Low-Level Program Verification using Matching Logic Reachability

Dwight Guth, Andrei Ștefănescu, and Grigore Roșu

University of Illinois at Urbana-Champaign

June 29, 2013
Motivation

- Operational semantics as models of programming languages
- Use operational semantics as basis for
  - interpreters
  - type-checking
  - model-checking
  - deductive program verification
Outline

1. Language Definition
2. Matching Logic Reachability
3. Program Verification
Simple Low Level Language

- implemented in the K framework
- standard arithmetic and logic operations
- registers
- load/store instructions for memory access
- branching instructions
- interrupts
- I/O instructions
- time units/operation
Basic Instructions

\[
\text{SYNTAX} \quad BInst ::= BOpCode \; Register, \; Exp, \; Exp \; [\text{strict}(3, \; 4)]
\]

\[
\text{SYNTAX} \quad BOpCode ::= \text{add} | \; \text{sub} | \; \text{mul} | \; \text{div} | \; \text{or} | \; \text{and}
\]

\[
\text{SYNTAX} \quad Exp ::= Register | \; \#\text{Int}
\]

\[
\text{SYNTAX} \quad Register ::= r\text{Int}
\]
Load/Store Instructions

\[
\text{SYNTAX } \text{MInst ::= load Register, Exp [strict(3)]}
\]

\[
\text{SYNTAX } \text{MInst ::= store Exp, Exp [strict(2, 3)]}
\]
Branching and Interrupt Instructions

**Syntax**

\[
\text{JInst ::= jmp \ Id}
\]

\[
\text{BrInst ::= BrOpCode \ Id, Exp, Exp [\text{strict}(3, 4)]}
\]

\[
\text{BrOpCode ::= beq | bne | blt | ble}
\]

\[
\text{BrOpCode ::= int}
\]

\[
\text{NOpCode ::= rfi}
\]
Sample Program with two Interrupts

main: li r0 , #100
    li r1 , #0
    li r2 , #0
    int t1, #7, #10
    int t2, #10, #15
    jmp loop

loop: sub r0 , r0 , #1
    bne loop , r0 , #0
    halt

t1:   add r1 , r1 , #1
    rfi

t2:   add r2 , r2 , #1
    rfi
Configuration

CONFIGURATION:

\[
\left\langle \text{load}($\text{PGM}$) \right\rangle_k \left\langle \text{jumpTo}($\text{main}$) \right\rangle \left\langle \text{Map} \right\rangle_{\text{pgm}} \left\langle \text{Map} \right\rangle_{\text{mem}} \left\langle \text{Map} \right\rangle_{\text{reg}} \\
\left\langle \text{$\text{TIMING}$} \right\rangle_{\text{timings}} \left\langle 0 \right\rangle_{\text{wcet}} \left\langle \text{$\text{INPUT}$} \right\rangle_{\text{input}} \left\langle \text{$\text{INITIAL}$} \right\rangle_{\text{status}} \left\langle \text{List} \right\rangle_{\text{timers}} \\
\left\langle 0 \right\rangle_{\text{priority}} \left\langle \text{List} \right\rangle_{\text{stack}} \left\langle \text{Set} \right\rangle_{\text{interrupts}}
\]

Dwight Guth, Andrei Ştefănescu, and Grigore Roşu (University of Illinois at Urbana-Champaign) – Low-Level Program Verification using Matching Logic Reachability – June 29, 2013
Evaluating Arithmetic Operations

\[
\text{RULE} \quad \begin{array}{c}
\frac{r1 \ldots}{I2} \\
\hline
\end{array} \quad \langle \ldots I \mapsto I2 \ldots \rangle \text{ reg}
\]

\[
\text{RULE} \quad \begin{array}{c}
\frac{\text{add } r1, I2, I3 \ldots}{\text{time(add)}} \\
\hline
\end{array} \quad \langle \frac{R}{R[I2 + I3/I]} \rangle \text{ reg}
\]
Evaluating load/store

RULE
\[ \left( \text{load } r, l_2 \ldots \right)_k \left( \cdots l_2 \leftrightarrow l_3 \cdots \right)_{\text{mem}} \left( \frac{R}{R[l_3/l]} \right)_{\text{reg}} \]

RULE
\[ \left( \text{store } l, l_2 \ldots \right)_k \left( \frac{M}{M[l_2/l]} \right)_{\text{mem}} \]
Evaluating Branching Instructions

RULE
\[
\begin{array}{c}
\begin{array}{c}
\text{bne } X, l, l2 \\
\text{time(bne)} \sim \text{branch}(l \neq l2, X)
\end{array}
\end{array}
\]}

RULE
\[
\begin{array}{c}
\begin{array}{c}
\text{branch(true, X)}
\end{array}
\end{array}
\]}
\[
\begin{array}{c}
\begin{array}{c}
\text{jumpTo(X)}
\end{array}
\end{array}
\]}

RULE
\[
\begin{array}{c}
\begin{array}{c}
\text{branch(false, _)}
\end{array}
\end{array}
\]}
\[
\begin{array}{c}
\begin{array}{c}
\cdot K
\end{array}
\end{array}
\]}

Low-Level Program Verification using Matching Logic Reachability

June 29, 2013 12 / 31
Evaluating int

\[
\text{int schedules an interrupt to fire } l \text{ cycles after executing, and then every } l_2 \text{ cycles thereafter. The timers cell stores the currently activated interrupts in a list of tuples.}
\]
Evaluating rfi

RULE

\[ \langle \text{rfi} \sim \_ \rangle_k \langle \text{time(rfi)} \sim K \rangle \langle (K, \text{Priority}) \_ \_ \_ \rangle \langle \text{List} \_ \_ \_ \rangle \langle \text{stack} \_ \_ \_ \rangle \langle \_ \_ \_ \rangle \langle \text{Priority} \_ \_ \_ \rangle \]

Restore the previously executing code from the stack cell, which also contains the previously-executing priority to restore to the priority cell. Interrupts are assigned numeric priority in the order they are scheduled by the program, and can interrupt only code running at a lower priority. The main program begins executing at priority 0.
I/O Instructions

- read/write from a number of buses
- each time cycle, the value on each bus is updated by an external environment
Each instruction takes a particular number of cycles. Afterwards, the I/O buses are updated and interrupts may fire.
Outline

1. Language Definition

2. Matching Logic Reachability

3. Program Verification
Matching Logic

- Logic for specifying *static properties* of program configuration and reasoning about them (generalizes separation logic)
- Extends first-order logic with *patterns*
  - Special predicates which are configuration terms with variables
  - Configurations satisfy patterns iff they match them
- Parametric in a model of program configurations (which is axiomatized)
Matching Logic Reachability Rules
(ICALP’12, OOPSLA’12, LICS’12)

“Rewrite” rules over matching logic patterns:

\[ \varphi \Rightarrow \varphi' \]

Semantics: any concrete configuration satisfying \( \varphi \) and terminating reaches a configuration satisfying \( \varphi' \), in the transition system induced by the operational semantics

Since patterns generalize terms, matching logic reachability rules capture term rewriting rules
Operational Semantics and Axiomatic Semantics as Reachability Rules

- Operational semantics rule $l \Rightarrow r$ if $b$ is syntactic sugar for reachability rule $l \land b \Rightarrow r$

- Hoare triple encoded in a reachability rule with the empty code in the right-hand-side
Reachability Logic

- Language-independent proof system for deriving sequents of the form

  \[ \mathcal{A} \vdash_{C} \varphi \Rightarrow \varphi' \]

- \( \mathcal{A} \) (axioms) and \( C \) (circularities) are sets of eachability rules

- Intuitively, *symbolic execution* with operational semantics + reasoning with *cyclic behaviors*
Proof System for Reachability

**Axiom:**
\[
\varphi \Rightarrow \varphi' \in A \\
\frac{}{\mathcal{A} \vdash C \varphi \Rightarrow \varphi'}
\]

**Transitivity:**
\[
\mathcal{A} \vdash C \varphi_1 \Rightarrow \varphi_2 \quad \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow \varphi_3 \\
\frac{}{\mathcal{A} \vdash C \varphi_1 \Rightarrow \varphi_3}
\]

**Reflexivity:**
\[
\frac{}{\mathcal{A} \vdash \varnothing \varphi \Rightarrow \varphi}
\]

**Circularity:**
\[
\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi' \\
\frac{}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}
\]

**Logic Framing:**
\[
\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \quad \psi \text{ is a (patternless) FOL formula} \\
\frac{}{\mathcal{A} \vdash_C \varphi \land \psi \Rightarrow \varphi' \land \psi}
\]

**Consequence:**
\[
\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash_C \varphi_1' \Rightarrow \varphi_2' \\
\frac{}{\models \varphi_2' \rightarrow \varphi_2} \\
\frac{}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2}
\]

**Case Analysis:**
\[
\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi \quad \mathcal{A} \vdash_C \varphi_2 \Rightarrow \varphi \\
\frac{}{\mathcal{A} \vdash_C \varphi_1 \lor \varphi_2 \Rightarrow \varphi}
\]

**Abstraction:**
\[
\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \quad X \cap FreeVars(\varphi') = \emptyset \\
\frac{}{\mathcal{A} \vdash_C \exists X \varphi \Rightarrow \varphi'}
Traditional vs Our Approach

- Traditional proof systems: *language-specific*

\[
\{\psi \land e \neq 0\} \vdash \{\psi\} \\
\{\psi\} \text{while}(e) \vdash \{\psi \land e = 0\}
\]

- Our proof system: *language-independent*

### Circularity:
\[
\begin{align*}
A \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi & \Rightarrow \varphi' \\
A \vdash_{C} \varphi & \Rightarrow \varphi'
\end{align*}
\]

### Transitivity:
\[
\begin{align*}
A \vdash_{C} \varphi_1 & \Rightarrow \varphi_2 \\
A \cup C & \vdash \varphi_2 \Rightarrow \varphi_3 \\
A & \vdash_{C} \varphi_1 \Rightarrow \varphi_3
\end{align*}
\]
Soundness and Completeness

- Sound (partial correct) with respect to the transition system induced by the semantics
- Relatively complete under some weak assumptions about the configuration model (it can express Gödel $\beta$ function)
- Proofs size comparable with Hoare logic (FM’12)
Outline

1. Language Definition
2. Matching Logic Reachability
3. Program Verification
Verifier for a Low-level Language

- Derives program specifications from the operational semantics using the proof system
- Implemented in the K framework as a set of rules added to the operational semantics
- Reasoning required by the Consequence proof rule
  - Maude, for structural matching
  - Z3, for arithmetic constraints
- Automated (the user only provides the specifications)
Sample program properties

- Upper bounds for the total number of cycles simple programs take to execute (computing the sum of the first "n" numbers, sorting an array, etc)
- Correctness of programs manipulating I/O buses
- Upper bound for the number of cycles a program with interrupts takes to terminate
Sample Program with two Interrupts

```assembly
main:  li r0 , M:Int
       li r1 , #0
       li r2 , #0
       int t1, #7, #10
       int t2, #10, #15
       jmp loop

loop:  sub r0 , r0 , #1
       bne loop , r0 , #0
       halt

t1:   add r1 , r1 , #1
       rfi

t2:   add r2 , r2 , #1
       rfi
```
Invariant

\[
\begin{align*}
\text{RULE} & : \begin{cases}
0 \mapsto N & 1 \mapsto R1 & 2 \mapsto R2 \\
\langle \text{add} \mapsto 1, \text{rfi} \mapsto 2 \rangle_{\text{timing}} & \langle \text{Time} \rangle_{\text{wcet}} & \langle \text{t1}, \text{T1}, 10, \text{t2}, \text{T2}, 15 \rangle_{\text{timers}}
\end{cases} \\
\langle \$ \rangle_k \langle \$ \rangle_{\text{pgm}} \langle 0 \rangle_{\text{priority}} \langle \text{List} \rangle_{\text{stack}}
\end{align*}
\]

\[
\begin{align*}
\text{when} & : \quad N > 0 \land T1 > \text{Time} \land T2 > \text{Time} \land D > 0 \land D = 3 \times N + 1 + \max\left(0, 3 \times \left\lceil \frac{(D-T1+\text{Time})}{10} \right\rceil \right) + \max\left(0, 3 \times \left\lceil \frac{(D-T2+\text{Time})}{15} \right\rceil \right)
\end{align*}
\]
Invariant

- Invariant derives $pg_m$ and $k$ cell contents from placement in program.
- Invariant depends on timing parameters: side condition uses integer 3.
- Current time is Time.
- N remaining loop iterations.
- All remaining loop iterations plus interrupts last D cycles.
- Next interrupts occur at T1 and T2.
- Invariant depends on timer frequency: 10 and 15 in denominators.
- Priority and stack derived from invariant beginning in normal code.
- Number of remaining interrupts derived from fixed-point equations.
Conclusions

- $\mathcal{K}$ definition of a low-level language
- Matching logic verifier constructed from the $\mathcal{K}$ definition
- Proofs of upper bound of the number of execution cycles and of correctness