Proving that programs eventually do something good

Byron Cook
Collaborators

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

- Domagoj Babic,
- Josh Berdine,
- Aziem Chawdhary,
- Dino Distefano,
- Alexey Gotsman,
- Sumit Gulwani,
- Alan Hu,
- Samin Ishtiaq,
- Eric Koskinen,
- Tal Lev-Ami,
- Peter O’Hearn,
- Matthew Parkinson,
- Andreas Podelski,
- Zvonimir Rakameric,
- Andrey Rybalchenko,
- Mooly Sagiv,
- Moshe Vardi,
- Viktor Vafeiadis,
- Hongseok Yang,
- & the East London Massive.
In contrast to popular belief, proving termination is not always impossible.

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

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

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

This problem rose to prominence before the invention of the modern computer, in the era of Hilbert’s Entscheidungsproblem:* the challenge to formalize all of mathematics and use algorithmic means to determine the validity of all statements. In hopes of either solving Hilbert’s challenge, or showing it impossible, logicians began to search for possible instances of undecidable problems. Turing’s proof\(^1\) of termination’s undecidability is the most famous of those findings.\(^2\)

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

Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe we are always unable to prove termination, rather than more benign consequence that we are unable to always prove termination. Please like “but that’s like the termination problem” are often used to end discussions that might otherwise lead to viable partial solutions for real but undecidable problems. While we cannot ignore termination’s undecidability, if we develop a slightly modified problem statement we can build useful tools. In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. If the termination proving tool cannot prove or disprove termination, it should return “unknown.” Using only a finite amount of time, determine whether a given program will always finish running or could execute forever, or return the answer “unknown.”

\(^1\) In English, “decision problem.”

\(^2\) There is a minor controversy as to whether or not Turing proved the undecidability in.\(^2\) Technically he did it, but termination’s undecidability is an easy consequence of the result that is proved. A simple proof can be found in [here].\(^3\)

Key insights:

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

review articles

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

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

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

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

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

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

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

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

\(^a\) In English: “Decision problem.”

\(^b\) There is a minor controversy over whether or not Turing proved the undecidability of \(\mu\)-calculus. Technically it did not, but termination’s undecidability is an easy consequence of the result that is proved. A simple proof can be found in the literature.\(^b\)

key insights

1. For decades, the same method was used for proving termination. It has never been applied successfully to large programs.
2. A deep theorem in mathematical logic, based on Skolem’s theorem, holds the key to a new method.
3. A new method can scale to large programs because it allows for the modular construction of termination arguments.
Automatic formal verification

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

Build tools that find proofs of correctness using mathematics and logic

100% testing coverage
- Faster and more scalable than brute force
- Allows for 100% coverage even for infinite-state systems
“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”
“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”
“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”
“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”
“The mouse device driver’s event-handling routine always eventually terminates”
“The mouse device driver’s event-handling routine always eventually terminates”
“The mouse device driver’s event-handling routine always eventually terminates”
“The mouse device driver’s event-handling routine always eventually terminates”
“The mouse device driver’s event handling routine always eventually terminates.”
for (entry = DeviceExtension->ReadQueue.Flink;
    entry != &DeviceExtension->ReadQueue;
    entry = entry->Flink) {

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

        cldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp() was called
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (cldCancelRoutine) {
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }
    }

    else {
        // This IRP was just cancelled and the cancel routine was (or will
        // be) called. The cancel routine will complete this IRP as soon as
        // we drop the spinlock. So don't do anything with the IRP.
        //
        // Also, the cancel routine will try to dequeue the IRP, so make the
        // IRP's listEntry point to itself.
        //
        ASSERT (irp->Cancel);
        InitializeListHead (&irp->Tail.Overlay.ListEntry);
    }
}
for (entry = DeviceExtension->ReadQueue.Flink;
    entry != &DeviceExtension->ReadQueue;
    entry = entry->Flink) {

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp() was called
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }

        else {
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

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

      oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

→ Introduction

→ Termination basics

→ New advances for program termination proving
  ▪ Proving termination argument validity
  ▪ Finding termination arguments

→ Conclusion
Outline

→ Introduction

→ Termination basics

→ New advances for program termination proving
  ▪ Proving termination argument validity
  ▪ Finding termination arguments

→ Conclusion
Traditional termination proving method originally proposed by the forefathers of computing

E.g. Turing, “Checking a large routine”, 1949
Traditional termination proving method originally proposed by the forefathers of computing.

E. O. Elachi, 1949
Proving termination
Proving termination
Proving termination
Proving termination
Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In this case the non-negative integers \((n > n')\) satisfy \((n > n')\) which
Proving termination
Proving termination
Proving termination
Proving termination
Proving termination
$R \subseteq \geq f$
Proving termination

$$\trianglerighteq_f \triangleq \{ (s, t) \mid f(s) > f(t) \}$$
\[ R \subseteq \geq f \]
Outline

→ Introduction

→ Termination basics

→ New advances for program termination proving
  ▪ Proving termination argument validity
  ▪ Finding termination arguments

→ Conclusion
Outline

→ Introduction

→ Termination basics & history

→ New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments

→ Conclusion
Difficulties:

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

- Finding an \( f \) such that \( R \subseteq \geq_f \) is even harder in practice (and undecidable in theory)
Difficulties:

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

- Finding an $f$ such that $R \subseteq \geq f$ is even harder in practice (and undecidable in theory)
Automating the search for proofs

→ Transition relations must be computed

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

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

\[ U^*(I) \subseteq Q \]

→ \( Q \) represents an infinite set of states, but has a compact expression
Automating the search for proofs

Transition relations must be computed:

\[ R = U \cap [(U^*(I) \times \mathbb{L}) \cup \mathbb{H}] \]

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

\[ U^*(I) \subseteq Q \]

\( Q \) represents an infinite set of states, but has a compact expression.
Automating the search for proofs

- Transition relations must be computed

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

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

\[ U^*(I) \subseteq Q \]

- \( Q \) represents an infinite set of states, but has a compact expression
Automating the search for proofs

Transition relations must be computed.

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

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

\[ U^*(I) \subseteq Q \]

\( Q \) represents an infinite set of states, but has a compact expression.
Automating the search for proofs

We use an over-approximation of the transition relation

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

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

$$R' \subseteq \triangleright_f$$

Meaning: there might be unrealistic transitions that we have to worry about.
Automating the search for proofs

→ In practice, it's extremely hard to find the right overapproximation $Q$.

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

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

→ Tools like SLAM can be used to prove validity.
Difficulties:

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

- Finding an $f$ such that $R \subseteq \trianglerighteq f$ is even harder in practice (and undecidable in theory)
Automating the search for proofs

Difficulties:

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

- Finding an $f$ such that $R \subseteq \exists_f$ is even harder in practice (and undecidable in theory)
Automating the search for proofs

Difficulties:

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

- Finding an $f$ such that $R \subseteq \gtrless_f$ is even harder in practice (and undecidable in theory)
Automating the search for proofs

- Proving the inclusion $R^+ \subseteq \geq f \cup \geq g \cup \geq h$ is hard in practice (and undecidable in theory)

- Finding an $f$ such that $R \subseteq \geq f$ is even harder in practice (and undecidable in theory)
Automating the search for proofs

- Proving the inclusion $R^+ \subseteq \geq_f \cup \geq_g \cup \geq_h$ is hard in practice (and undecidable in theory)

- Finding an $f$ such that $R \subseteq \geq_f$ is even harder in practice (and undecidable in theory)
Automating the search for proofs

- Proving the inclusion $R^+ \subseteq \triangleright f \cup \triangleright g \cup \triangleright h$ is hard in practice (and undecidable in theory).
- Finding an $f$ such that $R \subseteq \triangleright f$ is even harder in practice (and undecidable in theory).
Automating the search for proofs

- Proving the inclusion $R^+ \subseteq \gtrdot_f \cup \gtrdot_g \cup \gtrdot_h$ is hard in practice (and undecidable in theory)

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

- Modularity gives us freedom when looking for valid arguments

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

\[ R^+ \subseteq \emptyset \]
Modular termination arguments

\[ R^+ \subseteq \emptyset \]
Modular termination arguments

$R^+ \subseteq \emptyset$
Modular termination arguments

\[ R^+ \supseteq \emptyset \]
Modular termination arguments

$\mathbb{R}^{+} \not\subseteq \emptyset$
Modular termination arguments
Modular termination arguments

$R^+ \not\subseteq \emptyset$

$R^+ \subseteq \geq f$
Modular termination arguments

\[ R^+ \not\subseteq \emptyset \]

\[ R^+ \subseteq \geq f \]
Modular termination arguments

\[ R^+ \not\leq \emptyset \]

\[ R^+ \not\leq \geq f \]
Modular termination arguments

\[ R^+ \subseteq \emptyset \]

\[ R^+ \subseteq \geq f \]
Modular termination arguments

\[ R^+ \subseteq \geq f \]

\[ g, g \]
$\mathbb{R}^+ \subseteq \geq f$
Modular termination arguments

\[ R^+ \supseteq \geq f \]

\[ R^+ \subseteq \geq f \cup \geq g \]
Modular termination arguments

\[ R^+ \subseteq [\geq f \cup \geq g] \]
Modular termination arguments

\[ R^+ \subseteq f \cup g \]

Diagram showing the relationship between \( R^+ \), \( f \), and \( g \).
Modular termination arguments

\[ R^+ \subseteq f \cup g \]

\[ h \quad h \]
Modular termination arguments

\[ R^+ \subseteq \geq_f \cup \geq_g \]

\[ R^+ \subseteq \geq_f \cup \geq_g \cup \geq_h \]
Modular termination arguments

\[ R^+ \subseteq \triangleright_f \cup \triangleright_g \cup \triangleright_h \]
Modular termination arguments

\[ R^+ \supseteq f \cup g \cup h \]
Difficulties:

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

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

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

- Finding an $f$ such that $R \subseteq \nRightarrow_f$ is even harder in practice (and undecidable in theory)
copied = 0;

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

    while (x < y) {
        x = f(x,y);
        if (!copied) {
            g(&y,x);
            if (*) {
                H[x] = x;
                H[y] = y;
                copied = 1;
            }
        }
    }
}

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

copied = 0;
copied = 0;
.
.
while (x < y) {
    if (!copied) {
        if (*) {
            H[x] = x;
            H[y] = y;
            copied = 1;
        }
    } else {
        assert(T_1 || T_2 || T_3);
    }
}

x = f(x, y);
g(&y, x);
\[ R^+ \subseteq T_1 \cup T_2 \cup T_3 \]

copied = 0;

\[
\text{while}(x<y) \{ \\
\text{if } (!\text{copied}) \{ \\
\quad \text{if } (*) \{ \\
\quad \quad H[x] = x; \\
\quad \quad H[y] = y; \\
\quad \quad \text{copied} = 1;
\quad \}
\}
\} \text{ else } \{ \\
\quad \text{assert}(T_1 \parallel T_2 \parallel T_3); \\
\}
\]

\[
x = f(x,y); \\
g(&y,x); \\
\}
copied = 0;

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

x = f(x,y);
g(&y,x);
copied = 0;
    .
    .
    .

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

x = f(x,y);
g(&y,x);
copied = 0;
  
  while (x<y) {
    if (!copied) {
      if (*) {
        H[x] = x;
        H[y] = y;
        copied = 1;
      }
    } else {
      assert(T_1 || T_2 || T_3);
    }
  }

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

```
1: void main()
2: {
3:     int x = nondet();
4:     int * p = &x;
5:     while(x<100) {
6:         (*p)++;  // This line is highlighted.
7:     }
8: }
9: }
```
Example

```
unsigned int Ack(unsigned int x, unsigned int y) {
    if (x > 0) {
        int n;
        if (y > 0) {
            y--;
            n = Ack(x, y);
            x--;
            return Ack(x, n);
        } else {
            n = 1;
        }
    } else {
        return y + 1;
    }
}
void main() {
    int x = nondet();
    int y = nondet();
    Ack(x, y);
```
```c
1: unsigned int Ack(unsigned int x, unsigned int y)
2: {
3:     int n;
4:     if (y>0) {
5:         y--;
6:         n = Ack(x, y);
7:     } else {
8:         n = 1;
9:     }
10:     x--;
11:     return Ack(x, n);
12: } else {
13:     return y+1;
14: }
15: }
16: void main()
17: {
18:     int x = nondet();
19:     int y = nondet();
20:     Ack(x, y);
```
```c
unsigned int Ack(unsigned int x, unsigned int y) {
    if (x > 0) {
        int n;
        if (y > 0) {
            y--;
            n = Ack(x, y);
        } else {
            n = 1;
        }
    } else {
        return Ack(x, n);
    }
    return y + 1;
}

void main() {
    int x = nonet();
    int y = nonet();
    Ack(x, y);
}
```
Examples
Examples

```c
void main()
{
    int x = nondet();
    int y = nondet();
    int z = nondet();

    while(x < 100 && 100 < z)
    {
        if (nondet()) {
            x++;
        } else {
            x--;
            z--;
        }
    }
}```
Examples
Examples

```c
void main() {
    int x = nondet();
    int y = nondet();
    if (y > 0) {
        while (x < 100) {
            x = x + y;
        }
    }
}
```
Abstract

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

Categories and Subject Descriptors: D.4.4 [Software]: Software Engineering—Program Verification; B.4.5 [Software]: Operating Systems—Reliability

General Terms: Reliability, Verification

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

1. Introduction

Reactive systems (e.g., operating systems, web servers, mail servers, database engines, etc.) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive services. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines inside the Windows serializer enumeration device driver. This code calls other serialized device drivers by passing one or more of the functions listed in Figure 1. Each of these drivers is then executed serially. The function in Figure 1 is called from several dispatch routines inside the Windows serializer enumeration device driver. This code calls other serialized device drivers by passing one or more of the functions listed in Figure 1. Each of these drivers is then executed serially.

request packet and PktData -> TopOfStack is the pointer to another device-based device driver. In the case where the device driver returns a return-value that indicates success, but places 0 in PktData->TopOfStack, the serial enumeration driver will fail to increment the value pointed to by PktData (line 65), possibly causing the driver to immediately exit this loop and not return to its calling context. The consequence of this error is that the computer’s serial device could become non-responsive. Worse yet, depending on what actions the device driver takes, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc.) at high priority and excessive physical I/O activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now, no automated termination tool has even been able to provide a capacity for large program fragments (>20,000 lines) together with accurate support for programming language features such as incrementally nested loops, pointers, function pointers, side-effects, etc. In this paper we describe a tool called TERMINATOR.

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

Furthermore, TERMINATOR does not require to construct any exact correct termination argument but rather a set of partials of possible arguments, some of which may be bad guesses. That is, this set need not be the exact set of the ‘right’ ranking functions but only a superset. We find the same insensitivity of the refinement of the termination argument as with intuitive abstraction refinement for safety (the set of predicates need not be the exact set of ‘right’ predicates but only a superset).

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

2006

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

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

General Terms Reliability, Verification

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

1. Introduction

Reactive systems (e.g., operating systems, web servers, mail servers, database engines, etc.) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device-driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing 1/0 request packets via the kernel pseudo-device driver. The request packet and PStrData (TopOfStack) is sent to another serial-based device driver. In the case where the device driver returns a return-value that indicates successful completion places 0 in PStrStatusBlock-InformationSerial enumeration driver will fail to increment the virtual pointer by actualizar (line 69), possibly causing the driver to instantly execute this loop and not return to its calling context. The consequence of this error is that the computer’s serial devices could become non-responsive. Worse yet, depending on what actions the other device drivers take, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc.) at high priority and excessive physical bus activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now, some termination tools have even been shown to provide a capacity for large program fragments (>20,000 lines) together with accurate support for programming language features such as arbitrarily nested loops, pointers, functions, pointers, side-effects, etc. In this paper we describe such a tool, called TERMINATOR.

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

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

Checking the termination argument is the hard part of our method. This is because the termination argument is now a set of running functions, not a single running function. With a single running function one must show that the rank decreases from the initial call to the target function, but for a single transition step. In our setting it is not sufficient to look at a single transition step. Instead, we must consider all finite sequences of transitions. We must show that for every sequence, one of the ranking functions decreases...
Termination Proofs for Systems Code

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Abstract

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

Categories and Subject Descriptors D.2.4 [Software]; D.2.5 [Software]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Reliability, Verification

1. Introduction

Reactive systems (e.g., operating systems, web servers, mail servers, database engines, etc.) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device-driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other I/O-based device drivers by passing 1/0 request packets via the local request protocol (see Section 5.2). Lptr is a pointer to the

---

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

<table>
<thead>
<tr>
<th>Driver</th>
<th>Run-time (seconds)</th>
<th>True bugs found</th>
<th>False bugs reported</th>
<th>Lines of code</th>
<th>Capping set size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1K</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1K</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>410</td>
<td>0</td>
<td>1</td>
<td>8K</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>1475</td>
<td>0</td>
<td>1</td>
<td>7.5K</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>123292</td>
<td>1</td>
<td>11</td>
<td>5.5K</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
<td>0</td>
<td>0</td>
<td>5K</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>4174</td>
<td>0</td>
<td>0</td>
<td>8K</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>210</td>
<td>0</td>
<td>0</td>
<td>8K</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>1294</td>
<td>0</td>
<td>0</td>
<td>6K</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>158</td>
<td>0</td>
<td>0</td>
<td>8K</td>
<td>21</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>2.5K</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>204</td>
<td>0</td>
<td>0</td>
<td>2.5K</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>257</td>
<td>1</td>
<td>1</td>
<td>7.5K</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1K</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>141</td>
<td>0</td>
<td>1</td>
<td>6.5K</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>1.5K</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>800</td>
<td>1</td>
<td>4</td>
<td>4.5K</td>
<td>35</td>
</tr>
<tr>
<td>18</td>
<td>1503</td>
<td>0</td>
<td>0</td>
<td>6.5K</td>
<td>31</td>
</tr>
<tr>
<td>19</td>
<td>209</td>
<td>0</td>
<td>0</td>
<td>3K</td>
<td>28</td>
</tr>
<tr>
<td>20</td>
<td>4099</td>
<td>0</td>
<td>0</td>
<td>10K</td>
<td>63</td>
</tr>
<tr>
<td>21</td>
<td>1461</td>
<td>1</td>
<td>4</td>
<td>10K</td>
<td>56</td>
</tr>
<tr>
<td>22</td>
<td>114762</td>
<td>0</td>
<td>5</td>
<td>3.4K</td>
<td>65</td>
</tr>
<tr>
<td>23</td>
<td>158746</td>
<td>2</td>
<td>10</td>
<td>35K</td>
<td>75</td>
</tr>
</tbody>
</table>

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

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Abstract

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

Categories and Subject Descriptors D.2.4 [Software]. Software Engineering—Program Verification, B.4.5 [Software]. Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Reliability, Verification

1. Introduction

Response systems (e.g., operating systems, web servers, mail servers, database engines, etc.) are constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing 0/1 request packet via the kernel routine locat_locallname(via 50, pprev) is a pointer to the request packet and PData.TcpOffStack is another net-based device driver). In the case where the device driver returns a return value that indicates success 0 in PData.TcpOffStack, information, the notification driver will fail to increment the return value to be (false 56), possibly causing the driver to infinitely execute and not return to its calling context. The consequence is that the computer's serial devices could become non-functional. Worse yet, depending on what actions the other device driver invokes, the loop may cause repeated acquiring and releasing resources (memory, locks, etc.) at high priority and extendable bus activity. This extra work stresses the operation of the other drivers, and the user applications running on which may cause them to crash or become non-responsive.

This example demonstrates how a session of terminal is the process of ensuring that reactive systems can act. Until now we assume termination tool has only a capacity for large program fragments (e.g., 72) together with accurate support for program-language features such as conditionally nested loops, points-to, function return effects, etc. In this paper we describe a tool, called TOE.

3. TERMINATOR’s most distinguishing aspect is the new methods and tools for proving program termination shift the balance between the two tasks of construction and checking the termination argument. The goal is to construct an expression defining the time that the state machine will run, that is, in value decreases in any transition from state to a state. The construction of the machine that produces the task that is needed to be applied by the program. The checking part is relatively easy. In our task of constructing running functions is the selection of a value that is lower than any other running function. Furthermore, TERMINATOR is not required to cut off correct termination argument but rather a set of possible arguments, some of which may be bad guesses. This set must not be the exact set of the ‘right’ running function only a super set. We find the same insensitivity of the termination argument with iterative value for safety (the size of a predicate used to be the exact predicates but not only a super predicate).

Checking the termination argument is the hard task. This is because the termination argument of running functions, an exact running function. At running function one must show that the task decreases to post-state after making some single transition setting it is sufficient to look at a single transition. In some cases, we must consider all finite sequences of transitions. We that, for every sequence, one of the running function

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

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Abstract
Program termination is central to the process of ensuring that systems can always react. We describe and analyze a program termination
proof that performs a semi-sensitive and context-sensitive program analysis and provides capacity for large program fragments
(i.e., more than 30,000 lines of code) together with support for program
language features such as arbitrarily nested loops, pointers, function pointers, sub-programs, etc. We also present experimental
results on device driver dispatch routines from the Windows operating
system. The most distinguishing aspect of our tool is how it
shifts the balance between the two tasks of constructing and
respectively checking the termination argument. Checking becomes
the hard step. In this paper we show how we solve the corresponding
challenge of checking with binary reachability analysis.

Categories and Subject Descriptors D.4.4 [Software, Software
Engineering]—Program Verification, D.4.6 [Software, Operating
System—Reliability]

General Terms Reliability, Verification

Keywords Reliability, Verification

1. Introduction
Reactive systems (e.g., operating systems, web servers, mail servers,
database engines, etc.) are generally constructed from a set of
components that we expect will always terminate. Cases where
these functions unexpectedly do not return to their calling context lead
to non-responsive systems. Device driver dispatch routines, for
example, must eventually return to their caller. Consider the function
defined in Figure 1 which is called from several dispatch routines within
the Windows serial enumeration device driver. This code calls other
serial-based device drivers by passing 1/0 request packets via the
serial loopback device driver (Note 50). A request is to the device
request packet and PciData */topOfStack is another serial-based device driver). In the case where a
device driver returns a return value that indicates success
in PciStatusBlock *information the next
driver will fail to increment the mso value as it should (line 65), possibly causing the driver to
indefinitely execute and not return to its calling context. The consequence
is that the computer’s serial devices could become non-
responding, even if, depending on what actions the other device
in this loop may cause repeated acquiring and releasing
resources (memory, locks, etc.) at high priority and critical
bus activity. This extra work stress the operating
the other drivers, and the user applications running on
which may cause them to crash or become non-responsive.

This example demonstrates how a notion of termination
to the process of ensuring that reactive systems can
act. Until now no automatic termination tool has ever
provided a capacity for large program fragments (>2
lines) together with accurate support for program
language features such as arbitrarily nested loops, pointers,
function pointers, sub-programs, etc. In this paper we describe such a tool, called T3E.

TERMINATOR’s most distinguishing aspect, with
respect to existing tools for program termination,
is how it shifts the balance between the two tasks of constructing and
respectively checking the termination argument. Checking
becomes the hard step. In this paper we show how we solve the corresponding
challenge of checking with binary reachability analysis.

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[31] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.
Termination Proofs for Systems Code

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Abstract

Program termination is central to the process of ensuring that systems code can always terminate. We describe a new program termination

prove tool that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments

(> 20.000 lines of code) together with support for program

language features such as arbitrarily nested loops, pointers, function pointers, side effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating

system. The most distinguishing aspect of our tool is how it

shifts the balance between the two tasks of constructing and respectively checking the termination argument. Checking becomes

the hard step. In this paper we show how we solve the corresponding challenge of checking with binary reachability analysis.

Categories and Subject Descriptors D.2.4 [Software]; Software Engineering—Program Verification, D.2.5 [Software]; Operating System—Reliability

General Terms Reliability, Verification

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

1. Introduction

Reactive systems (e.g., operating systems, web servers, mail servers, database engines, etc.) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device-driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine kevent_block_dispatch (lines 50–52). pnp is a pointer to the request packet and PmData->TopOfStack is another serial-based device driver). In the case where device driver returns a return-value that indicates success, 0 in PmStatusBlock->Information, the termination

driver will fail to increment the usage count and thereby be (lines 65), possibly causing the driver to be in a situation and not return to their calling context. The consequence is that the computer's serial devices could become non-operational. Worse yet, depending on what actions the other device drivers perform, this may cause repeated acquiring and releasing resources (memory, locks, etc.) at high priority and exotic bus activity. This extra work stresses the entire system, and the user applications running on it may cause them to crash or become non-responsive.

This example demonstrates how a method to the process of ensuring that reactive systems can be

used. Until now, no automatic termination tool has been available to provide a capacity for large program fragments (<2000) together with accurate support for programming languages such as arbitrarily nested loops, pointers, function pointers, side effects, etc. In this paper we describe such a tool, called TDR.

TDR TERMINATOR's most distinguishing aspect, with respect to previous methods and tools for proving program termination, is that it shifts the balance between the two tasks of constructing and checking the termination argument. The casual construction of an expression defining the next state transition that it decreases is not the exact set of the 'right' state transition only a subset of it. We find the issue unavoidable and the termination argument as with iterative constriction for safety (the set of predicates must be the exact predicates but only a subset).

Checking the termination argument is the hard

method. This is because the termination argument is the result of ranking functions, not a single ranking function. A ranking function must show that the rank decreases at every step after assuming that it satisfies the single transition setting. It is not sufficient to look at a single transition; we must consider all finite sequences of transitions. We therefore, for each sequence, one of the ranking function

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

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Abstract
Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination technique that performs a partitioning-sensitive and context-sensitive program analysis as well as provides capacity for large program fragments (i.e., more than 20,000 lines of code) together with support for program analysis frameworks such as simulated loops, pointer, function-pointer, and variables, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing features of our tool are its ability to shift the balance between the two tasks of constructing and respectively checking the termination algorithm. Checking becomes the hard part. In this paper we show how we solve the corresponding request packet and PiData-TopOfStack is another sensitivity-based device driver). In the case we device driver returns a return-value that indicates a 0 in PiStatusFlie -imensional...
Send in the Terminator

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

Alan Turing, the mathematician who was among the founders of computer science, showed in 1956 that it is impossible to devise an algorithm to prove that any given program will always run to completion. The essence of his argument was that such an algorithm can always trip up if it analyzes itself and finds that it is unable to stop. "It leads to a logical paradox," remarks David Schalk, professor of computer science at Kansas State University. On a pragmatic level, the inability to "terminate," as it is called in computers, is familiar to any user of the Windows operating system who has clicked a mouse button and then stared indefinitely at the hourglass icon indicating that the program is looping endlessly through the same lines of code.

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

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

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

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

Patrick Cousot of the Ecole Normale Superieure in Paris, a pioneer in mathematical program analysis, notes that Terminator should work for a limited set of well-defined applications. "I doubt, for example, that Terminator is able to handle mathematically hard termination problems"—those for floating-point numbers—"nor programs that ran at the same time. Cook does not disagree, saying that the plans to develop termination proof methods for such programs. Finding a way to ensure that more complex programs don't freeze is such a difficult challenge, however, that Cook thinks it could consume the rest of his career.

---

**Computer Entomophobia**

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

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

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max Planck Institute for Informatics
podelski@mpi-inf.mpg.de

Abstract
Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination technique that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e., more than 28,000 lines of code) together with support for programming language features such as infinitely nested loops, pointers, function pointers, side effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of constructing and respectively checking the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of checking with binary reachability analysis.

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

General Terms: Reliability, Verification

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

1. Introduction

Reactive systems (e.g., operating systems, web servers, mail servers, database engines, etc.) are usually constructed from components that we will always terminate. Cases where these systems do not return to their calling context lead to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial transmission device driver. This code calls other serial-based device drivers by using I/O request packets via the kernel routine system_127_set

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine1, Byron Cook1, Dino Davidescu, and Peter W. O'Hearn1,2

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 configuration 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 depth 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 proven terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

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

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

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

3 Although hanging kernel-threads can trigger other bugs within the operating system.
Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jberdine@microsoft.com

Azem Chowdhary
Queen Mary, University of London
azem@cs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Dino Distefano
Queen Mary, University of London
distefano@cs.qmul.ac.uk

Peter O’Hearn
Queen Mary, University of London
peter@cs.qmul.ac.uk

Abstract

An invariance assertion for a program location \( l \) is a statement that always holds at \( l \) during execution of the program. Program invariance analyses infer invariance assertions that can be used when trying to prove safety properties. We use the term \( \text{variance analysis} \) to mean a statement that holds between any state at \( l \) and any previous state that was also at \( l \). This paper is concerned with the development of analyses for variance assertions and their applications to proving termination and liveness properties. We describe a method of constructing program variance analysis from invariance analyses. If we change the underlying invariance analysis, we get a different variance analysis. We describe several applications of the method, including variance analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variance analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today’s state-of-the-art termination provers.

Categories and Subject Descriptors D.4.2 [Software Engineering]: Software/Program Verification; F.1.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

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

1. Introduction

An invariance analysis takes in a program as input and infers a set of possibly disjunctive invariance assertions, i.e., invariants, that hold in program locations. Each location \( l \) in the program has an invariant that always holds during any execution at \( l \). These invariants can serve many purposes. They might be used directly to prove safety properties of programs. Or they might be used indirectly, for example, to aid the construction of abstract transition relations during symbolic program model checking [29]. Since a desired safety property is often directly provable from a given invariant, the user (or algorithm calling the invariance analysis) might try to refine the abstraction. For example, if the tool is based on abstract interpretation, it may then choose to improve the abstraction by delaying the widening operation [28] using dynamic partitioning [21], employing a different abstract domain, etc.

The aim of this paper is to develop an analogous set of tools for program termination and liveness: we introduce a class of tools called \( \text{variance analyses} \), which infer assertions, called \( \text{variance assertions} \), that hold between any state at \( l \) and any previous state that was also at \( l \). Note that a simple variance assertion may itself be a disjunction. We present a generic method of constructing variance analyses from invariance analyses. For each invariance analysis, we can construct what we call an \( \text{induced variance analysis} \).

This paper also introduces a condition on variance assertions called the \( \text{local termination predicate} \). In this work, we show how the variance assertions inferred during our analysis can be used to establish local termination predicates. If this predicate can be established for each variance assertion inferred for a program, the whole program termination has been proved; the correctness of this step relies on a result from [17] on the use of disjunctive well-founded order approximations. Analogously to invariance analysis, even if the induced variance analysis fails to prove whole program termination, it can still produce useful information. If the predicate can be established only for some subset of the variance assertions, this induces a different invariants property that holds of the program. Moreover, the information inferred can be used by other termination provers based on disjunctive well-foundedness, such as \text{T E R M I N A T O R} [14]. If the underlying invariance analysis is based on abstract interpretation, the user or algorithm could use the same abstraction refinement techniques that are available for invariance analyses.

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

Byron Cook
Microsoft Research
bycook@microsoft.com

Abstract
An invariant assertion for a program location $l$ is a statement that always holds at $l$ during execution of the program. Program invariant analysis infers invariant assertions that can be useful when trying to prove safety properties. We use the term variation assertion to mean a statement that holds between any state at $l$ and any previous state that was also at $l$. This paper is concerned with the development of analyses for variation assertions and their application to proving termination and liveliness properties. We describe a method of constructing program variation analysis from invariant analyses. If we change the underlying invariant analysis, we get a different variation analysis. We describe several applications of this method, including variation analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variation analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today’s state-of-the-art termination provers.

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jberdine@microsoft.com

Azem Chawdhry
Queen Mary, University of London
azem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Peter O’Hearn
Queen Mary, University of London
pohearn@dcs.qmul.ac.uk

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

Abstract
An invariant assertion for a program location $l$ is a statement that always holds at $l$ during execution of the program. Program invariant analysis infers invariant assertions that can be useful when trying to prove safety properties. We use the term variation assertion to mean a statement that holds between any state at $l$ and any previous state that was also at $l$. This paper is concerned with the development of analyses for variation assertions and their application to proving termination and liveliness properties. We describe a method of constructing program variation analysis from invariant analyses. If we change the underlying invariant analysis, we get a different variation analysis. We describe several applications of this method, including variation analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variation analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today’s state-of-the-art termination provers.

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsmann
University of Godiva
alexey.gotsmann@cam.ac.uk

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

Moshie Y Vardi
Purdue University
vardi@cs.purdue.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 had provided fully automatic support for proving properties that ensure that good events eventually happen (liveness properties). In this paper we present 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.

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O’Hearn

1 Microsoft Research
2 Queen Mary, University of London
bycook@microsoft.com

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 had provided fully automatic support for proving properties that ensure that good events eventually happen (liveness properties). In this paper we present 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.

Keywords Formal Verification, Software Model Checking, Liveness, Termination

1. Introduction
An invariant assertion for a program location $l$ is a statement that always holds at $l$ during execution of the program. Program invariant analysis infers invariant assertions that can be useful when trying to prove safety properties. We use the term variation assertion to mean a statement that holds between any state at $l$ and any previous state that was also at $l$. This paper is concerned with the development of analyses for variation assertions and their application to proving termination and liveliness properties. We describe a method of constructing program variation analysis from invariant analyses. If we change the underlying invariant analysis, we get a different variation analysis. We describe several applications of this method, including variation analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variation analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today’s state-of-the-art termination provers.

Windows kernel APIs that acquire resources and APIs that release resources may. For example, a counterexample to the property in a program trace in which KcsSpinLockSpin is called but not followed by a call to KcsSpinLockInSpin is called. Note that KcsSpinLockSpin will eventually be called in the case that a call to KcsSpinLockPinSpin occurs. Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions $f_0$, $g_0$, $f_1$, $g_1$, etc. It is easy to prove that the function is always called before $h$, in this case we need only look at the structure of the control-flow graph. It is much harder to prove that $h$ is called after $f_0$ has been called, or that $h$ is called after $f_1$ has been called, etc.

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace in which KcsSpinLockSpin is called but not followed by a call to KcsSpinLockInSpin. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will happen eventually. For example, that the kernel will eventually lock a semaphore. Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions $f_0$, $g_0$, $f_1$, $g_1$, etc. It is easy to prove that the function is always called before $h$, in this case we need only look at the structure of the control-flow graph. It is much harder to prove that $h$ is called after $f_0$ has been called, or that $h$ is called after $f_1$ has been called, etc.
Termination Proofs for Systems Code

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jberdine@microsoft.com

Azem Chowdhary
Queen Mary, University of London
azem@cs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Proving Termination by Divergence

Donnaroj Babirić, Alan J. Hu, Zvonimir Rakamarić
Department of Computer Science, University of British Columbia
{babiri,ahz,rakamaric}@cs.ubc.ca

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

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

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

This paper outlines a new proof procedure for cases of this sort. Using comparison techniques described in [1] and [2], our intention for this proposed procedure is to combine the existing termination analysis techniques—making future termination proofs a little less tempestual.

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

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine1, Byron Cook1, Dino Distefano2, and Peter W. O'Hearn1,2
1 Microsoft Research
2 Queen Mary, University of London

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

Andrey Rybakchenko
EPFL and MPI-Saarbrücken
ryba@epfl.ch

Moshe Z. 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 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 correct liveliness properties of Windows device drivers and found several previously unknown liveliness bugs.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software Development Tools - Program Verification, F.3.1 [Logics and Model Theories of Programs]: Specifying and Verifying Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Liveness, Termination

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

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

A driver should never call KernelCreateSpinlock unless it has already called KernelAcquireSpinlock. This is a safety property for the reason that a 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 KernelCreateSpinlock before calling KernelAcquireSpinlock). Note that SDV cannot check the equality of two related liveliness properties. If a driver calls KernelAcquireSpinlock then it must eventually make a call to KernelReleaseSpinlock.

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

Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: 

10; g(0); n(); k(). It is easy to prove that the function k is always called before h: in this case we only need to look at the structure of the control-flow graph. It is much harder to prove that k is eventually called after we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveliness property. Unfortunately, to practitioners, liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team.
1 Introduction

From the very beginnings of the formal analysis of programs in the 1950s, the task of formally verifying the correctness of a program has been decomposed into the tasks of proving termination and proving termination by divergence. Deciding termination, in general, is obviously undecidable, but thanks to considerable progress over the years (e.g., [9, 20, 5, 23, 6, 18, 21, 8]), a variety of techniques and heuristics have been developed to automatically prove termination of many loops in practice.
Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jberdine@mit.edu

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

Byron Cook
Microsoft Research
bycook@microsoft.com

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine1, Byron Cook1, Dino Distefano2, and Peter W. O'Hearn1,2
1 Microsoft Research
2 Queen Mary, University of London

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

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

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

Andrey Rybalchenko
EPFL and MPI-Saarbrücken

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

Ranking Abstractions

Aziem Chawdhary1, Byron Cook2, Sumit Gulwani2, Moshav Sagiv3, and Hongseok Yang2
1 Queen Mary, University of London
2 Microsoft Research
3 Tel Aviv University

Abstract. We propose an abstract interpretation algorithm for proving that a program terminates on all inputs. The algorithm uses a novel abstract domain which uses ranking relations to conservatively represent relations between intermediate program states. One of the attractive aspects of the algorithm is that it
Termination Proofs for Systems Code

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jb@microsoft.com

Azim Chawdhary
Queen Mary, University of London
azim@cs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine¹, Byron Cook¹, Dino Distefano¹, and Peter W. O'Hearn²

¹ Microsoft Research
² Queen Mary, University of London

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

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

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

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

Proving Termination by Divergence

Donnega Babić, Alan J. Hu, and Byron Cook
Department of Computer Science, University of California, Berkeley

Abstract

We describe a simple and efficient algorithm for proving the termination of a class of loops with nonlinear term assignments to variables. The method is based on divergences and we present a tool that works for each variable in the case of influence of all variables.

Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

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

Proving Conditional Termination

Byron Cook¹, Sumit Gulwani¹, Tal Lev-Ami²,³,⁴,
Andrey Rybalchenko⁵,⁶, and Moody Sayig⁷

¹ Microsoft Research
² Tel Aviv University
³ MPS-SWS

Abstract. We describe a method for synthesizing reasonable under-approximations of the set of all execution paths of a conditional loop.

Ranking Abstractions

Azim Chawdhary¹, Byron Cook², Sumit Gulwani², Moody Sagiv³, and Hongseok Yang⁴

¹ Queen Mary, University of London
² Microsoft Research
³ Tel Aviv University
⁴ TechX Labs

Abstract. We propose an abstract interpretation algorithm for proving that a program terminates on all inputs. The algorithm uses a novel abstract domain which uses ranking relations to conservatively validate relations between intermediate program states. One of the attractive aspects of the algorithm is that it unifies...
Temporal property verification as a program analysis task

Byron Cook¹, Erin Kuskima², and Moshe Vardi²

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

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

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g., [2,5,8,41,22]) together with techniques for discovering termination arguments (e.g., [1,9,21]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

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

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

Byron Cook1, Eric Koskinen2, and Mohso Vardis3

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

Abstract: We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, can be used as a program analysis tool to naturally reason about properties of programs. Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of programs (possibly infinite-state) transition systems. We observe that, with suitable use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction, backtracking, event checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are naturally performed by the program analysis tool. Using known safety analysis tools [2, 5, 8, 24, 32] along with techniques for discovering termination arguments [13, 15, 17], we can implement temporal logic prover whose power is essentially limited only by the power of the encoding technique.

Board on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

General Terms: Verification, Theory, Reliability

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

1. Introduction

The common wisdom amongst users and developers of tools that prove temporal properties of systems that have specific logic LTL [13, 15, 17] is more restrictive than CTL [15], but that properties expressible in the universal fragment of CTL (VCTL) with our formalization are often easier to prove than their LTL cousins.

1. Properties expressed in CTL without fairness can be proved in a purely syntactically-directed manner using state-based renaming techniques, whereas LTL requires deeper reasoning about sets of traces and the model relationships between families of them.

In this paper we aim to make an LTL prover for infinite-state systems with performance close to what one would expect from a C program. We use the observation that CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don’t always fit together; these cases arise when, due to some instances of nondeterminism in the transition system, the model of an LTL prover (e.g., C program) that we try to prove cannot be trusted.

Abstract

We propose an abstract version of an LTL prover for infinite-state systems with performance close to what one would expect from a C program. We use the observation that CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don’t always fit together: these cases arise when, due to some instances of nondeterminism in the transition system, the model of an LTL prover (e.g., C program) that we try to prove cannot be trusted.

In this paper we aim to make an LTL prover for infinite-state systems with performance close to what one would expect from a C program. We use the observation that CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don’t always fit together: these cases arise when, due to some instances of nondeterminism in the transition system, the model of an LTL prover (e.g., C program) that we try to prove cannot be trusted.

In this paper we aim to make an LTL prover for infinite-state systems with performance close to what one would expect from a C program. We use the observation that CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don’t always fit together: these cases arise when, due to some instances of nondeterminism in the transition system, the model of an LTL prover (e.g., C program) that we try to prove cannot be trusted.

In this paper we aim to make an LTL prover for infinite-state systems with performance close to what one would expect from a C program. We use the observation that CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don’t always fit together: these cases arise when, due to some instances of nondeterminism in the transition system, the model of an LTL prover (e.g., C program) that we try to prove cannot be trusted.

In this paper we aim to make an LTL prover for infinite-state systems with performance close to what one would expect from a C program. We use the observation that CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don’t always fit together: these cases arise when, due to some instances of nondeterminism in the transition system, the model of an LTL prover (e.g., C program) that we try to prove cannot be trusted.
Termination Proofs for Systems Composed of Programs

Byron Cook
Microsoft Research
bycook@microsoft.com

Abstract. We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with suitable use of recursion and non-determinism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction, search, backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g., [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (e.g., [1, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying program analysis.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with suitable use of recursion and non-determinism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction, search, backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g., [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (e.g., [1, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying program analysis.

2 Abstract

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with suitable use of recursion and non-determinism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction, search, backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g., [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (e.g., [1, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying program analysis.

3 Abstract

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with suitable use of recursion and non-determinism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction, search, backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g., [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (e.g., [1, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying program analysis.

4 Abstract

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with suitable use of recursion and non-determinism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (e.g., abstraction, search, backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (e.g., [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (e.g., [1, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying program analysis.
Termination Proofs for Systems Code

Byron Cook
Microsoft Research
bycook@microsoft.com

Proving stabilization for biological systems

Byron Cook1,2, Jasmin Fisher3, Elżbieta Krepela1,3, and Nir Piterman4

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

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

1 Introduction

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

119

In this paper we address the open challenge to find scalable algorithms for proving stabilization of biological systems. In computer science terms, we are trying to prove a liveliness property similar to termination of large parallel systems. The sizes of these systems force us to use some form of modular reasoning.

Unfortunately, because stabilization is a liveliness property, we must be careful when using the more powerful cyclic modal proof rules (e.g. [9,10]), as they are formally only sound in the context of safety [11]. Furthermore, we find that the complex temporal interactions between the modules are crucial to the stabilization of the system as a whole; meaning that we cannot use scalable techniques that simply abstract away the interactions altogether.

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

\[ \text{FG}(p_1) \land \ldots \land \text{FG}(p_n) \Rightarrow \text{FG}(q) \]

1 Non Predicates

\[ d \]

/ki u ng

\[ \text{spike lock} \]

on: No 2

Proving That Non-Blocking Algorithms Don't Block

Byron Cook · Andreas Podelski · Andrey Rybalchenko
Misunderstanding the halting problem

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

- Termination bugs found from failed proof attempts are usually more entertaining
for (entry = DeviceExtension->ReadQueue.Flink;
    entry != &DeviceExtension->ReadQueue;
    entry = entry->Flink) {

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp()
        // (or nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }

    } else {
        // This IRP was just cancelled and the cancel routine was (or will
        // be) called. The cancel routine will complete this IRP as soon as
        // we drop the spinlock. So don't do anything with the IRP.
        //
        // Also, the cancel routine will try to dequeue the IRP, so make the
        // IRP's listEntry point to itself.
        //
        ASSERT (irp->Cancel);
        InitializeListHead (&irp->Tail.Overlay.ListEntry);
    }
}
for (entry = DeviceExtension->ReadQueue.Flink;
    entry != &DeviceExtension->ReadQueue;
    entry = entry->Flink) {

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp()
        // (ie. nextIrp->Cancel is set), but whether IoCancelIrp()
        // is about to call (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //

        if (oldCancelRoutine) {
            //
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }
    } else {
        //
        // This IRP was just cancelled and the cancel routine was (or will
        // be) called. The cancel routine will complete this IRP as soon as
        // we drop the spinlock. So don't do anything with the IRP.
        //
        // Also, the cancel routine will try to dequeue the IRP, so make the
        // IRP's listEntry point to itself.
        //
        ASSERT (irp->Cancel);
        InitializeListHead (&irp->Tail.Overlay.ListEntry);
    }
}
for (entry = DeviceExtension->ReadQueue.Flink; entry != &DeviceExtension->ReadQueue; entry = entry->Flink) {

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp()
        // (ie. nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }
    }
    else {

        // This IRP was just cancelled and the cancel routine was (or will
        // be) called. The cancel routine will complete this IRP as soon as
        // we drop the spinlock. So don't do anything with the IRP.
        //
        // Also, the cancel routine will try to dequeue the IRP, so make the
        // IRP's listEntry point to itself.
        //
        ASSERT (irp->Cancel);
        InitializeListHead (irp->Tail.Overlay.ListEntry);
    }
}
Misunderstanding the halting problem

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        // IoCancelIrp() could have just been called on this IRP,
        // but whether IoCancelIrp() called (or is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            // Cancel routine not called for this IRP. Return this IRP.
            return irp;
        }

    } else {
        // This IRP was just cancelled and the cancel routine was (or will
        // be) called. The cancel routine will complete this IRP as soon as
        // we drop the spinlock. So don't do anything with the IRP.
        //
        // Also, the cancel routine will try to dequeue the IRP, so make the
        // IRP's listEntry point to itself.
        //
        ASSERT (irp->Cancel);
        InitializeListHead (&irp->Tail.Overlay.ListEntry);
    }
}
```
for (entry = DeviceExtension->ReadQueue.Flink;
    entry != &DeviceExtension->ReadQueue;
    entry = entry->Flink) {

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp
        // (ie. nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }
    }
}
else {

    //
    // This IRP was just cancelled and the cancel routine was (or will
    // be) called. The cancel routine will complete this IRP as soon as
    // we drop the spinlock. So don't do anything with the IRP.
    //
    // Also, the cancel routine will try to dequeue the IRP, so make the
    // IRP's listEntry point to itself.
    //
    ASSERT (irp->Cancel);
    InitializeListHead (&irp->Tail.Overlay.ListEntry);
}
for (entry = DeviceExtension->ReadQueue.Flink;
    entry != &DeviceExtension->ReadQueue;
    entry = entry->Flink) {

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

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

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

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            // Cancel routine not called for this IRP. Return this IRP.
            //
            return irp;
        }
    }
    else {
        //
        // This IRP was just cancelled and the cancel routine was (or will
        // be) called. The cancel routine will complete this IRP as soon as
        // we drop the spinlock. So don't do anything with the IRP.
        //
        // Also, the cancel routine will try to dequeue the IRP, so make the
        // IRP's listEntry point to itself.
        //
        ASSERT (irp->Cancel);
        InitializeListHead (&irp->Tail.Overlay.ListEntry);
    }
}
Still solving the halting problem I see. If you ever find spare time, you might also want to give a go at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

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

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

For extra credit, notice the O(n^2*m) condition created by the invocation by MouseClassCleanupQueue, where n is the number of non-FO matching objects in the beginning of the queue and m is the number of matching ones. DOS attack anyone?

- A

-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver
Still solving the halting problem I see. If you ever find spare time, you might also want to give a go at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

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

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

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

- A

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

Now to your actual question :)

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

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

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

- A

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

→ Introduction

→ Termination basics & history

→ New advances for program termination proving
  ▪ Proving termination argument validity
  ▪ Finding termination arguments

→ Conclusion
Outline

→ Introduction

→ Termination basics & history

→ New advances for program termination proving
  ▪ Proving termination argument validity
  ▪ Finding termination arguments

→ Conclusion
Future work

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

→ Now people are beginning to think that it’s effectively “solved”.

→ Much left to do, including
  - Complex data structures (safety)
  - Infinite-state systems w/ bit vectors (safety)
  - Binaries (safety)
  - Non-linear systems (liveness and safety)
  - Better support for concurrent programs
  - Modern programming features (e.g. closures)
  - Finding preconditions to termination
  - Scalability, performance, precision
Termination proving is at the heart of many undecidable problems (e.g. Wang’s tiling problem)

Modern termination proving techniques could potentially be used to building working tools

Challenge: “black-box” solutions to undecidable problems die in the most unpredictable ways
Conclusion

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

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

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

- http://research.microsoft.com/terminator
  - Research papers
  - Recorded technical lectures
  - Contact details

- CACM review article