# KISS: Keep it Simple and Sequential

## A tool for finding bugs in concurrent programs

Steffen Juilf Smolka

Technische Universität München

17. Juli 2012

# Overview

1. Motivation and Intuition behind KISS
2. The Transformation
3. Analysis and Conclusion

# Motivation

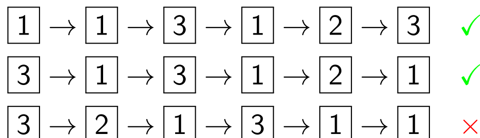
- Gordan E. Moore, Intel Co-Founder, 1965
- "The Free Lunch Is Over", Herb Sutter, 2005
- increasing number of transistors now manifests itself in an increasing number of cores
- performant software relies on concurrency

# Challenges of Concurrent Programs

- Difficult to get right
  - access to shared variables has to be synchronized
  - sometimes counter-intuitive

# Challenges of Concurrent Programs

- Difficult to get right
  - access to shared variables has to be synchronized
  - sometimes counter-intuitive
- Difficult to debug
  - nondeterministic in nature
  - combinatorical explosion in number of possible schedules
  - "Heisenbugs"




$\Rightarrow$ Testing is no longer an effective method for finding bugs

$\Rightarrow$ Tools for finding bugs are invaluable

# Traditional Approach

- Define safety properties
  - assertions
  - reachability, race conditions
- Explore all possible thread interleavings
  - forbidding complexity, exponential in # threads
  - Problem is undecidable [Ramalingam]

# Example

### Main Thread

```
// global variables
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {
  driver = init();
  is_running = true;
  is_in_use = false;
  async(thrA);
  async(thrB);
}
```

### Thread A

```
void thrA() {
  if(is_running){
  is_in_use = true;

  // use driver ...
     ⋮

  is_in_use = false;
  }
}
```

### Thread B

```
void thrB() {
  assume(!is_in_use);
  is_running = false;

  // clean up ...
  free(driver);
     ⋮

}
```

What assumptions are we making?

# Example

### Main Thread

```
// global variables
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {
  driver = init();
  is_running = true;
  is_in_use = false;
  async(thrA);
  async(thrB);
}
```

### Thread A

```
void thrA() {
  if(is_running){
  is_in_use = true;

  // use driver ...
    ⋮
  assert(is_running);
  is_in_use = false;
  }
}
```

### Thread B

```
void thrB() {
  assume(!is_in_use);
  is_running = false;

  // clean up ...
  free(driver);
    ⋮
  assert(!is_in_use);
}
```

What assumptions are we making?

# Example

### Main Thread

```
// global variables
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {
  driver = init();
  is_running = true;
  is_in_use = false;
  async(thrA);
  async(thrB);
}
```

### Thread A

```
void thrA() {


  if(is_running){
  is_in_use = true;









  // use driver ...
     .
     .
     .
  assert(is_running);
  is_in_use = false;
  }
}
```

### Thread B

```
void thrB() {
  assume(!is_in_use);



  is_running = false;
  // clean up ...
  free(driver);
     .
     .
     .
  assert(!is_in_use);






}
```

# The Kiss Aproach

Idea: Transform concurrent program $P$ to nondeterministic, sequential program $P_{\text{seq}}$ , which simulates subset of possbile executions of $P$.

$$\mathbf{P_{seq}} \begin{cases} \boxed{1} \to \boxed{1} \to \boxed{3} \to \boxed{1} \to \boxed{2} \to \boxed{3} \\ \boxed{3} \to \boxed{1} \to \boxed{3} \to \boxed{1} \to \boxed{2} \to \boxed{1} \\ \boxed{3} \to \boxed{2} \to \boxed{1} \to \boxed{3} \to \boxed{1} \to \boxed{1} \end{cases}$$

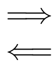
$P_{\text{seq}}$ is a nondeterministic scheduler executing $P$.

## Motivation

- Semantics of sequential program are easier
- Make problem decidable
- Maybe we can avoid exponential complexity ...
- *Keep it Simple and Sequential!*

# Properties of $P_{seq}$

$P_{seq}$ satisfies safety property



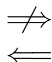
P satisfies safety property

# Properties of $P_{seq}$

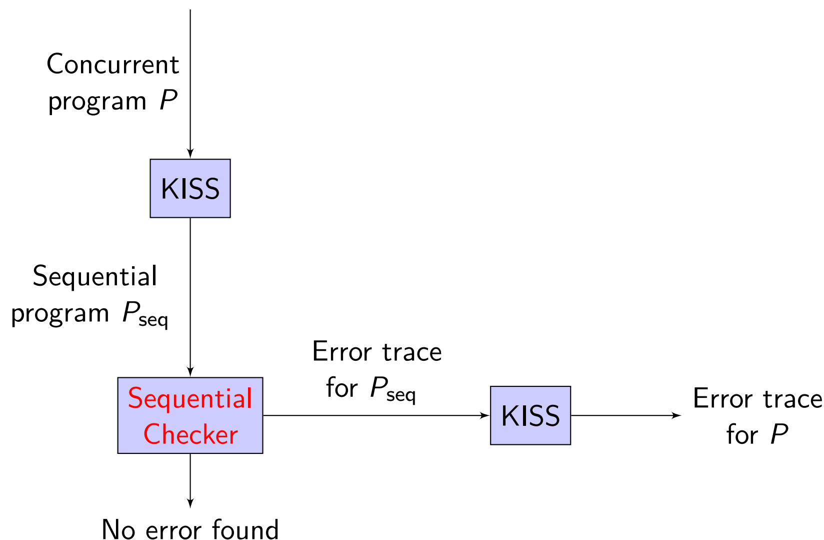
| **DECIDABLE** | | **UNDECIDABLE** |
|---|---|---|
| $P_{seq}$ satisfies | $\not\Rightarrow$ | P satisfies |
| safety property | $\Longleftarrow$ | safety property |

# Properties of $P_{seq}$

| DECIDABLE | | UNDECIDABLE |
|---|---|---|
| $P_{seq}$ satisfies safety property | $\not\Rightarrow$ $\Longleftarrow$ | P satisfies safety property |
| $P_{seq}$ violates safety property | $\Longrightarrow$ $\not\Leftarrow$ | P violates safety property |



# The KISS architecture



1 Motivation and Intuition behind KISS

2 The Transformation

3 Analysis and Conclusion

# Execution of Concurrent Programs

Basically, we want to implement a nondeterministic scheduler.
Problem: We need to manage n PCs and n Stacks, all with one Stack!

**Thread** 1 $\quad$ $\mathbf{PC_1} = 2$

```
1  void main1() {
2    f();
3    exit(0);
4  }
5
6  void f() {
7    g();
8    return;
9  }
10
11 void g() {
12   return;
13 }
```

main1

**Stack** 1

**Thread** *n* $\quad$ $\mathbf{PC_n} = 2$

```
1 void mainN() {
2   foo;
3   bar;
4   foobar;
5 }
```

mainN

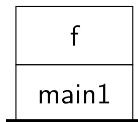
**Stack** *n*

# Execution of Concurrent Programs

Basically, we want to implement a nondeterministic scheduler.
Problem: We need to manage n PCs and n Stacks, all with one Stack!

**Thread** 1 **$PC_1 = 7$**

```
1  void main1() {
2    f();
3    exit(0);
4  }
5
6  void f() {
7    g();
8    return;
9  }
10
11 void g() {
12   return;
13 }
```




**Stack** 1

**Thread** *n* **$PC_n = 2$**

```
1  void mainN() {
2    foo;
3    bar;
4    foobar;
5  }
```

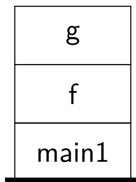
**Stack** *n*

# Execution of Concurrent Programs

Basically, we want to implement a nondeterministic scheduler.
Problem: We need to manage n PCs and n Stacks, all with one Stack!

**Thread** 1 $\mathbf{PC_1} = 12$

```
1  void main1() {
2    f();
3    exit(0);
4  }
5
6  void f() {
7    g();
8    return;
9  }
10
11 void g() {
12   return;
13 }
```




**Stack** 1

**Thread** $n$ $\mathbf{PC_n} = 2$

```
1 void mainN() {
2   foo;
3   bar;
4   foobar;
5 }
```

**Stack** $n$

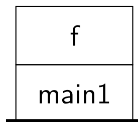
# Execution of Concurrent Programs

Basically, we want to implement a nondeterministic scheduler.

Problem: We need to manage n PCs and n Stacks, all with one Stack!

**Thread** 1 $\mathbf{PC_1} = 8$

```
1  void main1() {
2    f();
3    exit(0);
4  }
5
6  void f() {
7    g();
8    return;
9  }
10
11 void g() {
12   return;
13 }
```




**Stack** 1

**Thread** *n* $\mathbf{PC_n} = 2$

```
1  void mainN() {
2    foo;
3    bar;
4    foobar;
5  }
```

**Stack** *n*

# Execution of Concurrent Programs

Basically, we want to implement a nondeterministic scheduler.
Problem: We need to manage n PCs and n Stacks, all with one Stack!

**Thread** 1 — $\mathbf{PC_1} = 3$

```
1  void main1() {
2    f();
3    exit(0);
4  }
5
6  void f() {
7    g();
8    return;
9  }
10
11 void g() {
12   return;
13 }
```

main1

**Stack** 1

**Thread** *n* — $\mathbf{PC_n} = 2$

```
1  void mainN() {
2    foo;
3    bar;
4    foobar;
5  }
```

mainN

**Stack** *n*

# Execution of Concurrent Programs

Basically, we want to implement a nondeterministic scheduler.

Problem: We need to manage n PCs and n Stacks, all with one Stack!

**Thread** 1 $\quad$ $\mathbf{PC_1} = 3$

```
1  void main1() {
2    f();
3    exit(0);
4  }
5
6  void f() {
7    g();
8    return;
9  }
10
11 void g() {
12   return;
13 }
```

main1

**Stack** 1

**Thread** *n* $\quad$ $\mathbf{PC_n} = 2$

```
1 void mainN() {
2   foo;
3   bar;
4   foobar;
5 }
```

mainN

**Stack** *n*

Naive Solution: Store information in global variables.

But: Sequential model checking is exponential in # global variables!

# Scheduling with (almost) no Global Variables

Idea: Think of Threads as Functions!

At any point during execution we may nondeterministically

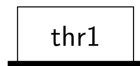
- Schedule a new thread by calling its starting function

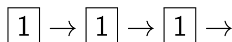
# Scheduling with (almost) no Global Variables

Idea: Think of Threads as Functions!

At any point during execution we may nondeterministically

- Schedule a new thread by calling its starting function



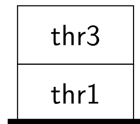



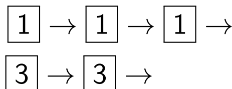
# Scheduling with (almost) no Global Variables

Idea: Think of Threads as Functions!

At any point during execution we may nondeterministically

- Schedule a new thread by calling its starting function



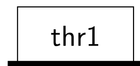



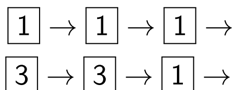
# Scheduling with (almost) no Global Variables

Idea: Think of Threads as Functions!

At any point during execution we may nondeterministically

- Schedule a new thread by calling its starting function
- Continue a thread already running by popping the current stack frame (`return;`)



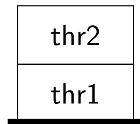



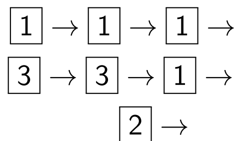
# Scheduling with (almost) no Global Variables

Idea: Think of Threads as Functions!

At any point during execution we may nondeterministically

- Schedule a new thread by calling its starting function
- Continue a thread already running by popping the current stack frame (return ;)



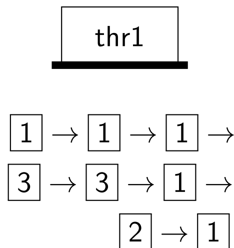



# Scheduling with (almost) no Global Variables

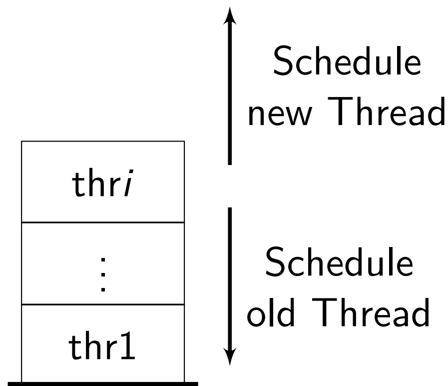
Idea: Think of Threads as Functions!

At any point during execution we may nondeterministically

- Schedule a new thread by calling its starting function
- Continue a thread already running by popping the current stack frame (return ;)



# Scheduling with (almost) no Global Variables (2)



# Example

## Main Thread

```
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {

  driver = init();

  is_running = true;

  is_in_use = false;

  async(thrA);

  async(thrB);
}
```

## Thread A

```
void thrA() {

  if(is_running){

  is_in_use = true;
     ⋮

  is_in_use = false;
  }
}
```

# Example

## Main Thread

```
Set ts = Set.init();
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {

  driver = init();

  is_running = true;

  is_in_use = false;

  async(thrA);

  async(thrB);
}
```

## Thread A

```
void thrA() {

  if(is_running){

  is_in_use = true;
     .
     .
     .

  is_in_use = false;
  }
}
```

# Example

## Main Thread

```
Set ts = Set.init();
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {

  driver = init();

  is_running = true;

  is_in_use = false;

  ts.add(thrA);

  ts.add(thrA);
}
```

## Thread A

```
void thrA() {

  if(is_running){

  is_in_use = true;
     ⋮

  is_in_use = false;
  }
}
```

```
CODE ≡
  sched_nondet_nr_of_thrs();
  if($) return;
```

# Example

## Main Thread

```
Set ts = Set.init();
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {
  CODE
  driver = init();
  CODE
  is_running = true;
  CODE
  is_in_use = false;
  CODE
  ts.add(thrA);
  CODE
  ts.add(thrA);
}
```

## Thread A

```
void thrA() {
  CODE
  if(is_running){
  CODE
  is_in_use = true;
     ⋮
  CODE
  is_in_use = false;
  }
}
```

```
CODE ≡
  sched_nondet_nr_of_thrs();
  if($) return;
```

# Example

### Main Thread

```
Set ts = Set.init(k);
usb_driver driver;
boolean is_running;
boolean is_in_use;

void main() {
  CODE
  driver = init();
  CODE
  is_running = true;
  CODE
  is_in_use = false;
  CODE
  if(ts.size<k) ts.add(thrA); else thrA();
  CODE
  if(ts.size<k) ts.add(thrB); else thrB();
}
```

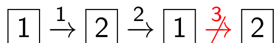
### Thread A

```
void thrA() {
  CODE
  if(is_running){
  CODE
  is_in_use = true;
      .
      .
      .
  CODE
  is_in_use = false;
  }
}
```

```
CODE ≡
  sched_nondet_nr_of_thrs();
  if($) return;
```

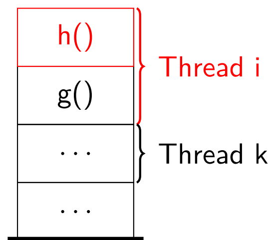
# Observations

- The currently executed thread is always on top of the stack $\Rightarrow$ the stack can grow without problems
- By exploiting the function concept, we can manage n PCs and n Stacks for free!
- We cannot simulate all possible executions:




- Between any two threads, we can simulate at most 2 interleavings
- $k = ts.max$ is "turning knob" for the number of executions we can simulate
  - $k = 0$ : async(f); $\rightsquigarrow$ f();

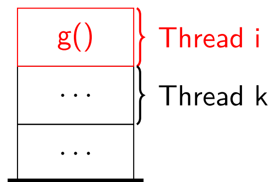
# Nondeterministic Returns




```
void g() {
   ⋮
   x = h();
   sched_nondet_nr_of_thrs();
   ⋮
}

void h() {
   ⋮
   sched_nondet_nr_of_thrs();
   if($) return;
   foo;
   ⋮
}
```

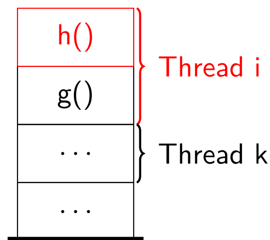
# Nondeterministic Returns




```
void g() {
   ⋮
   x = h();
   sched_nondet_nr_of_thrs();
   ⋮
}

void h() {
   ⋮
   sched_nondet_nr_of_thrs();
   if($) return;
   foo;
   ⋮
}
```

# Nondeterministic Returns

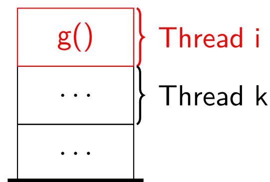



```
bool raise = false;

void g() {
  ⋮
  x = h(); if(raise) return;
  sched_nondet_nr_of_thrs();
  ⋮
}

void h() {
  ⋮
  sched_nondet_nr_of_thrs();
  if($) { raise = true; return; }
  foo;
  ⋮
}
```

# Nondeterministic Returns

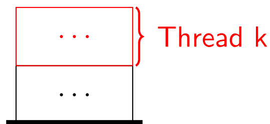



```
bool raise = false;

void g() {
   ⋮
   x = h(); if(raise) return;
   sched_nondet_nr_of_thrs();
   ⋮
}

void h() {
   ⋮
   sched_nondet_nr_of_thrs();
   if($) { raise = true; return; }
   foo;
   ⋮
}
```

# Nondeterministic Returns




```
bool raise = false;

void g() {
   ⋮
   x = h(); if(raise) return;
   sched_nondet_nr_of_thrs();
   ⋮
}

void h() {
   ⋮
   sched_nondet_nr_of_thrs();
   if($) { raise = true; return; }
   foo;
   ⋮
}
```

# Scheduling Function

```
void sched_nondet_nr_of_thrs() {
  var f;
  while($) {
    if(ts.size>0) {
      f = ts.get();
      f();
      raise = false;
    }
  }
}
```

1 Motivation and Intuition behind KISS

2 The Transformation

3 Analysis and Conclusion

# Complexity

- Sequential program with boolean variables: $O(|C| \cdot 2^{g+l})$
  - $|C| =$ size of the control flow graph
  - $g =$ number of global variables
  - $l =$ maximum number of local variables in scope at any time
- Complexity of KISS
  - small blow up of $|C|$ by constant factor
  - added small constant # global variables

  $\Rightarrow$ the complexity of using KISS on a concurrent program of a certain size is about the same as analyzing a sequential program of the same size (small constant factor)

# Conclusion

## Pros

- Low complexity, not exponential in # threads
- "Turning knob" `ts.max` ⇒ good scalability
- No false-positives
- Model checker only has to understand semantics of sequential programs
- Easy to implement

# Conclusion

## Pros

- Low complexity, not exponential in # threads
- "Turning knob" `ts.max` ⇒ good scalability
- No false-positives
- Model checker only has to understand semantics of sequential programs
- Easy to implement

## Cons

- Simulates executions with a maximum of 2 interleavings between any two threads
- May miss errors

# Thank you for listening!

Any Questions?

# Bibliography


[1] QUADEER, Shaz ; WU, Dinghao: KISS: Keep it simple and sequential. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2004)

[2] RAMALINGAM, G.: Context-sensitive synchronization-sensitive analysis is undecidable. In: *ACM Transactions on Programming Languages and Systems* 22 (2000), March, Nr. 2, S. 413–430

[3] REPS, T. ; HORWITZ, S. ; SAGIV, M.: Precise interprocedural dataflow analysis via graph reachability. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Software Engineering* (2003), S. 267–276

[4] SHARIR, M. ; PNUELI, A.: Two approaches to interprocedural data flow analysis. In: *Program Flow Analysis: Theory and Applications* (1981), S. 189–233