Rump Session: Efficient Checking of Fair Stuttering Refinements of Finite State Systems in ACL2

Rob Sumners

Centaur Technology

ACL2 Workshop 2017

Quick Review-1:

- Reviewing: "Proof Reduction of Fair Stuttering Refinement of Asynchronous Systems and Application"..
- Define specification and implementation as systems and *refinement* proof as goal.
  - *Refinement* encapsulates progress and correlation to specification while allowing abstraction of time and state details in specification.
- Reduce *refinement* proof to properties of single steps of a small number of tasks
  - Uses definition of blocking relation and additional definitions demonstrating absence of deadlock and starvation.

Quick Review-2:

- Limitations:
  - Placed requirements on system definitions which may be poor match for certain implementations.
    - Task updates were assumed to be asynchronous with a single task updating each step.
    - Task blocking was assumed to be a summation of potential blocks per task.
  - Required additional definitions of auxiliary predicates and ranking functions for progress.
  - Required invariant to be strengthened to an inductive invariant.
- Now to address these limitations.. and improve automation in finite-state cases.

Goals:

- Relax definition restrictions:
  - Allow synchronous updates of tasks via user specification.
  - Reduce definitional requirements on blocking relations.
  - Remove strict correlation of progress and change in task state.
  - Some other minor improvements (e.g. fewer structural assumptions of the task and system states)
- Establish checking procedures for finite-state systems:
  - Assuming fixed set of task IDs and finite task state set, split proof requirements into a large number of GL checks
  - When viable, reduce definitional requirements by transferring GL checks into GLMC checks

Supporting synchronous task updates

- User defines a selection set recognized by `(sel-p u)` which replaces task-id as parameter for next-state function/relation
- User defines predicate `(is-go k u)` which returns whether a task id `k` can update on selection id `u`
  - Also requires definition of `(id-sel k)` which ensures: `(thm (implies (...) (is-go k (id-sel k))))`
- Fully Synchronous: `(is-go k u) = 't`
- Fully Asynchronous: `(is-go k u) = (member k u)`
- Task Async. (as before): `(is-go k u) = (equal k u)`
- Limitation: stateless.. any "state" required for defining task update selection would need to be part of system state.

Remove strict correlation of progress and task state change

- Previous assumption: tasks made progress if and only if the task state changed.
  - Unfortunately, this precludes any cycles in task states that do not map to cycles in specification.
- Change: define separate notion of task "progress" by mapping task states to some progress label:
  - Prove that this label is preserved in the mapping to specification states.
  - Define ranking function which decreases until progress label changes when warranted.
  - We use simple instance by defining predicate `(actv x k)` – all active tasks eventually complete.
- Downside: the guarantee of progress is less clear in the specification and requires review of progress labels.

Reduce restrictions on blocking relations

- We assumed blocking based per-task: `(t-block a b)`
  - Task `a` was blocked iff `(t-block a b)` for some other task `b`.
  - This can be limiting.. e.g. if a task is blocked by two other tasks existence but not by each individually.
- Split needs of blocking relation into definitions of `(block x k)` and `(t-block x k l)`..
  - `(block x k)` defines when task `k` is blocked in state `x`.
  - `(t-block x k l)` defines when the blocking of task `k` involves task `l` in part...
    - `t-block` used to build rankings and properties which are relating specific tasks.
  - `(block x k)` must imply `(t-block x k l)` for some `l`.
- Similar split can be done in the case of `(noblk k x)` and `(t-noblk k l x)` but less likely to be useful.

Finite State Checking Automation using GL - 1

- Systems defined by:
  - `(init x)` – initial state predicate on state `x`
  - `(next x u j)` – next-state function takes state `x`, selector `u`, and free input `j`
  - `(actv x k)` – predicate returning if task `k` is active in state `x`
- In addition.. predicates relating to blocking: `(block x k)`, `(t-block x k l)`, `(t-noblk k l x)`..
- ...as well... refinement proof support functions such as invariants and ranking functions.
- Assume task-id set and selector set are fixed and finite and that task state space is finite.. Can we use GL to efficiently relieve proof obligations?

Finite State Checking Automation using GL - 2

- Take required refinement properties and generate instances appropriate for proof in GL.
  - Use user-defined functions to build explicit sets for enumerated variables and shape specs for symbolic variables.
- For example:

```
(defthm t-nstrv-decreases
  (implies (and (key-p k)
                (key-p l)
                (selp u)
                (iinv x)
                (block x k)
                (not (t-noblk k l x))
                (not (t-noblk k l (next x u j))))
           (bnl<< (t-nstrv k l (next x u j))
                  (t-nstrv k l x)
                  (nst-bnd)
                  (implies (is-go l u) (block x l)))))
```

Finite State Checking Automation using GL - 3

- For this theorem, we generate a DEF-GL-CHECK macro instance which is a make-event spawning instances of the property to be checked as def-gl-thms:

```
(DEF-GL-CHECK T-NSTRV-DECREASES
  :ENUM ((K (ENUM-VAL* ....))
         (L (ENUM-VAL* .. K ..))
         (U (ENUM-VAL* .. K L ..)))
  :VARS ((X (VAR-SH8P* .. K L U ..))
         (J (VAR-SH8P* .. K L U ..)))
  :FILTER 'T
  :DEBUG (M8K-DBUG* .. K L U X J ..)
  :DO-NOT-RANDOMIZE NIL
  :PROP (IMPLIES (AND (INV* X NAME K L U)
                      (BLOCK X K)
                      (NOT (T-NOBLK K L X))
                      (NOT (T-NOBLK K L (NEXT X U J))))
                 (BNL<< (T-NSTRV K L (NEXT X U J))
                        (T-NSTRV K L X)
                        (NST-BND)
                        (IMPLIES (IS-GO L U) (BLOCK X L)))))
```

Reducing Definition Requirement using GLMC

- When viable, we can significantly reduce definition requirements using model checking via GLMC.
  - Recent addition made by Sol Swords which allows export of finite-state invariant checks to an external model checker.

1. Invariant definitions do not need to be strengthened to be inductive..
   - Generate GLMC checks to show required invariants hold on reachable states
   - Use assume-guarantee to break up invariant check into smaller checks.
2. Ranking functions (e.g. `t-nlock`, `t-nstrv`, and `t-rank`) can be constructed..
   - Build a model check which fails on existence of certain bad cycles.
   - A passing check then ensures a topological sort of the state from which a ranking can be constructed.

Much more stuff to improve on..

- Structured way to automate proof of representatives for enumerated instances per property:
  - User defines representative mapping for enumerations.
  - Generate checks that representative returns same result..
  - ..and only generate checks for the representative enumerations.
- Generating definition of `block` and `t-block` from next-state:
  - Have an approach specifically for systems defined as SVEX from VL to SV in use at Centaur.. would like to generalize.
  - Would also like to generate definitions for `noblk` and `t-noblk`.. but this seems rather difficult without domain-specific assumptions.
- Structured way to lift step correlation from implementation to specification as a GLMC check.

Applications, Questions, Answers..

- Current Applications – finite state versions of previous work:
  - Concurrent programs: Bakery Algorithm, Concurrent Deque
  - Cache Coherence: German Coherence Protocol, TSO-CC
- Ongoing work: Verifying correctness of memory operations for RTL at Centaur.
  - Uses VL/SV/SVEX compilation from Verilog RTL to build implementation definitions.

Questions?