Proof Reduction of Fair Stuttering Refinement of Asynchronous Systems and Applications

Rob Sumners

Centaur Technology

ACL2 Workshop 2017

Motivation

- Hardware/software implementation systems attempt to optimize task execution:
  - break-up tasks into more manageable chunks..
  - ..schedule chunks for execution over time and resources
- Intuitive specification:
  - all tasks eventually complete..
  - ..with results consistent with atomic (as possible) task execution
- Assume specification defined as simpler system and show that the behaviors of the implementation are consistent with the specification.
  - Additional theorems or properties could be proven about the simpler specification system as needed..

Fair Stuttering Refinement

- Assume implementation and specification defined as systems and prove:
  - all *fair* runs of implementation map to *valid* runs of specification upto finite stutter:
    1. a run is *fair* if every task is eventually selected.
    2. a run is *valid* if every task is eventually selected AND changes state.
    3. specification either matches implementation or stutters.
  - A task which is selected must change state unless it is *blocked*
- *Refinement* compactly encapsulates safety and progress properties of the implementation.
- Unwieldy to prove properties on infinite runs directly..
- ..define functions and properties over single steps of a small number of tasks and derive results relating infinite runs.

Example: Bakery Algorithm

```
Algorithm  Bakery Task
1: $choosing \leftarrow$ 't
2: $temp \leftarrow shared.max$
3: $pos \leftarrow temp + 1$
4: if $(shared.max \leq temp)$ $shared.max \leftarrow pos$
5: $choosing \leftarrow$ 'nil
6: for every $task$ do
7:     wait if $task.choosing$
8:     wait if lex<$(task.pos, task.id, pos, id)$
9: ..critical section.. goto 1
```

Example: Bakery Specification

Algorithm Specification Task

```
$state \leftarrow$ 'interested
$state \leftarrow$ 'go if $task.state \neq$ 'go for all $task$
  ..critical section..
$state \leftarrow$ 'idle goto 1
```

- Ensures at most one task in critical section at any time..
  - A *fair run* does NOT ensure every task eventually reaches critical section.. BUT..
  - A *valid run* does ensure every task eventually reaches critical section!

Requirements for Refinement Proofs

1. Split step into an update function and blocking relation.
2. Prove that specification can match implementation
    - Specification can stutter a finite amount between steps
3. Prove that implementation has no deadlocks amongst tasks.
4. Prove that implementation has no starvation of tasks.
5. Prove sufficient conditions are invariant in implementation.

- Primary contribution is a theory that demonstrates *(fair stuttering) refinement* as a result of defining the necessary functions and proving these properties.

Bakery Algorithm: Update and Blocking

- Split step into update function and blocking relation:

```
1: $choosing \leftarrow$ 't
2: $temp \leftarrow shared.max$
3: $pos \leftarrow temp + 1$
4: if $(shared.max \leq temp)$ $shared.max \leftarrow pos$
5: $choosing \leftarrow$ 'nil
6: for every $task$ do
7:     wait if $task.choosing$
8:     wait if $\text{lex}{<}(task.pos, task.id, pos, id)$
9: $..critical\ section..$ goto 1
```

Bakery Algorithm: Blocking Relation

```
for every task do
    wait if task.choosing
    wait if lex<(task.pos, task.id, pos, id)
```

- Split task step into update and blocking relations..

```
(defun t-block (a b)
   (or (and (= (g :loc a) 5) (g :choosing b))
       (and (= (g :loc a) 6)
            (lex< (g :pos b) (ndx (g :id b))
                  (g :pos a) (ndx (g :id a))))))
```

Refinement Proof: Matching Specification-1

- Mapping Bakery Task states to 'idle, 'interested, and 'go:

```
1: $choosing \leftarrow$ 't
2: $temp \leftarrow shared.max$
3: $pos \leftarrow temp + 1$
4: if $(shared.max \leq temp)$ $shared.max \leftarrow pos$
5: $choosing \leftarrow$ 'nil
6: for every $task$ do
7:     wait if $task.choosing$
8:     wait if lex<$(task.pos, task.id, pos, id)$
9: ..critical section.. goto 1
```

Refinement Proof: Matching Specification-2

- Define `(t-map a)` and `(t-rank a)`:
  - `(t-map a)` maps a bakery task state to a specification task.
  - `(t-rank a)` returns ordinal decreases on bakery steps which are not matched in specification.
    - `t-rank` for 'interested states returns “distance” remaining to transition to 'go state
    - when specification match is blocked, then implementation must have been blocked..

```
(implies (and ... (t-next a b))
         (if (equal (t-map a) (t-map b))
             (o< (t-rank b) (t-rank a))
           (and (spec-next (t-map a) (t-map b))
                (implies (spec-block (t-map a) (t-map c))
                         (t-block a c)))))
```

Refinement Proof: Ensuring No Deadlocks

```
for every task do
    wait if task.choosing
    wait if lex<(task.pos, task.id, pos, id)
```

- Ensuring lack of deadlock: define a rank which decreases when one task blocks another..

```
  (defun t-nlock (a)
    (make-ord 2 (if (g :choosing a) 1 2)
    (make-ord 1 (1+ (nfix (g :pos a)))
                (ndx (g :id a)))))
....
  (thm (implies (and ... (t-block a b))
                (o< (t-nlock b) (t-nlock a)))
```

Refinement Proof: Ensuring No Starvation - 1

```
for every task do
    wait if task.choosing
    wait if lex<(task.pos, task.id, pos, id)
```

- Ensuring No Starvation: first define a predicate which defines when a task can no longer be blocked by another task..

```
  (defun t-noblk (a b)
    (or (and (!= (g :loc a) 5) (!= (g :loc a) 6))
        (and (not (g :choosing b))
             (> (g :pos b) (g :pos a)))))
....
  (thm (implies (and .. (t-next b c) (t-noblk a b))
                (and (not (t-block a b))
                     (t-noblk a c))))
```

Refinement Proof: Ensuring No Starvation - 2

```
for every task do
    wait if task.choosing
    wait if lex<(task.pos, task.id, pos, id)
```

- Ensuring No Starvation: ..and then define a rank which decreases until we reach `t-noblk` state.

```
  (defun t-nstrv (a b)
    ... "distance" from task state b to reach a state where
    ... b is no longer choosing and b.pos greater than a.pos)
....
  (thm (implies (and .. (t-next b c)
                        (not (t-noblk a b))
                        (not (t-noblk a c)))
                   (bnl< (t-nstrv a c) (t-nstrv a b) ..))
```

Refinement Proof: Prove Sufficient Conditions are Invariant

- For the sake of this paper.. no magic here.. we have to define an invariant which:
  - Implies the conditions sufficient to prove the other properties..
  - ..and is *inductive* – holds on initial states and across steps.
- For the Bakery.. the invariants were fairly straightforward properties relating task positions, code locations, and the shared variables..
  - ..but nonetheless relatively substantial compared to the other definitions and proofs

Comparison to Previous Efforts..

- Previous efforts at proving concurrent program refinements:

```
''Specification and Verification of Concurrent
  Programs Through Refinements''
-- S. Ray and R. Sumners, J. Autom. Reasoning, 2013
```

- In comparison, the previous efforts...
  - Supported more general forms of system definition with less assumptions.
  - Required bolting definition of specific fairness and progress tracking apparatus onto the system state.
  - Used simpler refinement properties, but required more complex rank functions and more components in invariants.
  - Muddled correctness of specification by need to review correctness of measures for fairness and progress.
  - Did not facilitate efficient finite-state property checking.

Further Considerations, Questions.

- This is one step along the path.. to take it further:
    - Relaxing system definition requirements?
        - For example, allowing synchronous task updates?
    - Efficiently reducing to finite-state checks?
        - Can we break properties down into smaller theorems, GL/GLMC checks
    - Many other considerations...
- Rump Session: Efficient Checking of Fair Stuttering Refinements of Finite State Systems in ACL2!

Questions?