Internal Note #327
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776
July 30, 1996

The Specification of a Simple Autopilot in ACL2

William D. Young

Abstract

We translate the specification of a simple autopilot from PVS into ACL2. By the judicious use of ACL2 macros, the notation can be made quite similar to the PVS notation.

1 Introduction

In his report "An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot"[But96], Ricky Butler describes the translation of some informal requirements specifications into a formal model using PVS. Butler makes the contention that "the modeling techniques used in this paper can be used with other formal specification languages that *are based upon higher-order logic*" (emphasis added). In fact, there are no obvious higher-order features in Butler's specification and the entire specification can easily be translated to a first-order formalism such as ACL2.[1] To demonstrate that this is so, we translated Butler's specification into ACL2 in about two hours work.

The results of the translation were illuminating. We found that by using the available `def-structure` macros in ACL2 and adding some macros for enumerated types, we were able to define an ACL2 specification that is very much a simple transliteration of the PVS specification. Moreover, the proof

[1] In a private correspondence, Butler reminded me that record structures in PVS are represented as functions. Therefore, there are higher-order aspects hidden by the record syntax. However, these are easily handled in a first-order context, as by the ACL2 def-structure macro facility.

performance of ACL2 was apparently quite superior to that of PVS. Butler reported for one lemma that: "The PVS system replays the proofs in 192 secs on a Sparc 20 with 32 Megabytes of memory." He then showed a proof script that "reduces the execution time to 57.0 secs." ACL2 dispatched the same lemma in under 5 seconds on a Sparc 10 workstation." Moreover, the given proof script suggested fairly low-level proof manipulation. ACL2 dispatched all of the lemmas of the specification fully automatically.

Our attempt here is not to denigrate Butler's efforts. His paper is a worthwhile tutorial example of the translation of informal specifications into a formal language. However, the misleading impression that the higher-order functions of PVS contributed in some substantial way to this exercise should not go unchallenged.

In the following section, we describe the macro package we introduced for adding enumerated types to ACL2. These macros have some interest independent of the autopilot specification. In Section 3 we give the complete ACL2 specification, using the PVS version as comments.

2 Def-enumeration

The syntax of ACL2 is that of Common Lisp. However, macros make this syntax extremely malleable. For example, record structures can be easily represented in ACL2 using the available `def-structure` macros. We used `def-structure` to represent the state component of our specification.

Because the autopilot specification makes use of several enumerated types, we decided to add a simple facility for conveniently expressing enumerated types in ACL2. This is very rudimentary mechanism and could be extended to handle other aspects of enumerated types, such as the ordering relation between type elements. Nevertheless, defining and refining these macros was the most time consuming aspect of this project.

Our `def-enumeration` macro[2] is fully defined in file:

```
/cli/project/os/java/Model/def-enumeration.lisp.
```

Given a form:

```
(def-enumeration name symbol-list)
```

we introduce:

[2] We earlier defined a simpler version which treats the enumeration literals as symbols rather as new constant function symbols.

- a new constant function for each member of the type;
- a recognizer for elements of the type;
- a recognizer for each particular element of the type.

We also introduce the axiom that all elements of the type are pairwise distinct.

For example, corresponding to the PVS declaration:

```
md_status: TYPE = off, armed, engaged
```

we have the ACL2 declaration:

```
(def-enumeration md-status (off armed engaged)).
```

This expands into the following sequence of definitions:

```
(ENCAPSULATE
   ;; We introduce three new function symbols.
   ((OFF NIL T)
    (ARMED NIL T)
    (ENGAGED NIL T))
   ;; The witnesses for these functions
   (LOCAL (DEFUN OFF NIL 'OFF))
   (LOCAL (DEFUN ARMED NIL 'ARMED))
   (LOCAL (DEFUN ENGAGED NIL 'ENGAGED))
   ;;
   (DEFTHM MD-STATUS-ELEMENTS-UNIQUE
     ;; The enumeration literals are pair-wise
     ;; distinct.
     (AND (NOT (EQUAL (OFF) (ARMED)))
          (NOT (EQUAL (OFF) (ENGAGED)))
          (NOT (EQUAL (ARMED) (ENGAGED))))))

(DEFUN MD-STATUS? (X)
  (OR (EQUAL X (OFF))
      (EQUAL X (ARMED))
      (EQUAL X (ENGAGED))))

(DEFUN OFF? (X) (EQUAL X (OFF)))
(DEFUN ARMED? (X) (EQUAL X (ARMED)))
(DEFUN ENGAGED? (X) (EQUAL X (ENGAGED))))
```

This provides us with a very convenient syntax for discussing the values of enumeration constants. With our extended syntax, we can ask whether a particular variable `x` has value `off` by asking (off? x), and whether a `x` has one of the values `off`, `armed`, or `engaged` by asking `(md-status? x)`. This syntactic sugar makes our version of the autopilot specification more readable.

3 Our Specification

This section contains the complete ACL2 specification of the simple autopilot. It is annotated with comments, principally in the form of the PVS expressions that were translated to yield the resulting ACL2 forms.

```
; Defines the def-structure and def-enumeration macros

(include-book "/slocal/src/acl2/v1-8/books/public/structures")
(ld "/cli/project/os/java/Model/def-enumeration.lisp")

; THE BASIC TYPES

; md_status: TYPE = off, armed, engaged

(def-enumeration md-status (off armed engaged))

; off_end: TYPE = md: md_status | md = off OR md = engaged

(defun off-eng? (x)
  (and (md-status? x)
       (or (off? x) (engaged? x))))

; disp_status: TYPE = pre_selected, current

(def-enumeration disp-status (pre-selected current))

; altitude_vals: TYPE = away, near_pre_selected, at_pre_selected

(def-enumeration altitude-vals
   (away near-pre-selected at-pre-selected))

; events: TYPE = press_att_cws, press_cas_eng, press_fpa_sel,
;                press_alt_eng, input_alt, input_fpa, input_cas,
```

```
;                alt_reached, alt_gets_near, fpa_reached

(def-enumeration events
   (press-att-cws press-cas-eng press-fpa-sel press-alt-eng
    input-alt input-fpa input-cas alt-reached alt-gets-near
    fpa-reached))

; DEFINITION OF STATE

; states: TYPE = [#  % RECORD
;                 att_cws: off_eng,
;                 cas_eng: off_eng,
;                 fpa_sel: off_eng,
;                 alt_eng: md_status,
;                 alt_disp: disp_status,
;                 fpa_disp: disp_status,
;                 cas_disp: disp_status,
;                 altitude: altitude_vals
;                #]

(defstructure states
  (att-cws    (:assert (off-eng? att-cws)))
  (cas-eng    (:assert (off-eng? cas-eng)))
  (fpa-sel    (:assert (off-eng? fpa-sel)))
  (alt-eng    (:assert (md-status? alt-eng)))
  (alt-disp   (:assert (disp-status? alt-disp)))
  (fpa-disp   (:assert (disp-status? fpa-disp)))
  (cas-disp   (:assert (disp-status? cas-disp)))
  (altitude   (:assert (altitude-vals? altitude)))
  (:options (:conc-name nil)
            (:keyword-updater update-state)
            :guards))

; AN ASIDE

; We need to enrich the defstructure theory so that any expression
; of the form:
;    (structure-access-xx (set-structure-field-y y-val struct))
; reduces appropriately.

; NOTE: the structures theory easily could be enhanced to generate
; these reductions automatically.

(defthm states-accessors-update-reductions
```

```
  ; takes 7-8 seconds to prove
  (and ; att-cws-reductions
       (equal (att-cws (update-state st :att-cws x))
              x)
       (equal (cas-eng (update-state st :att-cws x))
              (cas-eng st))

       ... 61 cases omitted ...

       (equal (altitude (update-state st :altitude x))
              x))
  :hints (("Goal" :in-theory
           (union-theories
            (current-theory :here)
            (theory 'defs-states-definition-theory)))))

; Another feature that we would like to have in defstructure is the
; ability to assure that the type assertions on the fields are
; preserved by each call to update-state.  That is, we want the
; following predicate to be shown an invariant:

(defun states-slots-typed (st)
  (and (weak-states-p st)
       (off-eng? (att-cws st))
       (off-eng? (cas-eng st))
       (off-eng? (fpa-sel st))
       (md-status? (alt-eng st))
       (disp-status? (alt-disp st))
       (disp-status? (fpa-disp st))
       (disp-status? (cas-disp st))
       (altitude-vals? (altitude st))))

; The way this can be used is shown by the lemma
; tran-att-cws-states-slots-typed below.

; THE SEMANTICS OF SYSTEM EVENTS

; tran_att_cws(st): states =
;   IF att_cws(st) = off THEN
;     st WITH [att_cws := engaged, fpa_sel := off, alt_eng := off,
;              alt_disp := current, fpa_disp := current]
;     ELSE st
;   ENDIF
```

```
(defun tran-att-cws (st)
  (if (off? (att-cws st))
      (update-state st
                    :att-cws  (engaged)
                    :fpa-sel  (off)
                    :alt-eng  (off)
                    :alt-disp (current)
                    :fpa-disp (current))
    st))

; It is possible to prove an analogous lemma for each of our
; operations below.

(defthm tran-att-cws-states-slots-typed
  (implies (states-slots-typed st)
           (states-slots-typed (tran-att-cws st))))

; tran_cas_eng(st): states =
;   IF cas_eng(st) = off THEN
;     st WITH [cas_eng := engaged]
;   ELSE
;     st WITH [cas_eng := off, :cas_disp := current]
;   ENDIF

(defun tran-cas-eng (st)
  (if (off? (cas-eng st))
      (update-state st
                    :cas-eng (engaged))
    (update-state st
                  :cas-eng  (off)
                  :cas-disp (current))))

; tran_fpa_sel(st): states =
; IF fpa_sel(st) = off THEN
;   st WITH [fpa_sel := engaged, att_cws := off,
;            alt_eng := off, alt_disp := current]
; ELSE
;   st WITH [fpa_sel := off, fpa_disp := current,
;            att_cws := engaged, alt_eng := off,
;            alt_disp := current]
; ENDIF

(defun tran-fpa-sel (st)
  (if (off? (fpa-sel st))
```

```
      (update-state st
                    :fpa-sel  (engaged)
                    :att-cws  (off)
                    :alt-eng  (off)
                    :alt-disp (current))
    (update-state st
                  :fpa-sel  (off)
                  :fpa-disp (current)
                  :att-cws  (engaged)
                  :alt-eng  (off)
                  :alt-disp (current))))

; tran_alt_eng(st): states =
; IF alt_eng(st) = off AND alt_disp(st) = pre_selected THEN
;   IF altitude(st) /= away THEN
;     st WITH [att_cws := off, fpa_sel := off,
;              alt_eng := engaged, fpa_disp := current]
;   ELSE
;     st WITH [att_cws := off, fpa_sel := engaged,
;              alt_eng := armed]
;   ENDIF
; ELSE st
; ENDIF

(defun tran-alt-eng (st)
  (if (and (off? (alt-eng st))
           (pre-selected? (alt-disp st)))
      (if (not (away? (altitude st)))
          (update-state st
                        :att-cws  (off)
                        :fpa-sel  (off)
                        :alt-eng  (engaged)
                        :fpa-disp (current))
        (update-state st
                      :att-cws  (off)
                      :fpa-sel  (engaged)
                      :alt-eng  (armed)))
    st))

; tran_input_alt(st): states =
; IF alt_eng(st) = off THEN
;     st WITH [alt_disp := pre_selected]
; ELSE
;   IF alt_eng(st)) = armed OR alt_eng(st) = engaged THEN
```

```
;     st WITH [alt_eng := off, alt_disp := pre_selected,
;              att_cws := engaged, fpa_sel := off,
;              fpa_disp := current]
;     ELSE st
; ENDIF

(defun tran-input-alt (st)
  (if (off? (alt-eng st))
      (update-state st
                    :alt-disp (pre-selected))
    (if (or (armed? (alt-eng st))
            (engaged? (alt-eng st)))
        (update-state st
                      :alt-eng (off)
                      :alt-disp (pre-selected)
                      :att-cws (engaged)
                      :fpa-sel (off)
                      :fpa-disp (current))
      st)))

; tran_input_fpa(st): states =
; IF fpa_sel(st) = off THEN
;   st WITH [fpa_disp := pre_selected]
; ELSE st
; ENDIF

(defun tran-input-fpa (st)
  (if (off? (fpa-sel st))
      (update-state st
                    :fpa-disp (pre-selected))
    st))

; tran_input_cas(st): states =
; IF cas_eng(st) = off THEN
;   st WITH [cas_disp := pre_selected]
; ELSE st
; ENDIF

(defun tran-input-cas (st)
  (if (off? (cas-eng st))
      (update-state st
                    :cas-disp (pre-selected))
    st))
```

```
; tran_alt_gets_near(st): states =
; IF alt_eng(st)) = armed
;   st WITH [altitude := near_pre_selected, alt_eng := engaged,
;            fpa_sel := off, fpa_disp := current]
; ELSE
;   st WITH [altitude := near_pre_selected]
; ENDIF

(defun tran-alt-gets-near (st)
  (if (armed? (alt-eng st))
      (update-state st
                    :altitude (near-pre-selected)
                    :alt-eng (engaged)
                    :fpa-sel (off)
                    :fpa-disp (current))
    (update-state st :altitude (near-pre-selected))))

; tran_alt_reached(st): states =
; IF alt_eng(st) = armed THEN
;   st WITH [altitude := at_pre_selected, alt_disp := current,
;            alt_eng := engaged, fpa_sel := off,
;            fpa_disp := current]
; ELSE
;   st WITH [altitude := at_pre_selected, alt_disp := current]
; ENDIF

(defun tran-alt-reached (st)
  (if (armed? (alt-eng st))
      (update-state st
                    :altitude (at-pre-selected)
                    :alt-disp (current)
                    :alt-eng (engaged)
                    :fpa-sel (off)
                    :fpa-disp (current))
    (update-state st :altitude (at-pre-selected)
                  :alt-disp (current))))

; tran_fpa_reached(st): states = st WITH [fpa_disp := current]

(defun tran-fpa-reached (st)
  (update-state st :fpa-disp (current)))

; nextstate(st, event): states =
;   CASES event OF
```

```
;     press_att_cws:  tran_att_cws(st),
;     press_alt_eng:  tran_alt_eng(st),
;     press_fpa_sel:  tran_fpa_sel(st),
;     press_cas_eng:  tran_cas_eng(st),
;     input_alt:      tran_input_alt(st),
;     input_fpa:      tran_input_fpa(st),
;     input_cas:      tran_input_cas(st),
;     alt_reached:    tran_alt_reached(st),
;     fpa_reached:    tran_fpa_reached(st),
;     alt_gets_near:  tran_alt_gets_near(st)
;   ENDCASES

;; Notice that the choice of functions rather than symbols for
;; enumeration literals makes this a bit more complicated.

(defun nextstate (st event)
  (cond
   ((press-att-cws? event)  (tran-att-cws st))
   ((press-alt-eng? event)  (tran-alt-eng st))
   ((press-fpa-sel? event)  (tran-fpa-sel st))
   ((press-cas-eng? event)  (tran-cas-eng st))
   ((input-alt? event)      (tran-input-alt st))
   ((input-fpa? event)      (tran-input-fpa st))
   ((input-cas? event)      (tran-input-cas st))
   ((alt-reached? event)    (tran-alt-reached st))
   ((fpa-reached? event)    (tran-fpa-reached st))
   ((alt-gets-near? event)  (tran-alt-gets-near st))
   (t                        st)))

; LEGAL INITIAL STATES

; st0: states = (#
;                att_cws := engaged,
;                cas_eng := off,
;                fpa_sel := off,
;                alt_eng := off,
;                alt_disp := current,
;                fpa_disp := current,
;                cas_disp := current,
;                altitude := away
;               #) ;

(defun st0 ()
  ; Create a concrete initial state
```

```
  (make-states
   :att-cws  (engaged)
   :cas-eng  (off)
   :fpa-sel  (off)
   :alt-eng  (off)
   :alt-disp (current)
   :fpa-disp (current)
   :cas-disp (current)
   :altitude (away)))

;; This is necessary because of some weirdness in the expansion of
;; constrained constant functions.

(in-theory (disable (:executable-counterpart st0)))

; is_initial(st): bool =      att_cws(st) = engaged
;                         AND cas_eng(st) = off
;                         AND fpa_sel(st) = off
;                         AND alt_eng(st) = off
;                         AND alt_disp(st) = current
;                         AND fpa_disp(st) = current
;                         AND cas_disp(st) = current

(defun is-initial (st)
  (and (engaged? (att-cws st))
       (off? (cas-eng st))
       (off? (fpa-sel st))
       (off? (alt-eng st))
       (current? (alt-disp st))
       (current? (fpa-disp st))
       (current? (cas-disp st))))

(defthm st0-is-initial
  (is-initial (st0)))

; GOOD STATES

; Butler also defines this notion on p. 22 with the name
; "reachable?"
;
; good(st): bool =
;     (att_cws(st) = engaged OR fpa_sel(st) = engaged
;         OR alt_eng(st) = engaged) AND
;     (alt_eng(st) /= engaged OR fpa_sel(st) /= engaged) AND
```

```
;     (att_cws(st) = engaged IMPLIES
;         alt_eng(st) /= engaged AND fpa_sel(st) /= engaged) AND
;     (alt_eng(st) = armed IMPLIES fpa_sel(st) = engaged)

(defun good (st)
  (and (or (engaged? (att-cws st))
           (engaged? (fpa-sel st))
           (engaged? (alt-eng st)))
       (or (not (engaged? (alt-eng st)))
           (not (engaged? (fpa-sel st))))
       (implies (engaged? (att-cws st))
                (and (not (engaged? (alt-eng st)))
                     (not (engaged? (fpa-sel st)))))
       (implies (armed? (alt-eng st))
                (engaged? (fpa-sel st)))))

; st0_good: LEMMA good(st0)

(defthm st0-good
  (good (st0)))

; nextstate_good: THEOREM
;    good(st) IMPLIES good(nextstate(st,event))

(defthm nextstate-good
  (implies (good st)
           (good (nextstate st event)))
  :hints (("Goal" :in-theory
           (union-theories (current-theory :here)
                           (theory 'defs-states-lemma-theory)))))

; REACHABLE STATES

; We now turn to the notion of reachability introduced by
; Butler on p. 22.  This is not in the final specification
; in Appendix B of Butler's paper.  We use a different notion of
; reachability than he does.

; n: VAR nat
; ev: VAR events
; pst: VAR states
; reachable_in(n,st): RECURSIVE bool =
;          IF n = 0 THEN is_initial(st)
;            ELSE
```

```
;             (EXISTS pst, ev: st = nextstate(pst,ev) AND
;                              reachable_in(n-1,pst))
;          ENDIF MEASURE n
;
; is_reachable(st): bool = (EXISTS n: reachable_in(n,st))

(defun nextstate* (st eventlist)
  ; This is the extension of nextstate to a list of
  ; events.
  (if (consp eventlist)
      (nextstate* (nextstate st (car eventlist))
                  (cdr eventlist))
    st))

(defun-sk reachable? (start end)
  ; A state end is reachable from state start if
  ; there is a sequence of events that takes you
  ; from start to end.
  (exists (eventlist)
    (equal end (nextstate* start eventlist))))

(defthm nextstate*-preserves-good
  (implies (good start)
           (good (nextstate* start eventlist)))
  :hints (("Goal" :induct (nextstate* start eventlist)
                  :in-theory (disable good nextstate))))

(defthm nextstate*-preserves-good-corollary
  (implies (and (equal end (nextstate* start eventlist))
                (good start))
           (good end))
  :hints (("Goal" :use nextstate*-preserves-good
                  :in-theory (disable good nextstate))))

; Butler has built-in the notion that the starting state must be
; initial, and hence good.
;
; reachable_good: THEOREM is_reachable(st) IMPLIES good(st)

(defthm reachable?-good
  (implies (and (reachable? start end)
                (good start))
           (good end))
  :hints (("Goal" :in-theory (disable good nextstate*))))
```

```
(defun-sk is-reachable? (end)
  (exists (start)
    (and (good start)
         (reachable? start end))))

(defthm is-reachable?-good
  (implies (is-reachable? end)
           (good end))
  :hints (("Goal" :in-theory (disable good nextstate))))

; TWO SAFETY PROPERTIES

; safety1: THEOREM reachable?(st) AND fpa_sel(st) = engaged AND
;                  fpa_sel(nextstate(st,event)) = off IMPLIES
;                     fpa_disp(nextstate(st,event)) = current

(defthm safety1
  (implies (and (is-reachable? st)
                (engaged? (fpa-sel st))
                (off? (fpa-sel (nextstate st event))))
           (current? (fpa-disp (nextstate st event)))))

; safety2: THEOREM reachable?(st) AND alt_eng(st) = engaged AND
;                  event /= input_alt AND
;                  alt_eng(nextstate(st,event)) = off IMPLIES
;                    alt_disp(nextstate(st,event)) = current

(defthm safety2
  (implies (and (is-reachable? st)
                (engaged? (alt-eng st))
                (not (input-alt? event))
                (off? (alt-eng (nextstate st event))))
           (current? (alt-disp (nextstate st event)))))
```

References

[But96] Ricky W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. Technical Memorandum 110255, NASA, May 1996.