

KISS : Keep It Simple And Sequential

Shaz Qadeer, Microsoft Research
Dinghao Wu, Princeton University

Speaker : Kaspar Rohrer

Overview

- Problem description
 - Proposed solution
 - Technical details
 - Example
 - Conclusion
 - Questions?
-

Problem Description

- ❑ Design of concurrent programs (CP) is difficult and error prone
 - ❑ Analysis of CPs is difficult and time consuming (sometimes even undecidable)
 - ❑ Traditional automated techniques have high computational complexity (exponential to number of threads)
-

Proposed Solution

- Transform CP into sequential program (SP)
 - Use well defined transformation rules
 - Language independent (as we will see)
 - Check SP with sequential model checker
 - Checker does not need to understand concurrent semantics
 - This reduction is complete but unsound
 - But allows to check safety properties on CPs
 - Such checks are normally undecidable
-

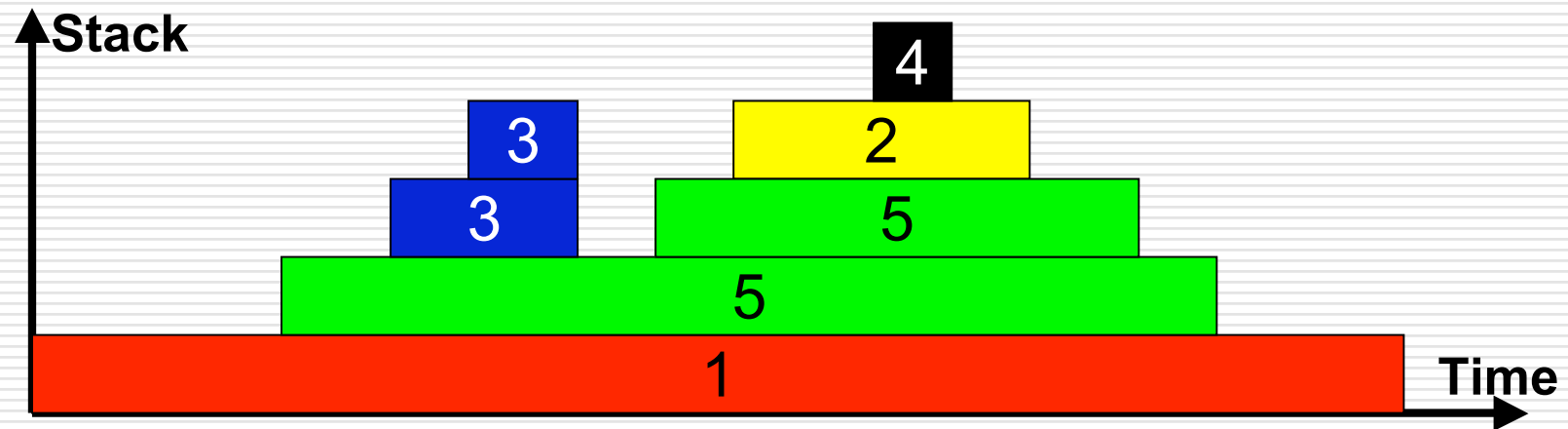
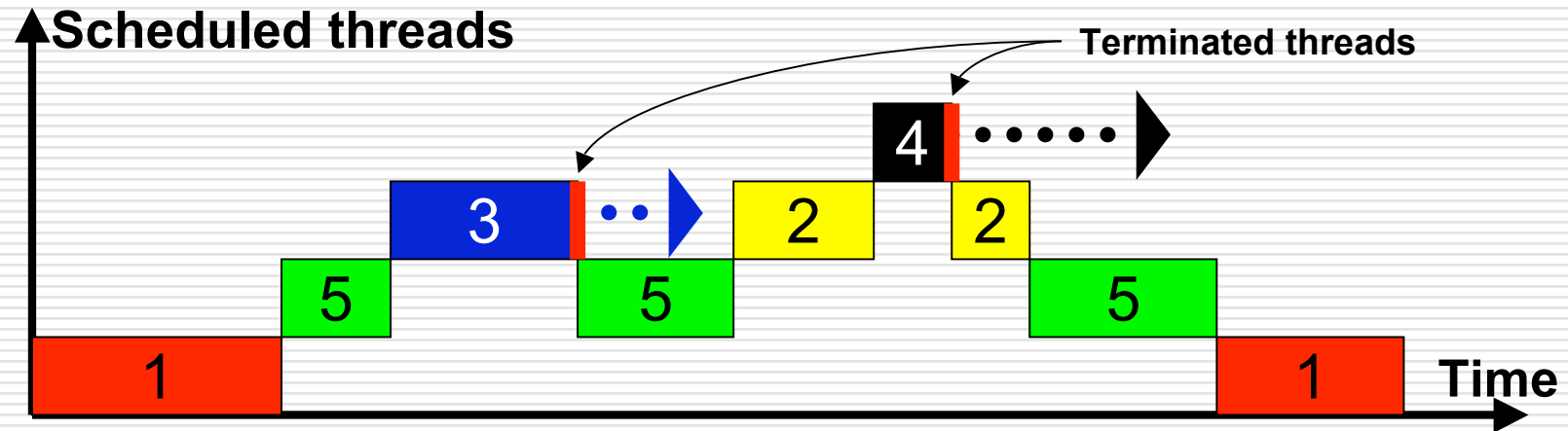
Tech - Transformation

- How do we transition from CP to SP?
 - A scheduler will be directly integrated into SP
 - At statement-level
 - No more concurrency, only function calls
 - Needs to be nondeterministic so model checker checks every possible path
 - Each path in SP will correspond to one possible execution of original CP
 - Transform each statement
 - Add scheduling code
 - Add safety checks
-

Tech - Scheduler

- As little additional state as possible (why?)
 - Limited resources and time for analysis!
 - Problem: Only a single stack for everything
 - Solution: Scheduling using stack discipline
 - We cannot have arbitrary interleaving of threads
 - Only partial thread resumption supported
-

Tech - Scheduler



Tech - Functionality

- Creating a new task Tx
 - Finite set TS that hold tasks to schedule
 - Add task Tx to TS if TS is not yet full
 - Immediately run task Tx if TS is already full
 - Scheduling a task Ts from TS (before each stmt)
 - Should we schedule before next statement? (ND)
 - If yes, choose a task Ts from TS and run it. (ND)
 - Terminating current task Tc (instead of next stmt)
 - Boolean RAISE indicating termination of task Tc
 - Should we terminate or execute next statement? (ND)
 - Subsequent statements of Tc will be ignored if RAISE
-

Tech - Functionality

- Transformation can be extended to check for race conditions
 - Checking for races on a variable R
 - Variable ACCESS indicating type of access
 - Possible checks before each statement (ND)
 - Checks depend on type of statement
 - Check if involved variables access R
 - Terminate task after a check is issued
 - Conflicting accesses only happen in different tasks
-

Example - SPL

Consider this simple parallel language (SPL):

Function names	$f ::= f_0 \mid f_1 \mid \dots$
Integers	$i ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$
Boolean constants	$b ::= \text{true} \mid \text{false}$
Constants	$c ::= i \mid f \mid b$
Primitives	$\text{op} ::= + \mid - \mid \times \mid ==$
Variables	$v ::= v_0 \mid v_1 \mid \dots$
Values	$u ::= v \mid c$
Statements	$s ::= \dots$

Example - SPL contd.

Statements $s ::=$

$v0 = c$
|
 $v0 = \&v1$
|
 $v0 = *v1$
|
 $*v0 = v1$
|
 $v0 = v1 \text{ op } v2$

|
 $v = v0()$
|
return
|
 $s1; s2$

|
assert($v0$)
|
assume($v0$)
|
atomic{ s }
|
async $v0()$
|
choice{ $s1 | \dots | sn$ }
|
iter{ s }

- choice and iter are ND
 - assume is blocking
 - The if and while statements can be emulated with assume and choice / iter.
-

Example - Transformation

Given a CP s , the SP to analyze is defined as follows:

Check(s) =_{def} RAISE=false; TS={}; [**s**]; schedule()

Auxiliary functions and definitions:

```
schedule() {  
  var f;  
  iter { if (size() > 0) { f = get(); [ $f$ ]; RAISE = false } }  
}
```

get() : remove a task from TS and return it (ND)

put() : insert a task into TS

size() : return the number of tasks in TS

raise =_{def} RAISE=true; return

Example - Transformation

The transformation function is given as (incomplete):

[v0 = c] = schedule(); choice{skip|raise}; v0 = c
[atomic{s}] = schedule(); choice{skip|raise}; s
[v = v0()] = schedule(); choice{skip|raise};
v = ***[v0]***(); if (RAISE) return;
[async v0()] = schedule(); choice{skip|raise};
if (size() < MAX) put(v0)
else { ***[v0]***(); RAISE = false }
[return] = schedule(); return;
[s1;s2] = ***[s1]***;***[s2]***
[iter{s}] = iter{***[s]***}
[choice{s1|...|sn}] = choice{***[s1]***|...|***[sn]***}
..... =

Example - Race Detection

We need an additional state variable ACCESS for race detection:

ACCESS = {0: No access, 1: Read, 2: Write}

Auxiliary functions for race detection:

check_r(x) { if (x == &R) { assert(access != 2); access = 1 } }

check_w(x) { if (x == &R) { assert(access == 0); access = 2 } }

We also need to rewrite the transformation:

Check(s) =_{def} RAISE=false; TS={}; ACCESS=0; **[s]**; schedule()

Example - Race Detection

The new transformation function is given as (incomplete):

[v = *v1] = schedule(); choice{skip
 |check_r(&v1);raise
 |check_r(v1);raise
 |check_w(&v);raise
 }; v0 = c

[v = v0()] = schedule(); choice{skip
 |check_r(&v0);raise
 |check_w(&v);raise
 }; v = ***[v0()]***(); if (RAISE) return;

[async v0()] = schedule(); choice{skip|check_r(&v0);raise};
 if (size() < MAX) put(v0)
 else {***[v0()]***(); RAISE = false}

Conclusion 1

- Feed transformed CP to sequential model checker
 - Error trace from model checker can be transformed back into CP
 - Might miss errors due to imposed restrictions
 - Complexity of model checking for a SP with boolean variables is $O(|C| * 2^{g+l})$
 - $|C|$ is the size of the control-flow graph
 - g is the number of global variables
 - l is the maximum number of local variables in scope at any location
-

Conclusion 2

- Complexity of KISS for a CP is about the same as for a SP of the same size
 - Only a constant factor (Why?)
 - KISS is independent of back-end
 - Language independent
 - After all, Windows device drivers are not written in SPL
 - Has successfully been used
 - Detected race conditions in Windows device drivers
 - But also reported some “false alarms”
-

Questions

