Appendix C

Source code

This appendix contains ACL2 scripts and Common Lisp programs encoding concepts and algorithms described in this dissertation.

C.1 fd.lisp

```
;*****************************************************************
;*       BASIC DEFINITIONS OF EVENT RECONSTRUCTION THEORY
;*****************************************************************

(in-package "ACL2")

; ======================== SYSTEM MODEL ============================

(encapsulate (((cp *) => *)
              ((psi *) => *)
              ((wc *) => *)
              ((dp *) => *)
              ((sid * *) => *)
              ((emp *) => *)
              ((int * *) => *)
              ((uni * *) => *)
              ((rev *) => *)
              ((fwd *) => *)
              (($) => *))

; System model is defined by three functions: cp(), psi() and wc()
;
; cp(c) => {t/nil}  is true for "proper" computations, and false for all
;                     other objects. The set of all proper computations
;                     is closed under psi(x), i.e. (cp x) <=> (cp (psi x))
;
; psi(c0) => c1       is a transition function that defines exactly one
;                     successor computation for every computation
;                     (either proper or not) c0. If input sequence of
;                     c0 is empty, c1 == c0.
;
; wc(c1) => c0        is a witness function that returns one of
;                     predecestor computations for the given
;                     computation c1.
```

```
;                    psi(wc(c1)) = c1.

(local (defun cp (c) (if (equal c 'comp-universe) nil t)))
(local (defun psi (c0) c0))
(local (defun wc (c1) c1))

(defthm cp-is-boolean
  (booleanp (cp c))
  :rule-classes :type-prescription)

(defthm psi-wc-cancel
  (equal (psi (wc c)) c))

; set of proper computations is closed under psi

(defthm cp-psi-cp-x
  (equal  (cp (psi x)) (cp x)))

; =============== DESCRIPTIONS OF COMPUTATION SETS ==================

; There are eight functions for manipulating descriptions of
; computation sets.
;
; dp (d) => {t/nil}    is true for all proper descriptions and
;                      false otherwise. Sets of proper descriptions and
;                      proper computations are disjoint:
;                      (dp x) => (not (cp x))
;
; $()                  this constant function denotes description
;                      of all proper computations C_T. Note that
;                      (dp ($)) and (cp x) <=> (sid x ($))
;
; sid (c d) =>{t/nil}  is true iff given computation
;                      satisfies given description.
;                      Only proper computations can be described:
;                      (sid c d) => (comp c)
;
; emp (d) => {t/nil}   is true iff there are
;                      no computations satisfying given
;                      description
;
; int (d1 d2) => d3    returns description of intersection
;                      of computation sets described by d1 and d2
;
; uni (d1 d2) => d3    returns description of union
;                      of computation sets described by d1 and d2
;
; fwd (d0) => d1       returns description of
;                      all immediate successors
;                      of computations in d0.
;
; rev (d1) => d0       returns description of all
;                      immediate predecestors
;                      of computaitons in d1.

; ----- base function witnesses -----

(local (defun dp (d) (if (equal d 'comp-universe) t nil)))

(local (defun $ () 'comp-universe))

(local (defun sid (c d)
         (if (cp c) (if (equal d 'comp-universe) t nil) nil)))

(defthm dp-is-boolean
  (booleanp (dp d))
  :rule-classes :type-prescription)
```

```
(defthm sid-is-boolean
  (booleanp (sid c d))
  :rule-classes :type-prescription)

(defthm $-is-description
  (dp ($)))

(defthm $-contains-all-and-only-comps
  (equal (sid c ($)) (cp c)))

(defthm descr-contain-only-comps
  (implies (sid c d) (cp c)))

(defthm descs-arent-comps
  (implies (dp x) (not (cp x))))

; ----- description manipulation function witnesses -----

(local (defun emp (d) (declare (ignore d)) nil))

(defthm emp-is-boolean
  (booleanp (emp d))
  :rule-classes :type-prescription)

(defthm emp-is-vacuous
  (implies (emp d)
           (not (sid c d))))

(defthm $-is-not-empty
  (not (emp ($))))

(local (defun int (d1 d2)
         (if (equal d1 ($)) d2 (if (equal d2 ($)) d1 nil))))

(defthm int-semantics
  (equal (sid c (int d1 d2))
         (and (sid c d1)
              (sid c d2))))

(in-theory (disable int-semantics))

(local (defun uni (d1 d2)
         (if (or (equal d1 ($)) (equal d2 ($))) ($) nil)))

(defthm uni-semantics
  (equal (sid c (uni d1 d2))
         (or (sid c d1)
             (sid c d2))))

(in-theory (disable uni-semantics))

(local (defun fwd (d0) d0))

(defthm fwd-semantics
  (equal (sid c (fwd d))
         (sid (wc c) d)))

(local (defun rev (d1) d1))

(defthm rev-semantics
  (equal (sid c (rev d))
         (sid (psi c) d)))
)

; ================== FORMALISATION OF EVIDENCE ====================
```

```
; 1. RUN
;
; A sequence of computations is a run if every computation in it
; (except the first computation) is the successor of the
; immediately preceding computation
;
; Runs are represented by a lists. This representation is
; appropriate, because event reconstruction only deals with
; finite runs.

(defun runp-tail (ctail c0)
   (if (atom ctail)
     t
     (and (equal (car ctail) (psi c0))
          (runp-tail (cdr ctail) (car ctail)))))

(defun runp (c) (runp-tail (cdr c) (car c)))


; 2. OBSERVATION

; An observation is a triple (p m o), where p is
; a description of observed property (set of computations
; possessing that property), m and o are natural
; numbers.
;
; To simplify formalisation, p, m and o are allowed to be
; arbitrary objects in ACL2 universe, assuming that
; non-descriptions represent empty sets, and that non-naturals
; represent 0.

; Thus, a proper observation is just a triple

(defun observp (ob)
    (= 3 (len ob)))

; A run satisfies an observation if
;
; 1) all its elements satisfy p

(defun list-in-desc (l p)
  (if (atom l)
    t
    (and (sid (car l) p)
         (list-in-desc (cdr l) p))))

; 2) if length L of the run us such that  m <= L <= (m+o)

(defun rio (r observ)
  (let ((p (car observ))
        (m (car (cdr observ)))
        (o (car (cdr (cdr observ)))))

     (and (runp r)
          (observp observ)
          (list-in-desc r p)
          (<= (nfix m) (len r))
          (<= (len r) (+ (nfix o) (nfix m))))))


; 3. PARTITIONED RUN

; To define a partitoined run we need a funciton that
; concatenates elements of list in the order of listing

(defun con (l)
  (if (atom l)
```

```
    nil
    (append (car l) (con (cdr l)))))

; A partitioned run is a list such that concatenation of
; its elements gives a run

(defun partitionp (r)
  (runp (con r)))

; Every element of a partitioned run is a run

(defun list-of-runs (l)
  (if (atom l)
    t
    (and (runp (car l))
         (list-of-runs (cdr l)))))

(defthm partition-list-of-runs
  (implies (partitionp r)
           (list-of-runs r)))


; 4. OBSERVATION SEQUENCE

; An observation sequence is a list of observations.

(defun obsp (s)
  (if (atom s)
      t
      (and (observp (car s))
           (obsp (cdr s)))))

; Observations in observation sequence are listed in temporal order.
; Thus, a partitioned run (rl) explains observation sequence (s) if their
; lengths are equal and every element of partitioned run explains corresponding
; element of observation sequence

(defun runlst-in-obs (rl s)
  (cond ((atom rl)(atom s))
        ((atom s)(atom rl))
        (t (and (rio (car rl) (car s))
                (runlst-in-obs (cdr rl) (cdr s))))))

(defun pio (r s)
  (and (partitionp r)
       (runlst-in-obs r s)))


; 6. SEQUENCE OF PARTITIONED RUNS

; A sequence of partitioned runs is list whose elements partition the same run

(defun plistp-t (r pl)
  (if (atom pl)
    t
    (and (partitionp (car pl))
         (equal r (con (car pl)))
         (plistp-t r (cdr pl)))))

(defun plistp (pl)
  (plistp-t (con (car pl)) pl))

; 5. EVIDENTIAL STATEMENT

; An evidential statement is a list of observation sequences.

(defun esp (e)
```

```
  (if (atom e)
      t
      (and (obsp (car e))
           (esp (cdr e)))))

; A sequence of partitioned runs (v) explains evidential statement (e) if
; their lengths are equal and
; 1) all elements of the sequence of partitioned runs partition the same run,
;    i.e. (con v_0) = (con v_1) = ... = (con v_n) = u
; 2) every partitioned run explains corresponding
;    observation sequence in the evidential statement

(defun vie-tail (u v e)
  (cond ((atom v)(atom e))
        ((atom e)(atom v))
        (t (and (equal (con (car v)) u)
                (pio (car v) (car e))
                (vie-tail u (cdr v) (cdr e))))))

; function vie(v e) calculates u = (con v_0) and uses
; vie-tail(u v e) to check that v explains e

(defun vie (v e)
  (vie-tail (con (car v)) v e))
```

C.2 util.lisp

```
;********************
;* Helper functions
;********************

; append suff to the end of each element in lst
(defun combine (lst suff)
  (if (atom lst)
    nil
    (cons (cons (car lst) suff)
  (combine (cdr lst) suff))))

; make list of all possible pairwise
; combinations of the  elements of two lists
(defun product (l1 l2)
  (if (atom l2)
    nil
    (append (combine l1 (car l2)) (product l1 (cdr l2)))))

; convert every element of list into a singleton list
(defun listify-elements (l)
  (if (atom l)
    nil
    (cons (cons (car l) nil) (listify-elements (cdr l)))))

; similar to combine, but uses append instead of cons-ing
; and listifies the result
(defun combine-append (lst suff)
  (if (atom lst)
    nil
    (cons (cons (append (car lst) suff) nil)
  (combine-append (cdr lst) suff))))

; prepends each element in the given gist with the given prefix
(defun prepend (pref lst)
   (if (atom lst)
     nil
     (cons (cons pref (cons (car lst) nil)) (prepend pref (cdr lst)))))

(defun zp (x)
  (if (integerp x) (eq x 0) t))

(defun take (n lst)
  (if (zp n)
    nil
    (cons (car lst) (take (- n 1) (cdr lst)))))

; defconst macro for use in ordinary Common Lisp environment
(defmacro defconst (x y) `(defvar ,x ,y))
```

C.3 rec.lisp

```
;*****************************
;* Event reconsturciton algorithm
;*****************************

; proper computation predicate

(defun cp (c)
  (if (atom c)
    t
    (if (atom (cdr c))
       (and (null (cdr c))
            (eventp (caar c))
    (statep (cadar c)))
      (and (eventp (caar c))
   (statep (cadar c))
           (equal (st (caar c) (cadar c)) (cadadr c))
   (cp (cdr c))))))


(defvar *ALL-SINGLE-STEP-PATT*
          (listify-elements *ALL-EVENT-STATE-PAIRS*))


;
; partial list description language
;
; "true" lists denote themselves
; "untrue" lists denote patterns -- sets of lists

; proper description -- a list of patterns

(defun dp (d)
  (if (atom d)
    t
    (and (cp (car d))
 (dp (cdr d)))))

; a pattern is a special kind of computation. It is assumed
; that a pattern denotes all computations that begin with it.
; the following function checks if given computation c
; matches given pattern p.

(defun matches (c p)
  (if (atom p)
    t
    (if (atom c)
      nil
      (and (equal (car c) (car p))
   (matches (cdr c) (cdr p))))))

; a computation matches description if it matches one of the
; patterns

(defun in (c d)
  (if (atom d)
    nil
    (or (matches c (car d))
(in c (cdr d)))))

; Intersection of two descriptions

(defun intpp (p1 p2)
  (if (matches p1 p2)
    p1
    (if (matches p2 p1)
```

```
      p2
      nil)))

(defun intpd (p d acc)
  (if (atom d)
    acc
    (let ((i (intpp p (car d))))
      (if (null i)
        (intpd p (cdr d) acc)
        (intpd p (cdr d) (cons i acc))))))

(defun intdd-helper (d1 d2 acc)
  (if (atom d1)
    acc
    (intdd-helper (cdr d1) d2 (intpd (car d1) d2 acc))))

(defun intdd (d1 d2)
  (intdd-helper d1 d2 nil))

; union of two descriptions

(defun uindd (d1 d2)
  (append d1 d2))

; test for emptiness of a description

(defun emp (d) (atom d))

; single-step reverser

(defun revcomp (c)
  (if (atom c)
    *ALL-SINGLE-STEP-PATT*
   (combine (rev-st (cadar c)) c)))

(defun rev (lst)
  (if (atom lst)
    nil
    (append (revcomp (car lst))
            (rev (cdr lst)))))

; multi-step reverser

(defun revers (os d)
  (if (emp d)
    nil
    (if (atom os)
      d
      (revers (cdr os) (intdd (car os) (rev d))))))


; convert observation into list of single-step observations

(defun single-step-obs (p n)
  (if (zp n)
    nil
    (cons p (single-step-obs p (1- n)))))

(defun single-step-os (fos)
  (if (atom fos)
    nil
    (append (single-step-obs (first (car fos))
                             (second (car fos)))
            (single-step-os (cdr fos)))))


; convert generic observation sequence into equivalent
```

```
; list of fixed-length observation sequences

(defun fix-obs (p min opt)
  (if (zp opt)
    (cons
       (cons p (cons min (cons opt nil)))
       nil)
    (cons
       (cons p (cons (+ min opt) (cons 0 nil)))
       (fix-obs p min (1- opt)))))

(defun fix-os (os)
  (if (atom os)
    (cons nil nil)
    (product (fix-obs (first (car os))
                      (second (car os))
                      (third (car os)))
             (fix-os (cdr os)))))


; solving observation sequence

(defun fos-lengths (fos)
  (if (atom fos)
    nil
    (cons (second (car fos)) (fos-lengths (cdr fos)))))

(defun fos-total-len (fos-length-list)
   (if (atom fos-length-list)
     0
     (+ (car fos-length-list)
        (fos-total-len (cdr fos-length-list)))))

(defun solve-fix-os (fos)
  (cons (cons (fos-lengths fos)
         (cons (revers (reverse (single-step-os fos)) '(nil)) nil)) nil))

(defun solve-fix-os-list (fix-os-list)
  (if (atom fix-os-list)
    nil
    (append (solve-fix-os (car fix-os-list))
            (solve-fix-os-list (cdr fix-os-list)))))

(defun solve-os (os)
  (solve-fix-os-list (fix-os os)))

(defun select-comps (l acc)
  (if (atom l)
    acc
    (if (cp (car l))
      (select-comps (cdr l) (cons (car l) acc))
      (select-comps (cdr l) acc))))

;(comp 'select-comps)

(defvar *ALL-TWO-STEP-PATT*
  (select-comps
    (product *ALL-EVENT-STATE-PAIRS*
             *ALL-SINGLE-STEP-PATT*) nil))


(defmacro defpatt1-helper (const-name tester-name vars body)
  `(progn
     (defun ,tester-name (comp-list acc)
       (if (atom comp-list)
         acc
         (if ((lambda ,vars ,body) (caar comp-list))
```

```
          (,tester-name (cdr comp-list) (cons (car comp-list) acc))
          (,tester-name (cdr comp-list) acc))))

     (defvar ,const-name
       (,tester-name *ALL-SINGLE-STEP-PATT* nil))))

(defmacro defpatt1 (name vars body)
  (let
    ((tester-name
      (intern
       (concatenate 'string (symbol-name name) "-TESTER"))))
    `(defpatt1-helper ,name ,tester-name ,vars ,body)))

(defmacro defpatt2-helper (const-name tester-name vars body)
  `(progn
     (defun ,tester-name (comp-list acc)
       (if (atom comp-list)
         acc
         (if ((lambda ,vars ,body)
                  (caar comp-list)
                  (cadar comp-list))
           (,tester-name (cdr comp-list)
                         (cons (car comp-list) acc))
           (,tester-name (cdr comp-list) acc))))

     (defvar ,const-name
       (,tester-name *ALL-TWO-STEP-PATT* nil))))

(defmacro defpatt2 (name vars body)
  (let
    ((tester-name
      (intern
       (concatenate 'string (symbol-name name) "-TESTER"))))
    `(defpatt2-helper ,name ,tester-name ,vars ,body)))


; intersecting solutions of two observation sequences


(defun singleton-es-chunk (os-chunk)
   (cons (cons (car os-chunk) nil)
      (cons (cadr os-chunk) nil)))

(defun singleton-es-sol (os-sol)
   (if (atom os-sol)
     nil
     (cons (singleton-es-chunk (car os-sol))
           (singleton-es-sol (cdr os-sol)))))

(defun add-chunk (os-chunk es-chunk)
   (if (equal (fos-total-len (car os-chunk))
              (fos-total-len (caar es-chunk)))
     (let ((intersection
            (intdd (cadr os-chunk) (cadr es-chunk))))
       (if (emp intersection)
         nil
         (cons
          (cons (cons (car os-chunk) (car es-chunk))
                (cons intersection nil))
          nil)))
     nil))

(defun add-chunk-to-sol (es-sol os-chunk)
   (if (atom es-sol)
     nil
     (append (add-chunk os-chunk (car es-sol))
             (add-chunk-to-sol (cdr es-sol) os-chunk))))
```

```
(defun add-sol (os-sol es-sol)
   (if (atom os-sol)
     nil
     (append (add-chunk-to-sol es-sol (car os-sol))
             (add-sol (cdr os-sol) es-sol))))

(defun solve-es (es)
  (if (atom es)
    nil
    (if (atom (cdr es))
      (singleton-es-sol (solve-os (car es)))
      (add-sol (solve-os (car es)) (solve-es (cdr es))))))


; --- debugging tools ---

; stepper function
(defun stn (cl s)
  (if (atom cl)
    s
    (stn (cdr cl) (st (car cl) s))))
```

C.4 acme.lisp

```
;**************************
;* Printer analysis example
;**************************

(load "util")

; ======== Finite State Machine =======


; Proper state predicate

(defun valuep (v)
  (or (equal v 'empty)
      (equal v 'A)
      (equal v 'B)
      (equal v 'B_deleted)
      (equal v 'A_deleted)))

(defun statep (s)
  (and (equal (cdr (cdr s)) nil)
       (valuep (first s))
       (valuep (second s))))

; Proper event predicate

(defun eventp (e)
  (or (equal e 'add_A)
      (equal e 'add_B)
      (equal e 'take)))


; Set of all possible event-state pairs

(defconst *ALL-EVENT-STATE-PAIRS*
  (product
    '(take add_A add_B)
    (listify-elements (product
                        '(empty A B B_deleted A_deleted)
                        (listify-elements
                          '(empty A B B_deleted A_deleted))))))

; Transition funciton

(defun st (c s)
  (let ((d1 (first s))
(d2 (second s)))
    (cond ((equal c 'add_A)
             (if (or (equal d1 'A)
       (equal d2 'A))
       s
       (if (or (equal d1 'empty)
       (equal d1 'A_deleted)
       (equal d1 'B_deleted))
                 (list 'A d2)
 (if (or (equal d2 'empty)
 (equal d2 'A_deleted)
 (equal d2 'B_deleted))
   (list d1 'A)
   s))))
          ((equal c 'add_B)
              (if (or (equal d1 'B)
       (equal d2 'B))
 s
 (if (or (equal d1 'empty)
 (equal d1 'A_deleted)
```

```
 (equal d1 'B_deleted))
   (list 'B d2)
   (if (or (equal d2 'empty)
   (equal d2 'A_deleted)
   (equal d2 'B_deleted))
     (list d1 'B)
     s))))
   ((equal c 'take)
       (if (equal d1 'A)
 (list 'A_deleted d2)
 (if (equal d1 'B)
   (list 'B_deleted d2)
   (if (equal d2 'A)
     (list d1 'A_deleted)
     (if (equal d2 'B)
       (list d1 'B_deleted)
       s)))))
   (t s))))


; Inverse transition function

(defun rev-st (s)
  (let ((d1 (first s))
(d2 (second s)))
    (append
       (if (equal d1 'A_deleted)
         (list (list 'take (list 'A d2)))
 nil)
       (if (equal d1 'B_deleted)
 (list (list 'take (list 'B d2)))
 nil)
      (if (and (not (equal d1 'A))
       (not (equal d1 'B))
       (equal d2 'A_deleted))
 (list (list 'take (list d1 'A)))
 nil)
      (if (and (not (equal d1 'A))
       (not (equal d1 'B))
       (equal d2 'B_deleted))
 (list (list 'take (list d1 'B)))
 nil)
      (if (and (or (equal d1 'A_deleted)
   (equal d1 'B_deleted)
   (equal d1 'empty))
       (or (equal d2 'A_deleted)
   (equal d2 'B_deleted)
   (equal d2 'empty)))
(list (list 'take s))
nil)
      (if (and (equal d1 'A) (not (equal d2 'A)))
         (list (list 'add_A (list 'empty d2))
       (list 'add_A (list 'A_deleted d2))
              (list 'add_A (list 'B_deleted d2)))
 nil)
      (if (and (equal d1 'B) (not (equal d2 'B)))
         (list (list 'add_B (list 'empty d2))
       (list 'add_B (list 'A_deleted d2))
              (list 'add_B (list 'B_deleted d2)))
 nil)
      (if (and (equal d1 'B)
       (equal d2 'A))
         (list (list 'add_A (list d1 'empty))
       (list 'add_A (list d1 'A_deleted))
              (list 'add_A (list d1 'B_deleted)))
 nil)
      (if (and (equal d1 'A)
```

```
       (equal d2 'B))
        (list (list 'add_B (list d1 'empty))
      (list 'add_B (list d1 'A_deleted))
             (list 'add_B (list d1 'B_deleted)))
nil)
      (if (or (equal d1 'A) (equal d2 'A))
        (list (list 'add_A s))
nil)
      (if (and (equal d1 'A) (equal d2 'A))
        (list (list 'add_B s))
nil)
     (if (and (equal d1 'B) (equal d2 'B))
        (list (list 'add_A s))
nil)
      (if (or (equal d1 'B) (equal d2 'B))
        (list (list 'add_B s))
nil))))


; ---- Loading generic reconsturciton algorithm ------

(load "rec")

; ---- Loading drawing utility -----

(load "draw")

; ===== Formalisation of evidence =====

 ; The value of infinitum

 (defconst *infinitum* 6)

 ; The set of all computations

 (defconst *C_T* *ALL-SINGLE-STEP-PATT*)

 ; Carl's story

 (defpatt1 *B-DELETED*
        (x) (and (equal (first (second x)) 'B_deleted)
                 (equal (second (second x)) 'B_deleted)))

 (defconst *OS-CARL*  `((,*C_T* 0 ,*infinitum*) (,*B-DELETED* 1 0)))

 ; Manufacturer's story

 (defpatt1 *EMPTY*
        (x) (and (equal (first (second x)) 'empty)
                 (equal (second (second x)) 'empty)))


 (defconst *OS-MANU* `((,*EMPTY* 1 0) (,*C_T* 0 ,*infinitum*)))

 (defconst *ES-ACME* `(,*OS-MANU* ,*OS-CARL*))

 ; ============ Investigative hypothesis =================

 ; Alice's claim restricted to exclude speculative transitions

 (defpatt2 *ALICE-PRIME* (x y)
        (and (not (or (equal (first (second x)) 'A)
                      (equal (second (second x)) 'A)))
             (not (equal (second x) (second y)))
             (not  (and (equal (first (second x)) 'B_deleted)
                        (equal (second (second x)) 'B_deleted)
                        (equal (first x) 'Add_B)
```

```
                      (equal (first (second y)) 'B)
                      (equal (second (second y)) 'B_deleted)
                      (equal (first y) 'take)))))

(defconst *OS-PRIME-ALICE* `((,*ALICE-PRIME* 0 ,*infinitum*)
                          (,*B-DELETED* 1 0)))

(defconst *ES-PRIME-PRIME-ACME* (cons *OS-PRIME-ALICE* *ES-ACME*))

; ========== Computing meanings of evidential statements =========

(defconst *ES-ACME-SOL* (solve-es *ES-ACME*))
(defconst *ES-PRIME-PRIME-ACME-SOL* (solve-es *ES-PRIME-PRIME-ACME*))

(print "Is the meaning of es_ACME empty ?")
(print (null *ES-ACME-SOL*))

(print "Is the meaning of es''_ACME empty ?")
(print (null *ES-PRIME-PRIME-ACME-SOL*))
```

C.5 ft.lisp

```
;*********************************************************
;*         TEMPORAL BOUNDING OF EVENTS
;*********************************************************

;(in-package "ACL2")

; ACL2 functions not defined in CMU CL

(DEFUN MEMBER-EQUAL (X LST)
  (COND ((ENDP LST) NIL)
        ((EQUAL X (CAR LST)) LST)
        (T (MEMBER-EQUAL X (CDR LST)))))


(DEFUN NFIX (X)
  (IF (AND (INTEGERP X) (>= X 0)) X 0))

;(include-book "fd-tst")

; make list of all observation id's for given observation sequence obs.
; m is the index of obs in the evidential statement;
; n is the index of the first element of obs.

(defun allobs (obs m n)
  (if (atom obs)
    nil
    (cons (list (nfix m) (nfix n))
          (allobs (cdr obs) (nfix m) (+ (nfix n) 1)))))

; make list of observation id's for given evidential statement es.
; n is the index of the first observation sequence in es.

(defun alles (es n)
  (if (atom es)
    nil
    (append (allobs (car es) (nfix n) 0)
            (alles (cdr es) (+ (nfix n) 1)))))

; intersection-equal (taken from books/data-structures/set-defuns.lisp)
; returns list whose elements are members of both x and y

(defun intersection-equal (x y)
  ;(declare (xargs :guard (and (true-listp x) (true-listp y))))
  (cond ((endp x) nil)
((member-equal (car x) y)
 (cons (car x) (intersection-equal (cdr x) y)))
(t (intersection-equal (cdr x) y))))

; sum first n elements of the list l

(defun sumpref (n l)
  (if (or (zp n) (atom l))
    0
    (+ (nfix (car l)) (sumpref (1- n) (cdr l)))))

; ==== Calculating the lower (earliest) boundary ====


; Find id's of all runs in the given partition map pm,
; whose last computation appears in the partitioned run at a position
; less than or equal to pos.
;
; pm is the partition map;
; i is the index of the partition map in the partition map list;
; j is the index of the first run in the partition;
```

```
; cnt is the position of the first run of pm in the partitioned run.

(defun befpm (pm cnt pos i j)
  (if (atom pm)
    nil
    (if (<= (+ cnt (car pm)) pos)
      (cons (list i j)
            (befpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))
      (befpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))))

; find id's of all runs in the given partition map list pml,
; whose last computation appears in the partitioned run at a position
; less than or equal to pos.
;
; i is the index of the first partition map in pml.

(defun befpml (pos pml i)
  (if (atom pml)
    nil
    (append (befpm (car pml) 0 pos i 0)
            (befpml pos (cdr pml) (+ i 1)))))

; For the given list of partition list chunks v,
; find id's of all runs that precede run with ID=(i j) in
; all partition chunks of v.
;
; bef is the set of id's that passed test so far. Initially
;       bef is set to the set of all observation ids.
;       At each step, it is intersected with the set of
;       id'satisfying test for the current partition list chunk.

(defun findbef (v bef i j)
  (if (atom v)
    bef
    (findbef (cdr v)
             (intersection-equal
                bef
                (befpml (sumpref j (nth i (car (car v))))
                        (car (car v))
                        0))
             i j)))

; Find in the associative list tim of known observation times
; the maximal time whose id belongs to the set l.

(defun maxtime (max l tim)
  (if (atom tim)
    max
    (let ((time (car (cdr (car tim))))
          (id (car (car tim))))
      (if (not (member-equal id l))
        (maxtime max l (cdr tim))
        (if (null max)
          (maxtime time l (cdr tim))
          (if (< max time)
            (maxtime time l (cdr tim))
            (maxtime max l (cdr tim)))))))))


; Find the earliest time boundary for observation
; with ID=(i j) in evidential statement es, given list v of
; partition list chunks and list of known observation times tim.

(defun lbound (i j es v tim)
  (maxtime nil
           (findbef v (alles es 0) i j)
           tim))
```

```
#|
(lbound 2 1
        '((((1 2 3) 1 25) ((3 4) 1 25))
          (((1 2 3 4) 3 25))
          ((universe 0 25) ((2) 1 25) (universe 0 25)))

        '(((4) ((1 3) (4) (2 1 1)))
          ((4) ((2 2) (4) (2 1 1)))
          ((3) ((1 2) (3) (1 1 1)))
          ((4) ((1 2) (3) (2 1 0)))
          ((4) ((2 1) (3) (2 1 0))))

        '(((0 0) 6) ((2 2) 7) ((0 0) 8)))

Answer: 8
|#

; ==== Calculating the upper (latest) boundary ====

; Find id's of all runs in the given partition map pm,
; whose first computation appears in the partitioned run at a position
; greater than or equal to pos.
;
; pm is the partition map;
; i is the index of the partition map in the partition map list;
; j is the index of the first run in the partition;
; cnt is the position of the first run of pm in the partitioned run.

(defun aftpm (pm cnt pos i j)
  (if (atom pm)
    nil
    (if (<= pos cnt)
      (cons (list i j)
            (aftpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))
      (aftpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))))

; find id's of all runs in the given partition map list pml,
; whose first computation appears in the partitioned run at a position
; greater than or equal to pos.
;
; i is the index of the first partition map in pml.

(defun aftpml (pos pml i)
  (if (atom pml)
    nil
    (append (aftpm (car pml) 0 pos i 0)
            (aftpml pos (cdr pml) (+ i 1)))))

; For the given list of partition list chunks v,
; find id's of all runs that precede run with ID=(i j) in
; all partition chunks of v.
;
; aft is the set of id's that passed test so far. Initially
;       aft is set to the set of all observation ids.
;       At each step, it is intersected with the set of
;       id'satisfying test for the current partition list chunk.

(defun findaft (v aft i j)
  (if (atom v)
    aft
    (findaft (cdr v)
             (intersection-equal
                aft
                (aftpml (sumpref (+ j 1) ; include length of run into sum
                                 (nth i (car (car v))))
                        (car (car v))
```

```
                        0))
             i j)))

; Find in the associative list tim of known observation times
; the minimal time whose id belongs to the set l.

(defun mintime (min l tim)
  (if (atom tim)
    min
    (let ((time (car (cdr (car tim))))
          (id (car (car tim))))
      (if (not (member-equal id l))
        (mintime min l (cdr tim))
        (if (null min)
          (mintime time l (cdr tim))
          (if (< time min)
            (mintime time l (cdr tim))
            (mintime min l (cdr tim))))))))

; Find the latest time boundary for observation
; with ID=(i j) in evidential statement es, given list v of
; partition list chunks and list of known observation times tim.

(defun ubound (i j es v tim)
  (mintime nil
           (findaft v (alles es 0) i j)
           tim))
```

C.6 slack.lisp

```
;*************************************
;* Formalisation of slack space analysis
;*************************************

(load "util")

; ======== Finite State Machine =======


; Proper state predicate

(defun lenvaluep (l)
  (or (equal l 0)
      (equal l 1)
      (equal l 2)))

(defun datavaluep (v)
  (or (equal v 1)
      (equal v 0)))

; The first element of state is the length of
; The rest of the elements is the data store
; e.g. (1 0 0)

(defun statep (s)
  (and (lenvaluep (car s))
       (datavaluep (cadr s))
       (datavaluep (caddr s))
       (null (cdddr s))))

; Proper event predicate

; Event is a sequence of 1s and 0s with at least
; 1 element and at most 2 elements


(defun eventp (e)
  (or (equal e 'del)
      (and (datavaluep (car e))
           (null (cdr e)))
      (and (datavaluep (car e))
           (datavaluep (cadr e))
           (null (cddr e)))))

; Set of all possible event-state pairs

(defconst *LENGTH*      '(0 1 2))

(defconst *ALL-STATES*  '((0 0 0) (0 0 1) (0 1 0) (0 1 1)
                         (1 0 0) (1 0 1) (1 1 0) (1 1 1)
                         (2 0 0) (2 0 1) (2 1 0) (2 1 1)))

(defconst *ALL-EVENTS*  '(del (0) (1) (0 0) (0 1) (1 0) (1 1)))

(defconst *ALL-EVENT-STATE-PAIRS*
  (product *ALL-EVENTS* (listify-elements *ALL-STATES*)))

; Transition funciton


(defun st (c s)
  (if (atom c)
    (if (equal c 'del)      (list 0  (cadr s) (caddr s))
                            s)
    (let ((l (length c)))
```

```
      (if (equal l 1)       (list 1 (car c) (caddr s))
      (if (equal l 2)       (list 2 (car c) (cadr c))
                            s)))))

; Inverse transition function

(defun rev-st (s)
  (let* ((l (car s))
         (data (cdr s)))
    (cond
      ((equal l 0)
         (list (list 'del (list* 0 data))
               (list 'del (list* 1 data))
               (list 'del (list* 2 data))))
      ((equal l 1)
         (list (list (list (car data)) (list* 0 0 (cdr data)))
               (list (list (car data)) (list* 0 1 (cdr data)))
               (list (list (car data)) (list* 1 0 (cdr data)))
               (list (list (car data)) (list* 1 1 (cdr data)))
               (list (list (car data)) (list* 2 0 (cdr data)))
               (list (list (car data)) (list* 2 1 (cdr data)))))
      ((equal l 2)
         (list (list data (list 0 0 0))
               (list data (list 0 0 1))
               (list data (list 0 1 0))
               (list data (list 0 1 1))
               (list data (list 1 0 0))
               (list data (list 1 0 1))
               (list data (list 1 1 0))
               (list data (list 1 1 1))
               (list data (list 2 0 0))
               (list data (list 2 0 1))
               (list data (list 2 1 0))
               (list data (list 2 1 1))))
      (t nil))))

; ---- Loading generic reconsturciton algorithm ------

(load "rec")

; ---- Loading drawing utility -----

(load "draw")

; ---- Loading event time bounding algorithms ----

(load "ft")

; ===== Formalisation of evidence =====

 ; The value of infinitum

 (defconst *infinitum* 4)

 ; The set of all computations

 (defconst *C_T* *ALL-SINGLE-STEP-PATT*)

 ; Observed properties

 (defpatt1 *FINAL*
        (x) (equal (second x) '(1 0 1)))

 (defpatt1 *BLACKMAIL-WRITE*
        (x) (equal (first x) '(1 1)))
```

```
(defpatt1 *UNRELATED-WRITE*
       (x) (equal (first x) '(0)))

(defpatt1 *NO-UNRELATED-WRITE*
       (x) (not (equal (first x) '(0))))

(defconst *OS-FINAL*           `((,*C_T* 0 ,*infinitum*) (,*FINAL* 1 0)))

(defconst *OS-UNRELATED*       `((,*C_T* 0 ,*infinitum*)
                               (,*UNRELATED-WRITE* 1 0)
                               (,*C_T* 0 ,*infinitum*)
                               (,*C_T* 0 0)
                               (,*C_T* 1 ,*infinitum*)))

(defconst *OS-BLACKMAIL*       `((,*C_T* 0 ,*infinitum*)
                               (,*BLACKMAIL-WRITE* 1 0)
                               (,*C_T* 1 ,*infinitum*)))

(defconst *OS-PRIME-UNRELATED* `((,*NO-UNRELATED-WRITE* 0 ,*infinitum*)
                               (,*UNRELATED-WRITE* 1 0)
                               (,*NO-UNRELATED-WRITE* 0 ,*infinitum*)
                               (,*NO-UNRELATED-WRITE* 0 0)
                               (,*NO-UNRELATED-WRITE* 1 ,*infinitum*)))

(defconst *ES-BLACKMAIL* `( ,*OS-FINAL*
                          ,*OS-UNRELATED*
                          ,*OS-BLACKMAIL-OBS* ))

(defconst *ES-PRIME-BLACKMAIL* `( ,*OS-FINAL*
                                  ,*OS-PRIME-UNRELATED*
                                  ,*OS-BLACKMAIL* ))


; Compute the meanings of the evidential statements

(defconst *SOL-4* (solve-es *ES-BLACKMAIL*))

(defconst *SOL-PRIME-4* (silve-es *ES-PRIME-BLACKMAIL*))

; Run the time bounding algorithm

; 5 is the time of the reception of unrelated event
; the times of all other events are unknown.
(defconst *tim* '(((1 3) 5)))

; time bouding of blackmail-write in es_blackmail

(defconst *l*
  (lbound 2 1 *ES-BLACKMAIL* *SOL-4* *tim*))

(defconst *u*
  (ubound 2 1 *ES-BLACKMAIL* *SOL-4* *tim*))

; time bounding of blackmail-write in es'_blackmail

(defconst *l-prime*
  (lbound 2 1 *ES-PRIME-BLACKMAIL* *SOL-PRIME-4* *tim*))

(defconst *u-prime*
  (ubound 2 1 *ES-PRIME-BLACKMAIL* *SOL-PRIME-4* *tim*))
```

C.7 draw.lisp

```
;*****************************************
;* Drawing reconstruction results using DOT
;*****************************************

(defun digit-name (digit)
  (case digit
    (0 "0")
    (1 "1")
    (2 "2")
    (3 "3")
    (4 "4")
    (5 "5")
    (6 "6")
    (7 "7")
    (8 "8")
    (9 "9")
    (10 "A")
    (11 "B")
    (12 "C")
    (13 "D")
    (14 "E")
    (15 "F")
    (otherwise "?")))

(defun number-name-helper (n radix)
  (if (equal n 0)
    ""
    (concatenate 'string
                 (number-name-helper (floor n radix) radix)
                 (digit-name (rem n radix)))))
(defun number-name (n radix)
  (if (equal n 0) "0" (number-name-helper n radix)))

(defun pretty (l)
  (if (null l)
    ""
    (if (atom l)
         (if (numberp l) (number-name l 10) (symbol-name l))
      (let ((s (car l)))
        (concatenate
           'string
           (if (symbolp s) (symbol-name s)
             (if (consp s) (concatenate 'string "(" (pretty s) ")")
               (if (numberp s)
                 (number-name s 10)
                 " ??? ")))
           (if (not (null (cdr l))) ", " "") (pretty (cdr l)))))))

(defun stringify (l acc)
   (if (atom l)
     acc
     (stringify (cdr l) (concatenate 'string acc "_"
       (if (numberp (car l))
         (number-name (car l) 10)
         (symbol-name (car l)))))))

(defun flatten (l)
  (if (atom l)
    l
    (if (consp (car l))
      (append (flatten (car l)) (flatten (cdr l)))
      (cons (car l) (flatten (cdr l))))))

(defun draw-comp (file comp name1)
   (if (atom comp)
```

```
     nil
      (progn
        (format
           file "n~A [label=\"~A\"];~%" name1 (pretty (cadar comp)))
        (if (atom (cdr comp))
          nil
          (let ((name2 (concatenate
                         'string
                         name1
                         (stringify (flatten (cadr comp)) ""))))
            (progn
              (format
                 file "n~A [label=\"~A\"];~%" name2 (pretty (cadadr comp)))
              (format
                 file "n~A -> n~A [label=\"~A\"];~%"
                      name2 name1
                        (pretty (car (cadr comp))))
              (draw-comp file (cdr comp) name2)))))))

(defun draw-comps (file comps)
   (if (atom comps)
     nil
     (let ((rv (reverse (car comps))))
       (progn
         (draw-comp file rv (stringify (flatten (cdar rv)) ""))
         (draw-comps file (cdr comps))))))


(defun draw-sol (file es-sol)
   (if (atom es-sol)
     nil
     (progn
       (draw-comps file (cadar es-sol))
       (draw-sol file (cdr es-sol)))))

(defun draw (es-sol)
   (with-open-file (f "t.dot" :direction :output :if-exists :supersede)
     (format f "strict digraph G { ~% size=\"8,11\";~%rankdir=LR;~%")
     (draw-sol f es-sol)
     (format f "}~%")
   ))
```