THE ABCON AND H-ABCON
SOURCE CODES

Gary Snavely
Panos Papalambros

UM-MEAM-92-04
This document contains the source code for both the ABCON and the H-ABCON configuration design methodologies. A complete description of these design methodologies as well as the Common LISP implementation can be found in "An Abstraction-Based Methodology for Mechanical Configuration Design," a 1992 PhD thesis written by Gary Snavely at The University of Michigan.

(odefvar *lower-limit* nil) ; lower limit on abstraction level
(odefvar *optional-components* nil)
(odefvar *required-states* nil)
(odefvar *database* nil) ; database of components
(odefvar *duplicate-list* nil) ; used to check for type II duplicates
(odefvar *component-counter* 0) ; used in make-name-unique
(odefvar *optional-component-limits* nil) ; used to control number of optional components of a single type in a topology

; THE FOLLOWING ARE USED TO CONTROL THE SIZE OF THE OPEN LIST
(odefvar *invalid-arc-limit* 1) ; used in topology-instantiation
(odefvar *open-port-limit* 4) ; used in topology-generation
(odefvar *component-limit* 6) ; used in topology generation
(odefvar *open-list-length-limit* 100) ; used in best-first-search

; THE FOLLOWING ARE USED TO RANK TOPOLOGIES
(odefvar *number-of-arcs-factor* 0)
(odefvar *number-of-components-factor* 0)
(odefvar *number-of-ports-factor* 0)
(odefvar *number-of-clusters-factor* 0)
(odefvar *partial-topology-factor* 0)
(odefvar *abstraction-level-factor* 0)

(odefvar *matching-spec-counter* 0) ; used to count # of times specs matched

; Send this function a list of required states, where a required state is a list of s-component names, and a list of optional component names, and it calls compile-problem.
(defun solve (problem-statement optional-component-name/limit-pairs)
  (let
    (optional-component-names
      partial-topology

ranked-partial-topology)

(setq *duplicate-list* nil) ; initialize the *duplicate-list*

(setq *optional-component-limits*
  optional-component-name/limit-pairs)

(setq optional-component-names
  (mapcar 'first
    optional-component-name/limit-pairs))

(setq *optional-components* 
  (create-optional-components
    optional-component-names))

(setq partial-topology 
  (compile-problem problem-statement))

(setq ranked-partial-topology 
  (rank-topology partial-topology))

(best-first-search ranked-partial-topology))

(defun create-optional-components (names &optional optional-components)
  (cond ((null names)
    (reverse optional-components))
    (t (create-optional-components
      rest names)
      (cons (assoc (first names) *database* :test #'string-equal)
        optional-components))))

(defun compile-problem (problem-statement)
  (let
    (partial-topology)
    (setq problem-statement (create-problem-statement problem-statement))
    (setq *lower-limit* (second (caaar (last problem-statement))))
    (setq *required-states* (make-required-states problem-statement))
    (setq partial-topology
      (convert-required-states-into-partial-topology (first problem-statement))))

(defun create-problem-statement 
  (lists-of-s-component-names
   &optional lists-of-single-state-required-components)
  (let
    (optional-problem-statement required-problem-statement)
  (cond ((null lists-of-s-component-names)
      (setq optional-problem-statement
        (create-problem-statement-aux2
          (reverse
            lists-of-single-state-required-components)
        nil
        (list
          (reverse
            lists-of-single-state-required-components))))
      (setq required-problem-statement
        (convert-to-required-components
          optional-problem-statement))
      (t (create-problem-statement
        (rest lists-of-s-component-names)
        (cons (create-problem-statement-aux1
          (first lists-of-s-component-names)
          lists-of-single-state-required-components))))))

; Send this function a list of s-component-names and a 1, and it returns a list of
; single-state-components with nil for children.

(defun create-problem-statement-aux1
  (s-component-names &optional single-state-components)
  (let
    (s-component-name
     component-name
     component
     s-component
     new-s-component
     new-single-state-component)
    (cond
      ((null s-component-names) (reverse single-state-components))
      (t (setf s-component-name (first s-component-names))
        (setf component-name (convert-oscn-to-ocn s-component-name))
        (setf component (assoc component-name *database* :test #'string-equal))
        (setf s-component
          (assoc s-component-name (third component):test #'string-equal))
        (setf new-s-component
          (list
           (first s-component)
           (second s-component)
           nil
           (fourth s-component)))
        (setf new-single-state-component
          (list (first component) (second component) (list new-s-component)))
        (create-problem-statement-aux1
         (rest s-component-names)
         (cons new-single-state-component single-state-components))))

(defun create-problem-statement-aux2
  (lists-of-single-state-required-components
   new-lists-of-single-state-required-components
   problem-statement)
  (let
    (list-of-single-state-required-components
     new-list-of-single-state-required-components)
    (cond
      ((null lists-of-single-state-required-components)
       (create-problem-statement-aux2
        (reverse new-lists-of-single-state-required-components)
        nil
        (cons new-lists-of-single-state-required-components
          problem-statement)))
      (t (setf list-of-single-state-required-components
        (first lists-of-single-state-required-components))
        (setf new-list-of-single-state-required-components
          (create-problem-statement-aux3
           list-of-single-state-required-components))
        (cond
          ((null new-list-of-single-state-required-components)
           problem-statement)
          (t (create-problem-statement-aux2
              (rest lists-of-single-state-required-components)
              (cons new-list-of-single-state-required-components
                new-lists-of-single-state-required-components)
              problem-statement))))))

; Send this function a list of single-state required-components and it returns a list
; of single-state required-components that are parents of the ones sent.

(defun create-problem-statement-aux3
  (single-state-components
   &optional new-single-state-components)
  (let
    (single-state-component
     required-s-component
     new-single-state-component
     (create-problem-statement-aux2
      (rest single-state-components)
      (cons new-single-state-component single-state-components)
      problem-statement)))

3
(defun convert-scn-to-urscn (s-component-name unique-suffix)
  (let
    (state-suffix component-name)
    (setf state-suffix (remove-component-name s-component-name))
    (setf component-name (remove-state-suffix s-component-name))
    (concatenate 'string component-name "@r" state-suffix unique-suffix))
  )
  )
(defun convert-to-required-components
  (optional-problem-statement \optional required-problem-statement)
  (let
    (required-states)
    (cond ((null optional-problem-statement)
      (reverse required-problem-statement))
      (t (setf required-states (cetr-aux1 (first optional-problem-statement))))
      (setf required-states (remove-duplicates required-states :test #'equalp))
      (convert-to-required-components
       (rest optional-problem-statement)
       (cons required-states required-problem-statement)))))

(defun cetr-aux1
  (required-states \optional new-required-states)
  (let
    (required-state)
    (cond ((null required-states) (reverse new-required-states))
      (t (setf required-state (cetr-aux2 (first required-states) 1))
        (cetr-aux1
         (rest required-states)
         (cons required-state new-required-states))))))

(defun cetr-aux2
  (required-state-counter \optional new-required-state)
  (let
    (optional-component
     optional-component-name
     optional-s-component
     optional-s-component-name
     state-suffix
     required-component-name
     unique-required-component-name
     required-s-component-name
     unique-required-s-component-name
     required-s-component
     required-component
     s-parent-name
     parent-name
     parent-state-suffix
     required-s-parent-name
     unique-required-s-parent-name
     new-parent-name)
    (cond ((null required-state)
      (reverse new-required-state))
      (t (setf optional-component (first required-state))
        (setf optional-component-name (first optional-component))
        (setf optional-s-component (first (third optional-component)))
        (setf optional-s-component-name (first optional-s-component))
        (setf state-suffix (remove-component-name optional-s-component-name))
        (setf required-component-name
          (convert-occ-to-rcn optional-component-name))
        (setf unique-required-component-name
          (make-name-unique required-component-name counter))
        (setf required-s-component-name
          (concatenate 'string required-component-name state-suffix))
        (setf unique-required-s-component-name
          (make-name-unique required-s-component-name counter))
        (setf s-parent-name (first (second optional-s-component)))
        (cond ((null s-parent-name)
          (setf new-parent-name nil))
          (t (setf parent-name (remove-state-suffix s-parent-name))
            (setf parent-state-suffix (remove-component-name s-parent-name))
            (setf required-s-parent-name
              (concatenate
; string parent-name "@r" parent-state-suffix))
(setf unique-required-s-parent-name
     (make-name-unique
      required-s-parent-name
      counter))
(setf new-parent-name (list unique-required-s-parent-name)))

(setf required-s-component
     (list
      unique-required-s-component-name
      new-parent-name
      nil
      (fourth optional-s-component)))

(setf required-component
     (list
      unique-required-component-name
      (second optional-component)
      (list required-s-component)))

(ctrc-aux2
 rest required-state
 (+ 1 counter)
 (cons required-component new-required-state)))))))

(defun convert-required-states-into-partial-topology
   (required-states)
   (let
     (first-required-state
      single-state-required-component
      abstraction-level
      partial-state-topologies)
     (setf first-required-state (first required-states))
     (setf single-state-required-component (first first-required-state))
     (setf abstraction-level (second single-state-required-component))
     (setf partial-state-topologies
          (make-some-partial-state-topologies required-states))
     (list "partial" abstraction-level partial-state-topologies 0)))))

; Send this function:
; 1. required states: a list of single state required components for each state
;    of a single abstraction level
; and it will return a partial-state-topology for each state at that level.

(defun make-some-partial-state-topologies
   (required-states optional-partial-state-topologies)
   (let
     (single-state-required-components
      s-clusters
      s-components
      s-component-names
      partial-state-topology)
   (cond ((null required-states)
          (reverse partial-state-topologies))
          (t (setf single-state-required-components (first required-states))
              (setf s-clusters (make-s-clusters
                                  single-state-required-components))
              (setf s-components (mapcar 'caaddr single-state-required-components))
              (setf s-component-names (mapcar 'car s-components))
              (setf partial-state-topology (list s-clusters nil s-component-names))
              (make-some-partial-state-topologies
               rest required-states)
              (cons partial-state-topology
                    partial-state-topologies))))))

6
(defun make-s-clusters (single-state-required-components &optional s-clusters)
  (let
    (single-state-required-component
     s-component
     s-component-name
     port-specs
     s-open-ports)
    (cond ((null single-state-required-components) (reverse s-clusters))
          (t (setf single-state-required-component
                (first single-state-required-components))
              (setf s-component (first (third single-state-required-component)))
              (setf s-component-name (first s-component))
              (setf port-specs (fourth s-component))
              (setf s-open-ports (make-s-open-ports port-specs s-component-name))
              (make-s-clusters
               (rest single-state-required-components)
               (cons s-open-ports s-clusters))))))

(defun top-level-delistify (list &optional new-list)
  (cond ((null list) new-list)
          (t (if (listp (first list))
               (setf new-list (append new-list (first list)))
               (setf new-list (append new-list (list (first list))))
               (top-level-delistify (rest list) new-list))))

(defun delistify (list &optional new-list)
  (cond ((null list) new-list)
          (t (cond ((listp (car list))
                     (delistify (cdr list) (append new-list (delistify (car list)))))
               (t (delistify (cdr list) (append new-list (list (car list))))))))

(defun make-s-open-ports
  (s-port-specs s-component-name &optional s-open-ports)
  (cond ((null s-port-specs) (reverse s-open-ports))
          (t (make-s-open-ports
              (rest s-port-specs)
              s-component-name
              (cons (list s-component-name (first s-port-specs)) s-open-ports))))

(defun make-open-ports (port-names component-name &optional open-ports)
  (cond ((null port-names) (reverse open-ports))
          (t (make-open-ports
              (rest port-names) component-name
              (cons (list (first port-names) component-name) open-ports))))

(defun make-required-states (problem-statement &optional required-states)
  (let ()
    (cond ((null problem-statement)
            (reverse required-states))
          (t (make-required-states
              (rest problem-statement)
              (cons (make-required-states-auxl
                     (first problem-statement)
                     (second (caar (first problem-statement))))
                     required-states))))))

(defun make-required-states-auxl (states abstraction-level &optional new-level)
  (let
    (state
     new-state)
    (cond ((null states)
(list abstraction-level
  (reverse new-level)))
(t (setf state (first states))
  (setf new-state (make-required-states-aux2 state))
  (make-required-states-aux1
   (rest states)
   abstraction-level
   (cons new-state new-level))))

(defun make-required-states-aux2 (state &optional new-state)
  (let
    (ss-required-component
     s-required-component
     s-component-w-parent-name)
    (cond ((null state)
      (reverse new-state))
      (t (setf ss-required-component (first state))
        (setf s-required-component (first (third ss-required-component)))
        (setf s-component-w-parent-name
          (list
           (first (second s-required-component))
           s-required-component))
        (make-required-states-aux2
         (rest state)
         (cons s-component-w-parent-name new-state))))))

(defun convert-ocn-to-rcn (name)
  (concatenate 'string (princ-to-string name) "@r"))

(defun convert-oscn-to-ocn (name)
  (remove-state-suffix name))

(defun convert-rscn-to-rcn (name)
  (remove-state-suffix name))

(defun convert-urscn-to-uscn (name)
  (let
    (unique-suffix rscn scn uscn)
    (setf unique-suffix (get-unique-suffix name))
    (setf rscn (remove-unique-suffix name))
    (setf scn (convert-rscn-to-oscn rscn))
    (setf uscn (concatenate 'string scn unique-suffix))))

(defun convert-rscn-to-oscn (name)
  (let
    (component-name state-suffix oscn)
    (setf component-name (remove-state-suffix name))
    (setf component-name (remove-required-suffix component-name))
    (setf state-suffix (remove-component-name name))
    (setf state-suffix (remove-required-prefix state-suffix))
    (setf oscn (concatenate 'string component-name state-suffix))))

(defun convert-oscn-to-rscn (name)
  (let
    (component-name state-suffix rscn)
    (setf component-name (remove-state-suffix name))
    (setf state-suffix (remove-component-name name))
    (setf rscn (concatenate 'string component-name "@r" state-suffix))))

(defun convert-urcn-to-cn (name)
  (remove-required-suffix (remove-unique-suffix name))

(defun convert-usn-to-un (name)
(let
 (namel name2 unique-suffix)
 (setf name1 (remove-unique-suffix name))
 (setf name2 (remove-state-suffix name2))
 (setf unique-suffix (remove-component-name name))
 (setf unique-suffix (remove-required-prefix unique-suffix))
 (setf unique-suffix (remove-state-prefix unique-suffix))
 (concatenate 'string name2 unique-suffix)))

(defun convert-usc-to-n (name)
 (remove-required-suffix (remove-state-suffix (remove-unique-suffix name))))

(defun make-name-unique (name number)
 (concatenate 'string (princ-to-string name) "=" (princ-to-string number))

(defun remove-component-name (name)
 (string-left-trim "abcdefghiklnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" name))

(defun get-required-names (names &optional required-names)
 (let
 (name namel name2)
 (cond ((null names) (reverse required-names))
 (t (setf name (first names))
 (setf namel (remove-component-name name))
 (setf name2 (remove-required-prefix name1))
 (cond ((string-equal namel name2)
 (get-required-names (rest names) required-names))
 (t (get-required-names (rest names)
 (cons name required-names))))))

; Send this function a name with a "##" on the end and it will remove the "##".

(defun remove-unique-suffix (name)
 (let
 (namel name2)
 (setf namel (string-right-trim "1234567890sS" name))
 (setf name2 (string-right-trim "##" namel))
 (if (string-equal name2 name1) name name2)))

; Send this function a name with a "-s#" on the end and it will remove the "-s#".

(defun remove-state-suffix (name)
 (let
 (namel name2)
 (setf namel (string-right-trim "1234567890sS" name))
 (setf name2 (string-right-trim "-" namel))
 (if (string-equal name2 name1) name name2)))

; Send this function a name with a "@r" on the end and it will remove the "@r".

(defun remove-required-suffix (name)
 (let
 (namel name2)
 (setf namel (string-right-trim "r" name))
 (setf name2 (string-right-trim "@" namel))
 (if (string-equal name2 name1) name name2)))

; Send this function a name with a "@r" at the beginning and it will remove the "@r".

(defun remove-required-prefix (name)
 (let
 (namel name2)
 (setf namel (string-left-trim "@" name)))
(setf name2 (string-left-trim "r" name1))
(if (string-equal name2 name1) name2)))

; Send this function a name with "-s#" at the beginning and it will remove the "-s#".

(defun remove-state-prefix (name)
(let
(name1 name2)
(setf name1 (string-left-trim "-" name))
(setf name2 (string-left-trim "0123456789s" name1))
(if (string-equal name2 name1) name name2)))

(defun get-unique-suffix (unique-name)
(let
(unique-suffix)
(setf unique-suffix (remove-component-name unique-name))
(setf unique-suffix (remove-required-prefix unique-suffix))
(setf unique-suffix (remove-state-prefix unique-suffix))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; BEST-FIRST SEARCH
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Send this function a partial topology and it will direct the generation of complete
; topologies and the instantiation of those complete topologies at lower levels of
; abstraction according to the rank of each topology, in a best-first manner.

(defun best-first-search (partial-topology)
(cond ((not (string-equal
(car partial-topology)
"partial"))
(print 'trouble-in-best-first-search))
(t (setf *start-time* (get-internal-real-time))
(best-first-search-aux1 (list partial-topology)))))

(defun best-first-search-aux1 (open-list)
(let
(first-topology
topology-type
topology-abstraction-level
new-topologies)
(if (> (length open-list) *open-list-length-limit*)
(setf open-list (trim-list open-list *open-list-length-limit*)))
(if (> (length open-list) *maximum-open-list-length*)
(setf *maximum-open-list-length* (length open-list)))
(cond ((null open-list)
(format t "-%-\%THAT'S ALL FOLKS!\%")
(t (setf first-topology (car open-list))
(setf topology-type (car first-topology))
(setf topology-abstraction-level (cadr first-topology))
(cond ((and (string-equal topology-type
"complete")
(string-equal topology-abstraction-level
*lower-limit*))
(format t "-%-\%HERE'S AN ANSWER!\%")
(format-complete-topology
first-topology)
(format t "-%")
(format t "-%It took: -a seconds")
(truncate
(/ (- (get-internal-real-time)
*start-time*)
internal-time-units-per-second))))
))
(defun trim-list (list length &optional (counter 0) new-list)
  (cond ((or (= counter length) (null list)) (reverse new-list))
    (t (trim-list
        (rest list)
        length
        (+ 1 counter)
        (cons (car list) new-list))))
)

; Send this function a topology and it returns the topology with the correct rank

(defun rank-topology (topology)
  (let
    (topology-type
      complete-state-topology
      partial-state-topology
      s-arcs
      s-component-names
      rank
      s-clusters
      number-of-open-ports
      ranked-topology)
    (setf topology-type (car topology))
    (cond ((string-equal topology-type "complete")
      (setf complete-state-topology (caaddr topology))
      (setf s-arcs (car complete-state-topology))
      (setf s-component-names (cadr complete-state-topology))
      (setf rank
        (+ (* (length s-arcs)
              *number-of-arcs-factor*)
        (* (length s-component-names)
            *number-of-components-factor*)
        0
        0
        0
        0)))))
(* (get-abstraction-level-number ; abstraction level
(cadr
  topology))
  *abstraction-level-factor*)
)

((string-equal topology-type "partial")
 (setf partial-state-topology (caaddr topology))
 (setf s-clusters (car partial-state-topology))
 (if (not (null s-clusters))
     (setf number-of-open-ports
           (length (top-level-delistify s-clusters))))
 (setf rank
      (+ (*
            (length ; number of arcs
                (cadr partial-state-topology))
            *number-of-arcs-factor*)
       (*
            (length ; number of components
                (caddr partial-state-topology))
            *number-of-components-factor*)
       (*
            (number-of-open-ports ; number of open ports
                s-clusters)
            *number-of-clusters-factor*)
       (* 1 *partial-topology-factor* ) ; complete topology factor
       (*
            (get-abstraction-level-number ; abstraction level
                (cadr
                  topology))
            *abstraction-level-factor*)
      )))

(t (print 'trouble-in-rank-topology) (print topology))
 (setf ranked-topology
      (list (car topology) (cadr topology) (caddr topology) rank)))

(defun item-rank (item)
  (car (last item)))

(defun format-complete-topology (complete-topology)
  (let
    (complete-state-topologies)
    (format t "%--%(~a)" (car complete-topology))
    (format t "%--% ~a" (cadr complete-topology))
    (setf complete-state-topologies (caddr complete-topology))
    (format-complete-state-topologies complete-state-topologies 't)
    (format t "%--% ~a" (caddr complete-topology))))

(defun format-arcs (arcs first-time)
  (cond ((null arcs) (format t ""))
      (first-time
       (format t "%--%(~a)" (car arcs))
       (format-arcs (cadr arcs) nil))
      (t (format t "%--% ~a" (car arcs))
          (format-arcs (cadr arcs) nil))))

(defun format-complete-state-topologies (complete-state-topologies first-time)
  (let
    (complete-state-topology)
    (cond ((null complete-state-topologies)
            (format t "")
       (t (setf complete-state-topology
         (car complete-state-topologies))
         (if first-time
             (format t "%--%(~a)"
                   (format t "%--% ~a")
             (format-s-arcs (car complete-state-topology) 't)
             (format t "%--% ~a" (caddr complete-state-topology))
             (format-complete-state-topologies}}}
(cdr complete-state-topologies
nil))))}

(defun format-s-arcs (s-arcs first-time)
 (cond ((null s-arcs)
 (format t "")
 (first-time
 (format t "(")
 (format t "(-a" (car s-arcs))
 (format-s-arcs (cdr s-arcs) nil)
 (format t "-a" (car s-arcs))
 (format-s-arcs (cdr s-arcs) nil))))))

; Send this function a partial topology of the form:
; ("partial" abstraction-level partial-state-topologies rank)
; and it returns a ranked and sorted list of topologies, both complete and partial,
; that result from expanding the first open port in three ways:
; 1. by merging clusters.
; 2. by finding optional-components that match the first open-port.
; 3. by connecting the first open-port to all other matching open-ports in cluster.

(defun generate-topologies (partial-topology)
 (let
 ((abstraction-level (cadr partial-topology))
  (number-of-clusters
   (length (car (caddr partial-topology))))
  (if (> number-of-clusters 1)
   (let
    (merged-cluster-topologies
     (connect-clusters
      partial-topology))
    (connect-port-topologies
     (connect-open-ports
      partial-topology))
    (optional-component-topologies
     (find-matching-components
      partial-topology))
    (new-partial-topologies
     (append connect-port-topologies
              merged-cluster-topologies
              optional-component-topologies))
    (eliminate-ungainly-topologies
     new-partial-topologies)
    (new-topologies
     (make-complete-topology-complete
      new-partial-topologies))
    (string-equal abstraction-level "level")
    (eliminate type II duplicates
     new-topologies)
    (rank topologies
     new-topologies)
    (sort topologies
     (mapcar 'rank-topology
              new-topologies)))
   number-of-clusters))
 (if (string-equal abstraction-level "level")
   (eliminate type II duplicates
    new-topologies)
   (eliminate-duplicate-topologies
    new-topologies)))
 (rank topologies
  (mapcar 'rank-topology
           new-topologies)))
 (sort topologies
  (mapcar 'rank-topology
           new-topologies)))
 (sort topologies
  (mapcar 'rank-topology
           new-topologies))))
(sort ranked-new-topologies
  #:<
  :key #:item-rank)))

; Send this function a partial topology that contains > two clusters and it will
; return a list of topologies created by merging all clusters with the main cluster
; if they have open-ports that match the first open-port of the main cluster.

(defun connect-clusters (topology)
  (let
    (partial-state-topologies
      clusters
      main-cluster
      other-clusters)
    (cond ((string-equal (car topology)
                           "complete")
           ;; if topology is not partial
           (print 'trouble-in-generate-required-topologies))
          (t (setq partial-state-topologies (caddr topology))
            (setq clusters
                   (mapcar 'car
                                partial-state-topologies))
            (setq main-clusters
                   (transposed-list
                    clusters))
            (setq other-clusters
                   (car
                    clusters))
            (setq other-clusters
                   (cdr
                    clusters))
            (connect-clusters-auxl
              (mapcar 'car main-clusters)
              (mapcar 'cdr main-clusters)
              other-clusters
              other-clusters
              (mapcar 'cadr partial-state-topologies)
              (mapcar 'caddr partial-state-topologies)
              (caddr topology))))))

; Send this function:
; 1. main-cluster-s-open-ports: all states of the first open port in the main cluster
; 2. other-main-s-open-ports: all states of the other open ports in the main cluster
; 3. other-clusters: ((s1-c2 s2-c2 s3-c2) (s1-c3 s2-c3 s3-c3))
; 4. rotated-clusters: ((s1-c3 s2-c3 s3-c3) (s1-c3 s2-c3 s3-c3))
; 5. s-arcs: (s1-s-arcs s2-s-arcs ....)
; 6. s-component-names: (s1-s-component-names s2-s-component-names ..)
; 7. abstraction-level
; and it parses thru the clusters to merge with the main cluster.

(defun connect-clusters-auxl
  (main-cluster-s-open-ports
  other-main-s-open-ports
  other-clusters
  rotated-clusters
  s-arcs
  s-component-names
  abstraction-level
  &optional results)
  (let
    (redundant-cluster-to-merge
     cluster-to-merge
     non-redundant-cluster-to-merge
     (rest results))
    ;; do something with the results
    (return results))
result)
(setf redundant-cluster-to-merge
   (car
      other-clusters))
(cond ((null redundant-cluster-to-merge)
      results)
   ((and (null (car other-main-s-open-ports))
         (= (length
             (car redundant-cluster-to-merge))
             1)
         (> (length rotated-clusters) 1))
      (connect-clusters-aux1
       main-cluster-s-open-ports
       other-main-s-open-ports
       (cdr other-clusters)
       (rotate-list rotated-clusters)
       s-arcs
       s-component-names
       abstraction-level
       results))
   (t (setf cluster-to-merge
        (transpose-list
          redundant-cluster-to-merge))
       (setf non-redundant-cluster-to-merge
            (remove-duplicates
             cluster-to-merge
             :test #'equalp))
       (setf result
            (connect-clusters-aux2
             main-cluster-s-open-ports
             other-main-s-open-ports
             non-redundant-cluster-to-merge
             redundant-cluster-to-merge
             (transpose-list
              (cdr rotated-clusters))
             s-arcs
             s-component-names
             abstraction-level))
       (cond ((null result)
                (connect-clusters-aux1
                 main-cluster-s-open-ports
                 other-main-s-open-ports
                 (cdr other-clusters)
                 (rotate-list rotated-clusters)
                 s-arcs
                 s-component-names
                 abstraction-level
                 results))
            (t (connect-clusters-aux1
                 main-cluster-s-open-ports
                 other-main-s-open-ports
                 (cdr other-clusters)
                 (rotate-list rotated-clusters)
                 s-arcs
                 s-component-names
                 abstraction-level
                 (append results
                    result)))))))

; This function parses thru the open-ports of a single cluster-to-merge
; Send this function:
; 1. main-cluster-s-open-ports: all states of the first open port in the main cluster
; 2. other-main-s-open-ports: all states of the other open-ports in the main cluster
(defun connect-clusters-aux2
  (main-cluster-s-open-ports
   other-main-s-open-ports
   non-redundant-cluster-to-merge
   redundant-cluster-to-merge
   other-s-clusters
   s-arcs
   s-component-names
   abstraction-level
   &optional results)
(let
  (result partial-topology)
  (cond ((null non-redundant-cluster-to-merge) (reverse results))
    (t (setf result
          (connect-clusters-aux3
           main-cluster-s-open-ports
           other-main-s-open-ports
           (car non-redundant-cluster-to-merge)
           redundant-cluster-to-merge
           other-s-clusters
           s-arcs
           s-component-names
           abstraction-level))
    (cond ((null result)
           (connect-clusters-aux2
            main-cluster-s-open-ports
            other-main-s-open-ports
            (cdr non-redundant-cluster-to-merge)
            redundant-cluster-to-merge
            other-s-clusters
            s-arcs
            s-component-names
            abstraction-level
            results))
      (t (setf partial-topology
            (list
             "partial"
             abstraction-level
             result
             0))
            (connect-clusters-aux2
             main-cluster-s-open-ports
             other-main-s-open-ports
             (cdr non-redundant-cluster-to-merge)
             redundant-cluster-to-merge
             other-s-clusters
             s-arcs
             s-component-names
             abstraction-level
             (cons partial-topology
              results)))))))))

; Send this function:
; 1. main-cluster-s-open-ports: all states of the first open port in the main cluster
; 2. other-main-s-open-ports: all states of the other open-ports in the main cluster
; 3. cluster-s-open-ports: all states of a single open port in the cluster to merge
; 4. all-s-open-ports: all states of all open-ports in the cluster-to-merge
; 5. other-s-clusters: ((s1-c3 s1-c4) (s2-c3 s2-c4) (s3-c3 s3-c4))
; 6. s-arcs: (state1-s-arcs state2-s-arcs ....)
; 7. s-component-names: (state1-s-component-names state2-s-component-names ....)
; 8. abstraction-level
; and it calls aux4 if all s-open-ports of both clusters have matching port
; specifications, else it returns nil.

(defun connect-clusters-aux3
  (main-cluster-s-open-ports
   other-main-s-open-ports
   cluster-s-open-ports
   all-s-open-ports
   other-s-clusters
   s-arcs
   s-component-names
   abstraction-level)
  (cond ((all-matching-port-specifications?
               (mapcar 'cadr main-cluster-s-open-ports)
               (mapcar 'cadr cluster-s-open-ports)
               abstraction-level)
               (connect-clusters-aux4
                main-cluster-s-open-ports
                other-main-s-open-ports
                cluster-s-open-ports
                all-s-open-ports
                other-s-clusters
                s-arcs
                s-component-names))
    (t nil)))

; This function works on multiple states of a single open port.
; Send this function:
; 1. main-cluster-s-open-ports: all states of the first open-port in the main cluster
; 2. other-main-s-open-ports: all states of the other open-ports in the main cluster
; 3. cluster-s-open-ports: all states of a single open-port in the cluster-to-merge
; 4. all-s-open-ports: all states of all open-ports in the cluster-to-merge
; 5. other-s-clusters: ((s1-c3 s1-c4) (s2-c3 s2-c4) (s3-c3 s3-c4))
; 6. s-arcs: (state1-s-arcs state2-s-arcs ....)
; 7. s-component-names: (state1-s-component-names state2-s-component-names ....)
; and it will return a partial-topology. Note that the aux3 function already
; determined that all main-cluster-s-open-ports and cluster-s-open-ports have
; matching port specifications.

(defun connect-clusters-aux4
  (main-cluster-s-open-ports
   other-main-s-open-ports
   cluster-s-open-ports
   all-s-open-ports
   other-s-clusters
   s-arcs
   s-component-names
   optional results)
  (let
   (result
    (cond ((null main-cluster-s-open-ports)
           (setf results (reverse results)))
           ((setf result
                     (connect-clusters-aux5
                      (car main-cluster-s-open-ports)
                      (car other-main-s-open-ports)
                      (car cluster-s-open-ports)
                      (remove (car cluster-s-open-ports)
                       (car all-s-open-ports)
                       :test #'equalp

17
(:count 1)
(car other-s-clusters)
(car s-arcs)
(car s-component-names))
(connect-clusters-aux4
 (cdr main-cluster-s-open-ports)
 (cdr other-main-s-open-ports)
 (cdr cluster-s-open-ports)
 (cdr all-s-open-ports)
 (cdr other-s-clusters)
 (cdr s-arcs)
 (cdr s-component-names)
 (cons result results)))
(t nil))

; This function works on a single state of a single open port.
; Send this function:
; 1. main-cluster-s-open-port: the first s-open-port from the main-s-cluster
; 2. main-cluster-s-open-ports: the rest of the s-open-ports from the main-s-cluster
; 3. cluster-s-open-port: the first s-open-port from the s-cluster-to-merge
; 4. cluster-s-open-ports: the rest of the s-open-ports from the s-cluster-to-merge
; 5. other-s-clusters: all s-clusters besides main-s-cluster and s-cluster-to-merge
; 6. s-arcs: s-arcs from the partial-state-topology
; 7. s-component-names: s-component-names from the partial-state-topology
; and it will return a partial-state-topology. Note that the aux3 function already
; determined that the main-cluster-s-open-port and the cluster-s-open-port have
; matching port specifications.

(defun connect-clusters-aux5
 (main-cluster-s-open-port
 main-cluster-s-open-ports
 cluster-s-open-port
 cluster-s-open-ports
 other-s-clusters
 s-arcs
 s-component-names)
 (let
 (new-s-cluster
 new-s-clusters
 new-s-arc
 new-s-arcs
 new-s-component-names
 new-partial-state-topology)
 (cond ((and (null main-cluster-s-open-ports) ; if 2 single port
 (null cluster-s-open-ports) ; clusters are to be merged and other
 (not (null other-s-clusters))) ; clusters remain, this is not a
 nil)
 (t (setf new-s-cluster
 (append main-cluster-s-open-ports cluster-s-open-ports))
 (if (null new-s-cluster)
 (setf new-s-clusters other-s-clusters)
 (setf new-s-clusters
 (append (list new-s-cluster) other-s-clusters)))
 (cond ((string< (car main-cluster-s-open-port) (car cluster-s-open-port))
 (setf new-s-arc
 (append main-cluster-s-open-port cluster-s-open-port)))
 (t (setf new-s-arc
 (append cluster-s-open-port main-cluster-s-open-port))))
 (setf new-s-arcs (append s-arcs (list new-s-arc)))
 (setf new-s-component-names
 (list
 (car main-cluster-s-open-port)
 (car cluster-s-open-port))))
(setf new-s-component-names
  (union s-component-names
    new-s-component-names
    test #'string-equal))
(setf new-partial-state-topology
  (list
    new-s-clusters
    new-s-arcs
    new-s-component-names))))

; Send this function a partial topology and it will return a list of topologies
; created by connecting all open-ports in the main cluster that match the first
; open-port of the main cluster.

(defun connect-open-ports (topology)
  (let
    (partial-state-topologies
      clusters
      main-cluster
      other-s-clusters
      first-s-open-ports
      rest-s-open-ports
      redundant-s-open-ports
      non-redundant-s-open-ports)
    (cond ((string-equal (car topology)
                             "complete") ; if topology is not partial
      (print
        'trouble-in-generate-required-topologies))
    (t (setf partial-state-topologies
              (caddr topology))
      (setf clusters
        (mapcar
          'car
          partial-state-topologies))
      (setf main-cluster
        (mapcar 'car
          (cadr topology)))
      (cond ((and
               (> (length (car clusters)) 1)
               (= (length (car main-cluster)) 2)) ; if > 1 cluster and
            ; exactly 2 open ports in
            ; main cluster, then
          (return nil (split graph))
        (t (setf other-s-clusters
              (mapcar 'cdr
                clusters))
          (setf first-s-open-ports
            (mapcar 'car main-cluster)
          (setf rest-s-open-ports
            (mapcar 'cdr main-cluster)
          (setf redundant-s-open-ports
            (transpose-list
              rest-s-open-ports)
          (setf non-redundant-s-open-ports
            (remove-duplicates
              redundant-s-open-ports
              :test #'equalp))
          (connect-open-ports-aux
            first-s-open-ports
            non-redundant-s-open-ports
            rest-s-open-ports
            other-s-clusters
            (mapcar 'cadr partial-state-topologies)
            (mapcar 'caddr partial-state-topologies)
            (cadr topology))))))))}
; This function parses thru the non-redundant open-ports of the main cluster
; Send this function:
; 1. first-s-open-ports: all states of the first open port in the main cluster
; 2. non-redundant-s-open-ports: all states of other open-ports in the main cluster
; 3. rest-s-open-ports: all states of the non-first open-ports in the main cluster
; 4. other-s-clusters: ((s1-c3 s1-c4) (s2-c3 s2-c4) (s3-c3 s3-c4))
; 5. s-arcs: (s1-s-arcs s2-s-arcs ....)
; 6. s-component-names: (s1-s-component-names s2-s-component-names ....)
; 7. abstraction-level
; and it returns a list of partial topologies.

(defun connect-open-ports-aux1
  (first-s-open-ports
   non-redundant-s-open-ports
   rest-s-open-ports
   other-s-clusters
   s-arcs
   s-component-names
   abstraction-level
   optional results)
  (let
    (result)
    (cond
      ((null non-redundant-s-open-ports) (reverse results))
      (setf result
        (connect-open-ports-aux2
         first-s-open-ports
         (car non-redundant-s-open-ports)
         rest-s-open-ports
         other-s-clusters
         s-arcs
         s-component-names
         abstraction-level))
      (connect-open-ports-aux1
       (cdr non-redundant-s-open-ports)
       rest-s-open-ports
       other-s-clusters
       s-arcs
       s-component-names
       abstraction-level
       (cons (list "partial" abstraction-level result 0) results)))
      (t (connect-open-ports-aux1
          first-s-open-ports
          (cdr non-redundant-s-open-ports)
          rest-s-open-ports
          other-s-clusters
          s-arcs
          s-component-names
          abstraction-level
          results))))

; Send this function:
; 1. first-s-open-ports: all states of the first open port in the main cluster
; 2. other-s-open-ports: all states of another open port in the main cluster
; 3. rest-s-open-ports: all states of the non-first open-ports in the main cluster
; 4. other-s-clusters: ((s1-c3 s1-c4) (s2-c3 s2-c4) (s3-c3 s3-c4))
; 5. s-arcs: (state1-s-arcs state2-s-arcs ....)
; 6. s-component-names: (state1-s-component-names state2-s-component-names ....)
; 7. abstraction-level
; and it calls aux3 if all s-open-ports have matching port specifications, else it
; returns nil.

(defun connect-open-ports-aux2
  (state1-s-arcs state2-s-arcs ....)
  (state1-s-component-names state2-s-component-names ....)
  abstraction-level
  (aux3)
(first-s-open-ports
  other-s-open-ports
  rest-s-open-ports
  other-s-clusters
  s-arcs
  s-component-names
  abstraction-level)

(let ()
  (cond ((string-equal (caar first-s-open-ports) ; if the open ports are from
      (caar other-s-open-ports)) ; the same component
    nil)
    ((all-matching-port-specifications? ; return nil
      (mapcar 'cadr first-s-open-ports)
      (mapcar 'cadr other-s-open-ports)
      abstraction-level)
      (connect-open-ports-aux3
       first-s-open-ports
       other-s-open-ports
       rest-s-open-ports
       other-s-clusters
       s-arcs
       s-component-names))
    (t nil))))

; This function works on multiple states of a single open port.
; Send this function:
; 1. first-s-open-ports: all states of the first open-port in the main cluster
; 2. other-s-open-ports: all states of another open-port in the main cluster
; 3. rest-s-open-ports: all states of the non-first open-ports in the main cluster
; 4. other-s-clusters: ((s1-c3 s1-c4) (s2-c3 s2-c4) (s3-c3 s3-c4))
; 5. s-arcs: (state1-s-arcs state2-s-arcs .........)
; 6. s-component-names: (state1-s-component-names state2-s-component-names .........)
; and it will return a partial-topology. Note that the aux2 function already
; determined that all first-s-open-ports and other-s-open-ports have
; matching port specifications.

(defun connect-open-ports-aux3
  (first-s-open-ports
   other-s-open-ports
   rest-s-open-ports
   other-s-clusters
   s-arcs
   s-component-names
   optional results)
  (let (result)
    (cond ((null first-s-open-ports) (setf results (reverse results)))
      (t (setf result
        (connect-open-ports-aux4
         (car first-s-open-ports)
         (car other-s-open-ports)
         (remove (car other-s-open-ports)
            (car rest-s-open-ports)
            :test #'equalp
            :count 1)
         (car other-s-clusters)
         (car s-arcs)
         (car s-component-names)))
        (connect-open-ports-aux3
         (cdr first-s-open-ports)
         (cdr other-s-open-ports)
         (cdr rest-s-open-ports)
         (cdr other-s-clusters)
         result
         s-arcs
         s-component-names
         results)))))

21
(cdr s-arcs)
(cdr s-component-names)
(cons result results))))

; This function works on a single state of a single open port.
; Send this function:
; 1. first-s-open-port: the first s-open-port from the main-s-cluster
; 2. other-s-open-port: another s-open-port from the main-s-cluster
; 3. rest-s-open-ports: the rest of the s-open-ports from the main-s-cluster
; 4. other-s-clusters: all other s-clusters besides main-s-cluster
; 5. s-arcs: s-arcs from the partial-state-topology
; 6. s-component-names: s-component-names from the partial-state-topology
; and it will return a partial-state-topology. Note that the aux2 function already
; determined that the first-s-open-port and the other-s-open-port have
; matching port specifications.

defun connect-open-ports-aux4
  (first-s-open-port
   other-s-open-port
   rest-s-open-ports
   other-s-clusters
   s-arcs
   s-component-names)
(let
  (new-s-cluster
   new-s-clusters
   new-s-arc
   new-s-arcs
   new-s-component-names
   new-partial-state-topology)
  (setf new-s-cluster rest-s-open-ports)
  (if (null new-s-cluster)
      (setf new-s-clusters other-s-clusters)
      (setf new-s-clusters (append (list new-s-cluster) other-s-clusters)))
  (cond ((string< (car first-s-open-port) (car other-s-open-port))
      (setf new-s-arc (append first-s-open-port other-s-open-port)))
    (t (setf new-s-arc (append other-s-open-port first-s-open-port))))
  (setf new-s-arcs (append s-arcs (list new-s-arc)))
  (setf new-s-component-names
      (list (car first-s-open-port) (car other-s-open-port)))
  (setf new-s-component-names
      (union s-component-names new-s-component-names :test #'string-equal))
  (setf new-partial-state-topology
      (list new-s-clusters new-s-arcs new-s-component-names))))

; Send this function a partial topology and it will return a list of topologies that
; result from merging the first open port and all matching components.

defun find-matching-components (topology)
(let
  (unique-s-component-names
   optional-component-names
   component-pairs
   same-level-optional-components
   under-limit-optional-components)
  (cond ((string-equal (car topology) "complete")
      (print 'trouble-in-find-matching-components))
    (t (setf unique-s-component-names
      (caddr (car (caddr topology))))
      (setf optional-component-names
      (get-optional-names
      (mapcar 'remove-state-suffix
      unique-s-component-names
      component-pairs
      (mapcar (partial-match-component-names
      (list topology unique-s-component-names))
      (list topology same-level-optional-components)
      (list topology under-limit-optional-components))))))
  (setf new-partial-state-topology
      (list new-s-clusters new-s-arcs new-s-component-names))))

22
(mapcar 'remove-unique-suffix
  unique-s-component-names))

(setf component-pairs
  (make-component-pairs
   optional-component-names))

(setf same-level-optional-components
  (get-optional-components-of-level
   (cadr topology))

(if (null component-pairs)
  (setq under-limit-optional-components
       same-level-optional-components)

  (setq under-limit-optional-components
       (delete-over-limit-components
        component-pairs
        same-level-optional-components))

(find-matching-components-aux1
  topology
  (cadr topology)
  (caddr topology)
  under-limit-optional-components
  (mapcar 'convert-usn-to-n
           unique-s-component-names)))))

; This function parses through the components.

(defun find-matching-components-aux1
  (topology
   abstraction-level
   partial-state-topologies
   components
   topology-component-names
   optional results)

  (let
   (component
    component-name
    occurences
    unique-component-name
    s-components
    unique-s-components
    list-of-s-open-ports
    transposed-list-of-s-open-ports
    list-of-non-redundant-s-open-ports
    result)
   (cond ((null components) results)
         (t (setf component (car components))
           (cond ((and (= (length (caaar (caddr topology)))
                          1)
                   (= (length (caaddr (caaddr component)))
                          1)
                   (> (length (caar (caddr topology))) 1)))
                 (find-matching-components-aux1
                  topology
                  abstraction-level
                  partial-state-topologies
                  (cadr components)
                  topology-component-names
                  results))
                 (t (setf component-name (car component))
                   (setf occurences
                           (count component-name
                                    topology-component-names
                                    :test #'string-equal))
                   (setf unique-component-name
                           (make-name-unique
                            results)))))))
component-name
(+ 1 occurences)))
(setf s-components (caddr component))
(setf unique-s-components
(make-some-s-components-unique
s-components
(+ 1 occurences)))
(setf list-of-s-open-ports
(create-s-open-ports-from-s-components
unique-s-components))
(setf transposed-list-of-s-open-ports
(t((p1-s1 p1-s2 p1-s3)
(transpose-list
(p2-s1 p2-s2 p2-s3)
list-of-s-open-ports))
p3-s1 p3-s2 p3-s3))
(if (string-equal abstraction-level "level1")
(setf list-of-non-redundant-s-open-ports
(remove-duplicates
transposed-list-of-s-open-ports
:test #'equalp))
(setf list-of-non-redundant-s-open-ports
transposed-list-of-s-open-ports))
(setf result
(find-matching-components-aux2
list-of-s-open-ports
list-of-non-redundant-s-open-ports
partial-state-topologies
topology
abstraction-level))
(find-matching-components-aux1
topology
abstraction-level
partial-state-topologies
(cdr components)
topology-component-names
(append results result))))))

; This function parses through the ports of a single component.
; Send this function:
; 1. list-of-s-open-ports: ((p1-s1 p2-s1 p3-s1) (p1-s2 p2-s2 p3-s2)
; (p1-s3 p2-s3 p3-s3))
; 2. list-of-non-redundant-s-open-ports: ((p1-s1 p1-s2 p1-s3) (p2-s1 p2-s2 p2-s3))
; 3. partial-state-topologies:
; 4. topology:
; 5. abstraction-level:

(defun find-matching-components-aux2
(list-of-s-open-ports
list-of-non-redundant-s-open-ports
partial-state-topologies
topology
abstraction-level
&optional results)
(let
(result)
(cond ((null list-of-non-redundant-s-open-ports) results)
((setf result
(find-matching-components-aux3
list-of-s-open-ports
(car
list-of-non-redundant-s-open-ports)
partial-state-topologies
abstraction-level))
(these are the non-redundant
states of a single port)
(find-matching-components-aux2
list-of-s-open-ports
(cdr list-of-non-redundant-s-open-ports)
partial-state-topologies
topology
abstraction-level
(append results
 (generate-some-new-topologies
 result
topology)))))
(t (find-matching-components-aux2
 list-of-s-open-ports
 (cdr list-of-non-redundant-s-open-ports)
 partial-state-topologies
topology
abstraction-level
results)))))

; This function parses through the partial-state-topologies.
; Send this function:
; 1. list-of-s-open-ports: ((p1-s1 p2-s1 p3-s1) (p1-s2 p2-s2 p3-s2)
; (p1-s3 p2-s3 p3-s3))
; 2. non-redundant-s-open-ports: (p1-s1 p1-s2 p1-s3)
; 3. partial-state-topologies:
; 4. abstraction-level:

defun find-matching-components-aux3
(list-of-s-open-ports
 non-redundant-s-open-ports
 partial-state-topologies
 abstraction-level
 &optional results)
(let
(partial-state-topology
 s-clusters
 main-s-cluster
 main-s-open-port
 new-main-s-cluster
 main-s-component-name
 main-open-port-specification
 new-state-topologies)
(cond ((null partial-state-topologies) (reverse results))
(t (setf partial-state-topology (car partial-state-topologies))
 (setf s-clusters (car partial-state-topology))
 (setf main-s-cluster (car s-clusters))
 (setf main-s-open-port (car main-s-cluster))
 (setf new-main-s-cluster (cdr main-s-cluster))
 (setf main-s-component-name (car main-s-open-port))
 (setf main-open-port-specification (cdr main-s-open-port))
 (setf new-state-topologies
 (find-matching-components-aux4
 list-of-s-open-ports
 non-redundant-s-open-ports
 s-clusters
 main-s-open-port
 new-main-s-cluster
 main-s-component-name
 main-open-port-specification
 partial-state-topology
 abstraction-level))
 (cond ((null new-state-topologies) nil)
 (t (find-matching-components-aux3
 list-of-s-open-ports
 non-redundant-s-open-ports
 (cdr partial-state-topologies)
 abstraction-level
 (cons new-state-topologies
 (find-matching-components-aux3
 list-of-s-open-ports
 non-redundant-s-open-ports
 partial-state-topologies
 abstraction-level))))))

25
This function parses through the states of a single port.
Send this function:
1. list-of-s-open-ports: ((p1-s1 p2-s1 p3-s1) (p1-s2 p2-s2 p3-s2) (p1-s3 p2-s3 p3-s3))
2. non-redundant-s-open-ports: (p1-s1 p1-s2 p1-s3)
3. s-clusters: all s-clusters for the partial-state-topology
4. main-s-open-port: the first s-open-port of the main-s-cluster
5. main-s-cluster: the main-s-cluster with main-s-open-port removed
6. main-s-component-name: the name of the main-s-component
7. main-open-port-specification: the topology open port specification
8. partial-state-topology: the partial-state-topology
9. abstraction-level: the abstraction level
and it will return a list of state-topologies created by merging component
states that have matching port specifications.

(defun find-matching-components-aux4
(list-of-s-open-ports
  non-redundant-s-open-ports
  s-clusters
  main-s-open-port
  main-s-cluster
  main-s-component-name
  main-open-port-specification
  partial-state-topology
  abstraction-level
  (optional results))
(let
  (non-redundant-s-open-port
    port-specification
    s-component-name
    new-main-s-cluster
    new-s-clusters
    new-s-arcs
    new-s-component-names
    new-partial-state-topology)
  (cond ((null non-redundant-s-open-ports) (reverse results))
    (t (setf non-redundant-s-open-port (car non-redundant-s-open-ports))
      (setf port-specification (cadr non-redundant-s-open-port))
      (cond ((matching-port-specifications?
          main-open-port-specification
          abstraction-level)
        (setf s-component-name
          (car
            non-redundant-s-open-port))
        (setf new-main-s-cluster
          (append
            main-s-cluster
            (remove
              non-redundant-s-open-port
              (car list-of-s-open-ports)
              :test #'equalp
              :count 1)))
        (if (null new-main-s-cluster)
          (setf new-s-clusters (cdr s-clusters))
          (setf new-s-clusters
            (append
              (list new-main-s-cluster)
              (cdr s-clusters))))
        (cond ((string< main-s-component-name
          s-component-name)
(setf new-s-arc
  (list
    main-s-component-name
    main-open-port-specification
    s-component-name
    port-specification))
(t (setf new-s-arc
  (list
    s-component-name
    port-specification
    main-s-component-name
    main-open-port-specification))))
(setf new-s-arcs (append
  (cadr partial-state-topology)
  (list new-s-arc)))
(setf new-s-component-names (union
  (cadr partial-state-topology)
  (list main-s-component-name
    s-component-name)
  :test #'string-equal))
(setf new-partial-state-topology (list
  new-s-clusters
  new-s-arcs
  new-s-component-names))
(find-matching-components-aux4
  (cdr list-of-s-open-ports)
  (cdr non-redundant-s-open-ports)
  s-clusters
  main-s-open-port
  main-s-cluster
  main-s-component-name
  main-open-port-specification
  partial-state-topology
  abstraction-level
  (cons new-partial-state-topology
    results))))
(t (find-matching-components-aux4
  (cdr list-of-s-open-ports)
  (cdr non-redundant-s-open-ports)
  s-clusters
  main-s-open-port
  main-s-cluster
  main-s-component-name
  main-open-port-specification
  partial-state-topology
  abstraction-level
  results))))))

; Send this function sets of state-topologies and the original topology and
; it will return a list of new topologies.

(defun generate-some-new-topologies
  (state-topologies topology)
  (let
    (state-topology-combinations)
    (setf state-topology-combinations (generate-combinations state-topologies))
    (generate-some-new-topologies-aux1 state-topology-combinations topology)))

(defun generate-some-new-topologies-aux1
  (state-topology-combinations topology &optional results)
  (let
(state-topologies
  new-topology)
(cond ((null state-topology-combinations) (reverse results))
  (t (setf state-topologies
    (car
      state-topology-combinations))
    ; this is one set of state-
    ; topologies for a single
    ; partial-topology
    (setf new-topology
      (substitute
        state-topologies
        (caddr topology)
        topology))
    ; replace old state-topologies
    ; with new state-topologies
    ; in the partial topology
    (generate-some-new-topologies-aux1
      (cdr state-topology-combinations)
      topology
      (cons new-topology results))))))

(defun create-s-open-ports-from-s-components
  (s-components &optional list-of-s-open-ports)
  (cond ((null s-components) (reverse list-of-s-open-ports))
    (t (create-s-open-ports-from-s-components
      (cons (create-s-open-ports-from-s-components-aux1
        (caar s-components)
        (caddr (car s-components)))
        list-of-s-open-ports))))

(defun create-s-open-ports-from-s-components-aux1
  (s-component-name s-port-specs &optional s-open-ports)
  (let
    (s-port-spec
      s-open-port)
    (cond ((null s-port-specs) (reverse s-open-ports))
      (t (setf s-port-spec (car s-port-specs))
      (setf s-open-port (list s-component-name s-port-spec))
      (create-s-open-ports-from-s-components-aux1
       s-component-name
       (cdr s-port-specs)
       (cons s-open-port s-open-ports))))))

(defun make-some-s-components-unique
  (s-components number &optional unique-s-components)
  (let
    (s-component
      s-component-name
      unique-s-component-name
      unique-s-component)
    (cond ((null s-components) (reverse unique-s-components))
      (t (setf s-component (car s-components))
      (setf s-component-name (car s-component))
      (setf unique-s-component-name (make-name-unique s-component-name number))
      (setf unique-s-component
        (list
          unique-s-component-name
          (caddr s-component)
          (caddr s-component)
          (caddr s-component)))
      (make-some-s-components-unique
       (cdr s-components)
       number
       (cons unique-s-component unique-s-components)))))

; Send this function the list ((a b c) (d e f) (g h i)) and it returns
; ((a d g) (b e h) (c f i)).
(defun transpose-list (list &optional results)
  (cond ((null (car list)) (reverse results))
        (t (transpose-list
            (mapcar 'cdr list)
            (cons (mapcar 'car list) results))))))

; Send this function the list (a b c) and it returns (b c a).

(defun rotate-list (list)
  (append (cdr list) (list (car list))))

; Send this function two lists of port specifications (of the same length) and it
; returns t if all port specifications match (on an ordered basis), else nil.

(defun all-matching-port-specifications?
  (list-of-port-specs-1 list-of-port-specs-2 abstraction-level)
  (cond ((null list-of-port-specs-1) t)
       ((matching-port-specifications?
         (car list-of-port-specs-1)
         (car list-of-port-specs-2)
         abstraction-level)
        (all-matching-port-specifications?
         (cdr list-of-port-specs-1)
         (cdr list-of-port-specs-2)
         abstraction-level))
       (t nil)))

(defun make-component-pairs (component-names &optional component-pairs)
  (let
    (new-component-names)
    (cond ((null component-names) (reverse component-pairs))
          (t (setf new-component-names
                   (remove (car component-names)
                         component-names
                         :test #'string-equal))
             (make-component-pairs
              new-component-names
              (cons
               (list (car component-names)
                     (- (length component-names)
                        (- (length new-component-names))
                        component-pairs)))))))

; Send this function a list of partial topologies and it returns only those that
; have fewer open-ports than the *open-port-limit* and that have fewer components
; than the *component-limit*.

(defun eliminate-ungainly-topologies (topologies &optional new-topologies)
  (let
    (topology)
    (cond ((null topologies) (reverse new-topologies))
          (t (setf topology (car topologies))
             (cond ((or (> (length (caaar (caddr topology)))
                           *open-port-limit*)
                     (> (length (caddar (caddr topology)))
                        *component-limit*))
                 (eliminate-ungainly-topologies
                  (cdr topologies)
                  new-topologies))
                 (t (eliminate-ungainly-topologies
                     (cdr topologies)
                     (cons topology
                      new-topologies))))))))
(defun get-optional-components-of-level (abstraction-level)
  (gocol-auxl abstraction-level *optional-components*))

(defun gocol-aux1
  (abstraction-level optional-components &optional same-level-components)
  (cond ((null optional-components) (reverse same-level-components))
        ((string-equal (cadar optional-components) abstraction-level)
          (gocol-aux1
           abstraction-level
           (cdr optional-components)
           (cons (car optional-components) same-level-components)))
        (t (gocol-aux1
            abstraction-level
            (cdr optional-components)
            same-level-components))))

; Send this function:
; 1. topology-components: a list of pairs, where each pair is an optional component
;  name and the number of times that component appears in
;  the topology
; 2. components-to-test: a list of optional components that you would like to
;  add to the topology
; and it returns those optional components that do not exceed the limit.

(defun delete-over-limit-components
  (topology-components components-to-test &optional under-limit-components)
(let
  (component-to-test
   component-name
   component-pair
   number-in-topology
   limit)
  (cond ((null components-to-test) (reverse under-limit-components))
        (t (setf component-to-test (car components-to-test))
            (setf component-name (car component-to-test))
            (setf component-pair
                  (assoc component-name
                          topology-components
                          :test #'string-equal))
            (if (null component-pair)
                (setf number-in-topology 0)
                (setf number-in-topology (cadr component-pair)))
            (setf limit
                   (cdr
                    (assoc component-name
                            *optional-component-limits*
                            :test #'string-equal)))
            (cond ((< number-in-topology limit)
                           (delete-over-limit-components
                            topology-components
                            (cdr components-to-test)
                            (cons component-to-test
                                   under-limit-components)))
                (t (delete-over-limit-components
                    topology-components
                    (cdr components-to-test)
                    under-limit-components))))))

; Send this function a partial topology and it returns a complete topology
; if it is one, else it returns the partial topology.

(defun make-complete-topology-complete (partial-topology)
  (let
(partial-state-topologies
  partial-state-topology
  s-clusters)
(setf partial-state-topologies (caddr partial-topology))
(setf partial-state-topology (car partial-state-topologies))
(setf s-clusters (car partial-state-topology))
(cond ((and
        (string-equal (car partial-topology) "partial")
        (null s-clusters))
        (make-topology-complete partial-topology))
       (t partial-topology)))

; Send this function an partial topology and it makes it complete and re-ranks it.

(defun make-topology-complete (partial-topology)
  (let
    (partial-state-topologies
     complete-state-topologies
     complete-topology)
    (setf partial-state-topologies (caddr partial-topology))
    (setf complete-state-topologies
         (mapcar 'make-topology-complete-aux1 partial-state-topologies))
    (setf complete-topology
         (list
          "complete"
          (cadr partial-topology)
          complete-state-topologies
          (cadddr partial-topology)))))

(defun make-topology-complete-aux1 (partial-state-topology)
  (list (cadr partial-state-topology) (caddr partial-state-topology)))

; Send this function a list of topologies and it returns those topologies that are
; not type II duplicates. It also adds to the *duplicate-list* every topology that
; is not on the list.

(defun eliminate-duplicate-topologies (topologies optional results)
  (let
    (topology
     state-topology
     s-arcs
     ordered-s-arcs)
    (cond ((null topologies) (reverse results))
          (t (setf topology (car topologies))
             (setf state-topology (caaddr topology))
             (if (string-equal (car topology) "partial")
                 (setf s-arcs (cadr state-topology))
                 (setf s-arcs (car state-topology)))
             (setf s-arcs
                  (mapcar #'(lambda (a)
                              (concatenate
                               'string
                               (car a)
                               (concatenate-list (adr a))
                               (caddr a)
                               (concatenate-list (cadddr a))))
                               s-arcs)))
             (setf ordered-s-arcs (sort s-arcs '#'<string>))
             (setf ordered-s-arcs (concatenate-list ordered-s-arcs))
             (cond ((member ordered-s-arcs
                           *duplicate-list*
                           :test #'<string-equal))
                   (eliminate-duplicate-topologies
                    (append topologies (make-topology-complete-aux1 partial-state-topology)
                                optional results)))))

31
(cdr topologies)
(results))
(t (setf *duplicate-list*
  (cons ordered-s-arcs
   *duplicate-list*))
  (eliminate-duplicate-topologies
   (cdr topologies)
   (cons topology
    results)))))))

(defun concatenate-list (list)
  (cond ((null list) "")
    (t (concatenate 'string (car list) (concatenate-list (cdr list))))))

******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************

(defun instantiate-topology (complete-topology)
  (let
    (instantiation-level
     unique-s-component-names
     unique-optional-s-component-names
     unique-optional-s-offspring
     required-states
     required-parent-s-names
     offspring-s-components-with-parents
     component-combinations
     s-arcs-to-replace
     offspring-s-components
     answers)
    (setf instantiation-level
      (get-instantiation-level
       (cdr complete-topology)))
    (setf unique-s-component-names
      (mapcar 'cadr
               (caddr complete-topology)))
    (setf unique-optional-s-component-names
      (mapcar 'get-optional-names
               (mapcar 'unique-s-component-names)))
    (setf unique-optional-s-offspring
      (mapcar 'get-unique-s-offspring
               (mapcar 'unique-optional-s-component-names)))
    (setf required-states
      (car
       (cdr
        (assoc instantiation-level
         "required-states" :test #'string-equal))))
    (setf required-parent-s-names
      (double-mapcar
       'car
       required-states))
    (setf offspring-s-components-with-parents
      (get-offspring-s-components-with-parents
       required-parent-s-names
       required-states
       unique-s-component-names
       unique-optional-s-offspring))
    (setf component-combinations
      ;unique s-component names of all
      ;s-components in topology to
      ;instantiate
      ;those unique s-component names
      ;that are optional s-components
      ;these are the optional s-offspring
      ;for each state, including rotated
      ;offspring, with parent s-component
      ;name consed to front
      ;required s-components with s-
      ;parent name consed to front for the
      ;instantiation level
      ;required parent s-names
      ;n lists (where n = number of
      ;required topology states at the
      ;instantiation level) of
      ;s-components (both optional and
      ;required) with parent s-names
      ;consed to front
      ;all component combinations
      ;for each state
      (list))))

32
(mapcar
 'generate-combinations
 offspring-s-components-with-parents))
(setf s-arcs-to-replace
 (get-s-arcs-to-replace
  required-parent-s-names
  unique-s-component-names
  (mapcar 'car
   (caddr complete-topology)))))

(setf
 offspring-s-components-with-parents
 (remove-duplicates
  (unlist-offspring-s-components-with-parents
   offspring-s-components-with-parents)))

(setf
 offspring-s-components
 (mapcar
  'cadr
  offspring-s-components-with-parents))

(setf
 answers
 (instantiate-component-combinations
  (double-mapcar
   'listify-required-components
   component-combinations
   s-arcs-to-replace
   offspring-s-components
   instantiation-level))))

(defun convert-s-arc-into-arc (s-arc)
 (list (convert-usn-to-un (car s-arc)) (cadr s-arc)
        (convert-usn-to-un (caddr s-arc)) (caddrr s-arc)))

(defun get-optimal-names (names optional optional-names)
 (cond ((null names) (reverse optional-names))
       ((required-name? (car names)) (get-optimal-names (cdr names) optional-names))
       (t (get-optimal-names (cdr names) (cons (car names) optional-names))))))

(defun required-name? (name)
 (let
  (name2 name)
  (setf name2 (remove-component-name name))
  (setf name2 (remove-required-prefix name2))
  (if (string-equal name2 nil t)))); Send this function a unique optional s-component name and it returns all unique
 ; s-component offspring with unique parent s-component name consed to front.

(defun get-unique-s-offspring (unique-optoinal-s-component-names optional results)
 (let
  (unique-optoinal-s-component-name unique-offspring)
  (cond ((null unique-optoinal-s-component-names) (reverse results))
        (t (setf unique-optoinal-s-component-name
                 (car unique-optoinal-s-component-names))
           unique-offspring
           (get-unique-offspring-from-unique-name
            unique-optoinal-s-component-name))
        (get-unique-s-offspring
         (cadr unique-optoinal-s-component-names)
         (cons unique-offspring results))))))
component-name
unique-suffix
unique-component-name
component
s-components
s-component
offspring-s-component-names)
(setf s-component-name (remove-unique-suffix unique-s-component-name))
(setf component-name (remove-state-suffix s-component-name))
(setf unique-suffix
     (remove-state-prefix (remove-component-name unique-s-component-name)))
(setf unique-component-name
     (concatenate 'string component-name unique-suffix))
(setf component
     (assoc component-name *optional-components* :test #'string-equal))
(setf s-components (caddr component))
(setf s-component
     (assoc s-component-name s-components :test #'string-equal))
(setf offspring-s-component-names (caddr s-component)
     (get-unique-offspring-from-unique-name-aux1
      offspring-s-component-names unique-s-component-name unique-suffix))))

(defun get-unique-offspring-from-unique-name-aux1
     (offspring-s-component-names unique-parent-s-name unique-suffix
      &optional results)
     (let
       (single-component-offspring)
       (cond ((null offspring-s-component-names) (reverse results))
             (t (setf single-component-offspring (car offspring-s-component-names))
                (get-unique-offspring-from-unique-name-aux1
                 (cadr offspring-s-component-names)
                 unique-parent-s-name
                 unique-suffix
                (cons (get-unique-offspring-from-unique-name-aux2
                       single-component-offspring
                       unique-parent-s-name
                       unique-suffix)
                      results))))))

(defun get-unique-offspring-from-unique-name-aux2
     (offspring-s-component-names unique-parent-s-name unique-suffix
      &optional results)
     (let
       (offspring-s-component-name
        offspring-name
        offspring
        s-offspring
        unique-s-offspring
        rotated-offspring
        unique-s-offspring-with-parent-name)
       (cond ((null offspring-s-component-names) results)
             (t (setf offspring-s-component-name (car offspring-s-component-names))
                (remove-state-suffix offspring-s-component-name))
                (setf offspring
                (assoc offspring-name *optional-components* :test #'string-equal))
                (cond ((null offspring)
                        (get-unique-offspring-from-unique-name-aux2
                         (cadr offspring-s-component-names)
                         unique-parent-s-name
                         unique-suffix
                         results))
                        (t (setf s-offspring
                               (assoc offspring-s-component-name
                                  offspring-name
                                  offspring
                                  s-offspring
                                  unique-s-offspring
                                  rotated-offspring
                                  unique-s-offspring-with-parent-name)
                                  unique-suffix
                                  results))))))

34
(caddr offspring) :test #'string-equal)
(setf unique-s-offspring
  (list
    (concatenate 'string
      offspring-s-component-name
      unique-suffix)
    (cadr s-offspring)
    (caddr s-offspring)
    (caddr s-offspring)))
(setf rotated-offspring (list unique-s-offspring))
(setf unique-s-offspring-with-parent-name
  (get-unique-offspring-from-unique-name-aux3
    unique-parent-s-name
    rotated-offspring))
(get-unique-offspring-from-unique-name-aux2
  (cdr offspring-s-component-names)
  unique-parent-s-name
  unique-suffix
  (append results
    unique-s-offspring-with-parent-name)))))))

(defun get-unique-offspring-from-unique-name-aux3
  (parent-s-name s-components &optional results)
  (cond ((null s-components) (reverse results))
        (t (get-unique-offspring-from-unique-name-aux3
            parent-s-name
            (cdr s-components)
            (cons (list parent-s-name (car s-components)) results))))

; Send this function an s-component and it returns all valid rotations of the
; s-component. This is where this code accommodates type I duplicates in topology
; instantiation instead of in topology generation, as it used to be done.

(defun rotate-s-component (s-component)
  (let
    (port-specs
      parent-port-specs
      combined-specs
      new-specs
      new-port-specs-list
      rotated-s-components)
    (setf port-specs (caddr s-component))
    (setf parent-port-specs (mapcar 'remove-last-item port-specs))
    (setf combined-specs (combine-specs port-specs parent-port-specs))
    (setf new-specs (rotate-s-component-aux1 combined-specs combined-specs))
    (setf new-specs (generate-combinations new-specs))
    (setf new-port-specs-list (rotate-s-component-aux2 port-specs new-specs))
    (setf rotated-s-components
      (rotate-s-component-aux3
        (car s-component)
        (cadr s-component)
        (caddr s-component)
        new-port-specs-list))))

(defun rotate-s-component-aux1 (combined-specs all-combined-specs &optional results)
  (let
    (result)
    (cond ((null combined-specs) (reverse results))
          (t (setf result (assoc-all (car combined-specs) all-combined-specs))
            (setf result (mapcar 'cdr result))
            (setf result (remove-duplicates result :test #'equalp))
            (rotate-s-component-aux1
              (cadr combined-specs)
              (caddr combined-specs)
              new-port-specs-list))))

35
(defun rotate-s-component-aux2 (port-specs new-specs-list &optional results)
  (let
    (new-specs)
    (cond ((null new-specs-list) (reverse results))
          (t (setf new-specs (car new-specs-list))
              (cond ((and (onto? port-specs new-specs) (onto? new-specs port-specs))
                    (rotate-s-component-aux2
                     port-specs
                     (cdr new-specs-list)
                     (cons new-specs results)))
                   (t (rotate-s-component-aux2
                        port-specs
                        (cdr new-specs-list)
                        results)))))))

(defun rotate-s-component-aux3
  (name parents children port-specs-list &optional results)
  (cond ((null port-specs-list) (reverse results))
        (t (rotate-s-component-aux3
            name
            parents
            children
            (cdr port-specs-list)
            (cons (list name parents children (car port-specs-list)) results))))))

(defun onto? (list1 list2)
  (cond ((null list1) (not list2))
        (t (onto?
            (cdr list1)
            (remove (car list1) list2 :test #'equalp :count 1))))))

(defun remove-last-item (list)
  (reverse (cdr (reverse list)))))

(defun combine-specs (port-specs parent-specs &optional combined-specs)
  (cond ((null port-specs) (reverse combined-specs))
        (t (combine-specs
            (cdr port-specs)
            (cdr parent-specs)
            (cons (list (car parent-specs) (car port-specs)) combined-specs))))))

(defun assoc-all (item list &optional associations)
  (cond ((null list) (reverse associations))
        (t (assoc-all item (cdr list) (cons (car list) associations))))))

(defun get-offspring-s-components-with-parents
  (required-parent-s-names
   required-states
   unique-s-component-names
   unique-optional-s-offspring
   &optional offspring-s-components-with-parents)
  (let
    ()
    (cond ((null required-parent-s-names)
            (reverse offspring-s-components-with-parents))
          (t (get-offspring-s-components-with-parents
              (cdr required-parent-s-names)
              (cdr required-states)
              unique-s-component-names
              required-states)))

(defun assoc-all (item list &optional associations)
  (cond ((null list) (reverse associations))
        (t (assoc-all item (cdr list) (cons (car list) associations))))))

(defun combine-specs (port-specs parent-specs &optional combined-specs)
  (cond ((null port-specs) (reverse combined-specs))
        (t (combine-specs
            (cdr port-specs)
            (cdr parent-specs)
            (cons (list (car parent-specs) (car port-specs)) combined-specs))))))

(defun get-offspring-s-components-with-parents
  (required-parent-s-names
   required-states
   unique-s-component-names
   unique-optional-s-offspring
   &optional offspring-s-components-with-parents)
  (let
    ()
    (cond ((null required-parent-s-names)
            (reverse offspring-s-components-with-parents))
          (t (get-offspring-s-components-with-parents
              (cdr required-parent-s-names)
              (cdr required-states)
              unique-s-component-names
              required-states)))

(defun assoc-all (item list &optional associations)
  (cond ((null list) (reverse associations))
        (t (assoc-all item (cdr list) (cons (car list) associations))))))

(defun get-offspring-s-components-with-parents
  (required-parent-s-names
   required-states
   unique-s-component-names
   unique-optional-s-offspring
   &optional offspring-s-components-with-parents)
  (let
    ()
    (cond ((null required-parent-s-names)
            (reverse offspring-s-components-with-parents))
          (t (get-offspring-s-components-with-parents
              (cdr required-parent-s-names)
              (cdr required-states)
              unique-s-component-names
              required-states)))
unique-optional-s-offspring
  (cons (get-offspring-s-components-with-parents-aux1
    (car required-parent-s-names)
    (car required-states)
    unique-s-component-names
    unique-optional-s-offspring)
  offspring-s-components-with-parents))))

(defun get-offspring-s-components-with-parents-aux1
  (required-parent-s-names
   required-state
   unique-s-component-names
   unique-optional-s-offspring)
  (let
    ()
    (cond ((null unique-s-component-names)
      (print 'trouble-in-get-offspring-s-components-with-parents-aux1))
      ((null (set-difference
        required-parent-s-names
        (car unique-s-component-names)
        :test #'string-equal))
      (append (mapcar 'list required-state)
        (car unique-optional-s-offspring))
      (t (get-offspring-s-components-with-parents-aux1
        required-parent-s-names
        required-state
        (cdr unique-s-component-names)
        (cdr unique-optional-s-offspring))))))

(defun unlist-offspring-s-components-with-parents
  (offspring-s-components-with-parents)
  (let
    ()
    (setf offspring-s-components-with-parents
      (top-level-delistify
       offspring-s-components-with-parents))
    (setf offspring-s-components-with-parents
      (top-level-delistify
       offspring-s-components-with-parents))
    (unlist-offspring-s-components-with-parents-aux1
     offspring-s-components-with-parents))))

(defun unlist-offspring-s-components-with-parents-aux1
  (offspring-s-components-with-parents &optional results)
  (let
    (first-item)
    (cond ((null offspring-s-components-with-parents) results)
      (t (setf first-item (car offspring-s-components-with-parents))
      (cond ((listp (car first-item))
          (unlist-offspring-s-components-with-parents-aux1
           (cdr offspring-s-components-with-parents)
           (append results first-item))
          (t (unlist-offspring-s-components-with-parents-aux1
              (cdr offspring-s-components-with-parents)
              (append results (list first-item))))))))

; This function returns n lists of s-arcs to replace where n is the number of
; required states at the instantiation level.

(defun get-s-arcs-to-replace
  (required-parent-s-names
   unique-s-component-names
   list-of-s-arcs
   &optional new-list-of-s-arcs)
(cond ((null required-parent-s-names) (reverse new-list-of-s-arcs))
  (t (get-s-arcs-to-replace
       (cdr required-parent-s-names)
       unique-s-component-names
       list-of-s-arcs
       (cons (get-s-arcs-to-replace-aux1
                (car required-parent-s-names)
                unique-s-component-names
                list-of-s-arcs)
             new-list-of-s-arcs))))

(defun get-s-arcs-to-replace-aux1
  (required-parent-s-names
   unique-s-component-names
   list-of-s-arcs)
  (cond ((null unique-s-component-names)
         (print 'trouble-in-get-s-arcs-to-replace-aux1))
        (null (set-difference
               required-parent-s-names
               (car unique-s-component-names)
               :test #'string-equal))
        (car list-of-s-arcs))
        (t (get-s-arcs-to-replace-aux1
             required-parent-s-names
             (cdr unique-s-component-names)
             (cdr list-of-s-arcs))))

; Send this function:
; 1. s-arcs: a list of s-arcs to instantiate
; 2. offspring-s-components: all offspring-s-components with parent s-names
; 3. instantiation-level: the level to instantiate to

(defun get-s-arcs-with-validity-tags
  (s-arcs offspring-s-components instantiation-level &optional results)
  (let
    (s-arc
     ways-to-instantiate-s-arc)
    (cond ((null s-arcs) results)
          (t (setf s-arc (car s-arcs))
             (setf ways-to-instantiate-s-arc
                    (get-s-arcs-with-validity-tags-aux1
                     s-arc
                     offspring-s-components
                     instantiation-level))
             (get-s-arcs-with-validity-tags
              (cdr s-arcs)
              offspring-s-components
              instantiation-level
              (append results ways-to-instantiate-s-arc))))

; Send this function:
; 1. s-arc: an s-arc to instantiate
; 2. offspring-s-components: all offspring-s-components with parent s-name
; 3. instantiation-level: the level to instantiate to

(defun get-s-arcs-with-validity-tags-aux1
  (s-arc offspring-s-components instantiation-level)
  (let
    (from-parent-s-name
     from-s-offspring
     to-parent-s-name
     to-s-offspring
     s-component-pairs
     s-arcs-with-validity-tags)
(setf from-parent-s-name (car s-arc))
(setf from-s-offspring
  (get-s-arcs-with-validity-tags-aux2
   from-parent-s-name
   offspring-s-components))
(setf to-parent-s-name (caddr s-arc))
(setf to-s-offspring
  (get-s-arcs-with-validity-tags-aux2
   to-parent-s-name
   offspring-s-components))
(setf s-component-pairs
  (generate-combinations
   (list from-s-offspring
         to-s-offspring)))
(setf s-arcs-with-validity-tags
  (get-s-arcs-with-validity-tags-aux3
   s-component-pairs
   s-arc
   instantiation-level)))

; Send this function:
; 1. name: the name of a parent s-component
; 2. s-components: all offspring-s-components (with parent s-name consed to front)
; and it returns all s-components (without parent name on front) that are offspring
; of that parent component name.

(defun get-s-arcs-with-validity-tags-aux2
  (name s-components &optional results)
(let
  (s-component)
  (cond ((null s-components) (reverse results))
        (t (setf s-component (car s-components))
            (cond ((string-equal name (car s-component))
                   (get-s-arcs-with-validity-tags-aux2
                    name
                    (cdr s-components)
                    (cons (cadr s-component) results)))
                  (t (get-s-arcs-with-validity-tags-aux2
                        name
                        (cdr s-components)
                        results))))))

; This function parses through a list of pairs of s-components

(defun get-s-arcs-with-validity-tags-aux3
  (s-component-pairs s-arc instantiation-level &optional results)
(let
  (s-component-pair
   s-arc-with-validity-tag)
  (cond ((null s-component-pairs) (reverse results))
        (t (setf s-component-pair (car s-component-pairs))
            (setf s-arc-with-validity-tag
                   (get-s-arcs-with-validity-tags-aux4
                    s-component-pair
                    s-arc
                    instantiation-level))
            (get-s-arcs-with-validity-tags-aux3
             (cdr s-component-pairs)
             s-arc
             instantiation-level
             (cons s-arc-with-validity-tag results))))))

; Send this function:
; 1. s-component-pair: a pair of s-components at the instantiation level
; 2. s-arc: an s-arc to instantiate
; 3. instantiation-level: the level to instantiate to
; and it returns an s-arc connecting the two s-components with a 't added to the end
; if the s-arc is valid, or a 'nil added to the end if the s-arc is invalid.

(defun get-s-arcs-with-validity-tags-aux4
  (s-component-pair s-arc instantiation-level)
  (let
    (from-s-component to-s-component from-port-spec to-port-spec new-s-arc)
    (setf from-s-component (car s-component-pair))
    (setf to-s-component (cadr s-component-pair))
    (setf from-port-spec (cadr s-arc))
    (setf to-port-spec (cadddr s-arc))
    (if (string< (car from-s-component) (car to-s-component))
      (setf new-s-arc
        (list (car from-s-component)
          from-port-spec
          (car to-s-component)
          to-port-spec))
      (setf new-s-arc
        (list (car to-s-component)
          to-port-spec
          (car from-s-component)
          from-port-spec))
      (cond ((matching-port-specifications?
              from-port-spec
              to-port-spec
              instantiation-level)
              (list new-s-arc 't))
            (t (list new-s-arc nil)))))))

(defun instantiate-component-combinations
  (component-combinations
   s-arcs-to-replace
   s-components
   instantiation-level
   &optional results)
  (let
    (component-combination result ranked-result sorted-result)
    (cond ((null (car component-combinations)) results)
      (t (setf component-combination (mapcar 'car component-combinations))
        result
        (instantiate-component-combination
          component-combination
          s-arcs-to-replace
          instantiation-level))
      (cond ((null result)
              (instantiate-component-combinations
               (mapcar 'cdr component-combinations)
               s-arcs-to-replace
               s-components
               instantiation-level
               results))
        (t (setf result
             (generate-new-topologies
              result
              instantiation-level
              s-components))
            (setf result (eliminate-duplicate-topologies result))
            (setf ranked-result (mapcar 'rank-topology result))
            (setf sorted-result
              (sort ranked-result #'<))))

40
(sort ranked-result #< :key #'item-rank))
(instantiate-component-combinations
 (mapcar 'cdr component-combinations)
 s-arcs-to-replace
 s-components
 instantiation-level
 (merge 'list
 results ranked-result #'< :key #'item-rank))))))

; Send this function:
; 1. component-combination: a list for each state that contains all s-components-
; with-parent-names for a single component combination.
; 2. s-arcs-with-validity-tags: all s-arcs with t or nil tagged to end to show
; if the arc is valid or invalid.
; 3. s-arcs-to-replace: parent component-s-arcs
; and it checks if too many optional components of any one name is used before it
; calls the aux function to instantiate.

(defun instantiate-component-combination
 (component-combination s-arcs-to-replace instantiation-level)
 (let
 (first-topology-state
 state-combination
 unique-s-component-names
 s-component-names
 component-names
 optional-component-names
 component-pairs)
 (setf first-topology-state (car component-combination))
 (setf state-combination (mapcar 'car first-topology-state))
 (setf unique-s-component-names (mapcar 'caadr state-combination))
 (setf s-component-names
 (mapcar 'remove-unique-suffix unique-s-component-names))
 (setf component-names (mapcar 'remove-state-suffix s-component-names))
 (setf optional-component-names
 (get-optional-names
 component-names))
 (setf component-pairs
 (make-component-pairs
 optional-component-names))
 (if (any-component-over-limit? component-pairs)
 nil
 (instantiate-component-combination-aux1
 component-combination
 s-arcs-to-replace
 instantiation-level))))

(defun any-component-over-limit? (component-pairs)
 (let
 (component-pair
 component-name
 number-in-topology
 limit)
 (cond ((null component-pairs) nil)
 (t (setf component-pair (car component-pairs))
 (setf component-name (car component-pair))
 (setf number-in-topology (cadr component-pair))
 (setf limit
 (cadr
 (assoc component-name
 *optional-component-limits*
 :test #'string-equal)))
 (if (> number-in-topology limit)
 t
 nil))))
(any-component-over-limit? (cdr component-pairs)))

(defun instantiate-component-combination-aux1
  (component-combination
   s-arcs-to-replace
   instantiation-level
   &optional results)
  (let
    (all-ways-to-instantiate
     all-valid-ways-to-instantiate
     state
     instantiated-state)
    (cond ((null component-combination)
           (setf all-ways-to-instantiate (generate-combinations (reverse results)))
           (setf all-valid-ways-to-instantiate
                 (instantiate-states all-ways-to-instantiate))
           (t (setf state (car component-combination))
              (setf instantiated-state
                   (instantiate-state
                    state
                    (car s-arcs-to-replace)
                    instantiation-level))
              (if (null instantiated-state) ; if one state instantiates to nil
                   nil
                   (instantiate-component-combination-aux1
                    (cdr component-combination)
                    (cdr s-arcs-to-replace)
                    instantiation-level
                    (cons instantiated-state
                     results))))))

(defun instantiate-states (lists-of-states &optional valid-instantiations)
  (let
    (instantiation)
    (cond ((null lists-of-states) (reverse valid-instantiations))
          (t (setf instantiation (invalidate-arcs (car lists-of-states) 0))
            (cond ((null instantiation)
                   (instantiate-states
                    (cdr lists-of-states)
                    valid-instantiations))
                  (t (instantiate-states
                      (cdr lists-of-states)
                      (cons instantiation
                       valid-instantiations)))))))))

(defun invalidate-arcs (states counter &optional new-states)
  (let ()
    (cond ((null (car states)) new-states)
          ((member nil (mapcar 'cadar states))
           (setf counter (+ 1 counter))
           (cond (> counter "invalid-arc-limit") nil)
           (t (invalidate-arcs
               (mapcar 'cdr states)
               counter
               (add-s-arcs-to-states
                (mapcar 'invalidate-first-s-arc states) new-states))))
          (t (invalidate-arcs
              (mapcar 'cdr states)
              counter
              (add-s-arcs-to-states (mapcar 'car states) new-states)))))))

(defun invalidate-first-s-arc (state)
  (list (caar state) nil))
(defun add-s-arcs-to-states (s-arcs states &optional new-states)
  (cond ((null s-arcs) (reverse new-states))
    (t (add-s-arcs-to-states
        (cdr s-arcs)
        (cdr states)
        (cons (append (car states) (list (car s-arcs))) new-states))))))

; Send this function:
; 1. state: a list of s-components-with-parent-name for a single state and a
;    single component combination.

(defvar *invalid-arc-limit* 10)

(defun instantiate-state (state s-arcs-to-replace instantiation-level)
  (let
    (s-component-combinations)
    (setf s-component-combinations (generate-combinations state))
    (instantiate-state-aux1
      s-component-combinations
      s-arcs-to-replace
      instantiation-level)))

(defun instantiate-state-aux1
  (s-component-combinations
   s-arcs-to-replace
   instantiation-level
   &optional results)
  (let
    (s-component-combination
     instantiated-s-component-combination)
    (cond ((null s-component-combinations) (reverse results))
      (t (setf s-component-combination (car s-component-combinations))
        (setf instantiated-s-component-combination
          (instantiate-s-component-combination
            s-component-combination
            s-arcs-to-replace
            0
            instantiation-level))
        (if (null instantiated-s-component-combination)
            (instantiate-state-aux1
              (cdr s-component-combinations)
              s-arcs-to-replace
              instantiation-level
              results)
            (instantiate-state-aux1
              (cdr s-component-combinations)
              s-arcs-to-replace
              instantiation-level
              (cons instantiated-s-component-combination results)))))))

; Send this function:
; 1. s-component-combination: a single combination of s-components with parent
;    name consed to front of each
; 2. parent-s-arcs: (parent-from-s-name port-spec parent-to-s-name port-spec)
; 3. counter: initialize to 0
; 4. instantiation-level
; and it returns a list of s-arcs-with-validity-tags if the number of invalid s-arcs
; is less than or equal to the *invalid-arc-limit*, else nil.

(defun instantiate-s-component-combination
  (s-component-combination
   parent-s-arcs
   counter
   instantiation-level
   &optional results)
(let
 (parent-s-arc
  new-from-s-component-with-parent
  new-to-s-component-with-parent
  new-from-s-component
  new-to-s-component
  new-from-port-specs
  new-to-port-specs
  new-from-port-spec
  new-to-port-spec
  new-s-arc
  new-s-arc-with-validity-tag
  altered-from-port-specs
  altered-to-port-specs
  altered-from-s-component
  altered-to-s-component
  altered-from-s-component-with-parent
  altered-to-s-component-with-parent
  altered-s-component-combination)
  (cond ((null parent-s-arcs) (reverse results))
    (t (setf parent-s-arc (car parent-s-arcs))
      (setf new-from-s-component-with-parent
            (assoc
             (car parent-s-arc)
             s-component-combination
             :test #'string-equal))
      (setf new-from-s-component (cadr new-from-s-component-with-parent))
      (setf new-to-s-component-with-parent
            (assoc
             (caddr parent-s-arc)
             s-component-combination
             :test #'string-equal))
      (setf new-to-s-component (cadr new-to-s-component-with-parent))
      (setf new-from-port-specs (cadddr new-from-s-component))
      (setf new-to-port-specs (cadddd new-to-s-component))
      (setf new-from-port-spec
            (get-offspring-port-spec new-from-port-specs (cadr parent-s-arc)))
      (setf new-to-port-spec
            (get-offspring-port-spec new-to-port-specs (cadddr parent-s-arc)))
      (if (string< (car new-from-s-component) (car new-to-s-component))
        (setf new-s-arc
              (list
               (car new-from-s-component)
               new-from-port-spec
               (car new-to-s-component)
               new-to-port-spec))
        (setf new-s-arc
              (list
               (car new-to-s-component)
               new-to-port-spec
               (car new-from-s-component)
               new-from-port-spec))
      (cond ((matching-port-specifications? new-from-port-spec
                                               new-to-port-spec
                                               instantiation-level)
              (setf new-s-arc-with-validity-tag (list new-s-arc 't)))
            (t (setf new-s-arc-with-validity-tag (list new-s-arc 'nil))
                (setf counter (+ 1 counter)))
            (cond ((> counter 'invalid-arc-limit*) nil)
                  (t (setf altered-from-port-specs
                       (remove new-from-port-spec
                                new-from-port-specs)
                       )
                       ))))
 44
(defun get-offspring-port-spec (offspring-port-specs port-spec)
  (cond ((null offspring-port-specs)
           (print port-spec)
           (print 'trouble-in-get-offspring-port-spec))
        ((equalp port-spec
                  (remove-last-item
                   (car offspring-port-specs)))
         (car offspring-port-specs))
        (t (get-offspring-port-spec (cdr offspring-port-specs) port-spec))))

(defun convert-to-parent-name (required-s-component-with-s-parent)
  (list
   (convert-un-to-un (car required-s-component-with-s-parent))
   (cadr required-s-component-with-s-parent)))

(defun double-mapcar (function list &optional new-list)
  (cond ((null list) (reverse new-list))
        (t (double-mapcar
            function
            (cdr list)
            (cons (mapcar function (car list)) new-list)))))

(defun make-s-components-unique
  (s-components unique-suffix &optional unique-s-components)
  (let
    (s-component
      (list
       (lag s-components unique-suffix unique-s-components))
      (list
       (lag s-components unique-suffix unique-s-components))
      (list
       (lag s-components unique-suffix unique-s-components))
      (list
       (lag s-components unique-suffix unique-s-components))
      (list
       (lag s-components unique-suffix unique-s-components))))

(defun altered-to-port-specs
  (remove new-to-port-spec new-to-port-specs)
  (test #'equalp
         :count 1))

(defun altered-from-s-component
  (substitute altered-from-port-specs
               (caddadr new-from-s-component) new-from-s-component))

(defun altered-to-s-component
  (substitute altered-to-port-specs
               (caddadr new-to-s-component) new-to-s-component))

(defun altered-from-s-component-with-parent
  (list (car new-from-s-component-with-parent)
         altered-from-s-component))

(defun altered-to-s-component-with-parent
  (list (car new-to-s-component-with-parent)
         altered-to-s-component))

(defun altered-s-component-combination
  (substitute altered-from-s-component-with-parent
               new-from-s-component-with-parent
               s-component-combination
               :test #'equalp))

(defun altered-s-component-combination
  (substitute altered-to-s-component-with-parent
               new-to-s-component-with-parent
               altered-s-component-combination
               :test #'equalp))

(defun instantiate-s-component-combination
  altered-s-component-combination
  (cdr parent-s-arcs)
  counter
  instantiation-level
  (cons new-s-arc-with-validity-tag results)))))))

(defun test #'equalp
  :count 1))
(cond ((null s-components) (reverse unique-s-components))
  (t (setf s-component (car s-components))
   (setf s-component-name (car s-component))
   (make-s-components-unique
    (cdr s-components)
    unique-suffix
    (cons (list (concatenate 'string s-component-name
                       unique-suffix)
            (make-parent-names-unique
             (car parent-names)
             unique-suffix)
            (add (caddr s-component)
            (caddadr s-component))
            unique-s-components))))
)

(defun make-parent-names-unique
  (parent-names unique-suffix &optional unique-parent-names)
  (cond ((null parent-names) (reverse unique-parent-names))
  (t (make-parent-names-unique
   (cdr parent-names)
   unique-suffix
   (cons (concatenate 'string (car parent-names)
            unique-suffix)
            unique-parent-names)))))

; this function generates topologies from s-arc lists

(defun generate-new-topologies
  (state-topology-combinations
   abstraction-level
   s-components
   &optional new-topologies)
  (let
    (state-topology-combination new-topology)
    (cond ((null state-topology-combinations) (reverse new-topologies))
  (t (setf state-topology-combination
            contains n lists of s-arcs,
            (car
            one list for each of n
            (car state-topology-combinations))
   topology states
    (setf new-topology
      (generate-new-topology
      state-topology-combination
      abstraction-level
      s-components))
    (generate-new-topologies
     (cdr state-topology-combinations)
     abstraction-level
     s-components
     (cons new-topology
     new-topologies))))))

; Send this function a list of lists of s-arcs (one list of s-arcs for each state
; of the topology) and the abstraction-level, and it will return a complete topology
; if all arcs are complete, else a partial topology if any arc is incomplete.

(defun generate-new-topology (s-arc-lists abstraction-level s-components)
  (let
    (open-s-arc-lists
     partial-state-topologies
     partial-topology
     complete-state-topologies
     complete-topology)
    (cond ((member nil
      (mapcar (if any arc has a nil
                ;in the last position
                nil
                46
'cadr ;then the arc is invalid
 (car s-arc-lists)) ;and topology is partial

(setf open-s-arc-lists
 (mapcar 'get-open-s-arcs s-arc-lists))
(setf s-arc-lists (mapcar 'get-s-arcs s-arc-lists))
(setf partial-state-topologies
 (make-partial-state-topologies
  s-arc-lists
  open-s-arc-lists
  s-components))
(setf partial-topology
 (list
  "partial"
  abstraction-level
  partial-state-topologies
  0)))))

(t (setf complete-state-topologies ;else the topology is
 (mapcar
  'make-complete-state-topology
  s-arc-lists))
 (setf complete-topology
 (list
  "complete"
  abstraction-level
  complete-state-topologies
  0))))))

(defun get-open-port-from-s-open-port (s-open-port)
 (list (car s-open-port) (convert-usn-to-un (caddr s-open-port))))

(defun get-open-s-arcs (s-arc-list &optional open-s-arcs)
 (let
  (s-arc
   (cond ((null s-arc-list) (reverse open-s-arcs))
     (t (setf s-arc (car s-arc-list)))
       (cond ((null (cadr s-arc)) ;if s-arc is invalid
         (get-open-s-arcs
          (cdr s-arc-list)
          (cons (car s-arc) open-s-arcs)))
       (t (get-open-s-arcs (cdr s-arc-list) open-s-arcs))))))

(defun get-s-arcs (s-arc-list &optional s-arcs)
 (let
  (s-arc
   (cond ((null s-arc-list) (reverse s-arcs))
     (t (setf s-arc (car s-arc-list)))
       (cond ((null (cadr s-arc)) (get-s-arcs (cdr s-arc-list) s-arcs))
         (t (get-s-arcs (cdr s-arc-list) (cons (car s-arc) s-arcs)))))))

(defun make-partial-state-topologies
 (s-arc-lists ;unique names
 open-s-arc-lists ;unique names
 s-components
 &optional partial-state-topologies)
 (let
  (s-arcs
   open-s-arcs
   s-open-ports
   s-clusters
   unique-s-component-names
   partial-state-topology)
  (cond ((null s-arc-lists) (reverse partial-state-topologies))
    (t (setf s-arcs (car s-arc-lists)))
    (setf open-s-arcs (car open-s-arc-lists))
    (cond (partial-state-topologies
      (make-partial-state-topologies
       s-arc-lists
       open-s-arc-lists
       s-components))
      partial-state-topologies
      (partial-topology
       (list
        "partial"
        abstraction-level
        partial-state-topologies
        0)))))

(defun get-open-port-from-s-open-port (s-open-port)
 (list (car s-open-port) (convert-usn-to-un (caddr s-open-port))))

(defun get-open-s-arcs (s-arc-list &optional open-s-arcs)
 (let
  (s-arc
   (cond ((null s-arc-list) (reverse open-s-arcs))
     (t (setf s-arc (car s-arc-list)))
       (cond ((null (cadr s-arc)) ;if s-arc is invalid
         (get-open-s-arcs
          (cdr s-arc-list)
          (cons (car s-arc) open-s-arcs)))
       (t (get-open-s-arcs (cdr s-arc-list) open-s-arcs))))))

(defun get-s-arcs (s-arc-list &optional s-arcs)
 (let
  (s-arc
   (cond ((null s-arc-list) (reverse s-arcs))
     (t (setf s-arc (car s-arc-list)))
       (cond ((null (cadr s-arc)) (get-s-arcs (cdr s-arc-list) s-arcs))
         (t (get-s-arcs (cdr s-arc-list) (cons (car s-arc) s-arcs)))))))

(defun make-partial-state-topologies
 (s-arc-lists ;unique names
 open-s-arc-lists ;unique names
 s-components
 &optional partial-state-topologies)
 (let
  (s-arcs
   open-s-arcs
   s-open-ports
   s-clusters
   unique-s-component-names
   partial-state-topology)
  (cond ((null s-arc-lists) (reverse partial-state-topologies))
    (t (setf s-arcs (car s-arc-lists)))
    (setf open-s-arcs (car open-s-arc-lists))
    (cond (partial-state-topologies
      (make-partial-state-topologies
       s-arc-lists
       open-s-arc-lists
       s-components))
      partial-state-topologies
      (partial-topology
       (list
        "partial"
        abstraction-level
        partial-state-topologies
        0)))))

(defun get-open-port-from-s-open-port (s-open-port)
 (list (car s-open-port) (convert-usn-to-un (caddr s-open-port))))

(defun get-open-s-arcs (s-arc-list &optional open-s-arcs)
 (let
  (s-arc
   (cond ((null s-arc-list) (reverse open-s-arcs))
     (t (setf s-arc (car s-arc-list)))
       (cond ((null (cadr s-arc)) ;if s-arc is invalid
         (get-open-s-arcs
          (cdr s-arc-list)
          (cons (car s-arc) open-s-arcs)))
       (t (get-open-s-arcs (cdr s-arc-list) open-s-arcs))))))

(defun get-s-arcs (s-arc-list &optional s-arcs)
 (let
  (s-arc
   (cond ((null s-arc-list) (reverse s-arcs))
     (t (setf s-arc (car s-arc-list)))
       (cond ((null (cadr s-arc)) (get-s-arcs (cdr s-arc-list) s-arcs))
         (t (get-s-arcs (cdr s-arc-list) (cons (car s-arc) s-arcs)))))))

(defun make-partial-state-topologies
 (s-arc-lists ;unique names
 open-s-arc-lists ;unique names
 s-components
 &optional partial-state-topologies)
 (let
  (s-arcs
   open-s-arcs
   s-open-ports
   s-clusters
   unique-s-component-names
   partial-state-topology)
  (cond ((null s-arc-lists) (reverse partial-state-topologies))
    (t (setf s-arcs (car s-arc-lists)))
    (setf open-s-arcs (car open-s-arc-lists))
    (cond (partial-state-topologies
      (make-partial-state-topologies
       s-arc-lists
       open-s-arc-lists
       s-components))
      partial-state-topologies
      (partial-topology
       (list
        "partial"
        abstraction-level
        partial-state-topologies
        0)))))
(setq s-open-ports
  (get-s-open-ports-from-open-s-arcs open-s-arcs s-arcs s-components))
(setq s-clusters
  (create-s-clusters
   s-arcs
   s-open-ports
   (append (mapcar 'car s-arcs)
           (mapcar 'caddr s-arcs))
           (mapcar 'car s-open-ports)))
(setq unique-s-component-names
  (remove-duplicates
   (append
    (mapcar 'car s-arcs)
    (mapcar 'caddr s-arcs)
    :test #'string-equal))
(setq partial-state-topology
  (list
   s-clusters
   s-arcs
   unique-s-component-names))
(make-partial-state-topologies
  (cdr s-arc-lists)
  (cdr open-s-arc-lists)
  s-components
  (cons partial-state-topology
        partial-state-topologies))))

(defun create-s-clusters
  (s-arcs s-open-ports s-arc-names s-open-port-names &optional name-clusters)
  (let
   (s-arc
    s-component-name-1
    s-component-name-2)
   (cond ((null s-arcs))
     (setq name-clusters
       (append name-clusters
                (mapcar
                 'list
                 (set-difference
                  s-open-port-names
                  s-arcs)
                 s-arc-names
                 :test #'string-equal)))
     (create-s-clusters-aux2 s-open-ports name-clusters))
     (t (setq s-arc (car s-arcs))
     (setq s-component-name-1 (car s-arc))
     (setq s-component-name-2 (caddr s-arc))
     (setq name-clusters
       (create-s-clusters-aux1
        s-component-name-1
        s-component-name-2
        name-clusters))
     (create-s-clusters
      (cdr s-arcs)
      s-open-ports
      s-arc-names
      s-open-port-names
      name-clusters)))))

(defun create-s-clusters-aux1
  (name1 name2 name-clusters
    &optional new-name-clusters)
  (let
   (name-cluster
    new-name-cluster)
   (cond ((null name-clusters)
(append new-name-clusters (list (list name1 name2)))
(t (setf name-cluster (car name-clusters))
  (cond ((member name1 name-cluster :test #'string-equal)
        (setf new-name-cluster
              (union (list name2) name-cluster :test #'string-equal))
         (append (cdr name-clusters)
                  (list new-name-cluster)
                  new-name-clusters)))
((member name2 name-cluster :test #'string-equal)
 (setf new-name-cluster
        (union (list name1)
               name-cluster
               :test #'string-equal))
       (append (cdr name-clusters)
                (list new-name-cluster)
                new-name-clusters)))
(t (create-s-clusters-aux1
    name1
    name2
    (cdr name-clusters)
    (cons name-cluster new-name-clusters)))))))

(defun create-s-clusters-aux2 (s-open-ports name-clusters &optional s-clusters)
  (let
    (name-cluster
      s-cluster)
    (cond ((null name-clusters) (reverse s-clusters))
      (t (setf name-cluster (car name-clusters))
         (setf s-cluster (create-s-clusters-aux3 name-cluster s-open-ports))
         (create-s-clusters-aux2
          s-open-ports
          (cdr name-clusters)
          (cons s-cluster s-clusters))))))

(defun create-s-clusters-aux3 (name-cluster s-open-ports &optional s-cluster)
  (let
    (name matching-s-open-ports)
    (cond ((null name-cluster) s-cluster)
      (t (setf name (car name-cluster))
          (setf matching-s-open-ports (create-s-clusters-aux4 name-cluster s-open-ports))
          (create-s-clusters-aux3
           (cdr name-cluster)
           s-open-ports
           (append s-cluster matching-s-open-ports))))))

(defun create-s-clusters-aux4 (name s-open-ports &optional matching-s-open-ports)
  (let
    (s-open-port)
    (cond ((null s-open-ports) matching-s-open-ports)
      (t (setf s-open-port (car s-open-ports))
          (cond ((string-equal name (car s-open-port))
                 (create-s-clusters-aux4
                  name
                  (cdr s-open-ports)
                  (cons s-open-port matching-s-open-ports))
                 (t (create-s-clusters-aux4
                     name
                     (cdr s-open-ports)
                     matching-s-open-ports))))))

(defun get-s-open-ports-from-open-s-arcs
  (open-s-arcs s-components &optional s-open-ports)
  (let
    (open-s-arc s-open-port-1 s-open-port-2)
(cond ((null open-s-arcs) s-open-ports)
(t (setf open-s-arc (car open-s-arcs))
 (setf s-open-port-1 (list (car open-s-arc) (cadr open-s-arc)))
 (setf s-open-port-2 (list (caddr open-s-arc) (cadddr open-s-arc)))
 (get-s-open-ports-from-open-s-arcs
 s-arcs
 s-components
 (append s-open-ports (list s-open-port-1) (list s-open-port-2))))
)

(defun make-complete-state-topology (s-arcs)
 (setf s-arcs
 (mapcar 'car
 s-arcs))
(list s-arcs
 (remove-duplicates
 (append
 (mapcar 'car s-arcs)
 (mapcar 'caddr s-arcs))
 :test #'string-equal)))

(defun make-unique-arc-from-unique-s-arc (unique-s-arc)
 (list (convert-usn-to-un (car unique-s-arc))
 (cadr unique-s-arc)
 (convert-usn-to-un (caddr unique-s-arc))
 (cadddr unique-s-arc)))

; Send this function a list of items and it returns the list with an additional set
; of parenthesis around each item that was a singular list.

(defun listify-required-components (list &optional results)
 (let
 ((first-item)
 (cond ((null list) (reverse results))
 (t (setf first-item (car list))
 (cond ((listp (car first-item))
 (listify-required-components
 (cdr list)
 (cons first-item results)))
 (t (listify-required-components
 (cdr list)
 (cons (list first-item results))))))))

; Send this function a list of lists and it will return all combinations.
; Example: send it the list:  ((a b) (c) (d e))
; and it returns:  ((a c d) (b c d) (a c e) (b c e))

(defun generate-combinations (lists &optional new-lists)
 (cond ((null lists) new-lists)
 (null new-lists)
 (generate-combinations (cdr lists) (mapcar 'list (car lists))))
 (t (generate-combinations
 (cdr lists)
 (generate-combinations-aux1 new-lists (car lists))))))

(defun generate-combinations-aux1 (lists list &optional new-lists)
 (cond ((null list) new-lists)
 (t (generate-combinations-aux1
 lists
 (cdr list)
 (append new-lists (generate-combinations-aux2 (car list) lists))))))

(defun generate-combinations-aux2 (item lists &optional new-lists)
 (cond ((null lists) (reverse new-lists))
 (t (generate-combinations-aux2
 (append new-lists (list item)) (cdr lists)))))

50
item
(cdr lists)
(cons (append (car lists) (list item)) new-lists))))

==========================================
==========================================