Extracting behaviour from an executable instruction set model

Brian Campbell    Ian Stark

REMS project     rems.io
EPSRC grant EP/K008528/1

FMCAD, October 6 2016
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:

Generate instruction sequence
↓
Extract instruction behaviour from model
↓
Calculate sequence’s constraints and effects
↓
Solve constraints to build test (SMT)
↓
Add test harness

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) =
(\state.
 (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 \neq 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

\[
\neg \text{if word\_bit } 31 (s.\text{gpr } 2w) \text{ then } (63 \gg 32) (s.\text{gpr } 2w) \neq 0xFFFFFFFFw \\
\quad \text{else } (63 \gg 32) (s.\text{gpr } 2w) \neq 0w, \\
\quad s.\text{MEM } s.\text{PC } = 36w, s.\text{MEM } (s.\text{PC } + 1w) = 65w, s.\text{MEM } (s.\text{PC } + 3w) = 3w, \\
\quad s.\text{MEM } (s.\text{PC } + 2w) = 0w, \\
\quad (1 \gg 0) s.\text{PC } = 0w, s.\text{exception } = \text{NoException}\]

\[\vdash \text{NextStateMIPS } s =
\begin{array}{l}
\quad \text{SOME} \\
\quad \quad \text{(s with} \\
\quad \quad \quad \langle |\text{PC } := s.\text{PC } + 4w; \\
\quad \quad \quad \quad \text{gpr } := (1w =+ \text{sw2sw } ((31 \gg 0) (s.\text{gpr } 2w) + 3w)) s.\text{gpr}|>\rangle
\end{array}\]

- 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:

\[ H, S \vdash t \rightsquigarrow (H', t') \]

\( H \) 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:

\[
\frac{H, S \vdash u \rightsquigarrow (H', u')}{H, S \vdash (t, u) \rightsquigarrow (H', (t, u'))}
\]

\( \text{PAIR} \)
Symbolic execution

For let, separate ordinary data from state:

\[
H, S \vdash t \leadsto (H', (t', s')) \quad \forall i. \quad H'_i, S \triangleleft s'_i \vdash u[t'_i/x] \leadsto (H''_i, u'_i) \\
H, S \vdash \text{let } (x, s) = t \text{ in } u \leadsto \bigcup_i (H''_i, u'_i)
\]

Let

\[
(H, t), S \vdash u \leadsto (H', u') \quad (H, \neg t), S \vdash v \leadsto (H'', v') \\
H, S \vdash \text{if } t \text{ then } u \text{ else } v \leadsto (H', u') \cup (H'', v')
\]

Cond

Similar rule for pattern matching
Symbolic execution

Function application unfolds the definition

\[
c \, x_1 \ldots x_{n+1} := t \quad H, S \vdash v \leadsto (H', v')
\]

\[
\forall i. \quad H'_i, S \vdash t[u_1/x_1, \ldots, u_n/x_n, v'_i/x_{n+1}] \leadsto (H''_i, t'_i)
\]

\[
H, S \vdash c \, u_1 \ldots u_n \, v \leadsto \bigcup_i (H''_i, t'_i)
\]

\[
H, S \vdash \text{raise'exception } t \, u \leadsto \emptyset
\]

Other unwanted constants are handled similarly.
Soundness and (in)completeness

**Soundness**
- By construction:
  \[ H, S \vdash t \rightsquigarrow (H', t') \]
  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 $\neq$ word_bit 31 v then SignalException 0v 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**

\[\neg \text{NotWordValue} \ (s.c\_gpr\ 2w)\]

**Term**

\[\text{state}\]

(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 0v state
         else write’GPR (sw2sw ((31 >> 0) v),1w) state

(Second part of let)
Example: addi $1, $2, 3

Hypotheses

\[ \neg \text{NotWordValue} (s.c\_gpr \ 2w) \]

Term

if word\_bit 32 ((32 \ll 0) (s.c\_gpr 2w) + 3w) \neq 
word\_bit 31 ((32 \ll 0) (s.c\_gpr 2w) + 3w) then
   SignalException 0v state
else
   write’GPR (sw2sw ((31 \ll 0) ((32 \ll 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 0v state

(First branch)

Processor exception - choose to discard case

(Can do processor exceptions, but not on one slide)
Example: `addi $1,$2,3`

**Hypotheses**

\[ \neg \text{NotWordValue (s.c\_gpr 2w)}, \]
\[ \text{word\_bit 32 ((32 }\ll 0) (s.c\_gpr 2w) + 3w) =} \]
\[ \text{word\_bit 31 ((32 }\ll 0) (s.c\_gpr 2w) + 3w) \]

**Term**

\[ \text{write’GPR (sw2sw ((31 }\ll 0) ((32 }\ll 0) (s.c\_gpr 2w) + 3w)),1w) state} \]

(Second branch)
Example: addi $1,$2,3

Hypotheses

\[
\neg \text{NotWordValue (s.c\_gpr 2w)},
\text{word\_bit 32 ((32} \ll 0) (s.c\_gpr 2w) + 3w) =
\text{word\_bit 31 ((32} \ll 0) (s.c\_gpr 2w) + 3w)
\]

Term

\[
((),
\text{state with}
\begin{align*}
c\_gpr &:= (1w =+ \text{sw2sw ((31} \ll 0) ((32} \ll 0) (s.c\_gpr 2w) + 3w))) \\
&\quad \text{state.c\_gpr}
\end{align*}
\]

Final result: register 1 updated by signed addition
Example: Symbolic state update ($S \triangleleft s$)

Per-field state information $S$ consists of equations:

```plaintext
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 \((S \triangleleft s)\)

Per-field state information \(S\) consists of equations:

\[
\begin{align*}
\text{state.} \text{c}_\text{gpr} &= \text{s0.} \text{c}_\text{gpr} \\
\text{state.} \text{c}_\text{state} &= \text{s0.} \text{c}_\text{state} \text{ with } \text{c}_\text{lo} \equiv \text{NONE} \\
&\ldots
\end{align*}
\]

relating current state \text{state} to initial state \text{s0}

The update for \text{addi} \ $1,$2,3 is

\[
\begin{align*}
\text{state with} \\
&\quad \text{c}_\text{gpr} := (1w \Rightarrow+ \text{sw2sw} ((31 \gg 0) ((32 \gg 0) (\text{s}. \text{c}_\text{gpr} \ 2w) + 3w))) \\
&\quad \text{state.} \text{c}_\text{gpr})
\end{align*}
\]

apply per-field to get

\[
\begin{align*}
\text{newstate.} \text{c}_\text{gpr} &= \\
&\quad (1w \Rightarrow+ \text{sw2sw} ((31 \gg 0) ((32 \gg 0) (\text{s}. \text{c}_\text{gpr} \ 2w) + 3w))) \text{s0.} \text{c}_\text{gpr}) \\
\text{newstate.} \text{c}_\text{state} &= \text{s0.} \text{c}_\text{state} \text{ with } \text{c}_\text{lo} \equiv \text{NONE} \\
&\ldots
\end{align*}
\]
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.
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