbf{do}$$
$$\quad \mathsf{y} := \mathbf{malloc}()$$
$$\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x}$$
$$\quad \mathsf{x} := \mathsf{y}$$
$$\mathbf{done}$$

$$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$$
$$\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next}$$
$$\mathbf{done}$$

$$\mathbf{emp} \wedge \mathsf{x} = 0$$
$$\vee$$
$$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$$

Separation logic based shape analysis

Microsoft Research Cambridge

$$\mathbf{emp} \wedge \mathsf{x} = 0$$
$$\vee$$
$$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$$

$$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next} :\_$$

$\mathsf{x} := 0$

while $*$ **do**
  $\mathsf{y} :=$ **malloc**()
  $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
  $\mathsf{x} := \mathsf{y}$
done

while $\mathsf{x} \neq 0$ **do**
  $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$
done

$$\mathbf{emp} \wedge \mathsf{x} = 0$$
$$\vee$$
$$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$$

$\mathsf{x} := 0$

$\mathbf{while} * \mathbf{do}$
    $\mathsf{y} := \mathbf{malloc}()$
    $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
    $\mathsf{x} := \mathsf{y}$
$\mathbf{done}$

$$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next} :\_$$

$$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next} :\mathsf{x}$$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$
    $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$
$\mathbf{done}$

Separation logic based shape analysis

Microsoft Research Cambridge

$\mathbf{emp} \wedge \mathsf{x} = 0$

$\vee$

$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$

$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next}: \_$

$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next}: \mathsf{x}$

$\exists n.\ \mathbf{ls}(n, 0) \wedge \mathsf{x}, \mathsf{y} \mapsto \mathsf{next}: n$

$\mathsf{x} := 0$

$\mathbf{while} * \mathbf{do}$
    $\mathsf{y} := \mathbf{malloc}()$
    $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
    $\mathsf{x} := \mathsf{y}$
$\mathbf{done}$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$
    $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$
$\mathbf{done}$

Separation logic based shape analysis

$\mathsf{x} := 0$

$\mathbf{while} * \mathbf{do}$

$\quad \mathsf{y} := \mathbf{malloc}()$

$\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x}$

$\quad \mathsf{x} := \mathsf{y}$

$\mathbf{done}$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$

$\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next}$

$\mathbf{done}$

$\mathbf{emp} \wedge \mathsf{x} = 0$

$\vee$

$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$

$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next} :\_$

$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{y} \mapsto \mathsf{next} : \mathsf{x}$

$\exists n.\ \mathbf{ls}(n, 0) \wedge \mathsf{x}, \mathsf{y} \mapsto \mathsf{next} : n$

$\downarrow \alpha$

$\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}$

Separation logic based shape analysis

$$\mathbf{emp} \wedge \mathsf{x} = 0$$
$$\vee$$
$$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$$

$\mathsf{x} := 0$

while $*$ **do**
  $\mathsf{y} := \mathbf{malloc}()$
  $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
  $\mathsf{x} := \mathsf{y}$
done

✓

while $\mathsf{x} \neq 0$ **do**
  $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$
done

$$\boxed{\mathbf{ls}(\mathsf{x}, 0) \wedge \mathsf{x} = \mathsf{y}}$$

Separation logic based shape analysis

Microsoft Research Cambridge

$$
\begin{array}{l}
\mathsf{x} := 0 \\
\\
\textbf{while } * \textbf{ do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done} \\
\\
\textbf{while } \mathsf{x} \neq 0 \textbf{ do} \qquad \qquad \mathbf{ls}(\mathrm{x}, 0) \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\textbf{done}
\end{array}
$$

Separation logic based shape analysis

$$
\begin{array}{l}
\mathsf{x} := 0 \\
\\
\mathbf{while} * \mathbf{do} \\
\quad \mathsf{y} := \mathbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\mathbf{done} \\
\\
\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\mathbf{done}
\end{array}
$$

$\mathbf{ls}(\mathsf{x}, 0)$

$\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0)$

Separation logic based shape analysis

$$\mathsf{x} := 0$$

$$\begin{array}{l}
\textbf{while} * \textbf{do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done}
\end{array}$$

$$\begin{array}{l}
\textbf{while } \mathsf{x} \neq 0 \textbf{ do} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\textbf{done}
\end{array}$$

$\mathbf{ls}(\mathsf{x}, 0)$

$\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0)$

$\exists m.\ m \mapsto \mathsf{next} : \mathsf{x} * \mathbf{ls}(\mathsf{x}, 0)$

Separation logic based shape analysis

$$\mathsf{x} := 0$$

$$\begin{array}{l}
\textbf{while} * \textbf{do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done}
\end{array}$$

$$\begin{array}{ll}
\textbf{while}\ \mathsf{x} \neq 0\ \textbf{do} & \longleftarrow \mathbf{ls}(\mathsf{x}, 0) \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} & \longleftarrow \exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0) \\
\textbf{done} & \longleftarrow \exists m.\ m \mapsto \mathsf{next} : \mathsf{x} * \mathbf{ls}(\mathsf{x}, 0)
\end{array}$$

X

Separation logic based shape analysis

Microsoft Research Cambridge

$$
\begin{array}{l}
\mathsf{x} := 0 \\
\\
\textbf{while} * \textbf{do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done} \\
\\
\textbf{while}\ \mathsf{x} \neq 0\ \textbf{do} \\
\quad \mathsf{t} := \mathsf{x} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\quad \textbf{free}(\mathsf{t}) \\
\textbf{done}
\end{array}
$$

$\mathbf{ls}(\mathsf{x}, 0)$

$\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0)$

$\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0) \wedge \mathsf{t} = \mathsf{x}$

$\mathsf{x} := 0$

$\mathbf{while} * \mathbf{do}$
    $\mathsf{y} := \mathbf{malloc}()$
    $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
    $\mathsf{x} := \mathsf{y}$
$\mathbf{done}$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$ $\mathbf{ls}(\mathsf{x}, 0)$
    $\mathsf{t} := \mathsf{x}$ $\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0)$
    $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$ $\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0) \wedge \mathsf{t} = \mathsf{x}$
    $\mathbf{free}(\mathsf{t})$ $\mathsf{t} \mapsto \mathsf{next} : \mathsf{x} * \mathbf{ls}(\mathsf{x}, 0)$
$\mathbf{done}$

Separation logic based shape analysis

$$
\begin{array}{l}
\mathsf{x} := 0 \\
\\
\textbf{while} * \textbf{do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{ls}(\mathsf{x}, 0) \\
\textbf{while}\ \mathsf{x} \neq 0\ \textbf{do} & \\
 & \exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0) \\
\quad \mathsf{t} := \mathsf{x} & \\
 & \exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(n, 0) \wedge \mathsf{t} = \mathsf{x} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} & \\
 & \mathsf{t} \mapsto \mathsf{next} : \mathsf{x} * \mathbf{ls}(\mathsf{x}, 0) \\
\quad \textbf{free}(\mathsf{t}) & \\
\textbf{done} & \mathbf{ls}(\mathsf{x}, 0)
\end{array}
$$

Separation logic based shape analysis

$\mathsf{x} := 0$

$\mathbf{while} * \mathbf{do}$
    $\mathsf{y} := \mathbf{malloc}()$
    $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
    $\mathsf{x} := \mathsf{y}$
$\mathbf{done}$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$ $\boxed{\mathrm{ls}(\mathsf{x}, 0)}$
    $\mathsf{t} := \mathsf{x}$
    $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$ ✓
    $\mathbf{free}(\mathsf{t})$
$\mathbf{done}$ $\boxed{\mathrm{ls}(\mathsf{x}, 0)}$

Separation logic based shape analysis

$$\mathsf{x} := 0$$

$$\begin{array}{l}
\mathbf{while} * \mathbf{do} \\
\quad \mathsf{y} := \mathbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\mathbf{done}
\end{array}$$

$$\begin{array}{l}
\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do} \\
\quad \mathsf{t} := \mathsf{x} \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} \\
\quad \mathbf{free}(\mathsf{t}) \\
\mathbf{done}
\end{array}$$

$\mathbf{ls}(\mathsf{x}, 0)$

Separation logic based shape analysis

$\mathsf{x} := 0$

$\mathbf{while} * \mathbf{do}$
$\quad \mathsf{y} := \mathbf{malloc}()$
$\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x}$
$\quad \mathsf{x} := \mathsf{y}$
$\mathbf{done}$

$\mathbf{while}\ \mathsf{x} \neq 0\ \mathbf{do}$ $\qquad \mathbf{ls}(k_1, \mathsf{x}, 0)$
$\quad \mathsf{t} := \mathsf{x}$
$\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next}$
$\quad \mathbf{free}(\mathsf{t})$
$\mathbf{done}$

Separation logic based shape analysis

$\mathsf{x} := 0$

$\textbf{while} * \textbf{do}$
  $\mathsf{y} := \textbf{malloc}()$
  $(*\mathsf{y}).\mathsf{next} := \mathsf{x}$
  $\mathsf{x} := \mathsf{y}$
$\textbf{done}$

$\textbf{while } \mathsf{x} \neq 0 \textbf{ do}$ — $\mathbf{ls}(k_1, \mathsf{x}, 0)$
  $\mathsf{t} := \mathsf{x}$ — $\exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(k_2, n, 0) \wedge k_2 = k_1 - 1$
  $\mathsf{x} := (*\mathsf{x}).\mathsf{next}$ — $\cdots$
  $\textbf{free}(\mathsf{t})$
$\textbf{done}$ — $\mathbf{ls}(k_5, \mathsf{x}, 0) \wedge k_5 = k_4$

Microsoft Research Cambridge

$$\mathsf{x} := 0$$

$$\begin{array}{l}
\textbf{while} * \textbf{do} \\
\quad \mathsf{y} := \textbf{malloc}() \\
\quad (*\mathsf{y}).\mathsf{next} := \mathsf{x} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{done}
\end{array}$$

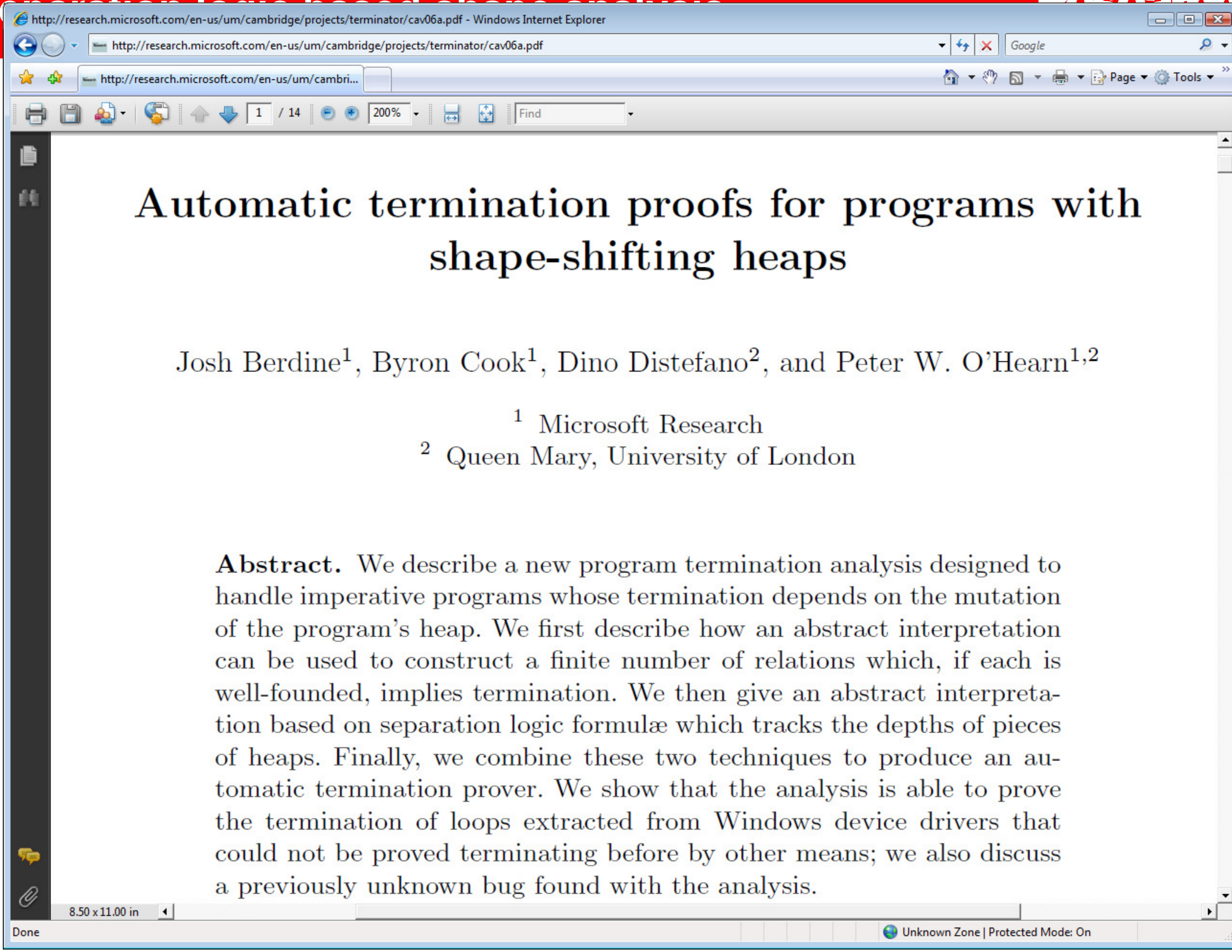
substitution: $k_1 := k_5$

$$\begin{array}{ll}
 & \boxed{\mathbf{ls}(k_1, \mathsf{x}, 0)} \\
\textbf{while } \mathsf{x} \neq 0 \textbf{ do} & \\
\quad \mathsf{t} := \mathsf{x} & \exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(k_2, n, 0) \wedge k_2 = k_1 - 1 \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} & \cdots \\
\quad \textbf{free}(\mathsf{t}) & \boxed{\mathbf{ls}(k_5, \mathsf{x}, 0) \wedge k_5 = k_4} \\
\textbf{done} &
\end{array}$$

Separation logic based shape analysis

$$\begin{array}{l}
\textbf{while } \mathsf{k}_1 \neq 0 \textbf{ do} \\
\quad \mathsf{k}_2 := \mathsf{k}_1 - 1 \\
\quad \mathsf{k}_3 := \mathsf{k}_2 \\
\quad \mathsf{k}_4 := \mathsf{k}_3 \\
\quad \mathsf{k}_5 := \mathsf{k}_4 \\
\quad \mathsf{k}_1 := \mathsf{k}_5 \\
\textbf{done}
\end{array}$$

$$\begin{array}{ll}
\textbf{while } \mathsf{x} \neq 0 \textbf{ do} & \boxed{\mathbf{ls}(k_1, \mathsf{x}, 0)} \\
\quad \mathsf{t} := \mathsf{x} & \exists n.\ \mathsf{x} \mapsto \mathsf{next} : n * \mathbf{ls}(k_2, n, 0) \wedge k_2 = k_1 - 1 \\
\quad \mathsf{x} := (*\mathsf{x}).\mathsf{next} & \cdots \\
\quad \mathbf{free}(\mathsf{t}) & \\
\textbf{done} & \boxed{\mathbf{ls}(k_5, \mathsf{x}, 0) \wedge k_5 = k_4}
\end{array}$$



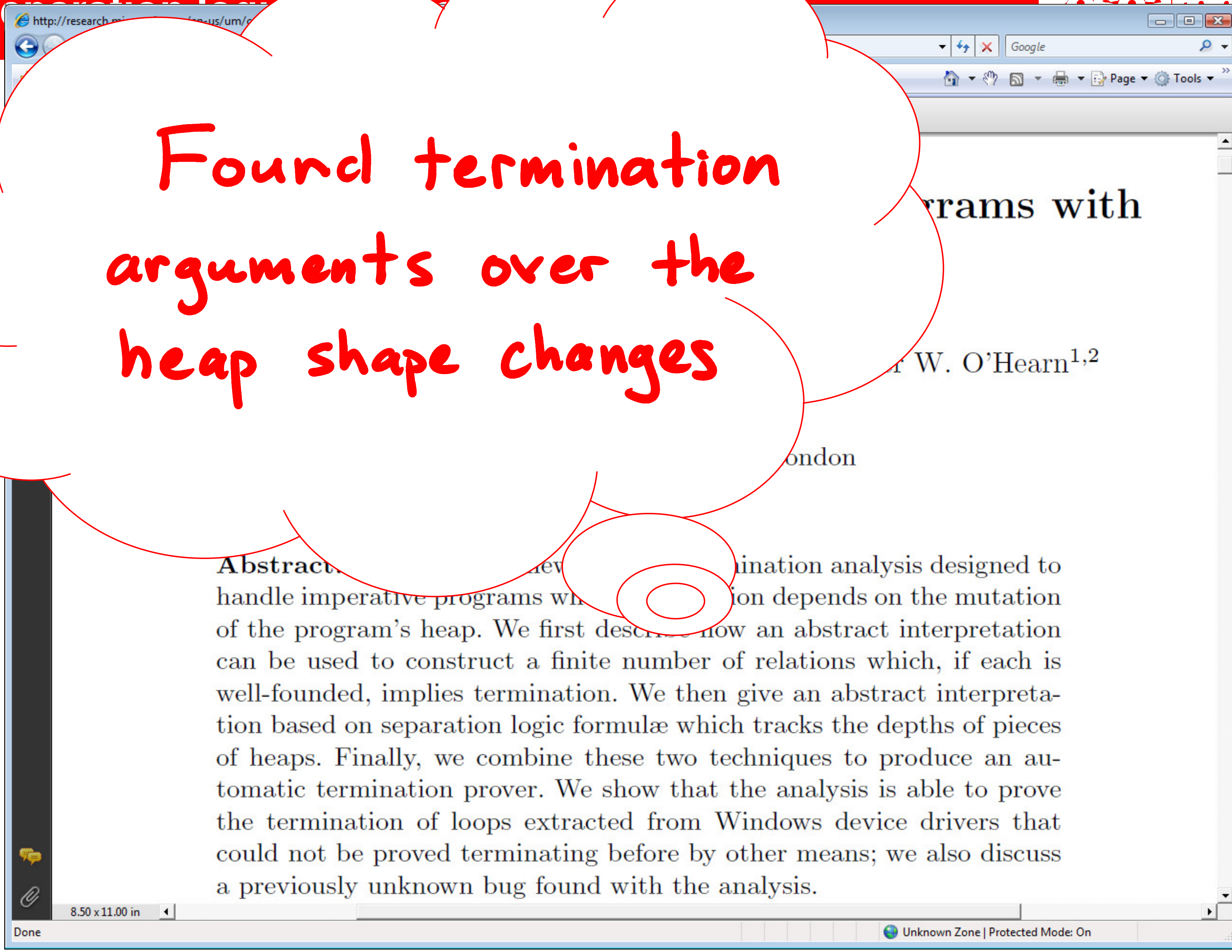
Automatic termination proofs for programs with shape-shifting heaps

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.

Found termination arguments over the heap shape changes



Abstract. ... ination analysis designed to handle imperative programs wh... ion depends on the mutation of the program's heap. We first desc... 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.

Separation logic based shape analysis

http://research.microsoft.com/en-us/um/cambridge/projects/slayer/sas07.pdf

Arithmetic Strengthening for Shape Analysis*

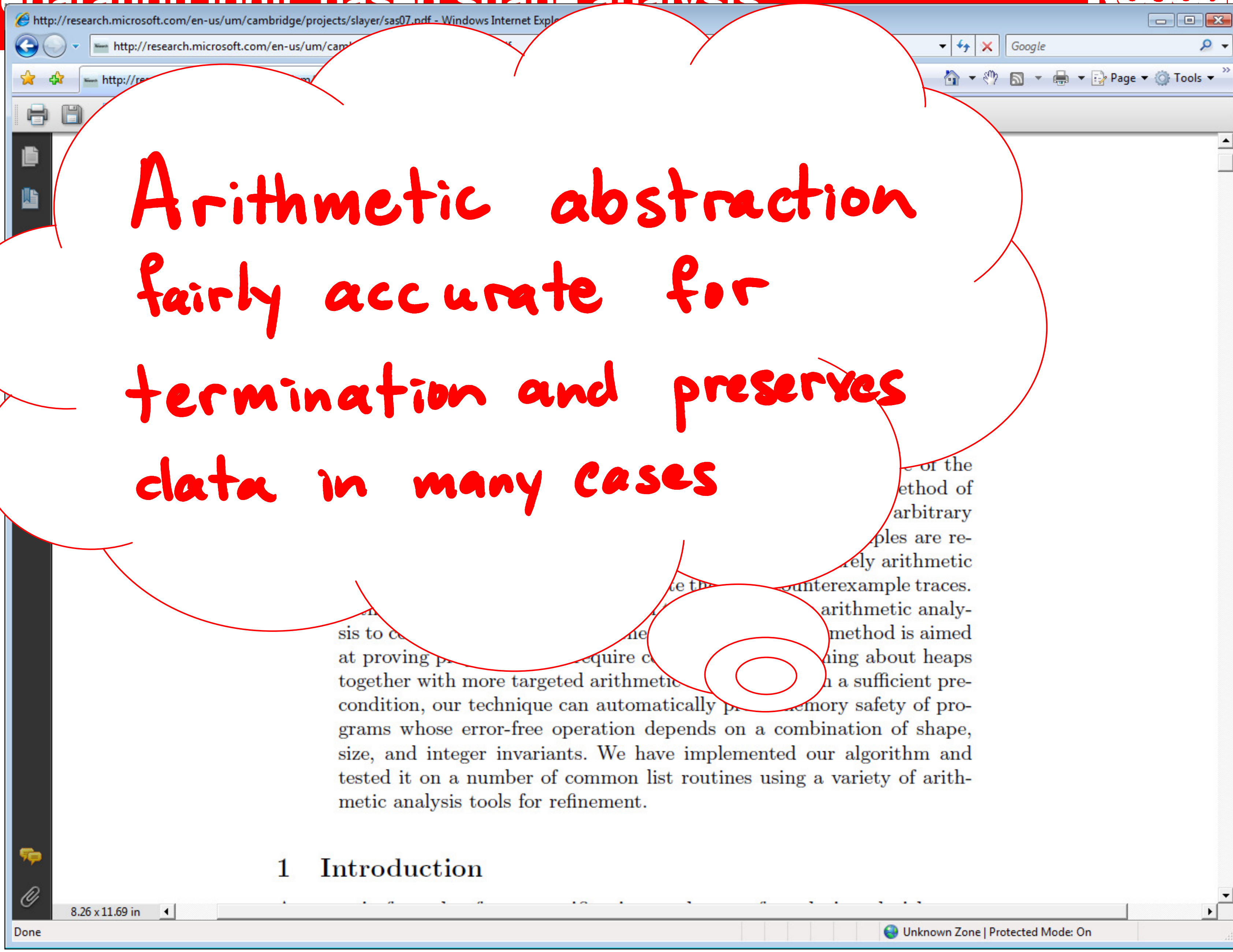
Stephen Magill[2], Josh Berdine[1], Edmund Clarke[2], and Byron Cook[1]

[1] Carnegie Mellon University
[2] Microsoft Research

Abstract. Shape analyses are often imprecise in their numerical reasoning, whereas numerical static analyses are often largely unaware of the shape of a program's heap. In this paper we propose a lazy method of combining a shape analysis based on separation logic with an arbitrary arithmetic analysis. When potentially spurious counterexamples are reported by our shape analysis, the method constructs a purely arithmetic program whose traces over-approximate the set of counterexample traces. It then uses this arithmetic program together with the arithmetic analysis to construct a refinement for the shape analysis. Our method is aimed at proving properties that require comprehensive reasoning about heaps together with more targeted arithmetic reasoning. Given a sufficient precondition, our technique can automatically prove memory safety of programs whose error-free operation depends on a combination of shape, size, and integer invariants. We have implemented our algorithm and tested it on a number of common list routines using a variety of arithmetic analysis tools for refinement.

1 Introduction

Separation logic based shape analysis



Arithmetic abstraction fairly accurate for termination and preserves data in many cases

Outline

- Fair termination
- Data structures
- Concurrency
- Conclusion

ioctl.c [Read Only]* - Microsoft Visual Studio

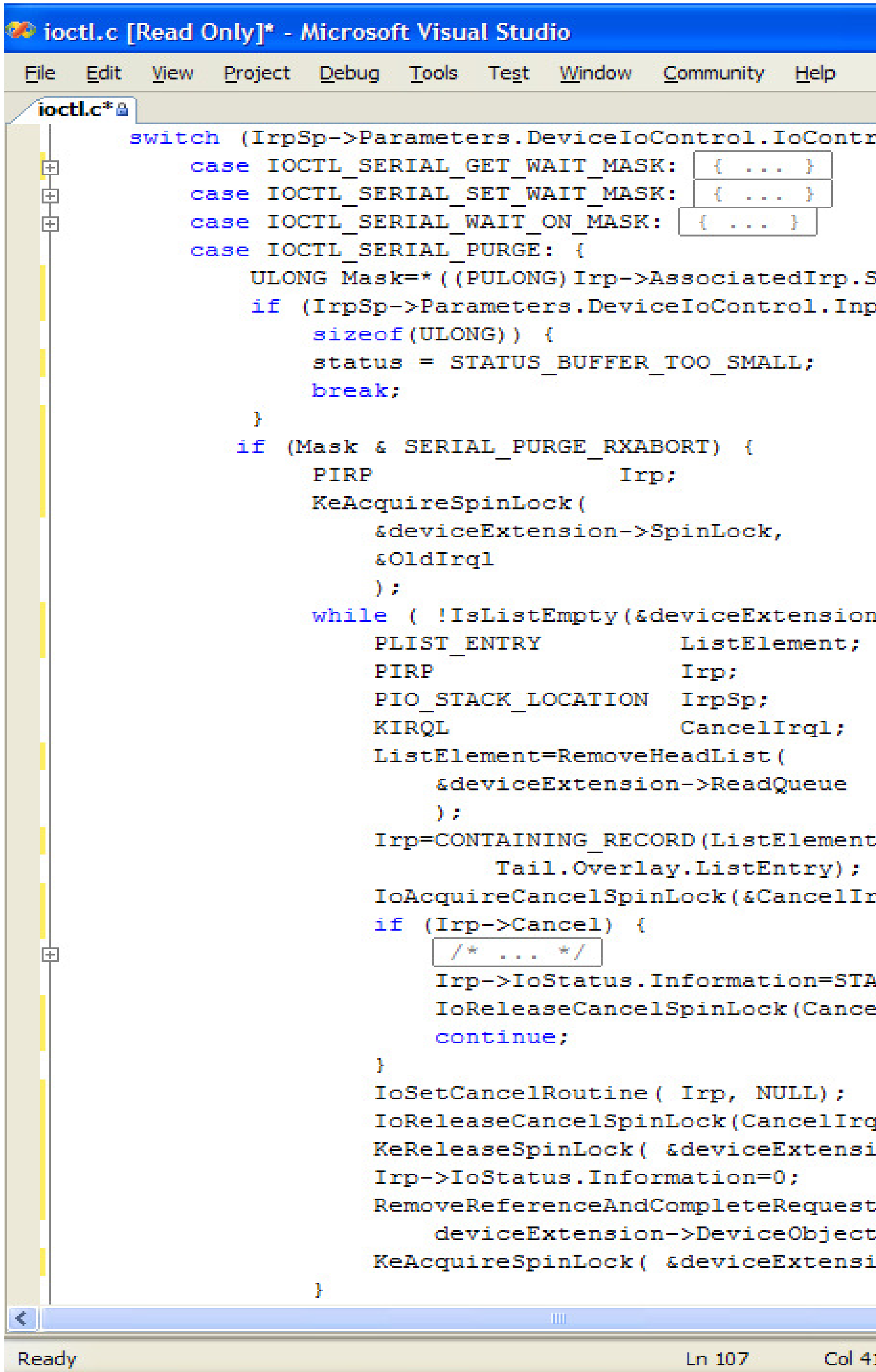
File Edit View Project Debug Tools Test Window Community Help

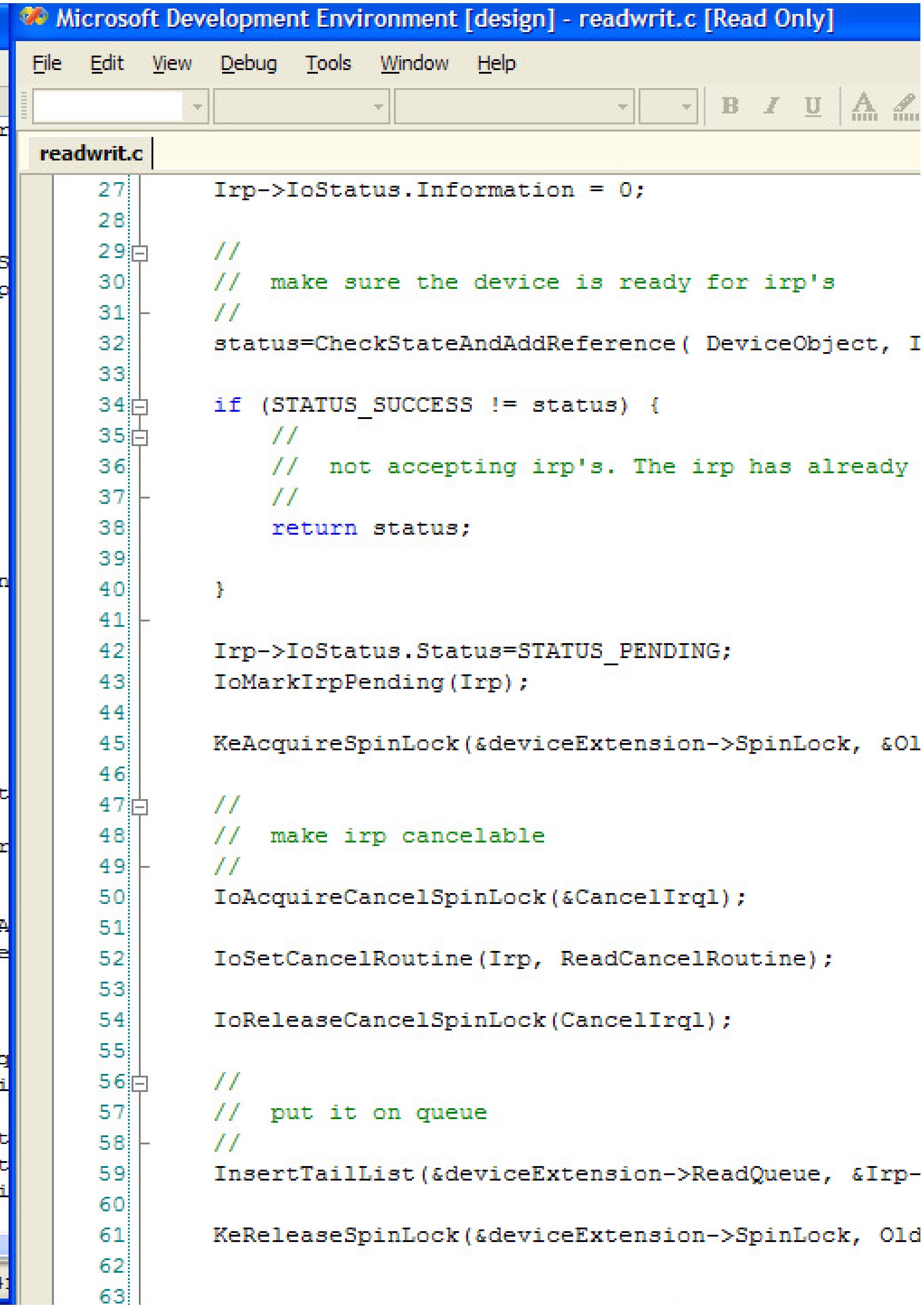
ioctl.c*

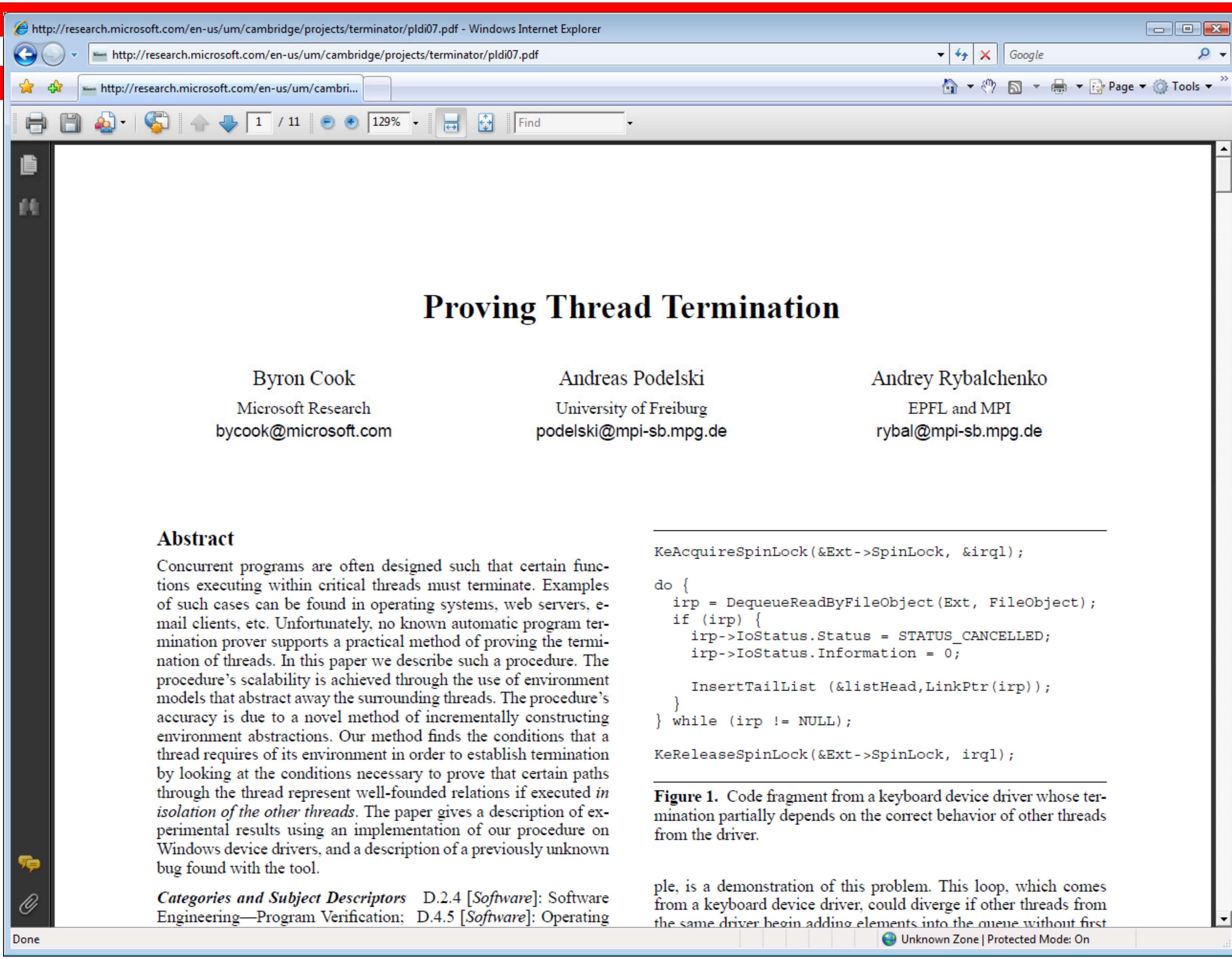
```c
    switch (IrpSp->Parameters.DeviceIoControl.IoControlCode) {
        case IOCTL_SERIAL_GET_WAIT_MASK: { ... }
        case IOCTL_SERIAL_SET_WAIT_MASK: { ... }
        case IOCTL_SERIAL_WAIT_ON_MASK: { ... }
        case IOCTL_SERIAL_PURGE: {
            ULONG Mask=*((PULONG)Irp->AssociatedIrp.SystemBuffer);
            if (IrpSp->Parameters.DeviceIoControl.InputBufferLength <
                sizeof(ULONG)) {
                status = STATUS_BUFFER_TOO_SMALL;
                break;
            }
           if (Mask & SERIAL_PURGE_RXABORT) {
                PIRP                Irp;
                KeAcquireSpinLock(
                    &deviceExtension->SpinLock,
                    &OldIrql
                    );
                while ( !IsListEmpty(&deviceExtension->ReadQueue)) {
                    PLIST_ENTRY          ListElement;
                    PIRP                 Irp;
                    PIO_STACK_LOCATION   IrpSp;
                    KIRQL                CancelIrql;
                    ListElement=RemoveHeadList(
                        &deviceExtension->ReadQueue
                        );
                    Irp=CONTAINING_RECORD(ListElement,IRP,
                            Tail.Overlay.ListEntry);
                    IoAcquireCancelSpinLock(&CancelIrql);
                    if (Irp->Cancel) {
                        /* ... */
                        Irp->IoStatus.Information=STATUS_CANCELLED;
                        IoReleaseCancelSpinLock(CancelIrql);
                        continue;
                    }
                    IoSetCancelRoutine( Irp, NULL);
                    IoReleaseCancelSpinLock(CancelIrql);
                    KeReleaseSpinLock( &deviceExtension->SpinLock, OldIrql);
                    Irp->IoStatus.Information=0;
                    RemoveReferenceAndCompleteRequest(
                        deviceExtension->DeviceObject, Irp, STATUS_CANCELLED);
                    KeAcquireSpinLock( &deviceExtension->SpinLock, &OldIrql);
                }
```

Ready Ln 107 Col 41 Ch 41 INS

Microsoft Research Cambridge


ioctl.c [Read Only]* - Microsoft Visual Studio
File Edit View Project Debug Tools Test Window Community Help
ioctl.c*
switch (IrpSp->Parameters.DeviceIoControl.IoContr
case IOCTL_SERIAL_GET_WAIT_MASK: { ... }
case IOCTL_SERIAL_SET_WAIT_MASK: { ... }
case IOCTL_SERIAL_WAIT_ON_MASK: { ... }
case IOCTL_SERIAL_PURGE: {
ULONG Mask=*((PULONG)Irp->AssociatedIrp.S
if (IrpSp->Parameters.DeviceIoControl.Inp
sizeof(ULONG)) {
status = STATUS_BUFFER_TOO_SMALL;
break;
}
if (Mask & SERIAL_PURGE_RXABORT) {
PIRP Irp;
KeAcquireSpinLock(
&deviceExtension->SpinLock,
&OldIrql
);
while (!IsListEmpty(&deviceExtension
PLIST_ENTRY ListElement;
PIRP Irp;
PIO_STACK_LOCATION IrpSp;
KIRQL CancelIrql;
ListElement=RemoveHeadList(
&deviceExtension->ReadQueue
);
Irp=CONTAINING_RECORD(ListElement
Tail.Overlay.ListEntry);
IoAcquireCancelSpinLock(&CancelIr
if (Irp->Cancel) {
/* ... */
Irp->IoStatus.Information=STA
IoReleaseCancelSpinLock(Cance
continue;
}
IoSetCancelRoutine(Irp, NULL);
IoReleaseCancelSpinLock(CancelIrq
KeReleaseSpinLock(&deviceExtensi
Irp->IoStatus.Information=0;
RemoveReferenceAndCompleteRequest
deviceExtension->DeviceObject
KeAcquireSpinLock(&deviceExtensi
}
Ready
Ln 107
Col 4


Microsoft Development Environment [design] - readwrit.c [Read Only]
File Edit View Debug Tools Window Help
readwrit.c
Irp->IoStatus.Information = 0;
//
// make sure the device is ready for irp's
//
status=CheckStateAndAddReference(DeviceObject, I
if (STATUS_SUCCESS != status) {
//
// not accepting irp's. The irp has already
//
return status;
}
Irp->IoStatus.Status=STATUS_PENDING;
IoMarkIrpPending(Irp);
KeAcquireSpinLock(&deviceExtension->SpinLock, &Ol
//
// make irp cancelable
//
IoAcquireCancelSpinLock(&CancelIrql);
IoSetCancelRoutine(Irp, ReadCancelRoutine);
IoReleaseCancelSpinLock(CancelIrql);
//
// put it on queue
//
InsertTailList(&deviceExtension->ReadQueue, &Irp-
KeReleaseSpinLock(&deviceExtension->SpinLock, Old



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

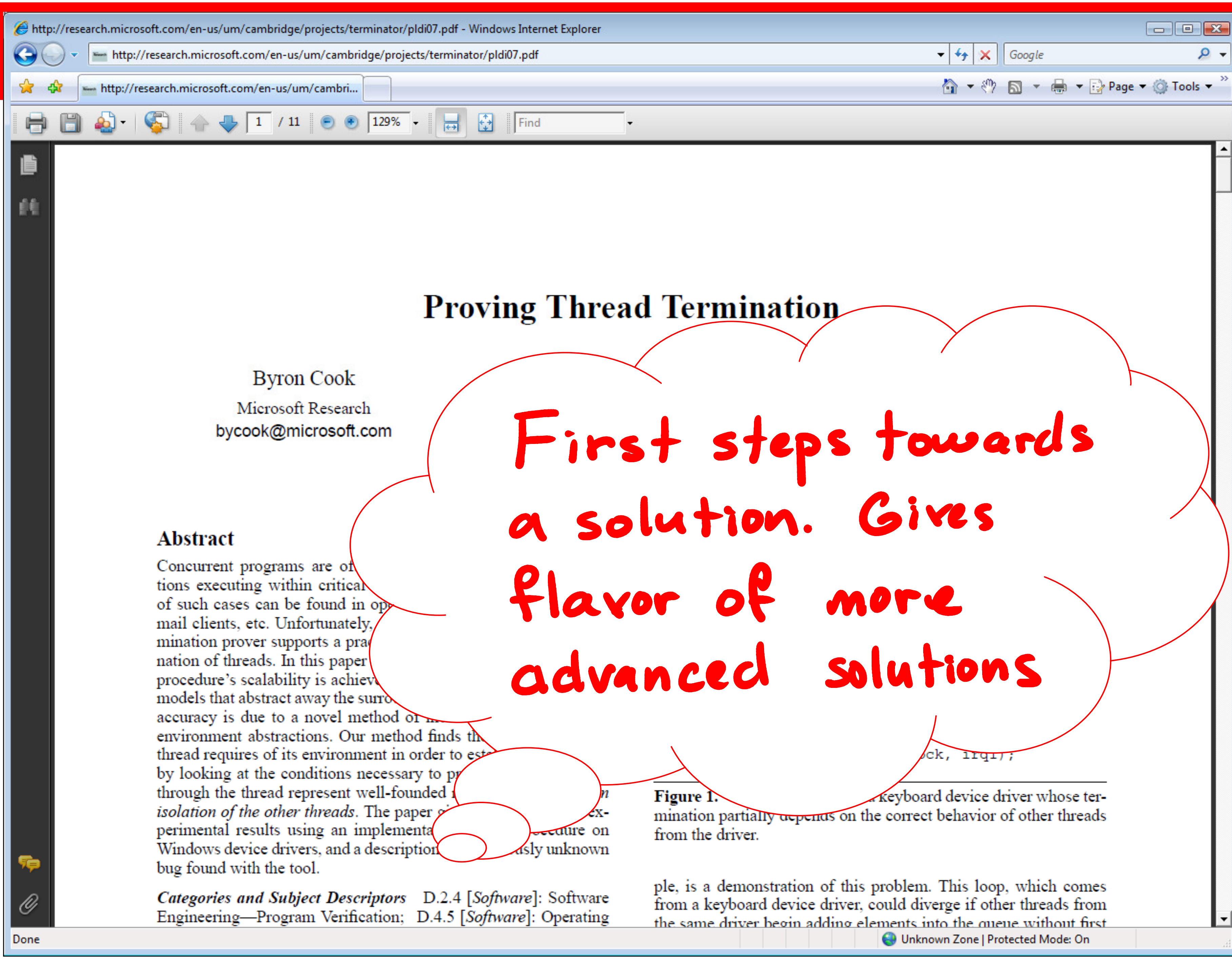
Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Examples of such cases can be found in operating systems, web servers, e-mail clients, etc. Unfortunately, no known automatic program termination prover supports a practical method of 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.

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

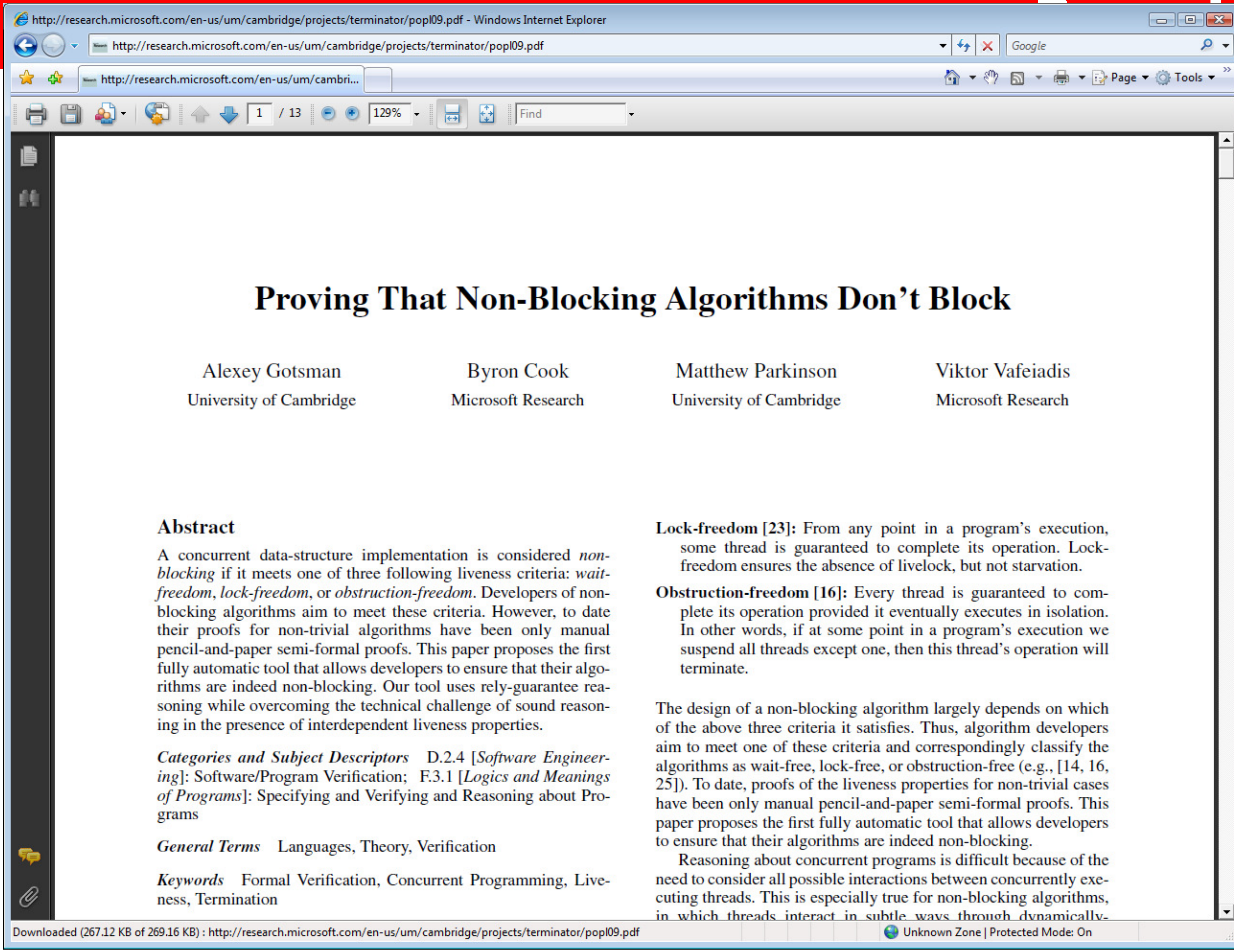
```
KeAcquireSpinLock(&Ext->SpinLock, &irql);

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 from a keyboard device driver, could diverge if other threads from the same driver begin adding elements into the queue without first



http://research.microsoft.com/en-us/um/cambridge/projects/terminator/pldi07.pdf - Windows Internet Explorer
1 / 11
129%
Proving Thread Termination
Byron Cook
Microsoft Research
bycook@microsoft.com
Abstract
First steps towards a solution. Gives flavor of more advanced solutions
Figure 1.
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 from a keyboard device driver, could diverge if other threads from
Categories and Subject Descriptors D.2.4 [Software]: Software Engineering—Program Verification; D.4.5 [Software]: Operating
Done
Unknown Zone | Protected Mode: On



Proving That Non-Blocking Algorithms Don't Block

Alexey Gotsman, University of Cambridge
Byron Cook, Microsoft Research
Matthew Parkinson, University of Cambridge
Viktor Vafeiadis, Microsoft Research

Abstract

A concurrent data-structure implementation is considered *non-blocking* if it meets one of three following liveness criteria: *wait-freedom*, *lock-freedom*, or *obstruction-freedom*. Developers of non-blocking algorithms aim to meet these criteria. However, to date their proofs for non-trivial algorithms have been only manual pencil-and-paper semi-formal proofs. This paper proposes the first fully automatic tool that allows developers to ensure that their algorithms are indeed non-blocking. Our tool uses rely-guarantee reasoning while overcoming the technical challenge of sound reasoning in the presence of interdependent liveness properties.

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 Languages, Theory, Verification

Keywords Formal Verification, Concurrent Programming, Liveness, Termination

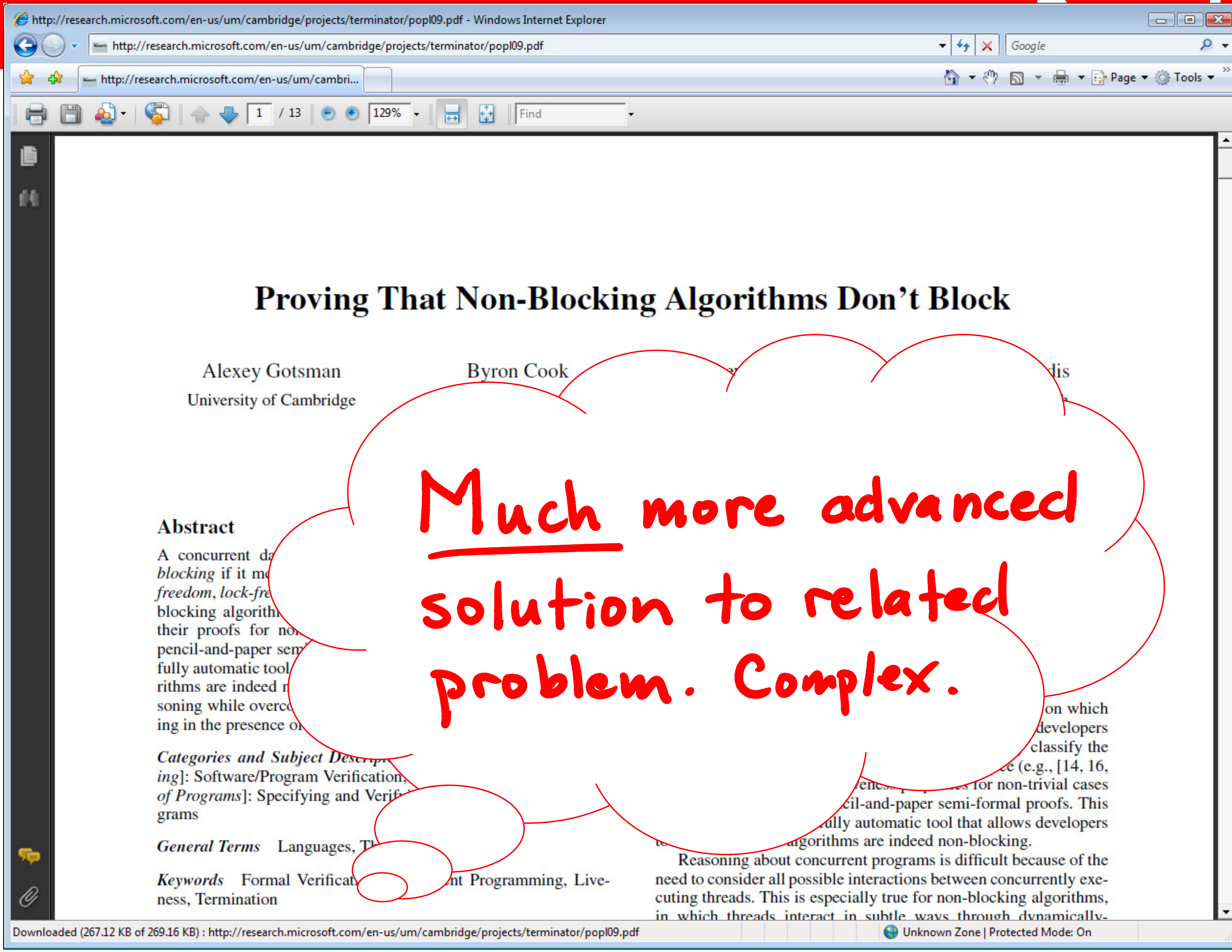
Lock-freedom [23]: From any point in a program's execution, some thread is guaranteed to complete its operation. Lock-freedom ensures the absence of livelock, but not starvation.

Obstruction-freedom [16]: Every thread is guaranteed to complete its operation provided it eventually executes in isolation. In other words, if at some point in a program's execution we suspend all threads except one, then this thread's operation will terminate.

The design of a non-blocking algorithm largely depends on which of the above three criteria it satisfies. Thus, algorithm developers aim to meet one of these criteria and correspondingly classify the algorithms as wait-free, lock-free, or obstruction-free (e.g., [14, 16, 25]). To date, proofs of the liveness properties for non-trivial cases have been only manual pencil-and-paper semi-formal proofs. This paper proposes the first fully automatic tool that allows developers to ensure that their algorithms are indeed non-blocking.

Reasoning about concurrent programs is difficult because of the need to consider all possible interactions between concurrently executing threads. This is especially true for non-blocking algorithms, in which threads interact in subtle ways through dynamically-

Microsoft



http://research.microsoft.com/en-us/um/cambridge/projects/terminator/popl09.pdf - Windows Internet Explorer

Proving That Non-Blocking Algorithms Don't Block

Alexey Gotsman
University of Cambridge

Byron Cook

Abstract

A concurrent da
blocking if it m
freedom, *lock-fre*
blocking algorith
their proofs for no
pencil-and-paper sem
fully automatic tool
rithms are indeed n
soning while overc
ing in the presence o

Categories and Subject Descr
ing]: Software/Program Verification,
of Programs]: Specifying and Verif
grams

General Terms Languages, T

Keywords Formal Verificat nt Programming, Live-
ness, Termination

on which
developers
classify the
e (e.g., [14, 16,
for non-trivial cases
cil-and-paper semi-formal proofs. This
ully automatic tool that allows developers
algorithms are indeed non-blocking.

Reasoning about concurrent programs is difficult because of the need to consider all possible interactions between concurrently executing threads. This is especially true for non-blocking algorithms, in which threads interact in subtle ways through dynamically-

Much more advanced solution to related problem. Complex.

Multithreaded programs

$$
\begin{aligned}
[\![\mathcal{P}_1]\!] &\triangleq (\mathcal{I}_1, \mathcal{R}_1, \mathcal{S}_1) \\
[\![\mathcal{P}_2]\!] &\triangleq (\mathcal{I}_2, \mathcal{R}_2, \mathcal{S}_2) \\
&\vdots \\
[\![\mathcal{P}_n]\!] &\triangleq (\mathcal{I}_n, \mathcal{R}_n, \mathcal{S}_n)
\end{aligned}
$$

Multithreaded programs

Microsoft Research Cambridge

$$
\begin{array}{lcl}
\mathcal{S} & \triangleq & \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n \\[1em]
\mathcal{I} & \triangleq & \{\ \ (s'_1, s'_2, \ldots, s'_n) \quad | \quad \forall i \in \{1, \ldots, n\}.\ s_i \in \mathcal{I}_i \\
 & & \wedge\ \ \forall i, j \in \{1, \ldots, n\}, v \in \textsc{Globals}(\mathcal{P}).\ s_i(v) = s_j(v) \\
 & & \wedge\ \ \forall i \in \{1, \ldots n\}.\ s'_i = s_i[\mathsf{tid} \mapsto i] \\
 & & \} \\[1em]
\mathcal{R} & \triangleq & \{\ \ ((s_1, s_2, \ldots s_n), (t_1, t_2, \ldots t_n)) \quad | \quad \exists i \in \{1, \ldots, n\}. \\
 & & \wedge\ \ \forall j \in \{1, \ldots, n\} - \{i\}, v \in \textsc{Locals}(\mathcal{P})\ \ s_j(v) = t_j(v) \\
 & & \wedge\ \ \forall j, k \in \{1, \ldots, n\}, v \in \textsc{Globals}(\mathcal{P}).\ t_j(v) = t_k(v) \\
 & & \wedge\ \ (s_i, t_i) \in \mathcal{R}_i \\
 & & \}
\end{array}
$$

Locks

Microsoft Research Cambridge

$$[\![x := e]\!]_V \triangleq \quad \{(s,t) \mid x \notin \textsc{Locks}(\mathcal{P}) \wedge \ldots$$

$$\begin{array}{rcll} [\![\mathbf{lock}(x)]\!]_V & \triangleq & \{ & (s,t) \\ & & \mid & x \in \textsc{Locks}(\mathcal{P}) \\ & & \wedge & \forall v \in V - \{x\}.\ s(v) = t(v) \\ & & \wedge & s(x) = 0 \\ & & \wedge & t(x) = s(\mathsf{tid}) \\ & & \} & \end{array}$$

$$\begin{array}{rcll} [\![\mathbf{unlock}(x)]\!]_V & \triangleq & \{ & (s,t) \\ & & \mid & x \in \textsc{Locks}(\mathcal{P}) \\ & & \wedge & \forall v \in V - \{x\}.\ s(v) = t(v) \\ & & \wedge & s(x) = s(\mathsf{tid}) \\ & & \wedge & t(x) = 0 \\ & & \} & \end{array}$$

Thread termination

Definition. Thread $\mathcal{P}_i$ is *thread-terminating* in $\mathcal{P}$ iff $\mathcal{P}_i$ can only make a finite number of steps in any $\mathcal{P}$-trace.

Definition. Thread $\mathcal{P}_i$ is *thread-terminating* in $\mathcal{P}$ iff $\mathcal{P}_i$ can only make a finite number of steps in any $\mathcal{P}$-trace.

Note: Not ruling out deadlock

Definition. Thread $\mathcal{P}_i$ is *thread-terminating* in $\mathcal{P}$ iff $\mathcal{P}_i$ can only make a finite number of steps in any $\mathcal{P}$-trace.

For simplicity also ignoring fairness

$x := 0 \;\|\; \text{while } (x \neq 0) \text{ skip}$

Thread termination

$$\mathcal{P}_1 \quad | \quad \mathcal{P}_2$$

Thread termination

$$\mathcal{P}_1 \quad \mathcal{A} \quad \mathcal{P}_2$$

Thread termination

$$\mathcal{P}_1 \quad \mathcal{A} \quad \mathcal{P}_2$$

Thread termination

Different types of checks

$$\mathcal{P}_1 \quad \mathcal{A} \quad \mathcal{P}_2$$

Thread termination

Different types of checks

$$\mathcal{P}_1 \quad \mathcal{A} \quad \mathcal{P}_2$$

Definition. An agreement $\mathcal{A}$ is a binary relation over states that constrains the change of global states, and leaves the change of local states unconstrained.

Abstract composition

Abstract composition. Assume that $P = (I, R, S)$. $P \mid_{\triangle} A \triangleq (I_{\triangle}, R_{\triangle}, S_{\triangle})$ such that

$$
\begin{array}{lcl}
I_{\triangle} & \triangleq & \{s \mid \exists s' \in I \wedge \forall v \in \textsc{Locals}(P).\ s(v) = s'(v)\} \\
S_{\triangle} & \triangleq & \{s \mid \exists s' \in S \wedge \forall v \in \textsc{Locals}(P).\ s(v) = s'(v)\} \\
R_{\triangle} & \triangleq & [A \cap \mathrm{Id}_{\textsc{Locals}(P)}]^{*}; R
\end{array}
$$

Abstract composition

Thread invariants.

$$
\begin{array}{lcl}
\textsc{Reach}(I, R, S) & \triangleq & R^*(I) \\
\textsc{Reach}_i(P) & \triangleq & \{s_i \mid (\ldots, s_i, \ldots) \in \textsc{Reach}(P)\} \\
\mathcal{R}_{\mathcal{I},i} & \triangleq & \mathcal{R}_i\!\downharpoonright_{\textsc{Reach}_i(\mathcal{P})}
\end{array}
$$

Microsoft Research Cambridge

Thread invariants.

$$
\begin{array}{lcl}
\textsc{Reach}(I, R, S) & \triangleq & R^*(I) \\
\textsc{Reach}_i(P) & \triangleq & \{s_i \mid (\ldots, s_i, \ldots) \in \textsc{Reach}(P)\} \\
\mathcal{R}_{\mathcal{I},i} & \triangleq & \mathcal{R}_i\!\downharpoonright_{\textsc{Reach}_i(\mathcal{P})}
\end{array}
$$

Lemma.

$$\forall j \in \{1, \ldots, n\} - \{i\}.\ \mathcal{R}_{\mathcal{I},j} \subseteq A$$

$$\Rightarrow$$

$$\textsc{Reach}_i(P) \subseteq \textsc{Reach}(P_i \mid_{\triangle} A)$$

Proving thread termination

Theorem. $\mathcal{P}_i$ is thread terminating if there exists an $\mathcal{A}$ such that

- $\forall j \in \{1, \ldots, n\} - \{i\}.\ \mathcal{R}_{\mathcal{I},j} \subseteq \mathcal{A}$, and
- $P_i \mid_{\triangle} \mathcal{A}$ is a terminating (sequential) program.

Proving thread termination

$\mathcal{A} := \textbf{true};$

while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**

  $\pi :=$ lasso counterexample;

  if strengthening of $\mathcal{A}$ not possible using π **then**

    return "$\mathcal{P}_i$ could not be proved terminating";

  else

    $\mathcal{A} :=$ strengthening of $\mathcal{A}$ using π;

  fi

  foreach $j \in \{1, \ldots n\} - i$ **do**

    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**

      $\mathcal{A} :=$ weakening using invariant for thread j;

    fi

  od

od

return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

$$
\begin{array}{l}
\textbf{lock}(\mathsf{lk});\\
\textbf{while} * \textbf{do}\\
\quad \textbf{assume}(\mathsf{x} > 0);\\
\quad \mathsf{x} := \mathsf{x} - 1;\\
\textbf{od}\\
\textbf{unlock}(\mathsf{lk});
\end{array}
$$

$\mathcal{P}_2$

$$
\begin{array}{l}
\textbf{while} * \textbf{do}\\
\quad \textbf{lock}(\mathsf{lk});\\
\quad \textbf{assume}(\mathsf{x} > 0);\\
\quad \mathsf{x} := \mathsf{x} - 1;\\
\quad \textbf{unlock}(\mathsf{lk});\\
\textbf{od}
\end{array}
$$

$\mathcal{P}_3$

$$
\begin{array}{l}
\textbf{while} * \textbf{do}\\
\quad \textbf{lock}(\mathsf{lk});\\
\quad \mathsf{x} := *;\\
\quad \textbf{unlock}(\mathsf{lk});\\
\textbf{od}
\end{array}
$$

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

$$
\begin{array}{l}
\textbf{lock}(\mathsf{lk}); \\
\textbf{while} * \textbf{do} \\
\quad \textbf{assume}(\mathsf{x} > 0); \\
\quad \mathsf{x} := \mathsf{x} - 1; \\
\textbf{od} \\
\textbf{unlock}(\mathsf{lk});
\end{array}
$$

$\mathcal{P}_2$

$$
\begin{array}{l}
\textbf{while} * \textbf{do} \\
\quad \textbf{lock}(\mathsf{lk}); \\
\quad \textbf{assume}(\mathsf{x} > 0); \\
\quad \mathsf{x} := \mathsf{x} - 1; \\
\quad \textbf{unlock}(\mathsf{lk}); \\
\textbf{od}
\end{array}
$$

$\mathcal{P}_3$

$$
\begin{array}{l}
\textbf{while} * \textbf{do} \\
\quad \textbf{lock}(\mathsf{lk}); \\
\quad \mathsf{x} := *; \\
\quad \textbf{unlock}(\mathsf{lk}); \\
\textbf{od}
\end{array}
$$

Which threads

"thread terminate"?

Example

Microsoft® Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($x > 0$);
  $x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume($x > 0$);
  $x := x - 1$;
  unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
  lock(lk);
  $x := *$;
  unlock(lk);

Which threads

"thread termina

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A}$:= weakening using invariant for thread j;
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume($x > 0$);
    $x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume($x > 0$);
    $x := x - 1$;
    un
od

$\mathcal{P}_3$

while $*$ **do**
    lock(lk);
    $x := *$;
    unlock(lk);

$$\mathcal{A} = \textbf{true}$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
    π := lasso counterexample;
    if strengthening of $\mathcal{A}$ not possible using π **then**
        return "$\mathcal{P}_i$ could not be proved terminating";
    else
        $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
    fi
    foreach $j \in \{1, \ldots n\} - i$ **do**
        if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
            $\mathcal{A}$:= weakening using invariant for thread j;
        fi
    od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

$$\begin{array}{l}\textbf{lock}(\mathsf{lk});\\ \textbf{while} * \textbf{do}\\ \quad \textbf{assume}(\mathsf{x} > 0);\\ \quad \mathsf{x} := \mathsf{x} - 1;\\ \textbf{od}\\ \textbf{unlock}(\mathsf{lk});\end{array}$$

$\mathcal{P}_2$

$$\begin{array}{l}\textbf{while}\\ \quad \textbf{lo}\\ \quad \textbf{as}\\ \quad \mathsf{x} :\\ \quad \textbf{un}\\ \textbf{od}\end{array}$$

$$\mathcal{A} = \textbf{true}$$

$$\begin{array}{l}[\mathcal{A} \cap \mathrm{I}\text{D}_{\{\mathsf{tid}\}}]^*;\\ \textbf{lock}(\mathsf{lk});\\ \textbf{while} * \textbf{do}\\ \quad [\mathcal{A} \cap \mathrm{I}\text{D}_{\{\mathsf{tid}\}}]^*;\\ \quad \textbf{assume}(\mathsf{x} > 0);\\ \quad [\mathcal{A} \cap \mathrm{I}\text{D}_{\{\mathsf{tid}\}}]^*;\\ \quad \mathsf{x} := \mathsf{x} - 1;\\ \textbf{od}\\ [\mathcal{A} \cap \mathrm{I}\text{D}_{\{\mathsf{tid}\}}]^*;\\ \textbf{unlock}(\mathsf{lk});\end{array}$$

$j;$

c := *;
while * **do**
 c := c − 1;
 assume(c > 0);
 use($\mathcal{A}$
 $\wedge\ '\mathsf{lk} = 1 \Rightarrow \mathsf{lk} = 1$
 $\wedge\ '\mathsf{lk} \neq 1 \Rightarrow \mathsf{lk} \neq 1$
 );
od

$[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
lock(lk);
while * **do**
 $[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
 assume(x > 0);
 $[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
 x := x − 1;
od
$[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
unlock(lk);

od
unlock(lk);

od

$$\mathcal{A} = \mathbf{true}$$

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($x > 0$);
  $x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while
  lo
  as
  x :
  un
od

$$\mathcal{A} = \textbf{true}$$

terminating?

$[\mathcal{A}$

lock(lk);
while $*$ **do**
  $[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
  assume($x > 0$);
  $[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
  $x := x - 1$;
od
$[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
unlock(lk);

j;

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while
  lo
  as
  x :
  un
od

$\mathcal{A} = \mathbf{true}$

$[\mathcal{A} \cap \mathrm{I}_\mathrm{D}$
lock(lk
while $*$
  $[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
  assume($\mathsf{x} > 0$);
  $[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
  $\mathsf{x} := \mathsf{x} - 1$;
od
$[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;
unlock(lk);

$[\![\pi_c]\!] = \exists x_1, x_3, \ldots$
$\mathsf{lk} = \mathsf{lk}' = 1$
$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_3$
$x_1 > 0$
$x_3 = x_2 - 1$

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while
  lo
  as
  x :
  u
od

$$\mathcal{A} = \mathbf{true}$$

$[\mathcal{A} \cap \mathrm{I}_{\mathrm{D}}$
lock(lk
while $*$
$[\mathcal{A} \cap \mathrm{I}_{\mathrm{D}_{\{\mathsf{tid}\}}}]^*$;

$$\begin{aligned} [\![\pi_c]\!] = \exists x_1, x_3, \ldots\ & \mathsf{lk} = \mathsf{lk}' = 1 \\ & \mathsf{x} = x_1 \wedge \mathsf{x}' = x_3 \\ & x_1 > 0 \\ & x_3 = x_2 - 1 \end{aligned}$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A}$:= weakening using invariant for thread j;
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while
  lo
  as
  x :
  ur
od

$$\mathcal{A} = \mathbf{true}$$

$[\mathcal{A} \cap \mathrm{ID}$
lock(lk
while $*$
$[\mathcal{A} \cap \mathrm{ID}_{\{\mathsf{tid}\}}]^*$;

$$\begin{aligned} [\![\pi_c]\!] = \exists x_1, x_3, \ldots\ & \\ \mathsf{lk} = \mathsf{lk}' = 1 & \\ \mathsf{x} = x_1 \wedge \mathsf{x}' = x_3 & \\ x_1 > 0 & \\ x_3 = x_2 - 1 & \end{aligned}$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A}$:= weakening using invariant for thread j;
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$$[\![\pi_c]\!] = \exists x_1, x_3, \ldots$$
$$\mathsf{lk} = \mathsf{lk}' = 1$$
$$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_3$$
$$x_1 > 0$$
$$x_3 = x_2 - 1$$

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

while $*$ **do**
  lock(lk);
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
  un...
od

x := ... ;
unlock(lk);

$$\mathcal{A} = \textbf{true}$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_\triangle \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A}$:= weakening using invariant for thread j
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume(x > 0);
    x := x − 1;
od
unlock(lk);

while $*$ **do**
    lock(lk);
    assume(x > 0);
    x := x − 1;
    unlock(lk);
od

x := ...;
unlock(lk);

$$[\![\pi_c]\!] = \exists x_1, x_3, \ldots$$
$$\mathsf{lk} = \mathsf{lk}' = 1$$
$$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_3$$
$$x_1 > 0$$
$$x_3 = x_2 - 1$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \,|_\triangle\, \mathcal{A}$ not provably terminating **do**

return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Key idea: we can examine the counterexample in isolation of the counterexample

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume(x $>$ 0)
  x := x − 1;
od
unlock(lk);

whi

x := ;
unlock(lk);

$[\![\pi_c^s]\!] = \exists x_1, \ldots$
lk = lk$'$ = 1
x $= x_1 \wedge$ x$' = x_2$
$x_1 > 0$
$x_2 = x_1 - 1$

$[\![\pi_c]\!] = \exists x_1, x_3, \ldots$
lk = lk$'$ = 1
x $= x_1 \wedge$ x$' = x_3$
$x_1 > 0$
$x_3 = x_2 - 1$

true;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**

;

d j

return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Key idea: we can examine the counterexample in isolation of the counterexample

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume(x > 0)
  x := x $- 1$;
od
unlock(lk);

whi

x :=
unlock(lk);

$[\![\pi_c^s]\!] \subseteq \trianglerighteq_{\mathsf{x}}$

$[\![\pi_c]\!] = \exists x_1, x_3, \ldots$
$\mathsf{lk} = \mathsf{lk}' = 1$
$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_3$
$x_1 > 0$
$x_3 = x_2 - 1$

$[\![\pi_c^s]\!] = \exists x_1, \ldots$
$\mathsf{lk} = \mathsf{lk}' = 1$
$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_2$
$x_1 > 0$
$x_2 = x_1 - 1$

true;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**

Key idea: we can examine the counterexample in isolation of the counterexample

return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$)
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$$\mathcal{A} = \textbf{true}$$

$[\![\pi_c^s]\!] \subseteq \trianglerighteq_{\mathsf{x}}$

$[\![\pi_c^s]\!] = \exists x_1, \ldots$
  $\mathsf{lk} = \mathsf{lk}' = 1$
  $\mathsf{x} = x_1 \wedge \mathsf{x}' = x_2$
  $x_1 > 0$
  $x_2 = x_1 - 1$

$[\![\pi_c]\!] = \exists x_1, x_3, \ldots$
  $\mathsf{lk} = \mathsf{lk}' = 1$
  $\mathsf{x} = x_1 \wedge \mathsf{x}' = x_3$
  $x_1 > 0$
  $x_3 = x_2 - 1$

$\mathsf{x} :=$ …;
unlock(lk);

… **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  $\pi :=$ lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A} :=$ strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A} :=$ weakening using invariant for thread j
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\llbracket \pi_c^s \rrbracket \subseteq \trianglerighteq_{\mathsf{x}}$

$\llbracket \pi_c \rrbracket = \exists x_1, x_3, \ldots$
$\mathsf{lk} = \mathsf{lk}' = 1$
$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_3$
$x_1 > 0$
$x_3 = x_2 - 1$

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$)
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$\llbracket \pi_c^s \rrbracket = \exists x_1, \ldots$
$\mathsf{lk} = \mathsf{lk}' = 1$
$\mathsf{x} = x_1 \wedge \mathsf{x}' = x_2$
$x_1 > 0$
$x_2 = x_1 - 1$

unlock(lk);

$$\mathcal{A} = \textbf{true} \wedge \trianglerighteq_{\mathsf{x}}$$

... **true**;
while $\mathcal{P}_i \mid_\triangle \mathcal{A}$ not provably terminating **do**
  $\pi :=$ lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A} :=$ strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A} :=$ weakening using invariant for thread j
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

$\mathbf{lock}(\mathsf{lk});$
$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\mathbf{od}$
$\mathbf{unlock}(\mathsf{lk});$

$\mathcal{P}_2$

$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{lock}(\mathsf{lk});$
$\quad \mathbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\quad \mathbf{un}$
$\mathbf{od}$

$\mathcal{P}_3$

$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{lock}(\mathsf{lk});$
$\quad \mathsf{x} := *;$
$\quad \mathbf{unlock}(\mathsf{lk});$

$$\mathcal{A} = \mathbf{true}$$
$$\wedge \ \trianglerighteq_{\mathsf{x}}$$

$\mathcal{A} := \mathbf{true};$
$\mathbf{while}\ \mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating $\mathbf{do}$
$\quad \pi :=$ lasso counterexample;
$\quad \mathbf{if}$ strengthening of $\mathcal{A}$ not possible using π $\mathbf{then}$
$\quad\quad \mathbf{return}$ "$\mathcal{P}_i$ could not be proved terminating";
$\quad \mathbf{else}$
$\quad\quad \mathcal{A} :=$ strengthening of $\mathcal{A}$ using π;
$\quad \mathbf{fi}$
$\quad \mathbf{foreach}\ j \in \{1, \ldots n\} - i\ \mathbf{do}$
$\quad\quad \mathbf{if}\ \mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}\ \mathbf{then}$
$\quad\quad\quad \mathcal{A} :=$ weakening using invariant for thread j;
$\quad\quad \mathbf{fi}$
$\quad \mathbf{od}$
$\mathbf{od}$
$\mathbf{return}$ "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume(x > 0);
    x := x − 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume(x > 0);
    x := x − 1;
    unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
    lock(lk);
    x := $*$;
    unlock(lk);

$$\mathcal{A} = \textbf{true} \wedge \trianglerighteq_{\mathsf{x}}$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
    π := lasso counterexample;
    if strengthening of $\mathcal{A}$ not possible using π **then**
        return "$\mathcal{P}_i$ could not be proved terminating";
    else
        $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
    fi
    foreach $j \in \{1, \ldots n\} - i$ **do**
        if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
            $\mathcal{A}$:= weakening using invariant for thread j;
        fi
    od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume(x > 0);
  x := x − 1;

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume(x > 0);
  x := x − 1;

$\mathcal{P}_3$

while $*$ **do**
  lock(lk);
  x := $*$;
  unlock(lk);

Checking a property like $R \subseteq [\geq_x$
is fairly easy in comparison to
$R^*(I) \subseteq V$

  $\mathcal{A}$:= weakening using invariant for thread j;
  fi
 od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

$\textbf{lock}(\mathsf{lk});$
$\textbf{while} \ast \textbf{do}$
$\quad \textbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\textbf{od}$
$\textbf{unlock}(\mathsf{lk});$

$\mathcal{P}_2$

$\textbf{while} \ast \textbf{do}$
$\quad \textbf{lock}(\mathsf{lk});$ ✓
$\quad \textbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\quad \textbf{unlock}(\mathsf{lk});$
$\textbf{od}$

$\mathcal{P}_3$

$\textbf{while} \ast \textbf{do}$
$\quad \textbf{lock}(\mathsf{lk});$
$\quad \mathsf{x} := \ast;$
$\quad \textbf{unlock}(\mathsf{lk});$

$$\mathcal{A} = \mathbf{true} \wedge \unrhd_{\mathsf{x}}$$

$\mathcal{A} := \textbf{true};$
$\textbf{while}\ \mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating $\textbf{do}$
$\quad \pi :=$ lasso counterexample;
$\quad \textbf{if}$ strengthening of $\mathcal{A}$ not possible using π $\textbf{then}$
$\quad\quad \textbf{return}$ "$\mathcal{P}_i$ could not be proved terminating";
$\quad \textbf{else}$
$\quad\quad \mathcal{A} :=$ strengthening of $\mathcal{A}$ using π;
$\quad \textbf{fi}$
$\quad \textbf{foreach}\ j \in \{1, \ldots n\} - i\ \textbf{do}$
$\quad\quad \textbf{if}\ \mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}\ \textbf{then}$
$\quad\quad\quad \mathcal{A} :=$ weakening using invariant for thread j;
$\quad\quad \textbf{fi}$
$\quad \textbf{od}$
$\textbf{od}$
$\textbf{return}$ "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume(x > 0);
    x := x − 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume(x > 0);
    x := x − 1;
    unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
    lock(lk);
    x := $*$;
    unlock(lk);

$$\mathcal{A} = \mathbf{true} \wedge \trianglerighteq_{x}$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
    π := lasso counterexample;
    if strengthening of $\mathcal{A}$ not possible using π **then**
        return "$\mathcal{P}_i$ could not be proved terminating";
    else
        $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
    fi
    foreach $j \in \{1, \ldots n\} - i$ **do**
        if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
            $\mathcal{A}$:= weakening using invariant for thread j;
        fi
    od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($x > 0$);
  $x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume($x > 0$);
  $x := x - 1$;
  unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
  lock(lk);
  $x := *$;
  unlock(lk);
od

$$\mathcal{A} = \mathbf{true}$$
$$\wedge \;\; \trianglerighteq_{x}$$

$\mathcal{A} :=$ **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  $\pi :=$ lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A} :=$ strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A} :=$ weakening using invariant for thread j;
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume($x > 0$);
    $x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume($x > 0$);
    $x := x - 1$;
    unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
    lock(lk);
    $x := *$;
    unlock(lk);

c := $*$;
while $*$ **do**
    c := c $- 1$;
    assume(c > 0);
    use(**true**
        $\wedge$ 'lk $= 2 \Rightarrow$ lk $= 2$
        $\wedge$ 'lk $\neq 2 \Rightarrow$ lk $\neq 2$
        );
od

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
    π := lasso counterexample;
    if strengthening of $\mathcal{A}$ not possible using π **then**
        return "$\mathcal{P}_i$ could not be proved terminating";
    else
        $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
    fi
    foreach $j \in \{1, \ldots n\} - i$ **do**
        if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
            $\mathcal{A}$:= weakening using invariant for thread j;
        fi
    od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft® Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume(x $>$ 0);
  x := x $-$ 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume(x $>$ 0);
  x := x $-$ 1;
  unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
  lock(lk);
  x := $*$;
  unlock(lk);

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_\triangle \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";

j;

return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

c := $*$;
while $*$ **do**
  c := c $-$ 1;
  assume(c $>$ 0);
  use(**true**
    $\wedge$ 'lk $= 2 \Rightarrow$ lk $= 2$
    $\wedge$ 'lk $\neq 2 \Rightarrow$ lk $\neq 2$
    );
od

$$\forall j \in \{1, \ldots, n\} - \{i\}.\ \mathcal{R}_{\mathcal{I},j} \subseteq A$$

$$\Rightarrow$$

$$\textsc{Reach}_i(P) \subseteq \textsc{Reach}(P_i \mid_\triangle A)$$

$\textsc{Reach}(\mathcal{P}_2 \mid_\triangle \textbf{true})$

lock(lk);
while $*$ **do**
assume($x > 0$);
$x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
lock(lk);
assume($x > 0$);
$x := x - 1$;
unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
lock(lk);
$x := *$;
unlock(lk);

$\mathcal{A} :=$ **true**;
while $\mathcal{P}_i \mid_\triangle \mathcal{A}$ not provably terminating **do**
$\pi :=$ lasso counterexample;
if strengthening of $\mathcal{A}$ not possible using π **then**
return "$\mathcal{P}_i$ could not be proved terminating";

return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

$$\forall j \in \{1, \ldots, n\} - \{i\}.\ \mathcal{R}_{\mathcal{I},j} \subseteq A$$

$$\Rightarrow$$

$$\textsc{Reach}_i(P) \subseteq \textsc{Reach}(P_i \mid_\triangle A)$$

c := $*$;
while $*$ **do**
c := c $- 1$;
assume(c > 0);
use(**true**
$\wedge$ 'lk $= 2 \Rightarrow$ lk $= 2$
$\wedge$ 'lk $\neq 2 \Rightarrow$ lk $\neq 2$
);
od

Ex

Microsoft Research Cambridge

$\textsc{Reach}(\mathcal{P}_2 \mid_{\triangle} \mathbf{true})$

lock(lk);
while $*$ **do**
    assume(x > 0);
    x := x − 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume(x > 0);
    x := x − 1;
    unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
    lock(lk);
    x := $*$;
    unlock(lk);

c := $*$;
while $*$ **do**
    c := c − 1;
    assume(c > 0);
    use(**true**
        $\wedge$ 'lk = 2 ⇒ lk = 2
        $\wedge$ 'lk ≠ 2 ⇒ lk ≠ 2
        );
od

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
    π := lasso counterexample;
    if strengthening of $\mathcal{A}$ not possible using π **then**
        return "$\mathcal{P}_i$ could not be proved terminating";
    else
        $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
    fi
    foreach $j \in \{1, \ldots n\} - i$ **do**
        if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
            $\mathcal{A}$:= weakening using invariant for thread j;
        fi
    od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

REACH($\mathcal{P}_2 \mid_\triangle$ **true**)

'x := x;
'lk := lk;
$\boxed{\text{x := x} - 1\text{;}}$
skip;

lock(lk);
while
as...
x :=
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
lock(lk);
assume(x > 0);
$\boxed{\text{x := x} - 1}$;
unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
lock(lk);
x := $*$;
unlock(lk);

c := $*$;
while $*$ **do**
c := c − 1;
assume(c > 0);
use(**true**
$\wedge$ 'lk = 2 $\Rightarrow$ lk = 2
$\wedge$ 'lk $\neq$ 2 $\Rightarrow$ lk $\neq$ 2
);
od

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_\triangle \mathcal{A}$ not provably terminating **do**
π := lasso counterexample;
if strengthening of $\mathcal{A}$ not possible using π **then**
return "$\mathcal{P}_i$ could not be proved terminating";
else
$\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
fi
foreach $j \in \{1, \ldots n\} - i$ **do**
if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
$\mathcal{A}$:= weakening using invariant for thread j;
fi
od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Ex... P... true)

$\delta \subseteq \trianglerighteq_{\mathsf{x}}$???

lock(...
while
as...
x :=
od
unlock(lk);

... x;
'lk := lk;
$\boxed{\mathsf{x} := \mathsf{x} - 1;}$
skip;

$\mathcal{P}_2$

while $*$ **do**
lock(lk);
assume(x > 0);
$\boxed{\mathsf{x} := \mathsf{x} - 1};$
unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
lock(lk);
x := $*$;
unlock(lk);
...

c := $*$;
while $*$ **do**
c := c − 1;
assume(c > 0);
use(**true**
$\wedge$ 'lk = 2 $\Rightarrow$ lk = 2
$\wedge$ 'lk $\neq$ 2 $\Rightarrow$ lk $\neq$ 2
);
od

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
π := lasso counterexample;
if strengthening of $\mathcal{A}$ not possible using π **then**
return "$\mathcal{P}_i$ could not be proved terminating";
else
$\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
fi
foreach $j \in \{1, \ldots n\} - i$ **do**
if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
$\mathcal{A}$:= weakening using invariant for thread j;
fi
od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

$\mathbf{lock}(\mathsf{lk});$
$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\mathbf{od}$
$\mathbf{unlock}(\mathsf{lk});$

$\mathcal{P}_2$

$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{lock}(\mathsf{lk});$ ✓
$\quad \mathbf{assume}(\mathsf{x} > 0);$ ✓
$\quad \mathsf{x} := \mathsf{x} - 1;$ ✓
$\quad \mathbf{unlock}(\mathsf{lk});$
$\mathbf{od}$

$\mathcal{P}_3$

$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{lock}(\mathsf{lk});$
$\quad \mathsf{x} := *;$
$\quad \mathbf{unlock}(\mathsf{lk});$
$\mathbf{od}$

$$\mathcal{A} = \mathbf{true}$$
$$\wedge \quad \trianglerighteq_{\mathsf{x}}$$

Example

$\mathcal{P}_1$

$$\begin{array}{l}\textbf{lock}(\mathsf{lk});\\ \textbf{while} * \textbf{do}\\ \quad \textbf{assume}(\mathsf{x} > 0);\\ \quad \mathsf{x} := \mathsf{x} - 1;\\ \textbf{od}\\ \textbf{unlock}(\mathsf{lk});\end{array}$$

$\mathcal{P}_2$

$$\begin{array}{l}\textbf{while} * \textbf{do}\\ \quad \textbf{lock}(\mathsf{lk}); \checkmark\\ \quad \textbf{assume}(\mathsf{x} > 0); \checkmark\\ \quad \mathsf{x} := \mathsf{x} - 1; \checkmark\\ \quad \textbf{unlock}(\mathsf{lk}); \checkmark\\ \textbf{od}\end{array}$$

$\mathcal{P}_3$

$$\begin{array}{l}\textbf{while} * \textbf{do}\\ \quad \textbf{lock}(\mathsf{lk});\\ \quad \mathsf{x} := *;\\ \quad \textbf{unlock}(\mathsf{lk});\\ \textbf{od}\end{array}$$

$$\mathcal{A} = \textbf{true}$$
$$\wedge \quad \trianglerighteq_{\mathsf{x}}$$

Example

$\mathcal{P}_1$

$\mathbf{lock}(\mathsf{lk});$
$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\mathbf{od}$
$\mathbf{unlock}(\mathsf{lk});$

$\mathcal{P}_2$

$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{lock}(\mathsf{lk});$
$\quad \mathbf{assume}(\mathsf{x} > 0);$
$\quad \mathsf{x} := \mathsf{x} - 1;$
$\quad \mathbf{unlock}(\mathsf{lk});$
$\mathbf{od}$

$\mathcal{P}_3$

$\mathbf{while} * \mathbf{do}$
$\quad \mathbf{lock}(\mathsf{lk});$
$\quad \mathsf{x} := *;$
$\quad \mathbf{unlock}(\mathsf{lk});$
$\mathbf{od}$

$$\mathcal{A} = \mathbf{true}$$
$$\wedge \quad \unrhd_{\mathsf{x}}$$

Example

$\mathcal{P}_1$

$$
\begin{array}{l}
\mathbf{lock}(\mathsf{lk});\\
\mathbf{while} * \mathbf{do}\\
\quad \mathbf{assume}(\mathsf{x} > 0);\\
\quad \mathsf{x} := \mathsf{x} - 1;\\
\mathbf{od}\\
\mathbf{unlock}(\mathsf{lk});
\end{array}
$$

$\mathcal{P}_2$

$$
\begin{array}{l}
\mathbf{while} * \mathbf{do}\\
\quad \mathbf{lock}(\mathsf{lk});\\
\quad \mathbf{assume}(\mathsf{x} > 0);\\
\quad \mathsf{x} := \mathsf{x} - 1;\\
\quad \mathbf{unlock}(\mathsf{lk});\\
\mathbf{od}
\end{array}
$$

$\mathcal{P}_3$

$$
\begin{array}{l}
\mathbf{while} * \mathbf{do}\\
\quad \mathbf{lock}(\mathsf{lk});\\
\quad \mathsf{x} := *;\\
\quad \mathbf{unlock}(\mathsf{lk});\\
\mathbf{od}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A} = \mathbf{true}\\
\quad \wedge \trianglerighteq_{\mathsf{x}}
\end{array}
$$

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume(x > 0);
    x := x − 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume(x > 0);
    x := x − 1;
    unlock(lk);
od

$\delta : \mathsf{lk} = 3$

$\mathcal{P}$

ile $*$ **do**
    lock(lk);
    x := $*$;
    unlock(lk);
od

$$\mathcal{A} = \mathbf{true}$$
$$\wedge \quad \unrhd_{x}$$

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($x > 0$);
  $x := x - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume($x > 0$);
  $x := x - 1$;
  unlock(lk);
od

$\delta : \mathsf{lk} = 3$

ile $*$ **do**
  lock(lk);
  $x := *$;
  unlock(lk);
od

$$\mathcal{A} = \mathbf{true}$$
$$\quad \wedge \ \unrhd_{x} \vee \mathsf{lk} = 3$$

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume$(\mathsf{x} > 0)$;
    $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume$(\mathsf{x} > 0)$;
    $\mathsf{x} := \mathsf{x} - 1$;
    unlock(lk);
od

$\delta : \mathsf{lk} = 3$

ile $*$ **do**
    lock(lk);
    $\mathsf{x} := *$;
    unlock(lk);
od

$$\mathcal{A} = \mathbf{true}$$
$$\wedge \;\; \unrhd_{\mathsf{x}} \vee \mathsf{lk} = 3$$

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

$$\begin{array}{l} \mathbf{lock}(\mathsf{lk}); \\ \mathbf{while} * \mathbf{do} \\ \quad \mathbf{assume}(\mathsf{x} > 0); \\ \quad \mathsf{x} := \mathsf{x} - 1; \\ \mathbf{od} \\ \mathbf{unlock}(\mathsf{lk}); \end{array}$$

$\mathcal{P}_2$

$$\begin{array}{l} \mathbf{while} * \mathbf{do} \\ \quad \mathbf{lock}(\mathsf{lk}); \checkmark \\ \quad \mathbf{assume}(\mathsf{x} > 0); \checkmark \\ \quad \mathsf{x} := \mathsf{x} - 1; \checkmark \\ \quad \mathbf{unlock}(\mathsf{lk}); \checkmark \\ \mathbf{od} \end{array}$$

$\mathcal{P}_3$

$$\begin{array}{l} \mathbf{while} * \mathbf{do} \\ \quad \mathbf{lock}(\mathsf{lk}); \checkmark \\ \quad \mathsf{x} := *; \checkmark \\ \quad \mathbf{unlock}(\mathsf{lk}); \checkmark \\ \mathbf{od} \end{array}$$

$$\begin{array}{l} \mathcal{A} = \mathbf{true} \\ \quad \wedge \ \ \lfloor\!\geqslant_{\mathsf{x}} \vee\ \mathsf{lk} = 3 \end{array}$$

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
    assume(x $>$ 0);
    x := x − 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
    lock(lk);
    assume(x $>$ 0);
    x := x − 1;
    un

od

$\mathcal{P}_3$

while $*$ **do**
    lock(lk);
    x := $*$;
    unlock(lk);

$$\mathcal{A} = \textbf{true}$$
$$\wedge \ \trianglerighteq_{\mathsf{x}} \vee \ \mathsf{lk} = 3$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
    π := lasso counterexample;
    if strengthening of $\mathcal{A}$ not possible using π **then**
        return "$\mathcal{P}_i$ could not be proved terminating";
    else
        $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
    fi
    foreach $j \in \{1, \ldots n\} - i$ **do**
        if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
            $\mathcal{A}$:= weakening using invariant for thread j;
        fi
    od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume($\mathsf{x} > 0$);
  $\mathsf{x} := \mathsf{x} - 1$;
  unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
  lock(lk);
  $\mathsf{x} := *$;
  unlock(lk);

$$\mathcal{A} = \textbf{true} \wedge \unrhd_{\mathsf{x}} \vee \mathsf{lk} = 3$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A}$:= weakening using invariant for thread j;
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

Microsoft Research Cambridge

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume(x > 0);
  x := x − 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(l…
  assu…
  …
  u…
od

$\mathcal{P}_3$

c := $*$;
while $*$ **do**
  c := c − 1;
  assume(c > 0);
  use(′x ≥ x ∨ lk = 3
    ∧ ′lk = 1 ⇒ lk = 1
    ∧ ′lk ≠ 1 ⇒ lk ≠ 1
    );
od

$$\mathcal{A} = \textbf{true}$$
$$\wedge \trianglerighteq_{\mathsf{x}} \vee \mathsf{lk} = 3$$

… **then**
…inating";
el…
$\mathcal{A}$ … using π;
fi
foreach $j \in \{1, \ldots n\} - i$ **do**
  if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
    $\mathcal{A}$:= weakening using invariant for thread j;
  fi
od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Example

$\mathcal{P}_1$

lock(lk);
while $*$ **do**
  assume(x $>$ 0);
  x := x $-$ 1;
od
unlock(lk);

$\mathcal{P}_2$

while $*$ **do**
  lock(lk);
  assume(x $>$ 0);
  x := x $-$ 1;
  unlock(lk);
od

$\mathcal{P}_3$

while $*$ **do**
  lock(lk);
  x := $*$;
  unlock(lk);

$$\mathcal{A} = \mathbf{true} \wedge \trianglerighteq_{\mathsf{x}} \vee \mathsf{lk} = 3$$

$\mathcal{A}$:= **true**;
while $\mathcal{P}_i \mid_{\triangle} \mathcal{A}$ not provably terminating **do**
  π := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using π **then**
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$:= strengthening of $\mathcal{A}$ using π;
  fi
  foreach $j \in \{1, \ldots n\} - i$ **do**
    if $\mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A}$ **then**
      $\mathcal{A}$:= weakening using invariant for thread j;
    fi
  od
od
return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";

Outline

- Fair termination
- Data structures
- Concurrency
- Conclusion

Conclusion

- Basics: *WF, ranking functions, disjunctive WF, decomposition, rank function synthesis*
- Sequential arithmetic, non-recursive programs: r*efinement, checking inclusions with transitive closure, induction, variance analysis*
- Fair termination (and liveness): *Modification to above techniques*
- Recursion: *via reduction to sequential non-recursive programs*
- Heap: *abstractions via shape analysis techniques*
- Non-termination: *proving, underapproximating weakest preconditions*
- Concurrency: *finding sound interdependent rely/guarantee conditions that use liveness*

Conclusion

Microsoft Research Cambridge

- Implementation using existing tools
  - Shape analysis engines, reachability engines, abstract interpreters, quantifier elimination procedures, decision procedures, LP solvers, etc.

- Termination tools:
  - ACL2
  - Polyrank
  - SpaceInvader
  - SatAbs (termination support in development)
  - ARMC
  - TERMINATOR
  - T2 (new version of TERMINATOR in development)
  - ....

Beyond static termination proving

Microsoft Research Cambridge

- Many problems are related to termination
  - Search for thread-scheduling that guarantees termination (operating systems)
  - Synthesis of compounds that kill targeted cells (medicine)
  - ....

- Perhaps advances in termination proving will lead to advances in other areas?

Conclusion

- Please contact me with questions or ideas!
  - byroncook@gmail.com
  - If I don't answer, just write again

- Thank you for your attention, questions