Finite-Domain Temporal Logic in ACL2: a semantics-based approach

Matteo Pradella

CNR IEIIT-MI
Dipartimento di Elettronica e Informazione
Politecnico di Milano
via Ponzio 34/5
20133 Milano, ITALY
pradella@elet.polimi.it

September 23, 2002

1 Introduction

This very sketchy document briefly presents an exercise written for the ACL2 theorem prover [4]. It introduces an encoding of a past/future linear temporal logic [1], based on a finite temporal domain semantics (presented in [2] for the logic language TRIO [3]).

2 The Encoding

The main idea behind the encoding is to see every logic proposition as a function defined on a history (i.e. a finite interpretation), and an instant. More precisely, a logic propostion is translated into a function $f : H, Z \rightarrow \{true, false, undefined\}$ being the finite-domain semantics based on a three-valued logic, with H the set of all finite interpretations, and Z the set of integer numbers.

2.1 Basic Elements

Definition 1. *Temporal Slice*: A temporal slice is a list of predicates/propositions. e.g. `(push (p 0))` is a temporal slice in which *push* and $p(0)$ hold.

Definition 2. *History*: A history (or finite temporal interpretation) h is a list of temporal slices. The first slice (i.e. `(car h)` in ACL2/Lisp notation) refers to temporal instant $i = 0$, the second (i.e. `(cadr h)`) to instant $i = 1$, and so on.

Example: a history

```
(setq hist
 '((timeout push) ; i=0
   (push)         ; i=1
   (push)         ; i=2
```

```
    (push)          ; i=3
    (push)          ; i=4
    (push)          ; i=5
    (push)          ; i=6
  ))
```

Definition 3. *Temporal Domain*: Let h be a history. Then its temporal domain (also called T) is the range of naturals: $[0..|h|-1]$.

Definition 4. *Formula*: Let H be the set of all finite histories and Z the set of integer numbers. Then, a *formula* is a function $f : H, Z \rightarrow \{t, nil, undef\}$, where t stands for $True$, nil stands for $False$, and $undef$ stands for $Undefined$.

The following functions/macros support the above definitions:

```
; Sup(T)
(defmacro sup-t (h)
  `(1- (length ,h)))

; is h a history (i.e. a list of lists)?
(defun k-histp (h)
  (cond
   ((not (true-listp h)) nil)
   ((null h) t)
   (t
    (and (true-listp (car h))
         (k-histp (cdr h))))))

; returns the temporal slice of h at time i
(defun at-inst (h i)
  (cond
   ((>= i (length h)) 'undef)
   ((< i 0) 'undef)
   (t (nth i h))))

; does the predicate 'pred' hold in h at i?
(defun k-atom (pred h i)
  (let ((hi (at-inst h i)))
    (cond
     ((equal hi 'undef) 'undef)
     (t (not (equal nil (member pred hi)))))))
```

2.2 Basic three-valued logic operators

The finite domain semantics presented in [2] is based on Kleene's three valued logic [5]. We need to distinguish *strict* from *loose* operators: loose propositional operators are used for defining unbound temporal operators, while strict ones are more suitable for expressing bounded temporal operators.

```
; Kleene's three valued propositional operators

(defun koolp (x)
  (not (null (member x '(t nil undef)))))

(defun k-and (a b) ; strict
  (cond
   ((null a) nil)
   ((null b) nil)
   ((equal a 'undef) 'undef)
   ((equal b 'undef) 'undef)
   (t t)))

(defun k-and-loose (a b)
  (cond
   ((null a) nil)
   ((null b) nil)
   ((equal a t) t)
   ((equal b t) t)
   (t 'undef)))

(defun k-not (a)
  (cond
   ((equal a 'undef) 'undef)
   (t (not a))))

(defun k-or (a b) ; strict
  (cond
   ((equal a t) t)
   ((equal b t) t)
   ((equal a 'undef) 'undef)
   ((equal b 'undef) 'undef)
   (t nil)))

(defun k-or-loose (a b)
  (cond
   ((equal a t) t)
   ((equal b t) t)
   ((null a) nil)
   ((null b) nil)
   (t 'undef)))

(defun k-implies (x y)
  (cond
   ((null x) t)
```

```
   ((equal y t) t)
   ((and (equal x t)
         (null y)) nil)
   (t 'undef)))

(defun k-iff (x y)
  (cond
   ((equal x 'undef) 'undef)
   ((equal y 'undef) 'undef)
   (t (equal x y))))
```

The following simple theorems on the three-valued propositional operators were introduced to check their definitions, and are automatically proved by ACL2.

```
(defthm comm-k-and
  (equal (k-and x y) (k-and y x))
  :rule-classes nil)

(defthm ass-k-and
 (equal (k-and x (k-and y z))
        (k-and (k-and x y) z))
 :rule-classes nil)

(defthm k-demorgan
 (equal (k-and x y)
        (k-not (k-or (k-not x) (k-not y))))
 :rule-classes nil)

(defthm comm-k-and-loose
 (equal (k-and-loose x y)
        (k-and-loose y x))
 :rule-classes nil)

(defthm ass-k-and-loose
 (equal (k-and-loose x (k-and-loose y z))
        (k-and-loose (k-and-loose x y) z))
 :rule-classes nil)

(defthm k-demorgan-loose
 (implies (and (koolp x) (koolp y))
          (equal (k-and-loose x y)
                 (k-not (k-or-loose (k-not x) (k-not y)))))
 :rule-classes nil)

(defthm k-implies-def
```

```
 (implies (and (koolp x) (koolp y))
          (equal (k-implies x y)
                 (k-or y (k-not x))))
 :rule-classes nil)

(defthm k-iff-def
 (implies (and (koolp x) (koolp y))
          (equal (k-iff x y)
                 (k-and (k-implies x y)
                        (k-implies y x))))
 :rule-classes nil)
```

2.3 Temporal operators

Temporal operators are introduced following the classical TRIO approach: TRIO formulae are constructed in the usual inductive way, starting from terms and atomic formulae. Besides the usual propositional operators and the quantifiers, one may compose TRIO formulae by using a single basic modal operator, called $Dist$, that relates the *current time*, which is left implicit in the formula, to another time instant: the formula $Dist(F, t)$, where F is a formula and t a term indicating a time distance, states that F holds at a time instant at t time units from the current instant.

A note on $Dist$: in our encoding, $Dist(p, t)$ is represented as $p(h, i+t)$, where i is the current time instant and h is a history.

The following table summarizes the TRIO definition of some of the most used temporal operators.

Operator	Definition
$Som(F)$	$\exists d\ Dist(F, d)$
$Alw(F)$	$\neg Som(\neg F)$
$SomF(F)$	$\exists d\ (d > 0 \wedge Dist(F, d))$
$SomP(F)$	$\exists d\ (d > 0 \wedge Dist(F, -d))$
$AlwF(F)$	$\neg SomF(\neg F)$
$AlwP(F)$	$\neg SomP(\neg F)$
$Lasts(F, t)$	$\forall d\ (0 < d < t \rightarrow Dist(F, d))$
$Lasted(F, t)$	$\forall d\ (0 < d < t \rightarrow Dist(F, -d))$
$WithinF(F, t)$	$\neg Lasts(\neg F, t)$
$WithinP(F, t)$	$\neg Lasted(\neg F, t)$
$Until(F, G)$	$\exists d\ (d > 0 \wedge Lasts(F, d) \wedge Dist(G, d))$
$Since(F, G)$	$\exists d\ (d > 0 \wedge Lasted(F, d) \wedge Dist(G, -d))$
$UpToNow(F)$	$\exists d\ (d > 0 \wedge Lasted(F, d))$; $Dist(F, -1)$ if T is not dense
$NowOn(F)$	$\exists d\ (d > 0 \wedge Lasted(F, d))$; $Dist(F, 1)$ if T is not dense
$Becomes(F)$	$F \wedge UpToNow(\neg F)$

ACL2 macros are a convenient way of representing derived operators.

For example, $Lasted(on, k)$ can be defined as:

```
; proposition 'on'
(defun on (h i)
  (k-atom 'on h i))

; Lasted(on, k)
(defun lasted_on (h i k)
  (declare (xargs
            :measure (past-op-measure h i)))
  (lasted_ii lasted_on on h i k))
```

The previous example uses a macro called `lasted_ii`. Function `pasted-op-measure` is used to tell ACL2 how to prove its termination for every input.

```
; measures for future/past temporal operators
(defun futr-op-measure (h i)
  (acl2-count (- (length h) i)))

(defun past-op-measure (h i)
  (acl2-count (- (- (length h)) i)))
```

Here is the complete list of macro definitions:

```
(defmacro alwf_i (rec f h i) ; AlwF(f)
  `(if (or
        (not (integerp ,i))
        (not (k-histp ,h))
        (< i 0)
        (> i (sup-t ,h)))
       'undef
     (k-and-loose (,f ,h ,i) (,rec ,h (1+ ,i)))))

(defmacro somf_i (rec f h i) ; SomF(f)
  `(if (or
        (not (integerp ,i))
        (not (k-histp ,h))
        (< i 0)
        (> i (sup-t ,h)))
       'undef
     (k-or-loose (,f ,h ,i) (,rec ,h (1+ ,i)))))

(defmacro alwp_i (rec f h i) ; AlwP(f)
  `(if (or
        (not (integerp ,i))
        (not (k-histp ,h))
        (< i 0)
```

```
        (> i (sup-t ,h)))
       'undef
     (k-and-loose (,f ,h ,i) (,rec ,h (1- ,i)))))

(defmacro somp_i (rec f h i) ; SomP(f)
  `(if (or
        (not (integerp ,i))
        (not (k-histp ,h))
        (< i 0)
        (> i (sup-t ,h)))
       'undef
     (k-or-loose (,f ,h ,i) (,rec ,h (1- ,i)))))

(defmacro until (rec a b h i) ; Until(a,b)
  `(if (or
        (not (integerp ,i))
        (not (k-histp ,h))
        (< i 0)
        (> i (sup-t ,h)))
       'undef
     (k-or-loose
      (,b ,h ,i)
      (k-and (,a ,h ,i) (,rec ,h (1+ ,i))))))

(defmacro since (rec a b h i) ; Since(a,b)
  `(if (or
        (not (integerp ,i))
        (not (k-histp ,h))
        (< i 0)
        (> i (sup-t ,h)))
       'undef
     (k-or-loose
      (,b ,h ,i)
      (k-and (,a ,h ,i) (,rec ,h (1- ,i))))))

(defmacro lasts_ii (rec f h i n) ; Lasts(f,n)
  `(cond
    ((or
      (not (integerp ,i))
      (not (k-histp ,h))
      (< i 0)
      (> i (sup-t ,h)))
     'undef)
    ((equal ,n 0)
     (,f ,h ,i))
```

```
    (t
     (k-and (,f ,h ,i) (,rec ,h (1+ ,i) (1- ,n))))))

(defmacro withinf_ii (rec f h i n) ; WithinF(f,n)
  `(cond
    ((or
      (not (integerp ,i))
      (not (k-histp ,h))
      (< i 0)
      (> i (sup-t ,h)))
     'undef)
    ((equal ,n 0)
     (,f ,h ,i))
    (t
     (k-or (,f ,h ,i) (,rec ,h (1+ ,i) (1- ,n))))))

(defmacro lasted_ii (rec f h i n) ; Lasted(f,n)
  `(cond
    ((or
      (not (integerp ,i))
      (not (k-histp ,h))
      (< i 0)
      (> i (sup-t ,h)))
     'undef)
    ((equal ,n 0)
     (,f ,h ,i))
    (t
     (k-and (,f ,h ,i) (,rec ,h (1- ,i) (1- ,n))))))

(defmacro withinp_ii (rec f h i n) ; WithinP(f,n)
  `(cond
    ((or
      (not (integerp ,i))
      (not (k-histp ,h))
      (< i 0)
      (> i (sup-t ,h)))
     'undef)
    ((equal ,n 0)
     (,f ,h ,i))
    (t
     (k-or (,f ,h ,i) (,rec ,h (1- ,i) (1- ,n))))))

(defmacro becomes (f h i) ; Becomes(f)
  `(k-and (,f ,h ,i)
          (k-not (,f ,h (1- ,i)))))
```

3 An Example: a lamp with a timer

Let us consider a lamp with a timer and a switch. When the lamp is off, pushing the button of the switch turns the lamp on. When the lamp is on, pushing the button turns the lamp off; moreover if the button is not pushed the lamp is turned off anyway by a timer after 4 time units. In the specification, the state of the lamp is modelled by the time dependent predicate *on*, which is true iff the lamp is on; the event of pushing the button is modelled by the time dependent predicate *push*. Finally, another time dependent predicate *timeout*, models the timer of the lamp. The specification of this system is the formula $Alw(A1 \wedge A2 \wedge A3)$, where $A1$, $A2$ and $A3$ are the following formulae:

A1: $timeout \leftrightarrow Lasted(on, 5)$.
A2: $Becomes(on) \leftrightarrow push \wedge Dist(\neg on, -1)$.
A3: $Becomes(\neg on) \leftrightarrow (push \wedge Dist(on, -1) \vee timeout)$.

$A1$ defines the timeout: the light has been on during the last 4 time units[1].

$A2$ states that the lamp becomes on iff the button is pushed and the light was off.

$A3$ states that the lamp becomes off iff either there is a timeout or the button is pushed while the light was on.

The translation formula to function(s) procedes in a bottom-up fashion, starting from propositions and predicates.

3.1 Definition of *A1, A2*, and *A3*

```
(include-book "trio-book/TRIO")
(in-package "TRIO")

(defun on (h i)
  (k-atom 'on h i))

(defun not-on (h i)
  (k-not (k-atom 'on h i)))

(defun timeout (h i)
  (k-atom 'timeout h i))

(defun lasted_on (h i k)
  (declare (xargs
            :measure (past-op-measure h i)))
  (lasted_ii lasted_on on h i k))

(defun a1 (h i)
```

[1] The definition of *Lasted* does not consider the current time instant, hence $Lasted(on, 5)$ requires *on* to be true in the previous four time instants.

```
  (k-iff
   (k-atom 'timeout h i)
   (lasted_on h (1- i) 4)))

(defun a2 (h i)
  (k-iff
   (becomes on h i)
   (k-and (k-atom 'push h i)
          (k-not (on h (1- i))))))

(defun a3 (h i)
  (k-iff
   (becomes not-on h i)
   (k-or
    (k-atom 'timeout h i)
    (k-and
     (k-atom 'push h i)
     (on h (1- i))))))
```

3.2 Encoding *Always* and trying the spec

We can express operators that are both past and future using the following technique:

```
; Alw(a1 & a2 & a3):

(defun a1a2a3 (h i) ; a1 & a2 & a3
  (k-and (a1 h i)
         (k-and (a2 h i)
                (a3 h i))))

(defun alwf_i-a1a2a3 (h i)
  (declare (xargs
            :measure (futr-op-measure h i)))
  (alwf_i alwf_i-a1a2a3 a1a2a3 h i))

(defun alwp_i-a1a2a3 (h i)
  (declare (xargs
            :measure (past-op-measure h i)))
  (alwp_i alwp_i-a1a2a3 a1a2a3 h i))

(defun alw-a1a2a3 (h i)
  (k-and-loose (alwf_i-a1a2a3 h i)
               (alwp_i-a1a2a3 h (1- i))))
```

Now we have indeed an interpreter for our simple specification:

```
(alw-a1a2a3
 '((push on) (on) (on) (on) (timeout) nil (push on)
   (on) (on) (push) nil)
 2)
```

Actually, the function obtained from the translation is nothing more than a Common Lisp implementation of the specification. This could be useful, e.g. for validation or testing, because it can directly exploit the usually quite efficient Common Lisp compiler.

3.3 A simple theorem

As a simple example, we can automatically prove a theorem.

Theorem 5. *Let h be a history, and i a time instant. If $A1 \wedge A2 \wedge A3$ holds at instant i in h, then $\neg timeout \wedge \neg push \rightarrow \neg Becomes(\neg on)$ holds.*

In ACL2:

```
(defthm lamp-th1
  (implies
   (k-histp h)
   (implies
    (equal (a1a2a3 h i) t)
    (equal
     (k-implies
      (k-and
       (k-not (k-atom 'timeout h i))
       (k-not (k-atom 'push h i)))
      (k-not
       (becomes not-on h i)))
     t)))
  :rule-classes nil)
```

It is proved in much less than one minute on an average PC running ACL2 under Linux.

An interesting characteristic of the previous theorem is that holds for every finite interpretation (and time instant).

Quite naturally, this approach could be used (with a probably greater effort) to prove much more complex properties of a specification. A typical proof approach, which comes directly form the encoding technique, could be based on induction on the history h.

References

1. E. Allen Emerson. Temporal and Modal Logic. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics 1990, J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, Pages 995-1072.

2. A. Coen-Porisini, M. Pradella, P. San Pietro, A finite domain semantics for testing temporal logic specifications, FTRTFT'98 Symposium Proceedings (Eds. A.P. Ravn and H. Rischel), LNCS Springer Verlag, September 1998, p 41-54
3. C. Ghezzi, D. Mandrioli, and A. Morzenti: TRIO, a logic language for executable specifications of real-time systems. Journal of Systems and Software 12, 2 (May 1990), 107-123.
4. M. Kaufmann and J. Moore, An Industrial Strength Theorem Prover for a Logic Based on Common Lisp, IEEE Transactions on Software Engineering 23(4), April 1997, pp. 203-213
5. A. Urquhart: Many valued Logic. D. Gabbay and F, Guenthner (eds), Handbook of Philosophical Logic, Vol. III, Kluwer, London, 1986.