Regression Verification for Multi-Threaded Programs

Arie Gurfinkel, Sagar Chaki — SEI/CMU

Ofer Strichman — Technion

Waterloo

Mentor Graphics

Dagstuhl Seminar on Program Equivalence
April 2018

Originally presented at VMCAI 2012

UNIVERSITY OF WATERLOO


Going
Retro

Regression Verification

Formal equivalence checking of two similar programs

Three selling points:

- Specification:
  - not needed
- Complexity:
  - depends on the semantic difference between the programs, and not on their size
- Invariants:
  - Easy to derive automatically high-quality loop/recursion invariants

Applications of an “ideal” RV tool

Regression Verification

- Validate refactoring
- Understand how changes propagate through the API

Semantic Impact Analysis

Generic Translation Validation

Proving Determinism

- P is equivalent to itself IFF P is deterministic

And many more…

Definition: Partial Equivalence (Strichman et al.)

Program P_1 and P_2 are *partially equivalent (p.e.)* IFF

- Executions of P_1 and P_2 on equal inputs
  - …which terminate,
  - result in equal outputs.

Undecidable

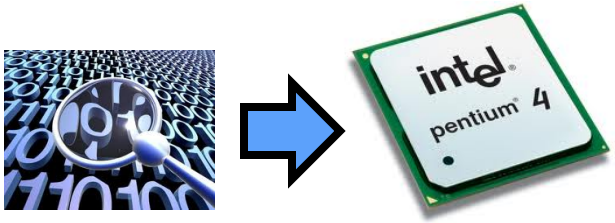
Supported by several tools

- RVT by Strichman et al.
- SymDiff by Lahiri et al.
- + LLRêve

Our Motivation

"Regression verification of Embedded Software"

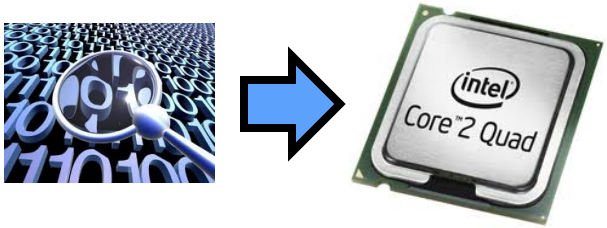
- Explore use of regression verification to aid in migration of real-time software from single-core to multi-core platforms



= ?

Need to extend regression verification to multi-threaded software first

- Real-time software is inherently multi-threaded
- Tasks run "in parallel", communicating via shared variables and locks
- Led to the research reported in this paper



Outline

What is a multi-threaded program?

$f_1 \parallel f_2$

Partial equivalence of multi-threaded programs

- Why the notion for sequential programs does not apply
- How to fix it

$f_1 \parallel f_2$

$=?$

$f'_1 \parallel f'_2?$

Sound proof rules for regression verification of multi-threaded programs

- What are the premises and the conclusion
- How to discharge the premises

$$\frac{\alpha, \beta, \gamma, \ldots}{f_1 \parallel f_2 = f'_1 \parallel f'_2}$$

Summary and Future Thoughts

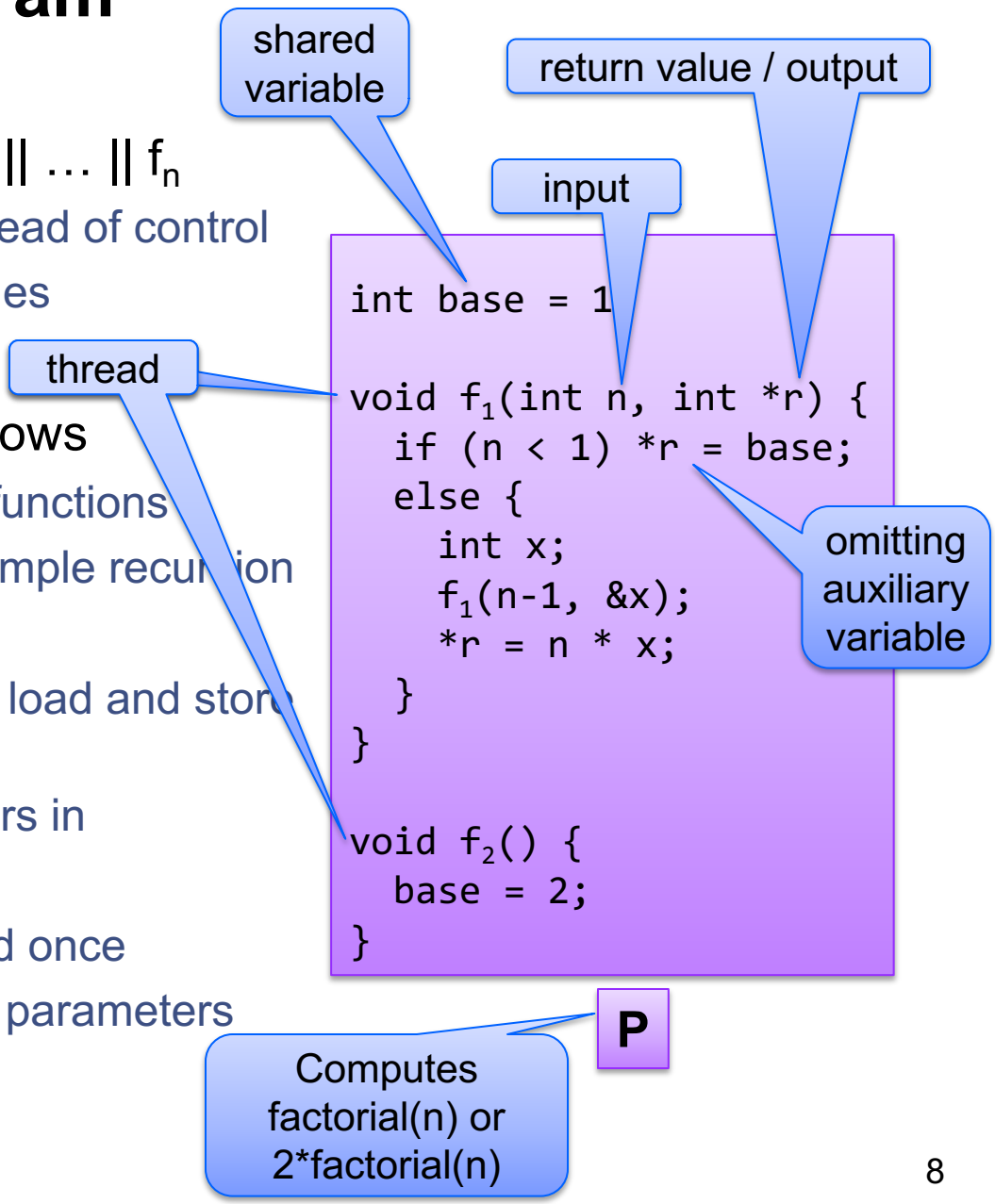
Multi-Threaded Program

Finite set of C functions : $P = f_1 \,||\, \ldots \,||\, f_n$

- Each executed in a separate thread of control
- Communicate via shared variables

Programs pre-processed as follows

- Loops are outlined to recursive functions
- Mutual recursion converted to simple recursion
- Non-recursive functions inlined
- Auxiliary variables introduced to load and store shared variables such that:
  - Shared variable x only appears in
    - t = x or x = exp
  - Each auxiliary variable is read once
- Functions return values through parameters passed by reference



Partial equivalence of multithreaded programs

Partial equivalence of sequential programs:

- P1 and P2 partially equivalent , all executions of P1 and P2 on equal inputs
  - …which terminate,
  - result in equal outputs

Consider multithreaded program P from previous slide

- Is P partially equivalent to itself?


```
int base = 1;

void f1(int n, int *r) {
  if (n < 1) *r = base;
  else {
    int x;
    f1(n-1, &x);
    *r = n * x;
  }
}

void f2() {
  base = 2;
}
```

Partial equivalence of multithreaded programs

P is a multi-threaded program (MTP)

Π(P): the set of terminating computations of P

R(P): relates inputs and outputs defined by Π(P):

- R(P) = { (*in*, *out*) | ∃π∈ Π(P). π begins in *in* and ends in *out* }

Definition: MTPs P_1, P_2 are *partially equivalent* if $R(P_1) = R(P_2)$

- denoted p.e. (P_1, P_2)

In our example P, is partially equivalent to itself

- R(P) = {(n, n!), (n, 2 * n!) | n ∈ nat }

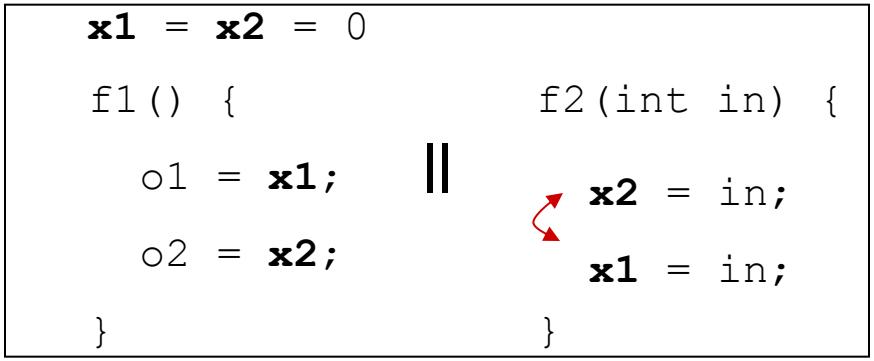
```
int base = 1;

void f1(int n, int *r) {
  if (n < 1) *r = base;
  else {
    int x;
    f1(n-1, &x);
    *r = n * x;
  }
}

void f2() {
  base = 2;
}
```

SOUND PROOF RULES FOR REGRESSION VERIFICATION OF MULTI-THREADED PROGRAMS

What affects partial equivalence of MTPs?



Swap write order

Before:

```
in = 1
o1 = 1, o2 = 0;
```

$(1, \langle 1, 0 \rangle) \in R(p)$

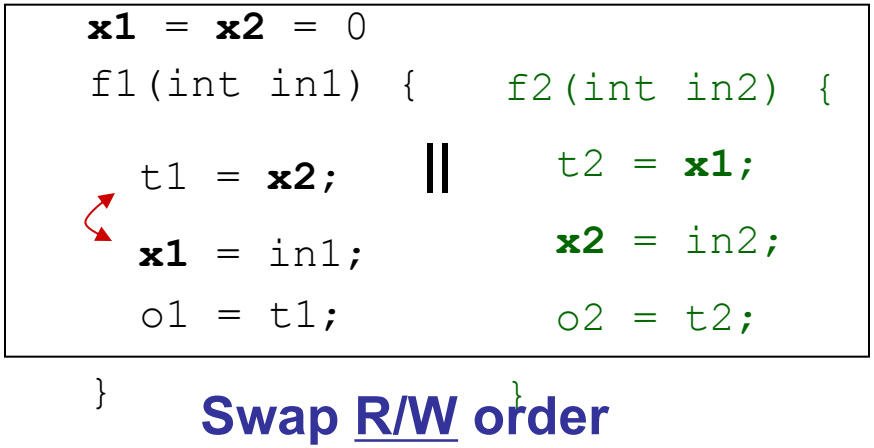
After:

```
in = 1
```

$o1 = 1 \Rightarrow o2 = 1$

$(1, \langle 1, 0 \rangle) \notin R(p)$

What affects partial equivalence of MTPs?



Before:

in1 = 1, in2 = 2

x1 = 1;

t2 = x1 = 1;

x2 = in2 = 2;

t1 = x2 = 2;

o1 = t1 = 2;

o2 = t2 = 1;

$(\langle 1,2\rangle, \langle 2,1\rangle) \in R(p)$

After:

in1 = 1, in2 = 2

o2 = 1 $\Rightarrow$

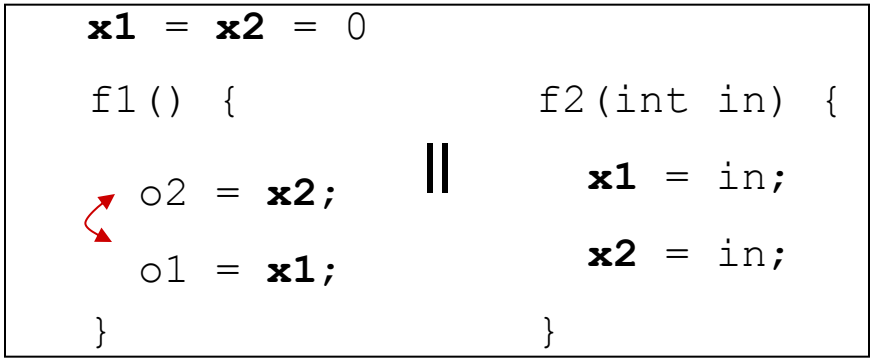
x1 = 1 < t2 = x1 $\Rightarrow$

t1 = 0 $\Rightarrow$

o1 = 0

$(\langle 1,2\rangle, \langle 2,1\rangle) \notin R(p)$

What affects partial equivalence of MTPs?



Swap read order

Before:

in = 1

o1 = 0, o2 = 1;

$(1, \langle 0, 1 \rangle) \in R(p)$

After:

in = 1

$o1 = 0 \Rightarrow o2 = 0$

$(1, \langle 0, 1 \rangle) \notin R(p)$

Mapping

Assume each function is used in a single thread.

- Otherwise, duplicate it

Find a mapping between the non-basic types

Find a bijective map between:

- threads
- shared variables
- functions (same prototype),
- in mapped functions: read globals, written-to globals

Without such a mapping: `goto end-of-talk`.

Function Semantics: Observable Stream

Consider a function *f* and input *in*

The observable stream of *f(in)*'s run is its sequence of

- function calls
- read/write of shared variables

Example: let *x* be a shared variable:

	The observable stream:
`x = in;`	
`t1 = t;`	
`t = x;`	
`g(t+1);`	

Observable Equivalence of Functions

Consider a function *f* and input *in*

The observable stream of *f(in)*'s run is its sequence of function calls and read/write of shared variables

- If the run is finite, we say it is a finite observable stream

f, f' are observably equivalent $\Leftrightarrow$
$\forall in.\ f(in)$, $f'(in)$ have equal sets of finite observable streams

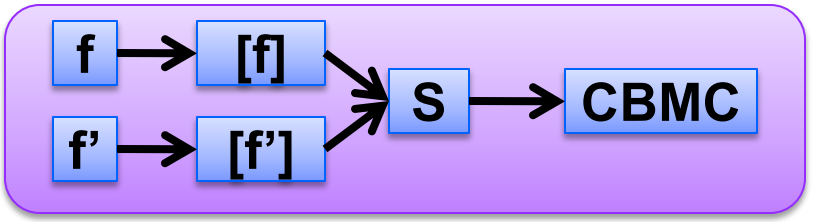
- Denoted by observe-equiv(f,f')

Assume: outputs are defined via shared variables (i.e., observable)
Then: observable equivalence $\Rightarrow$ partial equivalence

Checking Observable Equivalence of f and f'

Transform function f and f' to [f] and [f'] by:

- Reading shared variable x from an "input" stream UF_x
  - Maintain a location c in the stream
  - $t = x \leftarrow t = UF_x(c)$
- Recording outputs to the observable stream
  - shared variable accesses and function calls

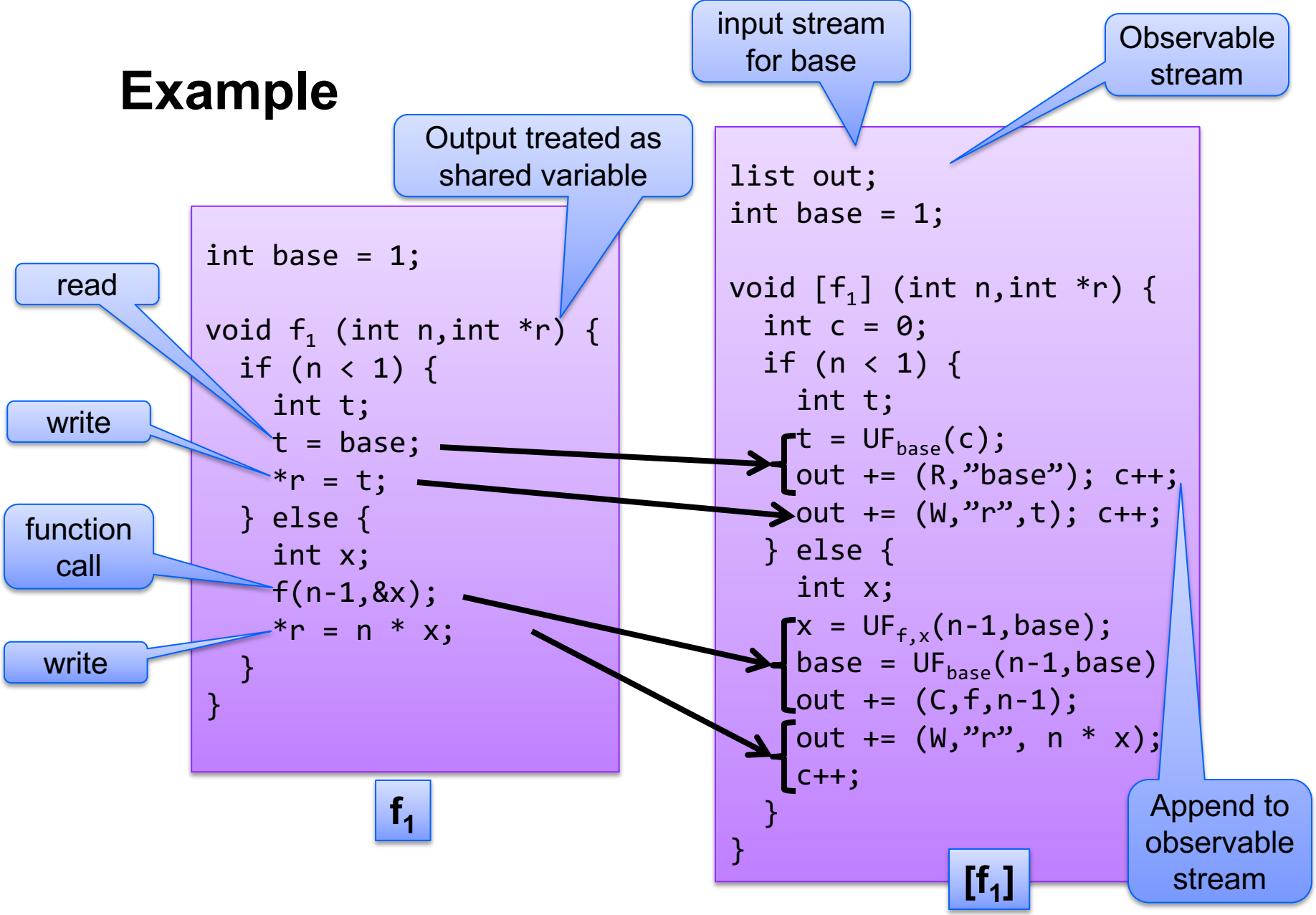


Generate sequential program S that calls [f];[f'] but also

- Ensures that inputs to f and f' are non-deterministic but equal
  - assume equal arguments and $UF_x = UF_{x'}$
- Asserts at the end that the observable streams of f and f' are equal
- S is linear in size of f and f'

Check validity (i.e., no assertion failure) of S (e.g., with CBMC)

Example


input stream for base
Observable stream
Output treated as shared variable
read
write
function call
write
int base = 1;
void f1 (int n,int *r) {
if (n < 1) {
int t;
t = base;
*r = t;
} else {
int x;
f(n-1,&x);
*r = n * x;
}
}
f1
list out;
int base = 1;
void [f1] (int n,int *r) {
int c = 0;
if (n < 1) {
int t;
t = UFbase(c);
out += (R,"base"); c++;
out += (W,"r",t); c++;
} else {
int x;
x = UFf,x(n-1,base);
base = UFbase(n-1,base)
out += (C,f,n-1);
out += (W,"r", n * x);
c++;
}
}
[f1]
Append to observable stream

```
void f1(int td, int *o) {
  int t1, t2, t3;
  if (td <= 0) t2 = x;
  else {
    t1 = x;
    t3 = t1 % td;
    x  = td;
    f1(t3, &t2);
  }
  *o = t2;
}

void f2() {
  int t;
  x = t;
}
```

```
void f1'(int td, int *o') {
  int t1, t2;
  t2 = x';
  if (td > 0) {
    t1 = x';
    x' = td;
    f1'(t1 % td, &t2);
  }

  *o' = t2;
}

f2'() {
  int t;
  x' = t;
}
```

Compute `gcd(td,x)`, where x is a shared variable set in another thread

```
void f1 (int td, int *o) {
  int t1, t2, t3, c = 0;
  if (td <= 0) {
    //2 ▷ t2 = x;
    t2 = UF_x(c);
    out1 += (R, "x"); c++;
  } else {
    //2 ▷ t1 = x;
    t1 = UF_x(c);
    out1 += (R, "x"); c++;
    //3 ▷ x  = td;
    t3 = t1 % td;
    x  = td;
    out1 += (W, "x", td); c++;
    //3 ▷ t2 = f1(t3, &t2);
    t2 = UF_f1_o(t3, x);
    x  = UF_f1_x(t3, x);
    out1 += (C, f1, t3, x);
  }
  //1 ▷ *o = t2;
  out1 += (W, "*o", t2); c++;
}
```

```
void f1'(int td, int *o') {
  int t1, t2, c = 0;
  //2 ▷ t2 = x';
  t2 = UF_x(c);
  out2 += (R, "x'"); c++;
  if (td > 0) {
    //2 ▷ t1 = x';
    t1 = UF_x(c);
    out2 += (R, "x'"); c++;
    //2 ▷ x' = td;
    x' = td;
    out2 += (W, "x'", td); c++;
    //3 ▷ f1'(t1 % td, &t2);
    t2 = UF_f1_o(t1 % td, x');
    x'  = UF_f1_x(t1 % td, x');
    out2 += (C, f1, t1 % td, x');
  }
  //1 ▷ *o' = t2;
  out2 += (W, "*o'", t2); c++;
}

main() {
  int in;
  f1(in);  f1'(in);
  rename(out2);
  assert(out1 == out2);
}
```

Proof Rule 1

Compositional:

- check one function pair at a time (supports recursion)
- requires verification of sequential program only (no thread composition)
- even when there are more than two threads
- shows that regression verification is “easier” than verification even for MTPs

ween the functions of P_1 and P_2

Sound (proof in the paper)

$$\frac{\forall\ f,f' \in \textit{map}.\ \textbf{observe-equiv}(f,f')}{\textbf{p.e.}\ (P_1, P_2)}$$

However:

- Premises are very strong
- f must be equivalent to f’ under all environments

Proof Rule 2

Premises are weaker than Rule 1 but harder to discharge

- More “complete” than Rule 1
- Details in the paper

Observable equivalence of f and f’ under environments compatible with the other threads in the program

Key insights:

since there are no loops, each function f reads shared variables atmost K times (K depends on f)

the environment of f compatible with another thread f_q is abstracted by a K-recursion-bounded abstraction of f_q

intuitively, the abstraction allows all behaviors with at most K recursive calls of f_q

$$\frac{\forall\ f,f' \in \mathit{map}.\ \Delta\ (f,f')}{\text{p.e. } (P_1, P_2)}$$

Compositional (like Proof Rule 1)

check one function pair at a time (supports recursion)

requires verification of sequential program only (no thread composition)

even when there are more than two threads

Other extensions

Support arbitrary many threads with dynamic thread creation

- Track thread creation in the output stream
- As long as same threads are crated, p.e. holds

Support atomic sections

- Necessary for modeling synchronization primitives such as locks
- Use local read/write (i.e., no UF) for shared variables written under atomic section

Sagar Chaki, Arie Gurfinkel, Ofer Strichman: Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). Formal Methods in System Design 47(3): 287-301 (2015)

Summary and Future Thoughts

Foundations of regression verification for multi-threaded programs

- Notion of partial equivalence of multi-threaded programs
- Two (sound) proof rules

Implementation and experimental validation

Synchronization primitives

- Locks, semaphores, atomic blocks

Real-time software

- Different execution (e.g., reactive) and scheduler model (e.g., priority-based)
- Different synchronization primitives (e.g., priority-ceiling, priority-inheritance)

THANK YOU!

UNIVERSITY OF
WATERLOO

Contact Information

Arie Gurfinkel
Senior Member of Technical Staff
RTSS Program
Telephone: +1 412-268-7788
Email: arie@cmu.edu

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Web
www.sei.cmu.edu/staff/arie

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this presentation is not intended in any way to infringe on the rights of the trademark holder.

This Presentation may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.