Partial Clock Functions in ACL2

John Matthews and Daron Vroon
ACL2 Workshop 2004
Goals

- *Given a state machine, we want:*
  - A termination proof: from a set of starting states, a desired goal state will always eventually be reached.
  - An efficient simulator: a function that steps machine until desired goal state is reached
  - Modularity: Be able to compose subroutine proofs and simulators
Goals

• **We don't want to:**
  • write a VCG (verification condition generator)
  • manually define a clock function
  • specify assertions or ordinal measures for every instruction in the subroutine
  • add a clock parameter to the simulator

• **Related work:**
  • First three conditions above met for partial correctness [Moore 2003]
  • First two conditions above met for total correctness [Ray & Moore 2004]
State machine model

- **State tuple**: represents current machine state
  - Defined as a stobj
  - Program, program counter are part of the state
    ```lisp
    (defstobj mstate
        (mem  :type (array (signed-byte 32) (1024))
        (progc :type integer)
        ...)
    ```
- **“next state” function**: executes one machine step
  ```lisp
  next : mstate => mstate
  ```
State machine model

- **Machine simulator (with clock parameter):** Executes machine for \( n \) steps
  - Returns current state if \( n \) is bogus

```
(defun run (n mstate)
  (declare (xargs :stobjs (mstate)
                   :guard (natp n)))
  (if (zp n)
      mstate
      (let ((mstate (next mstate))
            (let ((run (1- n) mstate)))
```

```
State machine model

- **State assertion**: predicate about a machine state

(defun entering-fib-routine (n mstate)
  (and (program-loaded *fib-addr* mstate)
       (equal (progc mstate)
              *fib-addr*)
       (equal (top-of-stack mstate) n))

(defun exiting-fib-routine (n mstate)
  (and (program-loaded *fib-addr* mstate)
       (equal (progc mstate)
              *fib-done-addr*)
       (equal (top-of-stack mstate) (fib n)))
State machine model

- **Cutpoints**: Finite collection of state assertions
  - Every program loop should be broken by at least one cutpoint
- **Exitpoint**: Desired end state assertion
  - Every exitpoint must be a cutpoint
  - Multiple exitpoints allowed
  - Exitpoints aren’t necessarily halting
- **Internal cutpoint**: A cutpoint that is not an exitpoint
Termination proof

- **Total correctness**: Every cutpoint always leads to an exitpoint.

- **Proof method**:
  - Assign an ordinal measure to every cutpoint
    \[
    \text{cutpoint-measure : mstate} \rightarrow \text{ordinal}
    \]
  - Symbolically simulate each control path from an internal cutpoint until another cutpoint is reached
  - Show that the newly-reached cutpoint is smaller according to cutpoint-measure
Symbolic simulation

• Symbolic simulation automated via a **partial clock function**
  • Has a generic, tail-recursive definition
  • Returns number of steps (- n) until next valid cutpoint state, if one is reachable
  • Undefined if no cutpoint state is reachable
  • Can be made “Executable”

```
(defun steps-to-cutpoint-tail (n mstate)
  (if (at-cutpoint mstate)
      n
      (steps-to-cutpoint-tail (+ n) (next mstate))))
```
Completed clock function

- Partial clock function is logically extended to a total function:
  - Tests whether value returned by steps-to-cutpoint-tail is a cutpoint:
    - If so, then return that value
    - If not, then return $\omega$

(defun steps-to-cutpoint (mstate)
  (let ((steps (steps-to-cutpoint-tail 0 mstate)))
    (if (at-cutpoint (run steps mstate))
      steps
      (omega))))
Clock function rewrites

- Completed clock function has simpler rewrite rules
- Rules use ordinal addition to handle unreachable cutpoints

(defun steps-to-cutpoint-zero
  (implies (at-cutpoint mstate)
    (equal (steps-to-cutpoint mstate) 0)))

(defun steps-to-cutpoint-nonzero-intro
  (implies (not (at-cutpoint mstate))
    (equal (steps-to-cutpoint mstate)
      (o+ 1
        (steps-to-cutpoint (next mstate))))))
Symbolic simulation

• Check termination by symbolically simulating machine, from each internal cutpoint to its next reachable cutpoint

(implies (and (at-cutpoint mstate)
             (not (at-exitpoint mstate)))
       (let* ((steps (steps-to-cutpoint (next mstate)))
              (cutpoint (run steps mstate))
              (and (at-cutpoint cutpoint)
                   (o< (cutpoint-measure cutpoint)
                        (cutpoint-measure mstate))))))

• But then machine gets simulated twice per internal cutpoint!
  • Once to compute number of steps to next cutpoint
  • Second time to compute next cutpoint’s state tuple
Symbolic simulation

- Solution: use clock function to define a **next-cutpoint** function
  - Returns next cutpoint, if it is reachable
  - Returns a non-cutpoint value, otherwise

```
(defun next-cutpoint (mstate)
  (let ((steps (steps-to-cutpoint mstate)))
    (if (natp steps)
        (run steps mstate)
        nil)))
```
Symbolic simulation

• Next-cutpoint function agrees with machine simulator...
   (thm
    (implies (at-cutpoint (next-cutpoint mstate))
     (equal (next-cutpoint mstate)
          (run (steps-to-cutpoint mstate) mstate)))))

...and still obeys good symbolic simulation rules
(deftm next-cutpoint-at-cutpoint
   (implies (at-cutpoint mstate)
     (equal (next-cutpoint mstate)
           mstate)))

(deftm next-cutpoint-intro-next
   (implies (not (at-cutpoint mstate))
     (equal (next-cutpoint mstate)
           (next-cutpoint (next mstate))))))
Symbolic simulation

- Now termination check symbolically simulates machine only once per internal cutpoint.

\[(\text{implies} \ (\text{and} \ (\text{at-cutpoint} \ mstate) \n(\text{not} \ (\text{at-exitpoint} \ mstate))) \n(\text{let} \ ((\text{cutpoint} \ (\text{next-cutpoint} \ (\text{next} \ mstate)))) \n(\text{and} \ (\text{at-cutpoint} \ \text{cutpoint}) \n(\text{o<} \ (\text{cutpoint-measure} \ \text{cutpoint}) \n(\text{cutpoint-measure} \ mstate)))))]
Termination

- Can now define function to count steps from cutpoint to next exitpoint

(defun steps-to-exitpoint-from-cutpoint (mstate)
  (declare (xargs :measure (cutpoint-measure mstate)))
  (cond
    ((not (at-cutpoint mstate))
      0)
    ((at-exitpoint mstate)
      0)
    (t
      (+ 1 (steps-to-cutpoint (next mstate))
           (steps-to-exitpoint-from-cutpoint
            (next-cutpoint (next mstate)))))))
Termination

- Main termination theorem:

(defthm total-correctness-from-cutpoint
 (implies (at-cutpoint mstate)
          (at-exitpoint
           (run (steps-to-exitpoint-from-cutpoint mstate) mstate))))
Efficient simulator

- Goal 2: Define an executable machine simulator function that doesn’t use a step counter
  - Simulator returns the first reachable exitpoint state
  - Simulator guard: input state must be a cutpoint
Efficient simulator

- Defining the simulator:
  - First define a cutpoint simulator, that steps the machine from one cutpoint to the next cutpoint
  - Main simulator calls cutpoint simulator until exitpoint is reached
  - Use cutpoint measure to prove termination
- Main challenge: stobj syntactic restrictions
Stobj restrictions

- Want to use `steps-to-cutpoint` in guards, but not execute them:

  ```lisp
  :guard (at-cutpoint
           (run (steps-to-cutpoint mstate) mstate))
  ```

- Problem: ACL2 requires guards to be executable
  - Difficult to make guards stobj-compliant
  - This definition doesn’t work, since `defpun` not stobj-compliant:

  ```lisp
  (defun steps-to-cutpoint (mstate)
    (declare (xargs :stobjs (mstate)))
    (let ((steps (steps-to-cutpoint-tail 0 mstate)))
      (if (at-cutpoint (run steps mstate))
          steps
          (omega))))
  ```
Stobj restrictions

- Need to write coercion functions between stobjs and ACL2 values

```
logical-mstatep : * => bool
copy-from-mstate : mstate => *
copy-to-mstate : (* mstate) => mstate

(defthm copy-from-mstate-correct
  (implies (mstatep mstate)
    (equal (copy-from-mstate mstate) mstate)))

(defthm copy-to-mstate-correct
  (implies (and (mstatep mstate)
                (logical-mstatep copy))
    (equal (copy-to-mstate copy mstate) copy)))
```
Stobj restrictions

- Next problem: guards are not allowed to modify stobjs

\begin{verbatim}
(defun steps-to-cutpoint (mstate)
  (declare (xargs :stobjs (mstate)))
  (let* ((mstate-copy (copy-from-mstate mstate))
         (steps
          (steps-to-cutpoint-tail 0 mstate-copy)))
    (if (at-cutpoint (run steps mstate))
        steps
        (omega)))))
\end{verbatim}

- "ACL2 value" version of run requires "ACL2 value" next
- Basically need to redefine the entire machine semantics
Stobj restrictions

• Solution: create a `with-copy-of-stobj` macro
  • allocates a local copy of stobj object
  • Executes a stobj-compliant `mv-let` form on the local copy
    • Discards the `mv-let`’s final stobj
    • Returns the `mv-let`’s final value

• Modified `steps-to-cutpoint` function is now stobj-compliant
  • Can be used in guards
  • ACL2 runtime error if executed (but still sound)
Efficient simulator

- Clockless simulator, useful for cutpoint-induction proofs:
  - next-cutpoint-exec defined with stobj-compliant guard
  - called by cutpoint simulator cutpoint-to-cutpoint-exec
- Main simulator calls cutpoint simulator until exitpoint

```lisp
(defun next-exitpoint-exec (mstate)
  (declare (xargs :stobjs (mstate)
                   :measure (cutpoint-measure mstate)
                   :guard (at-cutpoint mstate)))
  (if (mbt (at-cutpoint mstate))
      (if (at-exitpoint mstate)
          mstate
          (let ((mstate (cutpoint-to-cutpoint-exec mstate)))
            (next-exitpoint-exec mstate)))
      (dummy-mstate mstate)))
```
Efficient simulator

- **Clockless simulator**, useful for efficient execution (not in supporting materials):

```lisp
(defun next-exitpoint-exec (mstate)
  (declare (xargs :stobjs (mstate)
                   :guard (cutpoint-reachable mstate)
                   :measure (steps-to-exitpoint mstate)))
  (if (mbt (and (mstatep mstate)
                   (cutpoint-reachable mstate)))
    (if (at-exitpoint mstate)
      mstate
      (let ((mstate (next mstate)))
        (next-exitpoint-exec mstate))))
```

Conclusions

- Partial clock functions and cutpoint symbolic simulation increase automation and robustness of termination proofs
- Termination proofs are modular, because exitpoints need not halt
- Possible to define efficient, clockless machine simulators
- Clockless stobj-compliant simulators will be easier to write when ACL2
  - allows nonexecutable guards
  - removes stobj syntax restrictions in logical portions of guards, mbt, and mbe macros
- In the meantime, a defstobj+ ACL2 book has been written:
  - Automatically creates stobj coercion functions & theorems
  - Includes with-copy-of-stobj macro