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)

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)

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)

Liveness vs. safety

Safety properties always have a finite counterexample

Example: “Every Release() is preceded 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

Acquire $\Rightarrow$ $\Diamond$ Release is a termination-like property
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**:

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

- **Implemented as a transformation to reachability**
Example: SLIC

```c
static int l = 0;

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

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

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

Are these reachable?

```c
state ! @pt l = 0; }

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

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

```c
def AcquireLock()
{
    ............
}

def ReleaseLock()
{
    ............
}

def 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

```c
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;
    }
}
```
int l = 0;

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

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

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

\[ R^+_I \]
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 \( S, T \subseteq S \)
- A \( \mathcal{P} \)-trace, \( \tau \), is \textit{fair} if either only finite \( S \)-states occur in \( \tau \) or else an infinite number \( T \) states occur in \( \tau \).
Strong fairness

➔ Fair and unfair traces:

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

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

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

A program states $S, T \subseteq \tau$ and only finite $S$-states occur in $\tau$. If number $T$ states occur in $\tau$. 
state {}

fairness {
    // First Boolean expression: succeeds
    // on every return from IoCreateDevice
    IoCreateDevice.exit();

    // Second Boolean expression: succeeds
    // if IoCreateDevice returns
    // something other than
    // STATUS_OBJ_NAME_COLLISION
    IoCreateDevice.exit();
    return STATUS_OBJ_NAME_COLLISION;
}

A program states $S, T \subseteq \tau$ and only finite $S$-states occur in $\tau$ and finite $T$-states occur in $\tau$. 
Eliminating unfair paths
Eliminating unfair paths

\[ R^+ \subseteq \varnothing \doteq \forall (a, b) \in R^+. (a, b) \in \varnothing \]
Eliminating unfair paths

\[ \text{FAIR}(S, T) \triangleq \{(q_0, q_{k-1}) \mid \exists (q, k) \in \text{TRACESEGS}(P) \wedge (\exists i. \ q_i \in S) \Rightarrow (\exists j. \ q_j \in T)\} \]
Eliminating unfair paths

Is this an "unfair path"?
Is this an "unfair path"?
Eliminating unfair paths

Is this an "unfair path"?
Is this an "unfair path"?
Eliminating unfair paths

Is this an "unfair path"?
Eliminating unfair paths

Is this an "unfair path"?
Is this an “unfair path”? 

Eliminating unfair paths
Eliminating unfair paths

$$\text{FAIR}(S, T) \triangleq \{ (q_0, q_{k-1}) \mid \exists (q, k) \in \text{TRACESEGS}(P) \wedge (\exists i. q_i \in S) \Rightarrow (\exists j. q_j \in T) \}$$
Eliminating unfair paths

\[ \text{FAIR}(S, T) \triangleq \{(q_0, q_{k-1}) \mid \exists (q, k) \in \text{TRACESEGS}(P) \land (\exists i. q_i \in S) \Rightarrow (\exists j. q_j \in T)\} \]

**Observation.** \(R\) is well-founded with respect to fairness constraint \((S, T)\) iff \(R^+ \cap \text{FAIR}(S, T) \subseteq Q\) and \(Q\) is disjunctively well-founded.
Eliminating unfair paths

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

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

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

Observation. $R$ is well-founded with respect to fairness constraint $(S, T)$ iff $R^+ \cap \text{FAIR}(S, T) \subseteq Q$ 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: ....................?
void f()
{
    AcquireLock();
    ...
    ...
    ...
    ...
    ...
    ...
    ReleaseLock();
}

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

void main()
{
    s=NONE;

c

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

Infinite loops are added to exit points
Expressing fairness

Note that this can only happen once

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

Infinite loops are added to exit points
void f()
{
    AcquireLock();
    if (s==NONE) {
        if (nondet()) {
            s=PENDING;
        }
        .
        .
        .
        ReleaseLock();
    if (s==PENDING) {
        assume(false);
    }
    .
    .
}

void main()
{
    s=NONE;
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

```c
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

```c
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
$T := \emptyset$

while \texttt{REACHABLE}(\mathcal{R}(R, \ell, T), \ell_{err}) do
    let $\pi_s, \pi_c = \text{lasso in } \mathcal{R}(R, \ell, T)$ from 0 to $\ell$, and $\ell$ to $\ell_{err}$
    let $\rho = \omega([\pi_c]^*(\llbracket \pi_s \rrbracket(T)))$
    if \texttt{SYNTHESIS}(\llbracket \pi_c \rrbracket \cap \rho \times \rho) returns ranking function $f$ then
        $T := T \cup \triangleright_f$
    else
        report “potential counterexample found: $\pi_s, \pi_c$
    fi
od
report “termination proved with argument $T$
while(......) {

    body(*) {
        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;
Predicates, interpolants, etc are computed on demand.
Thus: fairness does not add much overhead.
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.

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
```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) {
            PIRQP Irp;
            KeAcquireSpinLock(&deviceExtension->SpinLock,
               &OldIrql);

            while ( !IsListEmpty(&deviceExtension->ReadQueue) ) {  
                PLIST_ENTRY ListElement;
                PIRQP 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
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”
  - $x \nrightarrow f : y, d : 5$: “The heaplet has exactly one cell $x$, holding a record with field $f=y$ and field $d=5$.”
  - $A \ast B$: “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 analysis

Separation logic

Classical logic (quantifiers, conjunction, etc)

Extension:

- \( \text{emp} \) : “The heaplet is empty”
- \( x \mapsto f : y, d : 5 \) : “The heaplet has exactly one cell \( x \), holding a record with field \( f = y \) and field \( d = 5 \).”
- \( A \star B \) : “The heaplet can be divided so \( A \) is true of exactly one partition, and \( B \) is true of the other”

Induction definitions

\[
\text{ls}(x, y) \triangleq \exists z. \ x \mapsto \text{next} : z \star \text{ls}(z, y) \\
\lor \quad x = y
\]
Separation logic based shape analysis

- Separation
  - Classical logic (quantifiers, conjunction, etc)
  - Extension:
    - emp: "The heaplet is empty"
    - $x \rightarrow f: y, d: 5$: "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 using emp, $\rightarrow$, *

\[
ls(x, y) \triangleq \exists z. x \rightarrow \text{next} : z * ls(z, y) \\
\lor x = y
\]

\[
ls(k, x, y) \triangleq k \geq 1 \land \exists z. x \rightarrow \text{next} : z * ls(k - 1, z, y) \\
\lor k = 0 \land x = y
\]
Separation logic based shape analysis

→ Cyclic lists?
  - $\exists y. ls(x, y) \ast ls(y, x)$

→ Acyclic lists?
  - $\exists y. ls(x, 0)$

→ "Pan handle lists"?
  - $\exists y, z. ls(x, y) \ast ls(y, z) \ast ls(z, y)$
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? ✗
Separation logic based shape analysis:  

- Sets of \(*\)-conjuncted formulae represent abstract heaps at program locations  
  - *e.g.* $\ell : \text{ls}(k, x, 0)$ “The program’s heap when executing the command at location $\ell$ consists only of an acyclic list pointed to by $x$”

- Forward symbolic simulation, *e.g.*  
  
  $$\{ \text{ls}(k, x, y) \wedge k \geq 1 \}$$
  $$x := (*x).\text{next}$$
  $$\{ \text{ls}(k, x, y) \wedge k \geq 0 \}$$
Separation logic based shape analysis:

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

\[ \alpha(\exists y. ls(x, y) \ast ls(y, z)) = ls(x, z) \]
\[ \alpha(x \mapsto \text{next} : y) = \exists k. ls(k, x, y) \land k \geq 1 \]

- Summaries for procedures, and “Frame Rule”:
  - if \( \{ A \} p(x) \{ B \} \), then forall \( H \), \( \{ A \ast H \} p(x) \{ B \ast H \} \)
\[ x := 0 \]

\[ \textbf{while} \ast \textbf{do} \]
\[ y := \text{malloc}() \]
\[ (\ast y).\text{next} := x \]
\[ x := y \]
\[ \textbf{done} \]

\[ \textbf{while} x \neq 0 \textbf{ do} \]
\[ x := (\ast x).\text{next} \]
\[ \textbf{done} \]
Separation logic based shape analysis

\[
x := 0
\]

while * do
  \[
  y := \text{malloc()}
  \]
  \[
  \ast y).next := x
  \]
  \[
  x := y
  \]
done

while \( x \neq 0 \) do
  \[
  x := (\ast x).next
  \]
done
Separation logic based shape analysis

\[ x := 0 \]

\[ \text{while } * \text{ do} \]
\[ y := \text{malloc()} \]
\[ (*y).\text{next} := x \]
\[ x := y \]
\[ \text{done} \]

\[ \text{while } x \neq 0 \text{ do} \]
\[ x := (*x).\text{next} \]
\[ \text{done} \]
Separation logic based shape analysis

\[
x := 0
\]
\[
\text{while } * \text{ do}
\]
\[
y := \text{malloc()}
\]
\[
(*y)\text{.next} := x
\]
\[
x := y
\]
\[
\text{done}
\]

\[
\text{while } x \neq 0 \text{ do}
\]
\[
x := (*x)\text{.next}
\]
\[
\text{done}
\]
Separation logic based shape analysis

\[ x := 0 \]
\[ \text{while } * \text{ do } \]
\[ y := \text{malloc()} \]
\[ (*y).next := x \]
\[ x := y \]
\[ \text{done} \]

\[ \text{while } x \neq 0 \text{ do } \]
\[ x := (*x).next \]
\[ \text{done} \]
Separation logic based shape analysis

\[ x := 0 \]
\[ \text{emp} \]
\[ \text{emp} \land x = 0 \]

\[ \text{while } * \text{ do} \]
\[ y := \text{malloc()} \]
\[ (*y).\text{next} := x \]
\[ x := y \]
\[ \text{done} \]

\[ \text{emp} * y \leftrightarrow \text{next} : \_ \land x = 0 \]
\[ \text{emp} * y \leftrightarrow \text{next} : 0 \land x = 0 \]

\[ \text{while } x \neq 0 \text{ do} \]
\[ x := (*x).\text{next} \]
\[ \text{done} \]

\[ \text{emp} * x, y \leftrightarrow \text{next} : 0 \]
Separation logic based shape analysis

\[ x := 0 \quad \text{emp} \]
\[ \text{while } \ast \text{ do} \]
\[ \quad y := \text{malloc()} \quad \text{emp } \ast y \leftrightarrow \text{next } : \_ \land x = 0 \]
\[ \quad (*y).\text{next} := x \quad \text{emp } \ast y \leftrightarrow \text{next } : 0 \land x = 0 \]
\[ \quad x := y \]
\[ \text{done} \]
\[ \text{while } x \neq 0 \text{ do} \]
\[ \quad x := (*x).\text{next} \]
\[ \text{done} \]
Separation logic based shape analysis

\[ x := 0 \]
\[ \text{while } * \text{ do} \]
\[ y := \text{malloc()} \]
\[ (*y).\text{next} := x \]
\[ x := y \]
\[ \text{done} \]
\[ \text{while } x \neq 0 \text{ do} \]
\[ x := (*x).\text{next} \]
\[ \text{done} \]
Separation logic based shape analysis

\[
x := 0
\]

\[
\text{while } * \text{ do}
\]

\[
y := \text{malloc()}
\]

\[
(*y).\text{next} := x
\]

\[
x := y
\]

done

\[
\text{while } x \neq 0 \text{ do}
\]

\[
x := (*x).\text{next}
\]

done

emp \land x = 0

ls(x, 0) \land x = y
Separation logic based shape analysis

\[ \text{emp} \land x = 0 \]
\[ \lor \]
\[ \text{ls}(x, 0) \land x = y \]

\[ x := 0 \]

**while** * do
\[ y := \text{malloc}() \]
\[ (*y).\text{next} := x \]
\[ x := y \]
**done**

**while** \[ x \neq 0 \] **do**
\[ x := (*x).\text{next} \]
**done**
Separation logic based shape analysis

\[ \text{emp} \land x = 0 \]
\[ \vee \]
\[ \text{ls}(x, 0) \land x = y \]

\[ x := 0 \]

\textbf{while} * \textbf{do}
\[ y := \text{malloc()} \]
\[ (\ast y).\text{next} := x \]
\[ x := y \]
\textbf{done}

\textbf{while} x \neq 0 \textbf{do}
\[ x := (\ast x).\text{next} \]
\textbf{done}
Separation logic based shape analysis

\[ \text{emp} \land x = 0 \]

\[ \forall \text{ls}(x, 0) \land x = y \]

\[ \text{ls}(x, 0) \land y \leftrightarrow \text{next} : - \]

\[
\begin{align*}
\text{x} & := 0 \\
\text{while } * \text{ do} \\
& \quad \text{y} := \text{malloc()} \\
& \quad (*y).\text{next} := x \\
& \quad x := y \\
\text{done}
\end{align*}
\]

\[
\begin{align*}
\text{while } x \neq 0 \text{ do} \\
& \quad x := (*x).\text{next} \\
\text{done}
\end{align*}
\]
Separation logic based shape analysis

\[
x := 0
\]

\[
\textbf{while } \ast \textbf{ do}
\]

\[
y := \text{malloc}()
\]
\[
(*y)\.next := x
\]
\[
x := y
\]

\[
\text{done}
\]

\[
\textbf{while } x \neq 0 \textbf{ do}
\]

\[
x := (*x)\.next
\]

\[
\text{done}
\]
Separation logic based shape analysis

\[
\begin{align*}
\text{emp} \land x &= 0 \\
\land (x, 0) \land x &= y \\
\text{while } * \text{ do} \\
\quad y &= \text{malloc}() \\
\quad (*y).\text{next} &= x \\
\quad x &= y \\
\text{done} \\
\text{while } x \neq 0 \text{ do} \\
\quad x &= (*x).\text{next} \\
\text{done}
\end{align*}
\]
Separation logic based shape analysis

\[ \text{emp} \wedge x = 0 \]
\[ \text{ls}(x, 0) \wedge x = y \]

\begin{align*}
x & := 0 \\
\text{while } * \text{ do} \\
\quad y & := \text{malloc()} \\
\quad (*y).\text{next} & := x \\
\quad x & := y \\
\text{done}
\end{align*}

\begin{align*}
\text{ls}(x, 0) \wedge y \leftrightarrow \text{next : } & \\
\text{ls}(x, 0) \wedge y \leftrightarrow \text{next : } x
\end{align*}

\[ \exists n. \text{ls}(n, 0) \wedge x, y \leftrightarrow \text{next : } n \]

\[ \text{ls}(x, 0) \wedge x = y \]

\begin{align*}
\text{while } x \neq 0 \text{ do} \\
\quad x & := (*x).\text{next} \\
\text{done}
\end{align*}
Separation logic based shape analysis

\[ \text{emp} \land x = 0 \]

\[ \text{ls}(x, 0) \land x = y \]

\[ x := 0 \]

\[ \text{while } * \text{ do} \]
\[ \quad y := \text{malloc()} \]
\[ \quad (*y).\text{next} := x \]
\[ \quad x := y \]
\[ \text{done} \]

\[ \text{while } x \neq 0 \text{ do} \]
\[ \quad x := (*x).\text{next} \]
\[ \text{done} \]
x := 0

while * do
    y := malloc()
    (*y).next := x
    x := y
done

while x ≠ 0 do
    x := (*x).next
done

ls(x, 0)
\begin{verbatim}
x := 0

while * do
    y := malloc()
    (*y).next := x
    x := y
\end{verbatim}

\begin{verbatim}
while x \neq 0 do
    x := (*x).next
\end{verbatim}

\begin{verbatim}
l_0(x, 0)
\exists n. x \mapsto next : n * l_0(n, 0)
\end{verbatim}
\[
x := 0
\]

\[
\textbf{while } * \textbf{ do}
\]
\[
y := \text{malloc}()
\]
\[
(*y)\text{.next} := x
\]
\[
x := y
\]
\[
\textbf{done}
\]

\[
\textbf{while } x \neq 0 \textbf{ do}
\]
\[
x := (\star x)\text{.next}
\]
\[
\textbf{done}
\]

\[
\exists n. \; x \mapsto \text{next} : n \times \text{ls}(n, 0)
\]

\[
\exists m. \; m \mapsto \text{next} : x \times \text{ls}(x, 0)
\]

\[
\text{Separation logic based shape analysis}
\]
\textbf{Separation logic based shape analysis}

\begin{align*}
x &:= 0 \\
\textbf{while} * \textbf{do} & \\
& y := \text{malloc}() \\
& (*y).\text{next} := x \\
& x := y \\
\textbf{done} \\
\textbf{while} x \neq 0 \textbf{do} & \\
& x := (*x).\text{next} \\
\textbf{done} \\
& \exists n. \ x \mapsto \text{next} : n * \text{ls}(n, 0) \\
& \exists m. \ m \mapsto \text{next} : x * \text{ls}(x, 0) \quad \checkmark
\end{align*}
\[\begin{align*}
x & := 0 \\
\textbf{while} \ x \neq 0 \ 	extbf{do} \\
& \quad y := \text{malloc}() \\
& \quad (\star y).\text{next} := x \\
& \quad x := y \\
\text{done}
\end{align*}\]
\( x := 0 \)

\[ \text{while } * \text{ do} \]
\[ \quad y := \text{malloc()} \]
\[ \quad (\ast y).\text{next} := x \]
\[ \quad x := y \]
\[ \text{done} \]

\[ \text{while } x \neq 0 \text{ do} \]
\[ \quad t := x \]
\[ \quad x := (\ast x).\text{next} \]
\[ \quad \text{free}(t) \]
\[ \quad \text{done} \]

\[ \text{ls}(x, 0) \]
\[ \exists n. \ x \mapsto \text{next} : n \ast \text{ls}(n, 0) \]
\[ \exists n. \ x \mapsto \text{next} : n \ast \text{ls}(n, 0) \land t = x \]
\[ t \mapsto \text{next} : x \ast \text{ls}(x, 0) \]
\[ x := 0 \]

\textbf{while} \* \textbf{do}
\hspace{1cm} y := \text{malloc}()
\hspace{1cm} (*y).\text{next} := x
\hspace{1cm} x := y
\hspace{1cm} done

\textbf{while} x \neq 0 \textbf{do}
\hspace{1cm} t := x
\hspace{1cm} x := (*x).\text{next}
\hspace{1cm} \text{free}(t)
\hspace{1cm} done

\text{ls}(x, 0)
\exists n. x \leftrightarrow \text{next} : n * \text{ls}(n, 0)
\exists n. x \leftrightarrow \text{next} : n * \text{ls}(n, 0) \land t = x
\text{free}(t)
\text{ls}(x, 0)
x := 0

while * do
    y := malloc()
    (*y).next := x
    x := y
done

while x \neq 0 do
    t := x
    x := (*x).next
    free(t)
done
\texttt{x := 0}

\texttt{while * do}
\begin{align*}
y & := \text{malloc}() \\
(*y).\text{next} & := x \\
x & := y
\end{align*}
\texttt{done}

\texttt{while x \neq 0 do}
\begin{align*}
t & := x \\
x & := (*x).\text{next} \\
\text{free}(t) \\
\texttt{done}
\end{align*}
\[ x := 0 \]

\[
\textbf{while} * \textbf{do} \\
\quad y := \text{malloc}() \\
\quad (\ast y).\text{next} := x \\
\quad x := y \\
\textbf{done}
\]

\[
\textbf{while} x \neq 0 \textbf{ do} \\
\quad t := x \\
\quad x := (\ast x).\text{next} \\
\quad \text{free}(t) \\
\textbf{done}
\]
\( x := 0 \)

\textbf{while} * do
\begin{align*}
  y & := \text{malloc}() \\
  (*y).\text{next} & := x \\
  x & := y
\end{align*}
\textbf{done}

\textbf{while} \( x \neq 0 \) do
\begin{align*}
  t & := x \quad \text{ls}(k_1, x, 0) \\
  x & := (*x).\text{next} \quad \exists n. x \mapsto \text{next} : n \ast \text{ls}(k_2, n, 0) \land k_2 = k_1 - 1 \\
  \text{free}(t) \quad \cdots
\end{align*}
\textbf{done}

\text{ls}(k_5, x, 0) \land k_5 = k_4
Separation logic based shape analysis

\[ x := 0 \]

while * do
  \[ y := \text{malloc()} \]
  \[ (*y).\text{next} := x \]
  \[ x := y \]
done

Substitution: \( k_1 := k_5 \)

\[ \text{while } x \neq 0 \text{ do} \]
  \[ t := x \]
  \[ \exists n. x \mapsto \text{next}: n \star \text{ls}(k_2, n, 0) \land k_2 = k_1 - 1 \]
  \[ x := (*x).\text{next} \]
  \[ \text{free}(t) \]
done

\[ \text{ls}(k_5, x, 0) \land k_5 = k_4 \]
Separation logic based shape analysis

while $k_1 \neq 0$ do
  $k_2 := k_1 - 1$
  $k_3 := k_2$
  $k_4 := k_3$
  $k_5 := k_4$
  $k_1 := k_5$
end

while $x \neq 0$ do
  $t := x$
  $x := (*x).next$
  free($t$)
end

$\exists n. x \leftrightarrow$ next : $n \ast ls(k_2, n, 0) \land k_2 = k_1 - 1$

$ls(k_1, x, 0)$

$ls(k_5, x, 0) \land k_5 = k_4$
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 formulae 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
Arithmetic Strengthening for Shape Analysis

Stephen Magill\textsuperscript{2}, Josh Berdine\textsuperscript{1}, Edmund Clarke\textsuperscript{2}, and Byron Cook\textsuperscript{1}

\textsuperscript{1} Carnegie Mellon University
\textsuperscript{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
Arithmetic abstraction fairly accurate for termination and preserves data in many cases
Outline

- Fair termination
- Data structures
- Concurrency
- Conclusion
Introduction
Introduction

Proving Thread Termination

Byron Cook  
Microsoft Research  
bycook@microsoft.com

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

Andrey Rybalchenko  
EPFL and MPI  
ryba@mpl-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.

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

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);

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

Byron Cook
Microsoft Research
bycook@microsoft.com

Abstract
Concurrent programs are often written by teams, with each team executing within critical sections of code that may be executed by other threads. Unfortunately, the current state of the art in thread abstraction is not well-equipped to handle this scenario. Our method finds that the thread requires its environment in order to exist, by looking at the conditions necessary to prove termination. The environment abstraction used in this paper is designed to abstract threads that abstract away the surrounding environment. Our method finds that the thread requires its environment in order to exist, by looking at the conditions necessary to prove termination. The environment abstraction used in this paper is designed to abstract threads that abstract away the surrounding environment.

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

First steps towards a solution. Gives flavor of more advanced solutions

Figure 1. A keyboard device driver whose termination partially depends on the correct behavior of other threads from the driver.
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
Proving That Non-Blocking Algorithms Don’t Block

Alexey Gotsman
University of Cambridge

Byron Cook

Abstract
A concurrent data structure is said to be non-blocking if it maintains freedom, lock-free execution, and their proofs for non-blocking are indeed non-blocking algorithms. Pencil-and-paper semi-formal proofs are most common in these situations while overcoming the presence of concurrency.

Categories and Subject Descriptors: Software/Program Verification [Software/Program Verification]: Specifying and Verifying Programs

General Terms Languages, Testers

Keywords Formal Verification, Concurrent Programming, Liveness, Termination

Much more advanced solution to related problem. Complex.
Multithreaded programs

\[
\begin{align*}
[P_1] & \triangleq (I_1, R_1, S_1) \\
[P_2] & \triangleq (I_2, R_2, S_2) \\
& \vdots \\
[P_n] & \triangleq (I_n, R_n, S_n)
\end{align*}
\]
$S \triangleq S_1 \times S_2 \times \ldots \times S_n$

$I \triangleq \{ (s'_1, s'_2, \ldots, s'_n) \mid \forall i \in \{1, \ldots, n\}. \ s_i \in I_i \\
\land \forall i, j \in \{1, \ldots, n\}, v \in \text{GLOBALS}(\mathcal{P}) . \ s_i(v) = s_j(v) \\
\land \forall i \in \{1, \ldots n\}. \ s'_i = s_i[\text{tid} \mapsto i] \}$

$R \triangleq \{ ((s_1, s_2, \ldots s_n), (t_1, t_2, \ldots t_n)) \mid \exists i \in \{1, \ldots, n\}. \\
\land \forall j \in \{1, \ldots, n\} - \{i\}, v \in \text{LOCALS}(\mathcal{P}) . \ s_j(v) = t_j(v) \\
\land \forall j, k \in \{1, \ldots, n\}, v \in \text{GLOBALS}(\mathcal{P}). \ t_j(v) = t_k(v) \\
\land (s_i, t_i) \in R_i \}$
\[
[x := e]_V \triangleq \{ (s, t) \mid x \not\in \text{LOCKS}(\mathcal{P}) \land \ldots \}
\]

\[
[\text{lock}(x)]_V \triangleq \{ (s, t) \mid x \in \text{LOCKS}(\mathcal{P}) \land \forall v \in V - \{x\}. s(v) = t(v) \land s(x) = 0 \land t(x) = s(\text{tid}) \}
\]

\[
[\text{unlock}(x)]_V \triangleq \{ (s, t) \mid x \in \text{LOCKS}(\mathcal{P}) \land \forall v \in V - \{x\}. s(v) = t(v) \land s(x) = s(\text{tid}) \land t(x) = 0 \}
\]
Definition. Thread $P_i$ is *thread-terminating* in $P$ iff $P_i$ can only make a finite number of steps in any $P$-trace.
**Definition.** Thread $P_i$ is *thread-terminating* in $P$ iff $P_i$ can only make a finite number of steps in any $P$-trace.

**Note:** Not ruling out deadlock
Definition. Thread $P_i$ is thread-terminating in $P$ iff $P_i$ can only make a finite number of steps in any $P$-trace.

For simplicity also ignoring fairness

$$x := 0 \parallel \text{while } (x \neq 0) \text{ skip}$$
Thread termination

\[ P_1 \quad | \quad P_2 \]
Thread termination

$\mathcal{P}_1 \ A \ \mathcal{P}_2$
Thread termination

\[ P_1 \rightarrow A \rightarrow P_2 \]
Different types of checks

\[ P_1 \quad A \quad P_2 \]
Different types of checks

Definition. An agreement $A$ is a binary relation over states that constrains the change of global states, and leaves the change of local states unconstrained.
Abstract composition. Assume that $P = (I, R, S)$. $P \triangleright A \triangleq (I_\triangle, R_\triangle, S_\triangle)$ such that

$I_\triangle \triangleq \{ s \mid \exists s' \in I \land \forall v \in \text{LOCALS}(P). \ s(v) = s'(v) \}$

$S_\triangle \triangleq \{ s \mid \exists s' \in S \land \forall v \in \text{LOCALS}(P). \ s(v) = s'(v) \}$

$R_\triangle \triangleq [A \cap \text{Id}_{\text{LOCALS}(P)}]^*; R$
Thread invariants.

\[
\begin{align*}
\text{REACH}(I, R, S) & \triangleq R^*(I) \\
\text{REACH}_i(P) & \triangleq \{ s_i \mid (\ldots, s_i, \ldots) \in \text{REACH}(P) \} \\
\mathcal{R}_{I,i} & \triangleq \mathcal{R}_{i \upharpoonright \text{REACH}_i(P)}
\end{align*}
\]
**Thread invariants.**

\[ \text{REACH}(I, R, S) \triangleq R^*(I) \]
\[ \text{REACH}_i(P) \triangleq \{ s_i \mid (\ldots, s_i, \ldots) \in \text{REACH}(P) \} \]
\[ \mathcal{R}_{I,i} \triangleq \mathcal{R}_i \upharpoonright \text{REACH}_i(P) \]

**Lemma.**

\[ \forall j \in \{1, \ldots, n\} - \{i\}. \mathcal{R}_{I,j} \subseteq A \]
\[ \implies \]
\[ \text{REACH}_i(P) \subseteq \text{REACH}(P_i \mid \triangle A) \]
Theorem. $P_i$ is thread terminating if there exists an $A$ such that

- $\forall j \in \{1, \ldots, n\} - \{i\}. R_{I,j} \subseteq A$, and

- $P_i \big|_\triangle A$ is a terminating (sequential) program.
\( \mathcal{A} := \text{true; } \)
\[ \text{while } \mathcal{P}_i \models_{\Delta} \mathcal{A} \text{ not provably terminating do} \]
\[ \pi := \text{lasso counterexample; } \]
\[ \text{if strengthening of } \mathcal{A} \text{ not possible using } \pi \text{ then} \]
\[ \text{return "}\mathcal{P}_i \text{ could not be proved terminating";} \]
\[ \text{else} \]
\[ \mathcal{A} := \text{strengthening of } \mathcal{A} \text{ using } \pi; \]
\[ \text{fi} \]
\[ \text{foreach } j \in \{1, \ldots n\} - i \text{ do} \]
\[ \text{if } \mathcal{R}_{\mathcal{I},j} \not\subseteq \mathcal{A} \text{ then} \]
\[ \mathcal{A} := \text{weakening using invariant for thread } j; \]
\[ \text{fi} \]
\[ \text{od} \]
\[ \text{od} \]
\[ \text{return "}\mathcal{P}_i \text{ terminates in } \mathcal{P}"; \]
Example

\( \mathcal{P}_1 \)
lock(\text{lk});
while * do
   assume(x > 0);
   x := x - 1;
od
unlock(\text{lk});

\( \mathcal{P}_2 \)
while * do
   lock(\text{lk});
   assume(x > 0);
   x := x - 1;
   unlock(\text{lk});
od

\( \mathcal{P}_3 \)
while * do
   lock(\text{lk});
   x := *
   unlock(\text{lk});
od
\[ P_1 \]
lock(lk);
while * do
  assume(x > 0);
  \textcolor{red}{x := x - 1;}
\textcolor{red}{\text{od}}
unlock(lk);

\[ P_2 \]
while * do
  lock(lk);
  assume(x > 0);
  \textcolor{red}{x := x - 1;}
  unlock(lk);
\textcolor{red}{\text{od}}

\[ P_3 \]
while * do
  lock(lk);
  \textcolor{red}{x := *;}
  unlock(lk);
\textcolor{red}{\text{od}}

Which threads "thread terminate"?
Example

\[ \mathcal{P}_1 \]
\[
\begin{align*}
\text{lock}(lk); \\
\text{while } * \text{ do} \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\text{od} \\
\text{unlock}(lk);
\end{align*}
\]

\[ \mathcal{P}_2 \]
\[
\begin{align*}
\text{while } * \text{ do} \\
\quad \text{lock}(lk); \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\text{unlock}(lk); \\
\text{od}
\end{align*}
\]

\[ \mathcal{P}_3 \]
\[
\begin{align*}
\text{while } * \text{ do} \\
\quad \text{lock}(lk); \\
\quad x := *; \\
\quad \text{unlock}(lk); \\
\quad \text{od}
\end{align*}
\]

Which threads "thread termina

\[
\begin{align*}
A := \text{true}; \\
\text{while } \mathcal{P}_i \models A \text{ not provably terminating do} \\
\quad \pi := \text{lasso counterexample}; \\
\text{if strengthening of } A \text{ not possible using } \pi \text{ then} \\
\quad \text{return } \text{"} \mathcal{P}_i \text{ could not be proved terminating"}; \\
\text{else} \\
\quad A := \text{strengthening of } A \text{ using } \pi; \\
\text{fi} \\
\text{foreach } j \in \{1, \ldots, n\} - i \text{ do} \\
\quad \text{if } \mathcal{R}_{I,j} \not\subseteq A \text{ then} \\
\quad \quad A := \text{weakening using invariant for thread } j; \\
\quad \text{fi} \\
\text{od} \\
\text{return } \text{"} \mathcal{P}_i \text{ terminates in } \mathcal{P}";
\end{align*}
\]
\( \mathcal{A} = \text{true} \)

\begin{align*}
\mathcal{P}_1 &\quad \text{lock}(lk); \\
&\quad \text{while } \ast \text{ do} \\
&\quad \quad \text{assume}(x > 0); \\
&\quad \quad x := x - 1; \\
&\quad \text{od} \\
&\quad \text{unlock}(lk); \\
\end{align*}

\begin{align*}
\mathcal{P}_2 &\quad \text{while } \ast \text{ do} \\
&\quad \quad \text{lock}(lk); \\
&\quad \quad \text{assume}(x > 0); \\
&\quad \quad x := x - 1; \\
&\quad \text{unlock}(lk); \\
&\quad \text{od} \\
\end{align*}

\begin{align*}
\mathcal{P}_3 &\quad \text{while } \ast \text{ do} \\
&\quad \quad \text{lock}(lk); \\
&\quad \quad x := \ast; \\
&\quad \quad \text{unlock}(lk); \\
&\quad \text{od} \\
\end{align*}

\begin{align*}
\mathcal{A} &:= \text{true}; \\
\text{while } \mathcal{P}_i \mid_\triangle \mathcal{A} \not\text{ provably terminating do} \\
&\quad \pi := \text{lasso counterexample}; \\
&\quad \text{if strengthening of } \mathcal{A} \not\text{ possible using } \pi \text{ then} \\
&\quad \quad \text{return } \text{“} \mathcal{P}_i \text{ could not be proved terminating”;} \\
&\quad \text{else} \\
&\quad \quad \mathcal{A} := \text{strengthening of } \mathcal{A} \text{ using } \pi; \\
&\quad \quad \text{fi} \\
&\quad \text{foreach } j \in \{1, \ldots n\} - i \text{ do} \\
&\quad \quad \text{if } \mathcal{R}_{i,j} \notin \mathcal{A} \text{ then} \\
&\quad \quad \quad \mathcal{A} := \text{weakening using invariant for thread } j; \\
&\quad \quad \text{fi} \\
&\quad \text{od} \\
&\quad \text{return } \text{“} \mathcal{P}_i \text{ terminates in } \mathcal{P}”; \\
\end{align*}
\[ P_1 \]
lock(lk);
while * do
  assume(x > 0);
  \( x := x - 1; \)
od
unlock(lk);

\( \mathcal{A} = \text{true} \)

\[ P_2 \]
while * do
  lock(lk);
  assume(x > 0);
  \( x := x - 1; \)
  \[ \mathcal{A} \cap \text{ID}\{\text{tid}\} \]*;
  unlatch(j);
  return \( \gamma \); terminates in \( \gamma \);
unlock(lk);
\texttt{c := *;}
\texttt{while * do}
\texttt{c := c - 1;}
\texttt{assume(c > 0);}
\texttt{use( A}
\texttt{\land 'lk = 1 \Rightarrow lk = 1}
\texttt{\land 'lk \neq 1 \Rightarrow lk \neq 1}
\texttt{);}
\texttt{od}
\texttt{unlock(lk);}

\textbf{A} = \texttt{true}

\texttt{\textbf{A} \cap ID\{tid\}^*;}  
\texttt{lock(lk);}  
\texttt{while * do}
\texttt{\textbf{A} \cap ID\{tid\}^*;}  
\texttt{assume(x > 0);}  
\texttt{\textbf{A} \cap ID\{tid\}^*;}  
\texttt{x := x - 1;}  
\texttt{od}
\texttt{unlock(lk);}  
\texttt{return \; j \; terminates \; in \; \eta ;}
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);
    \[ \mathcal{A} \cap ID_{\{tid\}} \] *;
    x := x - 1;
    unlock(lk);
od
\[ \mathcal{A} \cap ID_{\{tid\}} \] *;

\[ A = \text{true} \]
Example

\( P_1 \)

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

\[ A = \text{true} \]

\( P_2 \)

while *
  lock(\( lk \));
  assume(\( x > 0 \));
  \[ \pi_c = \exists x_1, x_3, \ldots \]
  \( lk = lk' = 1 \)
  \( x = x_1 \land x' = x_3 \)
  \( x_1 > 0 \)
  \( x_3 = x_2 - 1 \)
  \[ [A \cap ID\{tid\}]^* ; \]
  \[ [A \cap ID\{tid\}]^* ; \]
  \( x := x - 1 \);
od
unlock(\( lk \));
Example

$\mathcal{P}_1$

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

$\mathcal{P}_2$

lock($lk$);
while *

$[\mathcal{A} \cap \text{ID}\{\text{tid}\}]^*$;

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

\[ \mathcal{P}_1 \]
lock(\text{lk});
while * do
    assume(x > 0);
    x := x - 1;
od
unlock(\text{lk});

\[ \mathcal{P}_2 \]
lock(\text{lk});
while *
    assume(x > 0);
    x := x - 1;
un\text{lock}(\text{lk});

\( A = \text{true} \)

\[ [\mathcal{A} \cap \text{ID} \{\text{tid}\}]^*; \]

\[ [\pi_c] = \exists x_1, x_3, \ldots \]
\[ lk = \text{l}k' = 1 \]
\[ x = x_1 \land x' = x_3 \]
\[ x_1 > 0 \]
\[ x_3 = x_2 - 1 \]

\( A \leftarrow \text{true} \)

while \( \mathcal{P}_i \triangleright A \) not provably terminating do
    \( \pi \leftarrow \text{lasso counterexample}; \)
    if strengthening of \( A \) not possible using \( \pi \) then
        return "\( \mathcal{P}_i \) could not be proved terminating";
    else
        \( A \leftarrow \text{strengthening of } A \text{ using } \pi; \)
    fi
    foreach \( j \in \{1, \ldots n\} - i \) do
        if \( \mathcal{R}_{i,j} \not\subseteq A \) then
            \( A \leftarrow \text{weakening using invariant for thread } j; \)
        fi
    od
od
return "\( \mathcal{P}_i \) terminates in \( \mathcal{P} \)";
Example

\[ \pi_c = \exists x_1, x_3, \ldots \]
\[ l_k = l_{k'} = 1 \]
\[ x = x_1 \land x' = x_3 \]
\[ x_1 > 0 \]
\[ x_3 = x_2 - 1 \]

\[ x := x' \]
\[ \textbf{unlock}(l_k); \]

\[ \mathcal{P}_1 \]
\textbf{lock}(l_k);
\textbf{while} * \textbf{do}
\hspace{1cm} \textbf{assume}(x > 0);
\hspace{1cm} x := x - 1;
\hspace{1cm} \textbf{od}
\textbf{unlock}(l_k);

\[ A = \text{true} \]

\[ \forall i \in \{1, \ldots, n\} - i \]
\[ \textbf{if} \ \mathcal{R}_{i,j} \not\subseteq A \ 	extbf{then} \]
\[ \textbf{A := weakening using invariant for thread } j \]
\[ \textbf{fi} \]
\[ \textbf{od} \]

\[ \textbf{return} \ "\mathcal{P}_i \text{ terminates in } \mathcal{P}\"; \]
Example

$P_1$
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

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

x := x - 1,
unlock(lk);

$A := true;$
while $P_i \mid \Delta A$ not provably terminating do

return "$P_i$ terminates in $P$";

Key idea: we can examine the counterexample in isolation of the counterexample
\( \mathcal{P}_1 \)

\begin{verbatim}
lock(lk);

while * do
    assume(x > 0);
    x := x - 1;

od

unlock(lk);
\end{verbatim}

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

\[ x := x_2, \]

\[ unlock(lk); \]

\[ \text{true;} \]

\[ \text{while } \mathcal{P}_i | \triangle A \text{ not provably terminating do} \]

\[ \text{return } "\mathcal{P}_i \text{ terminates in } \mathcal{P}"; \]

**Key idea:** we can examine the counterexample in isolation of the counterexample.
Example

$P_1$

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

\[ [\pi_c^s] \subseteq \geq x \]
\[ [\pi_c] = \exists x_1, x_3, \ldots \]
\[ \text{lk} = \text{lk}' = 1 \]
\[ x = x_1 \land x' = x_2 \]
\[ x_1 > 0 \]
\[ x_2 = x_1 - 1 \]
\[ x_3 = x_2 - 1 \]
\[ x := \ldots ; \]
\[ \text{unlock}(\text{lk}); \]

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

return "$P_i$ terminates in $P$";
Example

\[ \mathcal{P}_1 \]

\hspace{1cm} \text{lock}(lk); \\
\hspace{1cm} \text{while } \ast \text{ do} \\
\hspace{2cm} \text{assume}(x > 0) \\
\hspace{2cm} x := x - 1; \\
\hspace{1cm} \text{od} \\
\hspace{1cm} \text{unlock}(lk); \\
\]

\[ \mathcal{A} = \text{true} \]

\[ [\pi^s_c] \subseteq \geq_X \]

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

\[ x := \ldots \\
\hspace{1cm} \text{unlock}(lk); \]

\[ \text{true;} \]

\[ \text{while } \mathcal{P}_i \vdash \mathcal{A} \text{ not provably terminating do} \]

\[ \pi := \text{lasso counterexample;} \]

\[ \text{if strengthening of } \mathcal{A} \text{ not possible using } \pi \text{ then} \]

\[ \text{return } "\mathcal{P}_i \text{ could not be proved terminating}"; \]

\[ \text{else} \]

\[ \mathcal{A} := \text{strengthening of } \mathcal{A} \text{ using } \pi; \]

\[ \text{fi} \]

\[ \text{foreach } j \in \{1, \ldots n\} - i \text{ do} \]

\[ \text{if } \mathcal{R}_{i,j} \not\subseteq \mathcal{A} \text{ then} \]

\[ \mathcal{A} := \text{weakening using invariant for thread } j \]

\[ \text{fi} \]

\[ \text{od} \]

\[ \text{return } "\mathcal{P}_i \text{ terminates in } \mathcal{P}"; \]
\( \mathcal{P}_1 \)

lock(lk);

while * do

assume(x > 0)

x := x - 1;

od

unlock(lk);

\[ [\pi^s_c] \subseteq \geq_x \]

\[ [\pi_c] = \exists x_1, x_3, \ldots \]

\[ lk = lk' = 1 \]

\[ x = x_1 \land x' = x_2 \]

\[ x_1 > 0 \]

\[ x_2 = x_1 - 1 \]

\[ x_3 = x_2 - 1 \]

unlock(lk);

\( \mathcal{A} = \text{true} \)

\[ \land \geq_x \]

\[ \text{true; while } \mathcal{P}_i \models \Delta \mathcal{A} \text{ not provably terminating do} \]

\[ \pi := \text{lasso counterexample;} \]

if strengthening of \( \mathcal{A} \) not possible using \( \pi \) then

\[ \text{return "\( \mathcal{P}_i \) could not be proved terminating";} \]

else

\[ \mathcal{A} := \text{strengthening of } \mathcal{A} \text{ using } \pi; \]

fi

foreach \( j \in \{1, \ldots n\} - i \) do

if \( \mathcal{R}_{I,j} \not\subseteq \mathcal{A} \) then

\[ \mathcal{A} := \text{weakening using invariant for thread } j \]

fi

od

od

\[ \text{return "\( \mathcal{P}_i \) terminates in } \mathcal{P};" \]
Example

\begin{align*}
\mathcal{P}_1 & \quad \text{lock}(lk); \\
& \quad \text{while } \ast \text{ do} \\
& \quad \quad \text{assume}(x > 0); \\
& \quad \quad x := x - 1; \\
& \quad \quad \text{od} \\
& \quad \text{unlock}(lk); \\
\mathcal{P}_2 & \quad \text{while } \ast \text{ do} \\
& \quad \quad \text{lock}(lk); \\
& \quad \quad \text{assume}(x > 0); \\
& \quad \quad x := x - 1; \\
& \quad \quad \text{unlock}(lk); \\
& \quad \quad \text{od} \\
\mathcal{P}_3 & \quad \text{while } \ast \text{ do} \\
& \quad \quad \text{lock}(lk); \\
& \quad \quad x := \ast; \\
& \quad \quad \text{unlock}(lk); \\
& \quad \quad \text{od} \\
\end{align*}

\[ A = \text{true} \land \models_x \]

\[ A := \text{true; while } \mathcal{P}_i \models A \text{ not provably terminating do} \]
\[ \pi := \text{lasso counterexample;} \]
\[ \text{if strengthening of } A \text{ not possible using } \pi \text{ then} \]
\[ \quad \text{return } \text{“} \mathcal{P}_i \text{ could not be proved terminating”;} \]
\[ \text{else} \]
\[ \quad A := \text{strengthening of } A \text{ using } \pi; \]
\[ \text{fi} \]
\[ \text{foreach } j \in \{1, \ldots, n\} - i \text{ do} \]
\[ \quad \text{if } R_{I,j} \not\in A \text{ then} \]
\[ \quad \quad A := \text{weakening using invariant for thread } j; \]
\[ \quad \text{fi} \]
\[ \text{od} \]
\[ \text{return } \text{“} \mathcal{P}_i \text{ terminates in } \mathcal{P}”; \]}
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

\[ A := \text{true; } \]
while \( \mathcal{P}_i \mid \Delta \) \( A \) not provably terminating do
  \( \pi := \text{lasso counterexample; } \)
  if strengthening of \( A \) not possible using \( \pi \) then
    return "\( \mathcal{P}_i \) could not be proved terminating";
  else
    \( A := \text{strengthening of } A \) using \( \pi \);
  fi
  foreach \( j \in \{1,\ldots,n\} - i \) do
  if \( \mathcal{R}_{l,j} \not\subseteq A \) then
    \( A := \text{weakening using invariant for thread } j \);
  fi
  od
  return "\( \mathcal{P}_i \) terminates in \( \mathcal{P} \)";
Example

$P_1$

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

$P_2$

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

$P_3$

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

Checking a property like $R \subseteq \preceq_x R^*(I) \subseteq V$

$K \forall j \in T$ then  
  $A :=$ weakening using invariant for thread $j$;  
  fi  
od  
od  
return "$P_i$ terminates in $P$";
\[ \mathcal{A} = \text{true} \]
\[ \land \quad \mathcal{G} \geq x \]

\[ \mathcal{P}_1 \]
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

\[ A := \text{true}; \]
while \( \mathcal{P}_i \mid \mathcal{A} \) not provably terminating do
  \( \pi := \text{lasso counterexample}; \)
  if strengthening of \( \mathcal{A} \) not possible using \( \pi \) then
    return "\( \mathcal{P}_i \) could not be proved terminating";
  else
    \( \mathcal{A} := \text{strengthening of } \mathcal{A} \text{ using } \pi; \)
  fi
  foreach \( j \in \{1, \ldots, n\} - i \) do
    if \( \mathcal{R}_{i,j} \not\in \mathcal{A} \) then
      \( \mathcal{A} := \text{weakening using invariant for thread } j; \)
    fi
  od
return "\( \mathcal{P}_i \) terminates in \( \mathcal{P} \)";
$$\mathcal{P}_1$$
lock(lk);
while * do
  assume(x > 0);
  x := x - 1;
endwhile
unlock(lk);

$$\mathcal{P}_2$$
while * do
  lock(lk);
  assume(x > 0);
  x := x - 1;
endwhile
unlock(lk);

$$\mathcal{P}_3$$
while * do
  lock(lk);
  x := *;
  unlock(lk);
endwhile

$$\mathcal{A} = \text{true}$$
∧ \begin{align*}
\begin{array}{c}
\exists x
\end{array}
\end{align*}

\begin{align*}
&\text{\begin{tabular}{l}
A := true; \\
while } P_i \mid A \text{ not provably terminating do} \\
\quad \pi := \text{lasso counterexample;}
\end{tabular}} \\
&\text{\begin{tabular}{l}
\quad if strengthening of } A \text{ not possible using } \pi \text{ then} \\
\quad \quad \text{return } "P_i \text{ could not be proved terminating}";
\end{tabular}} \\
&\text{\begin{tabular}{l}
\quad else \\
\quad \quad A := \text{strengthening of } A \text{ using } \pi;
\end{tabular}} \\
&\text{\begin{tabular}{l}
\quad fi
\end{tabular}} \\
&\text{\begin{tabular}{l}
\quad foreach } j \in \{1, \ldots n\} - i \text{ do} \\
\quad \quad if } R_{l,j} \not\subseteq A \text{ then} \\
\quad \quad \quad A := \text{weakening using invariant for thread } j;
\end{tabular}} \\
&\text{\begin{tabular}{l}
\quad fi
\end{tabular}} \\
&\text{\begin{tabular}{l}
\quad endwhile
\end{tabular}} \\
&\text{return } "P_i \text{ terminates in } P";
\end{align*}
Example

\[ P_1 \]
lock(lk);
while * do
  assume(x > 0);
x := x - 1;
od
unlock(lk);

\[ P_2 \]
while * do
  lock(lk);
  assume(x > 0);
x := x - 1;
unlock(lk);
od

\[ P_3 \]
while * do
  lock(lk);
  x := *;
unlock(lk);

\[ A := true; \]
while \( P_i \mid A \) not provably terminating do
  \( \pi := \) lasso counterexample;
  if strengthening of \( A \) not possible using \( \pi \) then
    return "\( P_i \) could not be proved terminating";
  else
    \( A := \) strengthening of \( A \) using \( \pi \);
  fi
  foreach \( j \in \{1, \ldots, n\} - i \) do
    if \( R_{i,j} \not\in A \) then
      \( A := \) weakening using invariant for thread \( j \);
    fi
  od
return "\( P_i \) terminates in \( P \)";
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;
  od
unlock(lk);

\[ \mathcal{P}_3 \]
while * do
    lock(lk);
    x := *;
    unlock(lk);
  od

\[ A := \text{true}; \]
while \( \mathcal{P}_i \mid \Delta A \) not provably terminating do
    \( \pi := \text{lasso counterexample}; \)
    if strengthening of \( A \) not possible using \( \pi \) then
        return \( \mathcal{P}_i \) could not be proved terminating;\n    else
        \( A := \text{strengthening of } A \text{ using } \pi; \)
    fi
    foreach \( j \in \{1, \ldots, n\} - i \) do
        if \( \mathcal{R}_{I,j} \not\in A \) then
            \( A := \text{weakening using invariant for thread } j; \)
        fi
    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

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

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

$\Rightarrow$

$\mathcal{REACH}_i(P) \subseteq \mathcal{REACH}(P_i |\triangle A)$

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

Reachability Analysis:

\[
\text{Reach}(P_2 | \triangle \text{ true})
\]

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

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

\[
\text{while * do}
  \text{lock(lk);
  x := *;
  unlock(lk);
od}
```

\[
A := \text{true;}
\text{while } P_i | \triangle A \text{ not provably terminating do}
  \pi := \text{lasso counterexample;}
  \text{if strengthening of } A \text{ not possible using } \pi \text{ then}
    \text{return “}P_i\text{ could not be proved terminating”;}
```

\[
\forall j \in \{1, \ldots, n\} - \{i\}. \mathcal{R}_{I,j} \subseteq A \\
\Rightarrow \\
\text{Reach}_i(P) \subseteq \text{Reach}(P_i | \triangle A)
```

\[
\text{return “}P_i\text{ terminates in } P\text{”;}
```
Example

\( \text{Reach}(P_2 | \triangle \text{true}) \)

\( P_2 \)

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

\( P_3 \)

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

\( A := \text{true}; \)

- while \( P_i | \triangle A \) not provably terminating do
  - \( \pi := \text{lasso counterexample}; \)
  - if strengthening of \( A \) not possible using \( \pi \) then
    - return “\( P_i \) could not be proved terminating”;
  - else
    - \( A := \text{strengthening of } A \text{ using } \pi; \)
  - fi
- foreach \( j \in \{1, \ldots, n\} - i \) do
  - if \( R_{I,j} \not\in A \) then
    - \( A := \text{weakening using invariant for thread } j; \)
  - fi
- od
- return “\( P_i \) terminates in \( P \)”;

\( \triangle \)
Reach($P_2 \mid \triangle true$)

\begin{align*}
\text{lock}(lk) & ; \\
\text{while} \star & \text{do} \\
\text{lock}(lk) & ; \\
\text{assume}(x > 0) & ; \\
x := x - 1 & ; \\
\text{unlock}(lk) & ; \\
\text{odd} & \\
\text{unlock}(lk) & ; \\
\text{end} & \\
\end{align*}

$P_3$

\begin{align*}
\text{while} \star & \text{do} \\
\text{lock}(lk) & ; \\
x := * & ; \\
\text{unlock}(lk) & ; \\
\text{end} & \\
\end{align*}

$A := true$;
while $P_i \mid \triangle A$ not provably terminating do
\begin{align*}
\pi & := \text{lasso counterexample}; \\
\text{if} & \text{strengthening of } A \text{ not possible using } \pi \text{ then} \\
\text{return} & \text{"$P_i$ could not be proved terminating"}; \\
\text{else} & \\
A & := \text{strengthening of } A \text{ using } \pi; \\
\text{fi} \\
\text{fi} \\
\text{foreach} & j \in \{1, \ldots, n\} - i \text{ do} \\
\text{if} & \mathcal{R}_{i,j} \not\subseteq A \text{ then} \\
A & := \text{weakening using invariant for thread } j; \\
\text{fi} \\
\text{fi} \\
\text{return} & \text{"$P_i$ terminates in $P$"}; \\
\end{align*}
\( \delta \subseteq \geq_x \) ?

\[\begin{align*}
\mathcal{P}_2 & \quad \text{while } x \neq 0 \text{ do} \\
& \quad \text{lock}(l_k); \\
& \quad x := x - 1; \\
& \quad \text{assume}(x > 0); \\
& \quad x := x - 1; \\
& \quad \text{unlock}(l_k); \\
& \quad \text{od} \\
\mathcal{P}_3 & \quad \text{while } \star \text{ do} \\
& \quad \text{lock}(l_k); \\
& \quad x := \star; \\
& \quad \text{unlock}(l_k); \\
& \quad \text{od}
\end{align*}\]

\[\begin{align*}
A := \text{true}; \\
\text{while } \mathcal{P}_i |_A A \text{ not provably terminating do} \\
& \quad \pi := \text{lasso counterexample}; \\
& \quad \text{if strengthening of } A \text{ not possible using } \pi \text{ then} \\
& \quad \quad \text{return } "\mathcal{P}_i \text{ could not be proved terminating}"; \\
& \quad \text{else} \\
& \quad \quad A := \text{strengthening of } A \text{ using } \pi; \\
& \quad \quad \text{fi} \\
& \quad \text{fi} \\
& \quad \text{od}
\end{align*}\]

\[\begin{align*}
\text{foreach } j \in \{1, \ldots, n\} - i \text{ do} \\
& \quad \text{if } R_{i,j} \not\subseteq A \text{ then} \\
& \quad \quad A := \text{weakening using invariant for thread } j; \\
& \quad \quad \text{fi} \\
& \quad \text{od}
\end{align*}\]

\[\text{return } "\mathcal{P}_i \text{ terminates in } \mathcal{P}";\]
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

\[ \mathcal{A} = \text{true} \]
\[ \land \]
\[ \geq x \]
Example

\( P_1 \)
lock(\text{lk});
while * do
  assume(x > 0);
  x := x - 1;
od
unlock(\text{lk});

\( P_2 \)
while * do
  lock(\text{lk});
  assume(x > 0);
  x := x - 1;
  unlock(\text{lk});
od

\( P_3 \)
while * do
  lock(\text{lk});
  x := *;
  unlock(\text{lk});
od

\[ A = \text{true} \]
\[ \land \quad \geq_x \]
Example

\( \mathcal{P}_1 \)
\[
\text{lock}(lk); \\
\text{while } * \text{ do} \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\text{od} \\
\text{unlock}(lk); 
\]

\( \mathcal{P}_2 \)
\[
\text{while } * \text{ do} \\
\quad \text{lock}(lk); \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\quad \text{unlock}(lk); \\
\text{od} 
\]

\( \mathcal{P}_3 \)
\[
\text{while } * \text{ do} \\
\quad \text{lock}(lk); \\
\quad x := *; \\
\quad \text{unlock}(lk); \\
\text{od} 
\]
Example

$\mathcal{P}_1$
\[
\text{lock}(|k|); \\
\text{while * do} \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\od \\
\text{unlock}(|k|);
\]

$\mathcal{P}_2$
\[
\text{while * do} \\
\quad \text{lock}(|k|); \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\quad \text{unlock}(|k|); \\
\od
\]

$\mathcal{P}_3$
\[
\text{while * do} \\
\quad \text{lock}(|k|); \\
\quad \text{assume}(x > 0); \\
\quad x := *; \\
\quad \text{unlock}(|k|); \\
\od
\]

$\mathcal{A} = \text{true}$
\[
\land \quad \geq x
\]
Example

\( P_1 \)
lock(lk);
while * do
  assume(x > 0);
  x := x - 1;
  od
unlock(lk);

\( P_2 \)
while * do
  lock(lk);
  assume(x > u);
  x := x - 1;
unlock(lk);
  od

\[ A = true \]
\[ \land \geq_x \]

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

\( \delta : lk = 3 \)
Example

$P_1$
lock($lk$);
while * do
  assume($x > 0$);
  $x := x - 1$;
  od
unlock($lk$);

$P_2$
while * do
  lock($lk$);
  assume($x > u$);
  $x := x - 1$;
  unlock($lk$);
  od

$\delta : lk = 3$

$A = true$
\land \exists_x \lor lk = 3$
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 > u);
    x := x - 1;
    unlock(lk);
od

\[ \delta : lk = 3 \]

\[ \mathcal{A} = \text{true} \]
\[ \wedge \bigodot_x \vee lk = 3 \]
\[ A = \text{true} \]
\[ \land \biggr\uparrow_{x} \lor \text{lk} = 3 \]
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

$\mathcal{A} = \text{true} \land \exists x \forall lk = 3$

$\mathcal{A}$ := true;
while $\mathcal{P}_i \mathcal{A}$ not provably terminating do
  $\pi$ := lasso counterexample;
  if strengthening of $\mathcal{A}$ not possible using $\pi$ then
    return "$\mathcal{P}_i$ could not be proved terminating";
  else
    $\mathcal{A}$ := strengthening of $\mathcal{A}$ using $\pi$;
    fi
  foreach $j \in \{1, \ldots n\} - i$ do
    if $\mathcal{R}_{i,j} \not\subseteq \mathcal{A}$ then
      $\mathcal{A}$ := weakening using invariant for thread $j$;
    fi
  od
  return "$\mathcal{P}_i$ terminates in $\mathcal{P}$";
end
\( \mathcal{P}_1 \)

\[
\begin{align*}
\text{lock}(lk); \\
\text{while } * \text{ do} \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\text{od} \\
\text{unlock}(lk);
\end{align*}
\]

\( \mathcal{A} = \text{true} \)

\[\wedge \quad \exists x \quad \bigvee lk = 3\]

\( \mathcal{P}_2 \)

\[
\begin{align*}
\text{while } * \text{ do} \\
\quad \text{lock}(lk); \\
\quad \text{assume}(x > 0); \\
\quad x := x - 1; \\
\text{od} \\
\text{unlock}(lk);
\end{align*}
\]

\( \mathcal{P}_3 \)

\[
\begin{align*}
\text{while } * \text{ do} \\
\quad \text{lock}(lk); \\
\quad x := *; \\
\quad \text{unlock}(lk);
\end{align*}
\]

\( \mathcal{A} := \text{true}; \)

\[
\text{while } \mathcal{P}_i \not\triangleright \mathcal{A} \not\text{ provably terminating do} \\
\pi := \text{lasso counterexample}; \\
\text{if strengthening of } \mathcal{A} \text{ not possible using } \pi \text{ then} \\
\quad \text{return } \text{“} \mathcal{P}_i \text{ could not be proved terminating”;} \\
\text{else} \\
\quad \mathcal{A} := \text{strengthening of } \mathcal{A} \text{ using } \pi; \\
\text{fi} \\
\text{foreach } j \in \{1, \ldots, n\} - i \text{ do} \\
\quad \text{if } \mathcal{R}_{i,j} \not\subseteq \mathcal{A} \text{ then} \\
\quad \quad \mathcal{A} := \text{weakening using invariant for thread } j; \\
\quad \text{fi} \\
\text{od} \\
\text{return } \text{“} \mathcal{P}_i \text{ terminates in } \mathcal{P}”;
Example

$P_1$
lock(lk);
while * do
  assume($x > 0$):
  $x := x - 1$;
  od
unlock(lk);

$P_2$
while * do
  lock(lk);
  assume($x > 0$):
  $c := *$;
  while * do
    $c := c - 1$;
    assume($c > 0$);
    use( $'x \geq x \lor lk = 3$
              $\land 'lk = 1 \Rightarrow lk = 1$
              $\land 'lk \neq 1 \Rightarrow lk \neq 1$
            );
    od
  od
unlock(lk);
  if $\forall i \in [1..n]$ then
    $A :=$ weakening using invariant for thread $j$;
  fi
od
return "$P_i$ terminates in $P$";
Example

\[ P_1 \]
lock(lk);
while * do
    assume(x > 0);
    x := x - 1;
od
unlock(lk);

\[ P_2 \]
while * do
    lock(lk);
    assume(x > 0);
    x := x - 1;
    unlock(lk);
od

\[ P_3 \]
while * do
    lock(lk);
    x := *;
    unlock(lk);
od

\[ \mathcal{A} = \text{true} \]
\[ \land \bigg[ \Rightarrow_{x} \lor lk = 3 \bigg] \]

\[ \mathcal{A} := \text{true}; \]
while \( P_i \mid \triangle \mathcal{A} \) not provably terminating do
\[ \pi := \text{lasso counterexample}; \]
if strengthening of \( \mathcal{A} \) not possible using \( \pi \) then
    return "\( P_i \) could not be proved terminating";
else
    \( \mathcal{A} := \) strengthening of \( \mathcal{A} \) using \( \pi \);
fi
foreach \( j \in \{1, \ldots, n\} - i \) do
    if \( R_{i,j} \not\subseteq \mathcal{A} \) then
        \( \mathcal{A} := \) weakening using invariant for thread \( j \);
    fi
od
return "\( P_i \) terminates in \( P \)";

→ Fair termination

→ Data structures

→ Concurrency

→ Conclusion
Conclusion

- Basics: WF, ranking functions, disjunctive WF, decomposition, rank function synthesis

- Sequential arithmetic, non-recursive programs: refinement, 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

➡ 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

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

Thank you for your attention, questions