Extracting behaviour from an executable instruction set model

Brian Campbell Ian Stark

lfcs Laboratory for Foundations of Computer Science

THE UNIVERSITY *of* EDINBURGH informatics

REMS project `rems.io`

EPSRC grant EP/K008528/1

FMCAD, October 6 2016

Introduction

Previously developed automated test generation for executable ISA models in HOL4 [FMICS 2014].

Want to automate extraction of instruction behaviour—

1. constraints for execution
2. results of execution

—from model in HOL4 theorem prover for new targets.

Successfully implement symbolic execution in HOL4, reusing its standard symbolic evaluation features.

Applied to simple MIPS model and experimental CHERI processor

Motivation: testing ISA models

Automatic randomised test generation in HOL4:

Generate instruction sequence

↓

Extract instruction behaviour from model

↓

Calculate sequence's constraints and effects

↓

Solve constraints to build test (SMT)

↓

Add test harness

Motivation: testing ISA models

Automatic randomised test generation in HOL4:

$$\begin{tikzcd}
\text{Generate instruction sequence} \arrow[d] \\
\text{Extract instruction behaviour from model} \arrow[d] \\
\text{Calculate sequence's constraints and effects} \arrow[d] \\
\text{Solve constraints to build test (SMT)} \arrow[d] \\
\text{Add test harness}
\end{tikzcd}$$

Previously:

+ Reused `stepLib` verification library for instruction behaviour
− Library needs to be written for new models
− Library skips some behaviour (exceptions, unaligned accesses)

Motivation: Testing CHERI

Experimental MIPS-compatible design with capability security extensions:

- Lots of new instructions, exceptions
- ISA model used for architectural exploration
- Bluespec design for processor

provide motivation for testing

Plain MIPS model has `stepLib`

- CHERI more than twice as large
- also more complete (e.g., memory)
- `stepLib` not ported

Model example: MIPS 32-bit signed immediate addition

L3 domain specific language, compiled to HOL4:

```
dfn'ADDI (rs,rt,immediate) =
(λstate.
   (let s =
      if NotWordValue (FST (GPR rs state)) then
        SND (raise'exception (UNPREDICTABLE "ADDI: NotWordValue")
             state)
      else state
    in
    let v = (32 >< 0) (FST (GPR rs s)) + sw2sw immediate
    in
      if word_bit 32 v ≠ word_bit 31 v then SignalException Ov s
      else write'GPR (sw2sw ((31 >< 0) v),rt) s))
```

- State threaded through definition

Model example: MIPS 32-bit signed immediate addition

L3 domain specific language, compiled to HOL4:

```
dfn'ADDI (rs,rt,immediate) =
(λstate.
   (let s =
      if NotWordValue (FST (GPR rs state)) then
        SND (raise'exception (UNPREDICTABLE "ADDI: NotWordValue")
             state)
      else state
    in
    let v = (32 >< 0) (FST (GPR rs s)) + sw2sw immediate
    in
      if word_bit 32 v ≠ word_bit 31 v then SignalException Ov s
      else write'GPR (sw2sw ((31 >< 0) v),rt) s))
```

- State threaded through definition
- 64-bit behaviour unspecified
- Overflow processor exception

Pre-existing library: `addiu $1,$2,3`

```
[¬if word_bit 31 (s.gpr 2w) then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
  else (63 >< 32) (s.gpr 2w) ≠ 0w,
 s.MEM s.PC = 36w, s.MEM (s.PC + 1w) = 65w, s.MEM (s.PC + 3w) = 3w,
 s.MEM (s.PC + 2w) = 0w,
 (1 >< 0) s.PC = 0w, s.exception = NoException]
⊢ NextStateMIPS s =
   SOME
     (s with
      <|PC := s.PC + 4w;
        gpr := (1w =+ sw2sw ((31 >< 0) (s.gpr 2w) + 3w)) s.gpr|>)
```

- Hypotheses contain assumptions, well-definedness constraints
- ⊢ Stylised conclusion: next = series of record updates
- One theorem per branch

A rough rule-based operational semantics

Pre-existing library implementation

Semi-automatic

- Assumptions and cases fed in manually
- Primarily uses symbolic evaluation
- Builds up results for
  - each instruction implementation
  - instruction fetch
  - decode

  then combines them into next step function

For faster development, we want to

- Avoid writing per-instruction information
- Case split automatically
- Avoid specifying intermediate results

Symbolic execution in HOL4

Symbolic evaluation

- general computation rules (including bitvectors, ...)
- specialisation, e.g., restricting memory accesses
- single result, leaves the structure intact

Symbolic state

- Set of rewrites, one per field

Symbolic execution

- follows threading of state
- case splits at conditionals, pattern matching
- discard unspecified/uninteresting cases
- keeps path condition in hypotheses
- one result per path

Symbolic evaluation

Uses

- HOL4 theories for booleans, bitvectors, naturals, integers, datatypes, . . .
- custom conversions to
  - FOR loops only once bound known
  - extra bitvector simplification
- model-specific conversions which
  - ⋆ may introduce hypotheses to limit behaviour
  - simplify memory mapping
  - inject instructions into memory

Instruction injection uses rewrite generated by applying symbolic execution to instruction fetch function.

Symbolic execution

Recursive procedure; described below with rules:

$$H, S \vdash t \rightsquigarrow \overline{(H', t')}$$

H	General hypotheses incorporates path condition
S	Per-field state information (equations)
t	Source term (also u, v below)

One result (H', t') per path

State always appears to the right:

$$\frac{H, S \vdash u \rightsquigarrow \overline{(H', u')}}{H, S \vdash (t, u) \rightsquigarrow \overline{(H', (t, u'))}} \quad \textsc{Pair}$$

Symbolic execution

For let, separate ordinary data from state:

$$\frac{H, S \vdash t \leadsto \overline{(H', (t', s'))} \qquad \forall i.\quad H'_i, S \lhd s'_i \vdash u[t'_i/x] \leadsto \overline{(H''_i, u'_i)}}{H, S \vdash \mathsf{let}\ (x, s) = t\ \mathsf{in}\ u \leadsto \bigcup_i \overline{(H''_i, u'_i)}}\ \textsc{Let}$$

S has per-field state information, $S \lhd s$ updates symbolic state

$$\frac{(H, t), S \vdash u \leadsto \overline{(H', u')} \qquad (H, \neg t), S \vdash v \leadsto \overline{(H'', v')}}{H, S \vdash \mathsf{if}\ t\ \mathsf{then}\ u\ \mathsf{else}\ v \leadsto \overline{(H', u')} \cup \overline{(H'', v')}}\ \textsc{Cond}$$

Similar rule for pattern matching

Symbolic execution

Function application unfolds the definition

$$\frac{\begin{array}{c} c\, x_1 \ldots x_{n+1} := t \qquad H, S \vdash v \rightsquigarrow \overline{(H', v')} \\ \forall i. \quad H'_i, S \vdash t[u_1/x_1, \ldots, u_n/x_n, v'_i/x_{n+1}] \rightsquigarrow \overline{(H''_i, t'_i)} \end{array}}{H, S \vdash c\, u_1 \ldots u_n\, v \rightsquigarrow \bigcup_i \overline{(H''_i, t'_i)}} \quad \textsc{App}$$

$$\frac{}{H, S \vdash \texttt{raise'exception}\; t\; u \rightsquigarrow \emptyset} \quad \textsc{Undef}$$

Other unwanted constants are handled similarly

Soundness and (in)completeness

Soundness

- By construction:

$$H, S \vdash t \rightsquigarrow \overline{(H', t')}$$

produces theorems for each i,

$$H'_i \vdash t = t'_i$$

Completeness

Incomplete by construction:

- e.g., deliberately simplify memory accesses

Complete up to specialisation?

- No formal results
- Systematic construction
  avoids overly strong assumptions about cases

Example: `addi $1,$2,3`

Hypotheses

Term

```
dfn'ADDI (2w,1w,3w) s
```

State only changes at the end

Example: `addi $1,$2,3`

Hypotheses

Term

```
let s =
    if NotWordValue (FST (GPR 2w state)) then
      SND (raise'exception (UNPREDICTABLE "ADDI: NotWordValue") state)
    else state
  in
  let v = (32 >< 0) (FST (GPR 2w s)) + 3w
  in
    if word_bit 32 v ≠ word_bit 31 v then SignalException Ov s
    else write'GPR (sw2sw ((31 >< 0) v),1w) s
```

Example: addi $1,$2,3

Hypotheses

Term

```
if NotWordValue (FST (GPR 2w state)) then
  SND (raise'exception (UNPREDICTABLE "ADDI: NotWordValue") state)
else state
```

(First part of let, rest on stack)

Example: `addi $1,$2,3`

Hypotheses

```
NotWordValue (s.c_gpr 2w)
```

Term

```
    SND (raise'exception (UNPREDICTABLE "ADDI: NotWordValue") state)
```

(First branch of if, first part of let, rest on stack)

Example: `addi $1,$2,3`

Hypotheses

```
NotWordValue (s.c_gpr 2w)
```

Term

```
        raise'exception (UNPREDICTABLE "ADDI: NotWordValue") state
```

(First part of if, let, rest on stack)

Undefined - discard case

Example: `addi $1,$2,3`

Hypotheses

```
¬NotWordValue (s.c_gpr 2w)
```

Term

```
      state
```

(Second part of if, first of let, rest on stack)

Example: addi $1,$2,3

Hypotheses

```
¬NotWordValue (s.c_gpr 2w)
```

Term

```
let v = (32 >< 0) (FST (GPR 2w state)) + 3w
  in
    if word_bit 32 v ≠ word_bit 31 v then SignalException Ov state
    else write'GPR (sw2sw ((31 >< 0) v),1w) state
```

(Second part of let)

Example: `addi $1,$2,3`

Hypotheses

```
¬NotWordValue (s.c_gpr 2w)
```

Term

```
if word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) ≠
   word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w) then
    SignalException Ov state
else
   write'GPR (sw2sw ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)),1w) state
```

(let evaluated)

Example: `addi $1,$2,3`

Hypotheses

```
¬NotWordValue (s.c_gpr 2w),
word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) ≠
word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
```

Term

```
  SignalException Ov state
```

(First branch)

Processor exception - choose to discard case

(Can do processor exceptions, but not on one slide)

Example: `addi $1,$2,3`

Hypotheses

```
¬NotWordValue (s.c_gpr 2w),
word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) =
word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
```

Term

```
write'GPR (sw2sw ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)),1w) state
```

(Second branch)

Example: addi $1,$2,3

Hypotheses

```
¬NotWordValue (s.c_gpr 2w),
word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) =
word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
```

Term

```
((),
 state with
   c_gpr := (1w =+ sw2sw ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)))
            state.c_gpr)
```

Final result: register 1 updated by signed addition

Example: Symbolic state update ($S \lhd s$)

Per-field state information S consists of equations:

```
state.c_gpr = s0.c_gpr
state.c_state = s0.c_state with c_lo := NONE
...
```

relating current state `state` to initial state `s0`

Example: Symbolic state update ($S \lhd s$)

Per-field state information S consists of equations:

```
state.c_gpr = s0.c_gpr
state.c_state = s0.c_state with c_lo := NONE
...
```

relating current state `state` to initial state `s0`

The update for `addi $1,$2,3` is

```
state with
  c_gpr := (1w =+ sw2sw ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)))
             state.c_gpr)
```

apply per-field to get

```
newstate.c_gpr =
  (1w =+ sw2sw ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w))) s0.c_gpr)
newstate.c_state = s0.c_state with c_lo := NONE
...
```

Performance

Compare existing library with combined approach on ‘plain’ MIPS:

- behaviour extraction much longer (old 0.23s, new 3.16s)
- but only rises to 17% of total test generation time
- even without caching, etc

(times median over 500 8-instruction tests)

CHERI times rise again (32.3s; 33% of total test generation time)

Still acceptable for batch production

Results

Found model bugs

- in instructions we couldn't test before
- on processor exceptions (esp. unintended writeback)

Successfully

- generates tests automatically
- less than two minutes per test
- ⋆ tracks new versions of the model with few adjustments

Instruction generation and harness generation phases still require manual maintenance; symbolic execution is robust against changes.

Summary

Automated extraction of instruction behaviour from an executable model

- combining prover's symbolic evaluation with symbolic execution
- reducing amount of model-specific input required
- with acceptable performance cost, and scope for improvement

Successfully applied to large CHERI ISA model, finding bugs in model and processor design.

HOL4 turns out to be a good environment for symbolic execution