Proving Skipping Refinement with ACL2s

Mitesh Jain and Pete Manolios

Northeastern University

ACL2 2015

Motivation

Motivation

- Property-based
  e.g., Temporal logics

Motivation

- Property-based
  e.g., Temporal logics
- Refinement-based

Refinement

Specification

Instruction Set Architecture

- *add rd, ra, rb*
- *sub rd, ra, rb*
- *jnz imm*
- $\ldots$

High-level abstract system ($\mathcal{A}$)

Refinement

Specification

Implementation

Instruction Set Architecture

- *add rd, ra, rb*
- *sub rd, ra, rb*
- *jnz imm*
- ...

High-level abstract system ($\mathcal{A}$)

Lower-level concrete system ($\mathcal{C}$)

Refinement

Specification

Implementation

Instruction Set Architecture

- *add rd, ra, rb*
- *sub rd, ra, rb*
- *jnz imm*
- ...

High-level abstract system ($\mathcal{A}$)

Lower-level concrete system ($\mathcal{C}$)

$\mathcal{C}$ <u>refines</u> $\mathcal{A}$ if every behavior of $\mathcal{C}$ is a behavior of $\mathcal{A}$.

Refinement in ACL2 community

- Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation, Manolios, Namjoshi, Sumners, 1999
- Verification of Pipelined Machines in ACL2, Manolios, 2000
- An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2, Sumners, 2000
- Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof, Goerigk, Wolfgang, 2000
- Deductive Verification of Pipelined Machines Using First-Order Quantification, Sandip, Warren, 2004
- Verification of Executable Pipelined Machines with Bit-Level Interfaces, Manolios, Srinivasan, 2005
- ...

Superscalar Microprocessor

IF	ID	RF	EX	WB		
IF	ID	RF	EX	WB		
	IF	ID	RF	EX	WB	
	IF	ID	RF	EX	WB	
		IF	ID	RF	EX	WB
		IF	ID	RF	EX	WB

Superscalar Microprocessor

IF	ID	RF	EX	WB		
IF	ID	RF	EX	WB		
	IF	ID	RF	EX	WB	
	IF	ID	RF	EX	WB	
		IF	ID	RF	EX	WB
		IF	ID	RF	EX	WB

- Pipelining

- Superscalar Execution

Superscalar Microprocessor

IF	ID	RF	EX	WB		
IF	ID	RF	EX	WB		
	IF	ID	RF	EX	WB	
	IF	ID	RF	EX	WB	
		IF	ID	RF	EX	WB
		IF	ID	RF	EX	WB

- Pipelining $\rightsquigarrow$ Stuttering
  Many concrete steps $\approx$ One abstract step
  Well-founded stuttering simulation and bisimulation
- Superscalar Execution

Superscalar Microprocessor

IF	ID	RF	EX	WB		
IF	ID	RF	EX	WB		
	IF	ID	RF	EX	WB	
	IF	ID	RF	EX	WB	
		IF	ID	RF	EX	WB
		IF	ID	RF	EX	WB

- Pipelining $\rightsquigarrow$ Stuttering
  Many concrete steps $\approx$ One abstract step
  Well-founded stuttering simulation and bisimulation
- Superscalar Execution $\rightsquigarrow$ Skipping
  One concrete step $\approx$ Many abstract steps

Superscalar Microprocessor

IF	ID	RF	EX	WB		
IF	ID	RF	EX	WB		
	IF	ID	RF	EX	WB	
	IF	ID	RF	EX	WB	
		IF	ID	RF	EX	WB
		IF	ID	RF	EX	WB

- Pipelining $\rightsquigarrow$ Stuttering
  Many concrete steps $\approx$ One abstract step
  Well-founded stuttering simulation and bisimulation
- Superscalar Execution $\rightsquigarrow$ Skipping
  One concrete step $\approx$ Many abstract steps

Existing notions of refinement do not account for "skipping"

Skipping Refinement

- Skipping refinement[1], a notion of refinement that directly accounts for **finite stuttering and finite skipping**

[1] CAV 2015

Skipping Refinement

- Skipping refinement[1], a notion of refinement that directly accounts for **finite stuttering and finite skipping**
- Sound and complete proof method that is amenable for **automated reasoning**

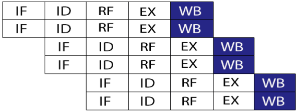
[1] CAV 2015

Skipping Refinement

We develop the notion in the framework of labeled transition systems $\mathcal{M} = \langle S, \rightarrow, L \rangle$, where:

- S is a set of states
- $\rightarrow \subseteq S \times S$ is the transition relation
- L is the labeling function
  Its domain is S, and tells us what is observable in a state.

Skipping Refinement



$\lesssim_r$

Instruction Set Architecture

- *add rd, ra, rb*
- *sub rd, ra, rb*
- *jnz imm*
- ...

$\mathcal{M}_C$ is a *skipping refinement* of $\mathcal{M}_A$ with respect to a refinement map $r : S_c \rightarrow S_A$, if there exists a relation $B \subseteq S_C \times S_A$ such that the following holds.

Skipping Refinement

IF	ID	RF	EX	WB		
IF	ID	RF	EX	WB		
	IF	ID	RF	EX	WB	
	IF	ID	RF	EX	WB	
		IF	ID	RF	EX	WB
		IF	ID	RF	EX	WB

$\lesssim_r$

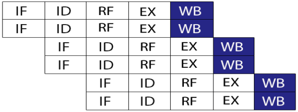
Instruction Set Architecture

- *add rd, ra, rb*
- *sub rd, ra, rb*
- *jnz imm*
- ...

$\mathcal{M}_C$ is a *skipping refinement* of $\mathcal{M}_A$ with respect to a refinement map $r : S_c \to S_A$, if there exists a relation $B \subseteq S_C \times S_A$ such that the following holds.

- $\langle \forall s \in S_C :: sBr.s \rangle$ *and*

Skipping Refinement



$\lesssim_r$

Instruction Set Architecture

- *add rd, ra, rb*
- *sub rd, ra, rb*
- *jnz imm*
- ...

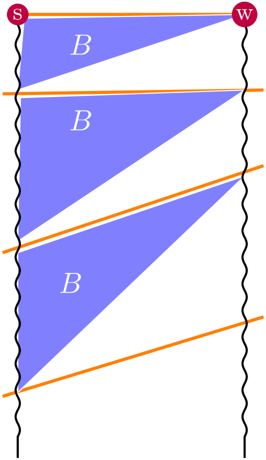
$\mathcal{M}_C$ is a *skipping refinement* of $\mathcal{M}_A$ with respect to a refinement map $r : S_c \rightarrow S_A$, if there exists a relation $B \subseteq S_C \times S_A$ such that the following holds.

- $\langle \forall s \in S_C :: sBr.s \rangle$ *and*
- B is a skipping simulation relation on the disjoint union of $\mathcal{M}_C$ and $\mathcal{M}_A$

Skipping Simulation (SKS)

$B \subseteq S \times S$ is an SKS on $\mathcal{M}$ iff for all s, w, such that sBw following holds.

- $L.s = L.w$ and
- $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : \underline{match(B, \sigma, \delta)} \rangle \rangle$

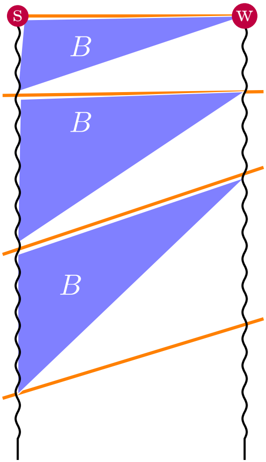


Skipping Simulation (SKS)

$B \subseteq S \times S$ is an SKS on $\mathcal{M}$ iff for all s, w, such that sBw following holds.

- $L.s = L.w$ and
- $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : \underline{match(B, \sigma, \delta)} \rangle \rangle$

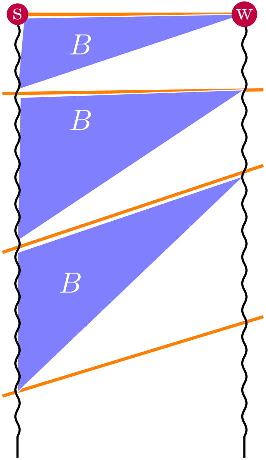
Reason about infinite behaviors.



Skipping Simulation (SKS)

$B \subseteq S \times S$ is an SKS on $\mathcal{M}$ iff for all s, w, such that sBw following holds.

- $L.s = L.w$ and
- $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : \underline{match(B, \sigma, \delta)} \rangle \rangle$



Reason about infinite behaviors.

Define an alternate characterization

Well-founded Skipping Simulation (WFSK)

$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$

Well-founded Skipping Simulation (WFSK)

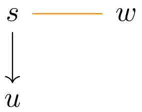
$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$
- There exist functions, $rankT \colon S \times S \rightarrow W$, $rankL \colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and

Well-founded Skipping Simulation (WFSK)

$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

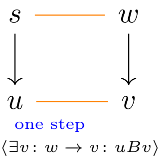
- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$
- There exist functions, $rankT \colon S \times S \rightarrow W$, $rankL \colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and $\langle \forall s, u, w \in S : sBw \wedge s \rightarrow u$:



Well-founded Skipping Simulation (WFSK)

$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

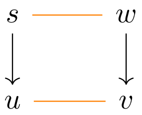
- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$
- There exist functions, $rankT \colon S \times S \rightarrow W$, $rankL \colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and $\langle \forall s, u, w \in S : sBw \wedge s \rightarrow u \colon$



Well-founded Skipping Simulation (WFSK)

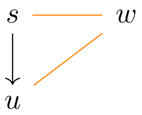
$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$
- There exist functions, $rankT \colon S \times S \rightarrow W$, $rankL \colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and $\langle \forall s, u, w \in S : sBw \wedge s \rightarrow u$:



one step

$\langle \exists v \colon w \rightarrow v \colon uBv \rangle$



stuttering on left

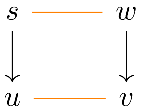
$(uBw \wedge rankT(u, w) \prec rankT(s, w))$

Well-founded Skipping Simulation (WFSK)

$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

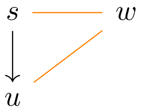
- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$
- There exist functions, $\mathit{rankT} \colon S \times S \rightarrow W$, $\mathit{rankL} \colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and

  $\langle \forall s, u, w \in S : sBw \wedge s \rightarrow u$:



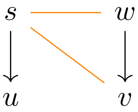
one step

$\langle \exists v \colon w \rightarrow v \colon uBv \rangle$



stuttering on left

$(uBw \wedge \mathit{rankT}(u, w) \prec \mathit{rankT}(s, w))$



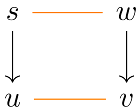
stuttering on right

$\langle \exists v \colon w \rightarrow v \colon sBv \wedge \mathit{rankL}(v, s, u) < \mathit{rankL}(w, s, u) \rangle$

Well-founded Skipping Simulation (WFSK)

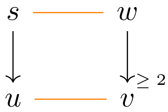
$B \subseteq S \times S$ is a WFSK on $\mathcal{M} = \langle S, \to, L \rangle$ iff :

- $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$
- There exist functions, $rankT \colon S \times S \to W$, $rankL \colon S \times S \times S \to \omega$, such that $\langle W, \prec \rangle$ is well-founded and $\langle \forall s, u, w \in S : sBw \wedge s \to u$:



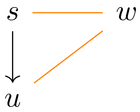
one step

$\langle \exists v \colon w \to v \colon uBv \rangle$



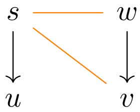
skipping on right

$\langle \exists v : w \to^{\geq 2} v \colon uBv \rangle \rangle$



stuttering on left

$(uBw \wedge rankT(u, w) \prec rankT(s, w))$



stuttering on right

$\langle \exists v \colon w \to v \colon sBv \wedge rankL(v, s, u) < rankL(w, s, u) \rangle$

Case Studies

- **Optimized Memory controller**
  Buffers read/write requests to the memory and updates multiple memory location in a page simultaneously
- **JVM-inspired (buffered) Stack Machine**
  Buffers instructions and eliminates redundant operations on stack
- **Vectorizing compiler transformation**
  Vectorizes a sequence of scalar instructions to a Single Instruction Multiple Data (SIMD) instruction

Vectorizing compiler transformation

Analyze the source program and when possible replace scalar instructions with SIMD instructions.

$$\begin{array}{ccccc} a & = & b & + & c \\ d & = & e & + & f \end{array} \rightarrow \boxed{\begin{array}{c} a \\ d \end{array}} = \boxed{\begin{array}{c} b \\ e \end{array}} +_{SIMD} \boxed{\begin{array}{c} c \\ f \end{array}}$$

- Correctness of the transformation:
  Given a scalar program, the target program generated by the transformation is equivalent to the scalar program.

Vectorizing compiler transformation

Analyze the source program and when possible replace scalar instructions with SIMD instructions.

$$\begin{array}{ccccc} a & = & b & + & c \\ d & = & e & + & f \end{array} \rightarrow \quad \boxed{\begin{array}{c} a \\ d \end{array}} = \boxed{\begin{array}{c} b \\ e \end{array}} +_{SIMD} \boxed{\begin{array}{c} c \\ f \end{array}}$$

- Correctness of the transformation:
  Given a scalar program, the target program generated by the transformation is equivalent to the scalar program.
- Target program can run faster than the source program.

Vectorizing compiler transformation

Analyze the source program and when possible replace scalar instructions with SIMD instructions.

$$\begin{array}{ccccc} a & = & b & + & c \\ d & = & e & + & f \end{array} \rightarrow \boxed{\begin{array}{c} a \\ d \end{array}} = \boxed{\begin{array}{c} b \\ e \end{array}} +_{SIMD} \boxed{\begin{array}{c} c \\ f \end{array}}$$

- Correctness of the transformation:
  Given a scalar program, the target program generated by the transformation is equivalent to the scalar program.
- Target program can run faster than the source program.

Proof of correctness by input-output equivalence can be tedious.

Skipping refinement gives a “local” proof method.

Scalar Machine: Operational semantics

State

```
(defdata scalar-op (enum '(add sub mul ...)))

(defdata scalar-prog (listof scalar-inst))

(defdata sprg-state (record (pc . program-counter)
                            (regs . register-file)
                            (sprg . scalar-prog)))
```

Scalar Machine: Operational semantics

State

```
(defdata scalar-op (enum '(add sub mul ...)))

(defdata scalar-prog (listof scalar-inst))

(defdata sprg-state (record (pc . program-counter)
                            (regs . register-file)
                            (sprg . scalar-prog)))
```

Transition relation for deterministic scalar machine

```
(defun step-sprg (s)
  (let* ((inst (nth (sprg-state-pc s) (sprg-state-sprg s)))
         (op (inst-scalar-op inst))
         ...)
    (case op
      (add  (execute-add ... ))
      ... )))
```

Vector Machine: Operational semantics

State

```
(defdata vector-ops (enum '(vadd vsub vmul ...)))

(defdata inst (oneof scalar-inst vector-inst))

(defdata vector-prog (listof inst))

(defdata vprg-state (record (pc . program-counter)
                            (regs . register-file)
                            (vprg . vector-prog)))
```

Transition relation for deterministic vector machine

```
(defun step-vprg (s)
  (let* ((inst (nth (vprg-state-pc s) (vprg-state-vprg s)))
         (op (get-op inst))
          ... )
    (case op
      (add  (execute-add ...))
      (vadd (execute-vadd ...))
      ... )))
```

Vector machines refines scalar machine

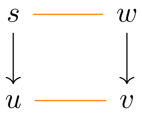
Refinement map

```
(defun ref-map (s)
  (let* ((rf (vprg-state-regs s))
         (vprg (vprg-state-vprg s))
         (vprg-pc (vprg-state-pc s))
         (sprg-pc (pcT (1- vprg-pc) vprg)))
         (sprg-state sprg-pc
                     rf
                     (scalarize-vprg vprg))))
```

`pcT` maps value of the vector machine's program counter to the corresponding value of the scalar machine's program counter.

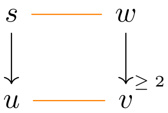
Vector Machines Refines Scalar Machine

Define $B = \{(s, w) |\ \mathrm{w} = \texttt{(ref-map s)}\}$.



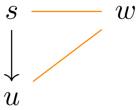
one step

$\langle \exists v \colon w \rightarrow v \colon uBv \rangle$



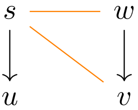
skipping on right

$\langle \exists v : w \rightarrow^{\geq 2} v : uBv \rangle \rangle$



stuttering on left

$(uBw \wedge rankT(u, w) \prec rankT(s, w))$

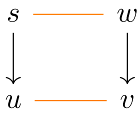


stuttering on right

$\langle \exists v \colon w \rightarrow v \colon sBv \wedge$
$rankL(v, s, u) < rankL(w, s, u) \rangle$

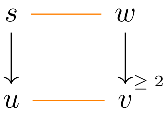
Vector Machines Refines Scalar Machine

Define $B = \{(s, w) | \text{ w} = \texttt{(ref-map s)}\}$.



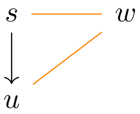
one step

$\langle \exists v \colon w \rightarrow v \colon uBv \rangle$



skipping on right

$\langle \exists v : w \rightarrow^{\geq 2} v : uBv \rangle \rangle$



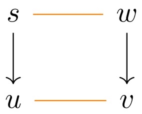
stuttering on left

$(uBw \wedge rankT(u, w) \prec rankT(s, w))$

`sprg` does not stutter

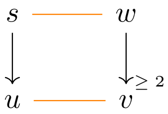
Vector Machines Refines Scalar Machine

Define $B = \{(s, w) |\ \mathrm{w} = \texttt{(ref-map s)}\}$.



one step

$\langle \exists v \colon w \rightarrow v \colon uBv \rangle$



skipping on right

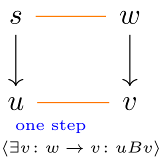
$\langle \exists v : w \rightarrow^{\geq 2} v : uBv \rangle \rangle$

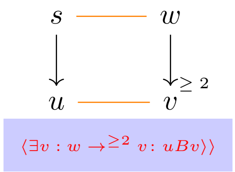
`vprg` does not stutter

`sprg` does not stutter

Vector Machines Refines Scalar Machine

Define $B = \{(s, w) |\ \mathrm{w} = \texttt{(ref-map s)}\}$.



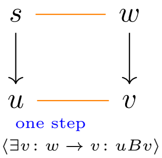


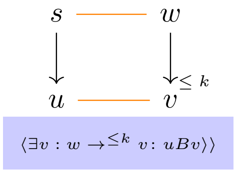
`vprg` does not stutter

`sprg` does not stutter

Vector Machines Refines Scalar Machine

Define $B = \{(s, w) |\ \mathrm{w} = \texttt{(ref-map s)}\}$.





`vprg` does not stutter

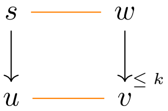
`sprg` does not stutter

An upper bound on skipping (k)

Maximum width of a vector instruction

Vector Machines Refines Scalar Machine

Final Theorem



bounded skipping on right

$\langle \exists v : w \rightarrow^{\leq k} v : uBv \rangle\rangle$

```
(defthm vprg-skip-refines-sprg
  (implies (and (vprg-statep s)
                (equal w (ref-map s))
                (equal u (step-vprg s)))
           (step-sprg-k-skip-rel w (ref-map u))))
```

Main lemmas

Let `s` be a `vprg-state`, `vpc` be the program counter in `s` and `inst` be the instruction pointed by `vpc` in `vprg`.

Let `w` = `(ref-map s)` and `spc` be the program counter in `w`.

- Lemma 1: If `inst` is a scalar instruction, then the corresponding instruction pointed by `spc` in `w` is also `inst`.
- Lemma 2: If `inst` is a vector instruction composed of k scalar instructions, say $s_0, \ldots, s_{k-1}$, then the corresponding instruction pointed by `spc + i` in `w` is s_i, for $i \in [0, k-1]$.

Skipping refinement is amenable for mechanical reasoning.

- An a priori knowledge of upper bound on skipping avoids reasoning about unbounded reachability.
- The proof obligations can often be simplified based on domain specific knowlege.

Other case studies

- Optimized Memory Controller

```
(defthm optmemc-skip-refines-memc
  (implies (and (good-statep s)
                (equal w (ref-map s))
                (equal u (impl-step s))
                (not (and (equal w (ref-map u))
                          (< (rank u) (rank s)))))
           (spec-step-k-skip-rel w (ref-map u))))
```

- JVM-inspired stack machine

```
(defthm bstk-skip-refines-stk
  (implies (and (good-statep s)
                (equal w (ref-map s))
                (equal u (impl-step s))
                (not (and (equal w (ref-map u))
                          (< (rank u) (rank s)))))
           (spec-step-k-skip-rel w (ref-map u))))
```

- Same WFSK to analyze correctness of systems.

Other case studies

- Optimized Memory Controller

```
(defthm optmemc-skip-refines-memc
  (implies (and (good-statep s)
                (equal w (ref-map s))
                (equal u (impl-step s))
                (not (and (equal w (ref-map u))
                          (< (rank u) (rank s)))))
           (spec-step-k-skip-rel w (ref-map u))))
```

- JVM-inspired stack machine

```
(defthm bstk-skip-refines-stk
  (implies (and (good-statep s)
                (equal w (ref-map s))
                (equal u (impl-step s))
                (not (and (equal w (ref-map u))
                          (< (rank u) (rank s)))))
           (spec-step-k-skip-rel w (ref-map u))))
```

- Same WFSK to analyze correctness of systems.
- ACL2s automatically proves the theorem with *no additional lemmas* for buffer depth upto 3.

Conclusion

- A notion of refinement that directly accounts for skipping behavior in optimized reactive systems.
- A sound and complete proof method for reasoning about skipping refinement.
- Validated the proof method by mechanically reasoning correctness of three optimized systems with ACL2s.

Future Work

- Complete local characterization of skipping refinement.
- Compositionality of skipping refinement.
- Use GL-framework for finite state models of systems.
- Refinement-based testing framework.

Thank You