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
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

Scheduled threads

Terminated threads

Stack

Time
Creating a new task \( T_x \)
- Finite set \( T_S \) that hold tasks to schedule
- Add task \( T_x \) to \( T_S \) if \( T_S \) is not yet full
- Immediately run task \( T_x \) if \( T_S \) is already full

Scheduling a task \( T_s \) from \( T_S \) (before each stmt)
- Should we schedule before next statement? (ND)
- If yes, choose a task \( T_s \) from \( T_S \) and run it. (ND)

Terminating current task \( T_c \) (instead of next stmt)
- Boolean \( R_A I S E \) indicating termination of task \( T_c \)
- Should we terminate or execute next statement? (ND)
- Subsequent statements of \( T_c \) will be ignored if \( R_A I S E \)
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  \( f ::= f_0 | f_1 | \ldots \)
Integers  \( i ::= \ldots | -1 | 0 | 1 | \ldots \)
Boolean constants  \( b ::= \text{true} | \text{false} \)
Constants  \( c ::= i | f | b \)
Primitives  \( \text{op} ::= + | - | x | == \)
Variables  \( v ::= v_0 | v_1 | \ldots \)
Values  \( u ::= v | c \)
Statements  \( s ::= \ldots \)
### Example - SPL contd.

**Statements** $s ::=$

- $v_0 = c$
- $v_0 = \&v_1$
- $v_0 = *v_1$
- $*v_0 = v_1$
- $v_0 = v_1 \text{ op } v_2$
- $v = v_0()$
- $\text{return}$
- $s_1; s_2$

| assert($v_0$) |
| assume($v_0$) |
| atomic{$s$} |
| async $v_0()$ |
| choice{$s_1|...|s_n$} |
| 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:*

$\text{Check}(s) = \begin{cases} \text{RAISE}=false; \text{TS} = \{\}; [s]; \text{schedule()} \end{cases}$

*Auxiliary functions and definitions:*

$\text{schedule}() \begin{cases} \text{var f;} \begin{cases} \text{iter} \begin{cases} \text{if (size() > 0) \{ f = \text{get()}; [f](); \text{RAISE} = false \} \} \end{cases} \end{cases} \end{cases}$

- $\text{get()}$ : remove a task from TS and return it (ND)
- $\text{put()}$ : insert a task into TS
- $\text{size()}$ : return the number of tasks in TS
- $\text{raise} = \begin{cases} \text{RAISE}=true; \text{return} \end{cases}$
Example - Transformation

The transformation function is given as (incomplete):

\[
\begin{align*}
[v_0 = c] &= \text{schedule()}; \text{choice}\{\text{skip}|\text{raise}\}; v_0 = c \\
[\text{atomic}\{s\}] &= \text{schedule()}; \text{choice}\{\text{skip}|\text{raise}\}; s \\
[v = v_0()] &= \text{schedule()}; \text{choice}\{\text{skip}|\text{raise}\}; \\
&\hspace{1cm} v = [v_0]() \text{ if (RAISE) return; } \\
[\text{async } v_0()] &= \text{schedule()}; \text{choice}\{\text{skip}|\text{raise}\}; \\
&\hspace{1cm} \text{if (size() < MAX) put(v0) } \\
&\hspace{1cm} \text{else } \{ [v_0]() \text{ RAISE = false } \} \\
[\text{return}] &= \text{schedule()} \text{ return; } \\
[s_1; s_2] &= [s_1];[s_2] \\
[\text{iter}\{s\}] &= \text{iter}\{[s]\} \\
[\text{choice}\{s_1|...|s_n\}] &= \text{choice}\{[s_1]|...|[s_n]\} \\
........................... &= ........................... 
\end{align*}
\]
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) = \text{def RAISE=false; TS=\{}; ACCESS=0; } [s]; \text{ schedule()}
Example - Race Detection

The new transformation function is given as (incomplete):

\[ v = *v1 \]

\[ schedule(); \text{choice}\{\text{skip} \]
\[ | \text{check}_r(&v1); \text{raise} \]
\[ | \text{check}_r(v1); \text{raise} \]
\[ | \text{check}_w(&v); \text{raise} \]
\[ \} \]

\[ v_0 = c \]

\[ v = v0() \]

\[ schedule(); \text{choice}\{\text{skip} \]
\[ | \text{check}_r(&v0); \text{raise} \]
\[ | \text{check}_w(&v); \text{raise} \]
\[ \} \]

\[ v = [v0]() \]; if (RAISE) return;

\[ \text{async} \]

\[ v0() \]

\[ schedule(); \text{choice}\{\text{skip} \]
\[ | \text{check}_r(&v0); \text{raise} \]
\[ \} \]

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