Purdue University
Purdue e-Pubs

ECE Technical Reports | Electrical and Computer Engineering

7-1-1993

A TUTORIAL ON LISP OBJECT-ORIENTED PROGRAMMING FOR BLACKBOARD COMPUTATION (SOLVING THE RADAR TRACKING PROBLEM)

P. R. Kersten
Naval Undersea Warfare Center Division

A. C. Kak
Purdue University School of Electrical Engineering

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

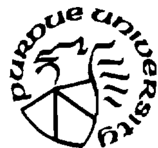
Kersten, P. R. and Kak, A. C., "A TUTORIAL ON LISP OBJECT-ORIENTED PROGRAMMING FOR BLACKBOARD COMPUTATION (SOLVING THE RADAR TRACKING PROBLEM) " (1993). *ECE Technical Reports.* Paper 233.
http://docs.lib.purdue.edu/ecetr/233

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

A TUTORIAL ON LISP OBJECT-ORIENTED PROGRAMMING FOR BLACKBOARD COMPUTATION (SOLVING THE RADAR TRACKING PROBLEM)

P. R. KERSTEN
A. C. KAK

TR-EE 93-24
JULY 1993



SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

A TUTORIAL ON LISP OBJECT-ORIENTED PROGRAMMING FOR BLACKBOARD COMPUTATION
(SOLVING THE RADAR TRACKING PROBLEM)*

P. R. Kersten
Code 2211
Naval Undersea Warfare Center Division
Newport, RI 02841-5047

A. C. Kak
Robot Vision Lab
1285 EE Building
Purdue University
W. Lafayette, IN 47907-1285

ABSTRACT

This exposition is a tutorial on how object-oriented programming (OOP) in Lisp can be used for programming a blackboard. Since we have used Common Lisp and the Common Lisp Object System (CLOS), the exposition demonstrates how object classes and the primary, before, and after methods associated with the classes can be used for this purpose. The reader should note that the different approaches to object-oriented programming share considerable similarity and, therefore, the exposition should be helpful to even those who may not wish to use CLOS.

We have used the radar tracking problem as a 'medium' for explaining the concepts underlying blackboard programming. The blackboard database is constructed solely of classes which act as data structures as well as method-bearing objects. Class instances form the nodes and the levels of the blackboard. The methods associated with these classes constitute a distributed monitor and support the knowledge sources in modifying the blackboard data. A rule-based planner is used to construct knowledge source activation records from the goals residing in the blackboard. These activation records are enqueued in a cyclic queueing system. A scheduler cycles through the queues and selects knowledge sources to fire.

* **Approved for public release, distribution unlimited, by the Naval Undersea Warfare Center.**

TABLE OF CONTENTS

Page

1. Introduction 3

2. The Representational Problem -- CLOS 10

3. Representation of The Abstraction Levels 19

4 Knowledge Sources 27

5. The Blackboard Control 31

6. Conclusions 43

7. Acknowledgment 44

8. References 45

Appendix A 47

Appendix B 67

1. INTRODUCTION

The blackboard (BB) approach to problem solving has been used in a number of systems dealing with a diverse set of applications, which include speech understanding [9], image understanding [1, 2, 14, 24], planning [15, 23], high-level signal processing [8,25-27], distributed problem solving [22], and general problem solving [6,7]. Usually, a blackboard system consists of three parts, a ***global database, knowledge sources (KS's)*** and the ***control.*** The global database is usually referred to as the blackboard and is, in most cases, the ***only*** means of communication between the KS's. The KS's are procedures capable of modifying the objects on the blackboard and are the only entities that are allowed to read or write on the blackboard. Control of the blackboard may be event driven, goal driven, or expectation driven. ***Events*** are changes to the BB, such as the arrival of data or modifications of data by one of the KS's. In an event driven BB, a scheduler uses the events as the primary information source to schedule the KS's for invocation. A goal driven BB system, on the other hand, is a more refined computational structure, which uses a composite mapping from the events to goals and then from goals directly to KS activations or indirectly from goals to subgoals and then to KS's. This refinement permits a more sophisticated planning algorithm to choose the next KS activation. By using goals, one can bias the blackboard or generate other goals to fetch or generate other components of the solution [5]. Note that if goals are isomorphic to the events, then a BB is essentially event driven.

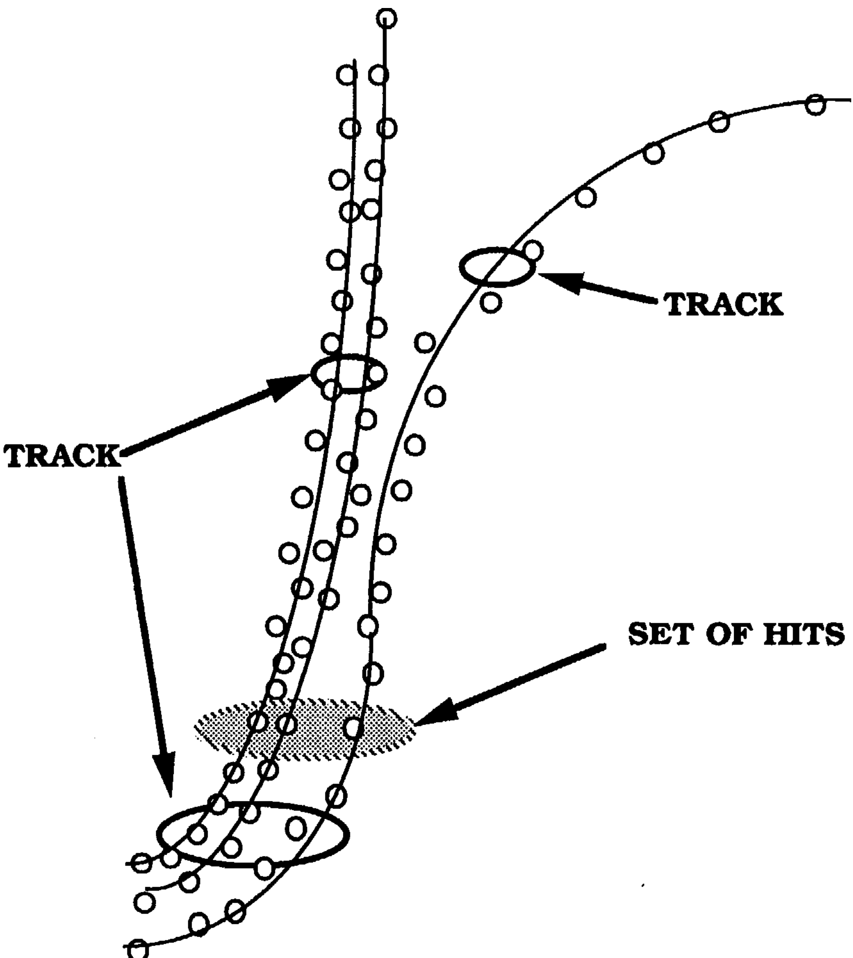
As was eloquently pointed out by Nii [27], there is a great difference between understanding the concept of a blackboard model and its implementation. Implementation is made all the more difficult by the lack in the current literature of a suitable exposition on how to actually go about writing a computer program for a blackboard. A blackboard is a complex computational structure, not amenable to a quick description as an algorithm. To program a blackboard, one must specify data structures for the items that are posted on the blackboard, explicitly state the nature of interaction between the data on the blackboard and the KS's, clearly define how the high-level goals get decomposed into lower-level goals during problem solving, etc. The purpose of this tutorial exposition is to rectify this deficiency in the literature, at least from the standpoint of helping someone to get started with the task of programming a blackboard.

For this tutorial, we have used the radar tracking problem (RTP) to illustrate how object-oriented programming (OOP) in Lisp can be used to establish the flow of control required for blackboard-based problem solving. The RTP is defined as follows: Given the radar returns, find the best partition of these returns into disjoint time sequences that represent the trajectories of craft or any other moving body. For craft flying in tight formations, we will associate a single trajectory with each formation. Each trajectory, whether associated with a single craft or a formation, will be called a track. Since craft

may break away fi-om a formation, any single track can lead to multiple tracks. Shown in Fig. 1 is a flight of three craft. Originally, their tight formation results in a single track. But, as the flight progresses, one of the craft breaks away to the right and, then, we have two tracks. The RTP problem then consists of assigning a radar return to one of the existing tracks or allowing it to initiate a new track. This problem is not new and has been solved with varying degrees of success and implemented in numerous systems. In fact, a blackboard solution of the RTP may already exist, although it is probably proprietary. In fact, there is indication [8,27,30] that TRICERO has a radar tracker embedded in it.

On the basis of the criteria advanced by Nii [27], it can be rationalized that the RTP problem is well suited to the blackboard approach. We will now provide this rationalization in the following paragraph; the blackboard-suitability criteria, as advanced by Nii, will be expressed as italicized phrases.

The radar returns vary widely in quality. Returns may have high signal-to-noise ratio (SNR) in uncluttered backgrounds but may also be noisy, cluttered, and weak. Obviously you design for the worst case that includes ***noisy and unreliable data.*** While it is true that tracks can legitimately cross, merge, and split, noise and clutter can also induce these as anomalies in the actual tracks, in addition to, of course, causing the tracks to fade (Fig. 2). Track formation in a noisy environment requires not only significant signal processing but, in general, also requires forward and backward reasoning at a syrnbolic level. For example, backward reasoning can verify a track by a hypothesize-and-test scheme that may invoke procedures requiring higher spatial resolution and longer signal integration times for hypothesis verification. In other words, under noisy conditions we may use coarse resolution and forward reasoning to form track hypotheses, and then invoke backward reasoning to verify strongly-held hypotheses. So, there is a need to use ***multiple reasoning methods;*** combined forward and backward reasoning steps can be easily embedded in a goal-driven blackboard. In addition to multiple reasoning methods, the system must also reason simultaneously along multiple lines. For example, when track splits occur, it may be desirable to watch and maintain several alternative track solutions before modifying the track information. ***Multiple lines of reasoning,*** as can be easily incorporated in a blackboard system, can play a natural role in searching for the optima. solution under these conditions. It is generally believed these days that tracking systems of the future will be equipped with multi-sensor capability. Therefore, future target tracking systems will have to allow fusion of information from diverse sensors, not to speak of the intelligence information that will also have to be integrated. With these additional inputs, the solution space quickly becomes large and complex, necessitating modularized computational structures, like blackboards, that are capable of handling ***a variety of input data.***



igure1. Shown here are radar returns from a flight of three craft. The circles represent the returned echos. Initially, the three craft are in a tight formation and form a single track. But subsequently, one of the craft breaks away to the right, the result being two tracks.

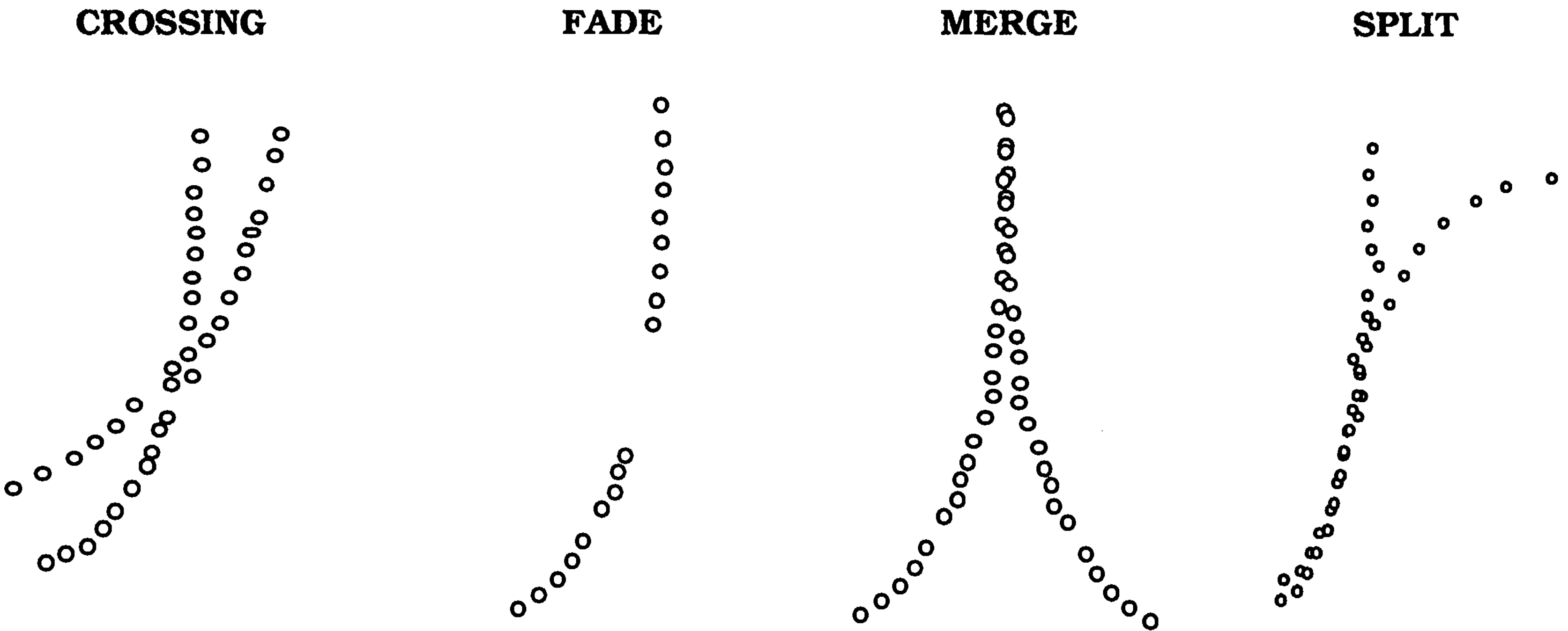


Figure 2. While it is true that tracks can legitimately cross, merge and split, low signal-to-noise ratios can also induce these as anomalies in the actual tracks. Note noise can also cause the tracks to fade.

In addition to using the above rationalization for justifying a blackboard-based solution to the RTP problem, one must also bear in mind the fact that the use of blackboards can simplify software development. The blackboard system solves a problem subject to the constraint that the processes, as represented by KS's, are independent enough so that they interact only through the blackboard database. This independence amongst processes has the advantage of allowing for independent development. There is, however, a price to be paid for maximally separating the KS's with respect to the BB database -- overhead. For example, if there is no shared memory, the cost of data transfer between the BB and KS's can be very high in terms of real time, not to mention software design time. While for research and development this may be a small price to pay, in real-time environments this may not be acceptable. Also, the opportunistic control made possible by a blackboard architecture may be ideal from a conceptual viewpoint and may increase solution convergence, but, because opportunistic control is difficult to model mathematically, it can lead to unpredictable behavior by a BB under circumstances not taken into account during the test phase of the system. Yet, in spite of these drawbacks, it is probably inevitable that BB systems will work their way into system designs of the future.

Our radar tracking blackboard (RTBB), the subject of this tutorial exposition, is constructed in CLOS (Common Lisp Object System) [13,18,20] with KS's written either in Common Lisp [12,29,31] or in C. The overall organization of RTBB is shown in Fig. 3. The database part of RTBB consists of two panels, the data panel and the goal panel, each containing three abstraction levels. Time-stamped radar returns reside in the form of beam nodes at the lowest level of abstraction in the data panel, the hit level. Spatially adjacent returns are grouped together into segments and reside as segment nodes at the next level of data abstraction. Finally, segments are grouped into track-level nodes at the highest level of abstraction in the data panel. A track-level data-panel node is capable of representing a formation of craft; multiple formations will require multiple track-level nodes. The data abstraction hierarchy is shown in greater detail in Fig. 4. On the goal side in Fig. 3, goal nodes at the hit level are simply requests to generate time-stamped radar returns. Goals at the segment level are more varied: there can be goal nodes that are requests to assign incoming radar returns to already existing segments, goals to deal with the problem of fading in radar returns, etc. Goals at the track level are also varied: goal nodes may request that new segments be merged with existing tracks or be allowed to form new tracks, or goal nodes may spawn sub-goals to verify that the currently held segments in a track indeed belong to the track if the track is deemed to be a threat. The ability to decompose a goal into sub-goals is a special benefit of a goal-driven BB. A rule-based planner maps the goals into either sub-goals or knowledge source activation records (KSAR's). A KSAR is simply a record of the fact that a goal node is ready with the appropriate data for firing a KS. RTBB enqueues all the KSAR's and the scheduler then cycles through the queues and selects the KS's to fire. The main BB process runs in

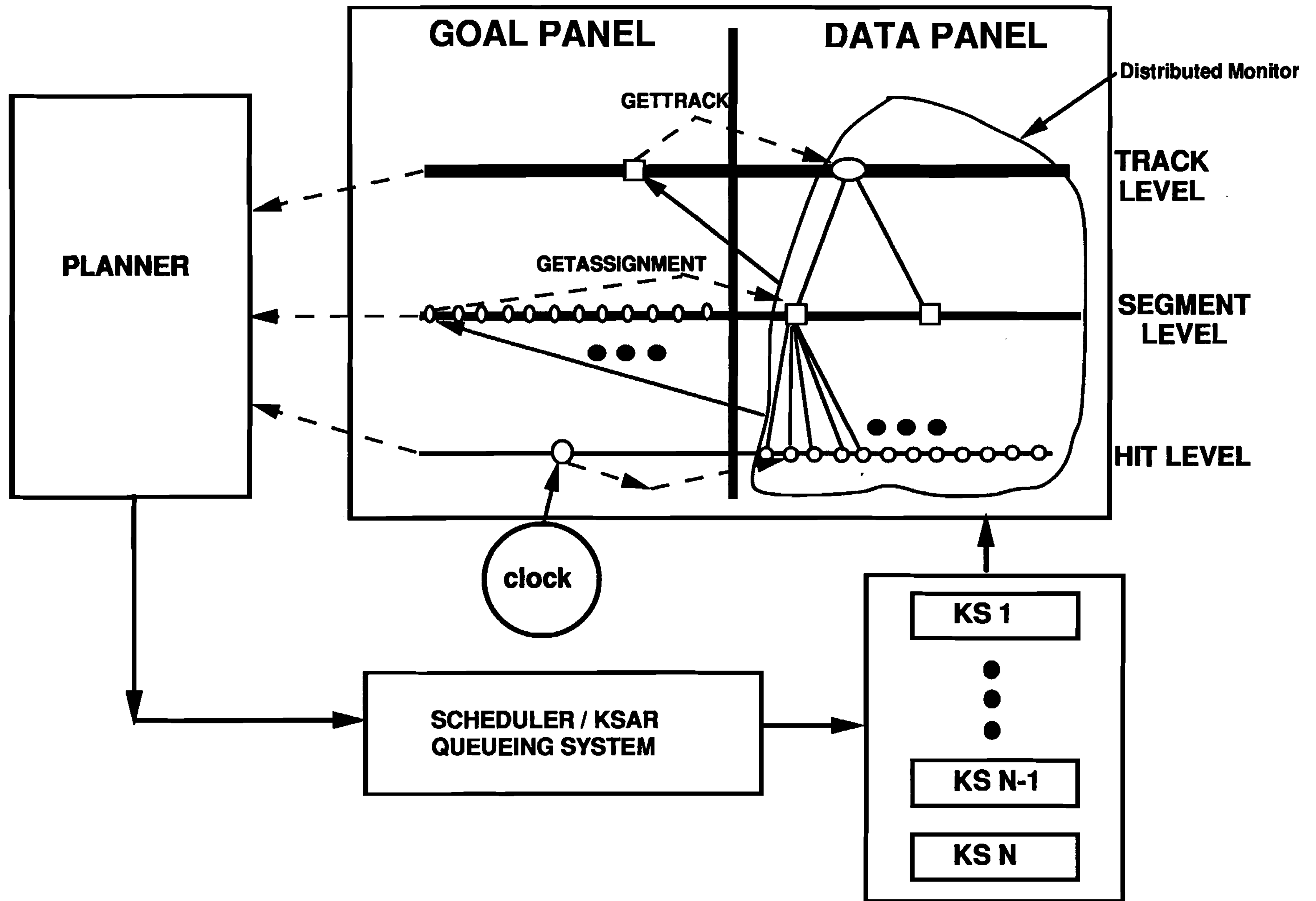


Figure 3. **Architecture of Radar Tracking Blackboard (RTBB).**

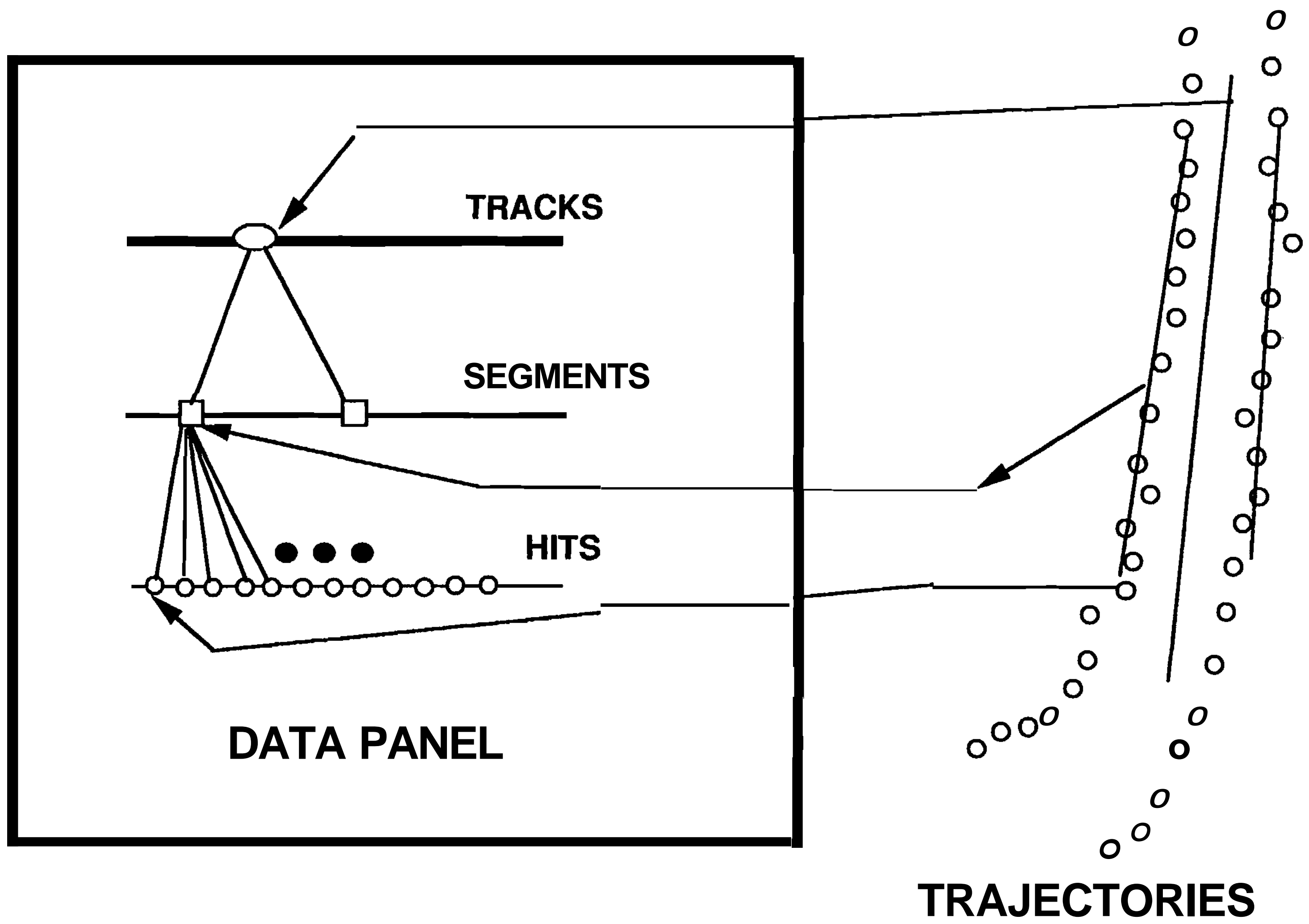


Figure 4. Shown here is the data abstraction hierarchy in greater detail. Hits that are spacially adjacent are grouped into segments. And segments that are approximately co-directional and spatially adjacent are grouped into tracks.

Lisp, and the KS's are either children of the main BB process or are threaded into the BB process itself. The database, BB monitor, and the scheduler are all part of the main BB process. All the processes run under the UNIX operating system. Each level and each *node* on the BB is an instantiation of some object class. These class instantiations are method-bearing data structures that are part of **CLOS.** The methods associated with BB nodes act as local monitors, collectively forming a distributed BB monitor, or as scribes for the **KS's** in updating the BB **information,** or even as information agents for the **rule**-based planner. ***After-methods*** written for the data nodes trigger after a node is altered and report the changes to the goal BB. This implementation of the monitor using CLOS is one of the more interesting aspects of RTBB.

In the rest of this tutorial, we will start in Section 2 with a brief introduction to CLOS. As we mentioned in the abstract, the different approaches to object-oriented programming share considerable similarities, and, therefore, even a reader who does not use CLOS should find this tutorial useful; such a reader may want to browse through Section 2 if only to become familiar with some of the main data structures used for RTBB. In Section 3, we describe the different abstraction levels used in RTBB. Section 4 briefly discusses the different **KS's** used. Control flow and scheduling **are** presented in Section 5. Finally, Section 6 contains the conclusions. We have also included an appendix where we have discussed four examples of increasing complexity. After a first pass through the main body of this paper, we believe the reader would find it very helpful to go through the examples in Appendix A for a fuller comprehension of the various aspects of the blackboard.

For those wishing to see the source code, it is included in Appendix B of this report. A **journal** paper based on this report appears in [19].

2. THE REPRESENTATION PROBLEM -- CLOS

The representation problem is central to problem solving in general and the implementation of a chosen representation requires suitable data structures. In RTBB, to represent the nodes at the different levels of the blackboard, we **first** define a generic node called node. The objects obtained by instantiating a ***node*** **will** be the simplest possible and probably not very useful data entities. The more useful object classes that would represent the nodes on the three levels of the right hand side of the blackboard shown in Fig. 3 **are** then defined as subclasses of the generic object class node, as shown in Fig. 5. The subclass corresponding to the beam level nodes is called bnode, the one **correspond**-ing to the segment level nodes ***snode,*** and the one corresponding to the **track** level nodes ***tnode.*** With this hierarchical organization, those properties of all the nodes that **are** common to all three levels can now be assigned to the generic node node and those properties

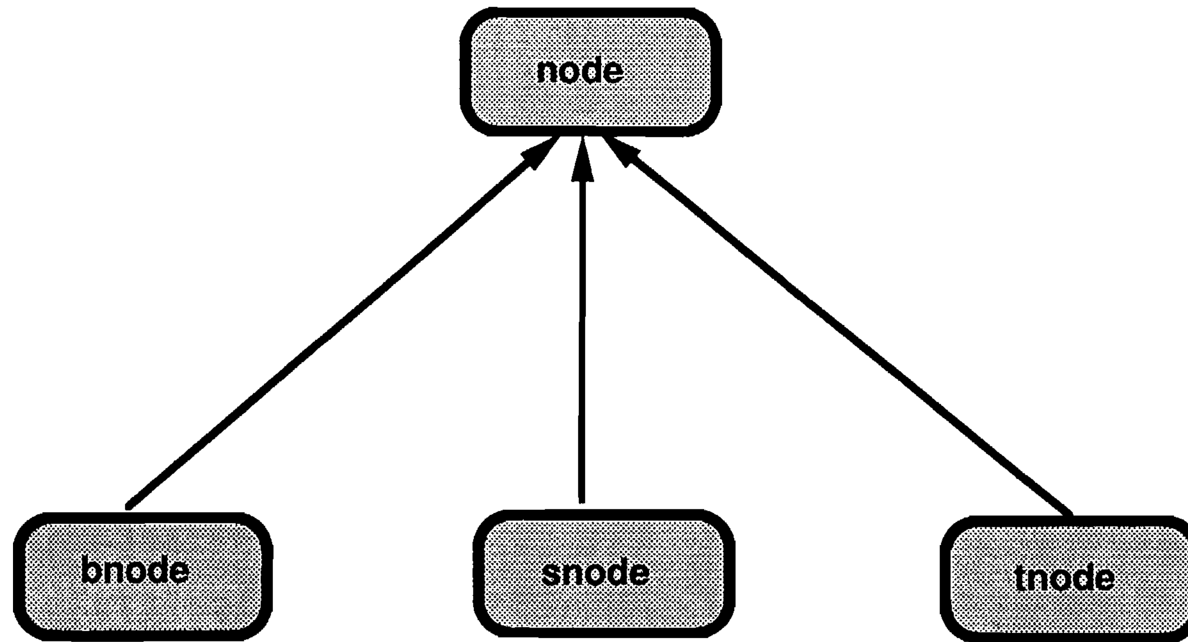


Figure 5. Organization of the node classes. The data abstractions used in RTBB are subclasses of the generic class node.

that are unique to each of the three classes individually can be so declared. To see how this can be done in **CLOS**, we now show how the generic class ***node*** is created by the ***defclass*** macro.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The is a generic class -- the class is the superclass of all data classes.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass node ()
  (
    (level :initarg :level :accessor level)
    (event-time :initarg :event-time :accessor event-time)
    )
  (:documentation "The node is superclass of all data data classes")
  )
```

This generic class has only two slots, ***level*** and ***event-time.*** For data nodes, the value of the ***level*** slot designates the level at which the nodes reside. The slot ***event-time*** is the clock time, in the sense that will be defined in Section 4. For each of the slots, the symbols ***:initarg*** and ***:accessor*** are called the slot options. The option declaration ***:initarg :level*** allows the slot ***level*** to be initialized with a value at the moment an instance of the class ***node*** is created and for the symbol ***:level*** to be used as the key word. The option declaration ***:accessor level*** makes it possible to read the value of this slot by the generic function ***(level*** node-instance) and to change the value by the function call ***(setf (level*** node-instance) ***new-value).*** Similarly for the slot options for the slot ***event-time.***

The subclass ***tnode*** is now defined in the following manner:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The class tnode is for track level data nodes. This class corresponds
;;  to the highest data abstraction on RTBB.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass tnode (node)
  (
     (checklyst :initarg :checklyst :initform nil :accessor checklyst)
     (check :initarg :check :initfonn nil :accessor check)
     (cpa-bracket :initarg :cpa-bracket :accessor cpa-bracket)
     (threat :initarg :threat :initform nil :accessor threat)
     (snode :initarg :snode :initform nil :accessor snode)
     (last-velocity :initarg :last-velocity :accessor last-velocity)
     (last-coord :initarg :last-coord :accessor last-coord)
     )
  (:documentation "The tnode class is for track level data nodes")
  )
```

The argument ***node*** in the first line of this ***defclass*** macro asserts that the ***tnode*** class has ***node*** as its superclass, in accordance with Fig. 5. Because the class ***node*** is a superclass of the class ***tnode***, the latter inherits all the slots of the former, together with the read and write **accessor** functions if any. We will explain in the next section the semantics of the seven local slots defined explicitly for the class ***tnode.*** What we wish to point out here is

the new slot option ***:initform*** that appears in four of the local slots. This option permits us to give a default initial value to a slot, these being all nil for the four slots carrying this option. Without the ***:initform*** option and an associated default value, the value of a slot is left unbound at the time an instance of a class is created, unless the ***:initarg*** option allows a value to be assigned at that time. If the value of a slot for an object instance is unbounded, and if an attempt is made to read the values of such slots, CLOS will signal an error.

An instance of a node at any level of the blackboard may be created by using the generic function ***make-instance*** in the following manner

```
(setq track1 (make-instance 'tnode :event-time 2222 :threat 'true))
```

which would bind an instance of class ***tnode*** to the symbol ***track1***. For this instance, the value of the slot ***event-time*** would be set to **2222** and the value of the ***threat*** slot to ***true.*** Note that this initialization of these two slot values would not have been possible if we had not used the ***:initarg*** option for the slots ***event-time*** and ***threat.***

Besides the notion of object classes that can inherit characteristics from other classes, the other most significant notion in object oriented programming deals with endowing objects with behaviors by the use of methods. Since methods, defined for specific classes, can also be inherited, CLOS provides what are called ***generic functions*** for controlling the flow of inheritance of methods. Before explaining more precisely the purpose of generic functions, we would like to mention the following important facts: 1) a primary method defined for a class will be inherited by all its subclasses; **2)** an inherited primary method from a superclass may be adapted to better serve the needs of a class by defining an after-method; and 3) a before-method may be used to carry out set-up work for a primary method. Methods are invoked for execution by calls to a generic function. By matching the parameter list in the generic function called with the parameter lists of all the methods, CLOS collects together all the applicable before, primary, and after methods and sequences them appropriately for execution; this is done by what is called a generic dispatch procedure. When a sequence of before, primary, and after methods is executed in response to a generic function call, the value returned by the generic function is the same as the value returned by the primary **method;** the before and after methods can only produce side effects. In the event there are multiple primary methods available for a given class owing to the existence of multiple superclasses, the generic dispatch procedure invokes rules of precedence that select that **procedure** which corresponds to the most specific superclass.

Since an after-method may be invoked automatically after initializing or altering critical slots in a node, it is possible to have such specialized methods report the changes to a queue or another portion of the BB. In an event-driven BB the changes are reported to an event queue and in a goal-driven BB the changes are reported to either a buffer in a

centralized monitor or directly to the goal side of the BB. Since RTBB is a goal-driven blackboard, any changes in the data are reported directly to the goal panel of the BB by after-methods associated with classes defining the data objects. One can think of these after-methods as constituting a distributed monitor. The methods may also be visualized as being part of the KS's or as a shared utility of these KS's for reporting changes in the data. The reader should note however that there do exist alternatives for designing monitors. For example, polling techniques along with change bits or variables in the class instantiations could be used to create a centralized monitor. As another alternative, KS's themselves could report all the changes to a centralized monitor since KS's are the only entities allowed to alter the blackboard.

The following ***defmethod*** is an example of an after-method which places a **node** in the goal panel after the event-time slot has been given a value by a primary method.* In the definition of the after-method, the role of the method is declared by the qualifier keyword :after. The primary method to which the defined method is an after-method appears immediately before the keyword ***:after***; in this case the primary method is ***(setf*** event-time), which is the generic writer function for altering the value of the slot event-time. Note in particular the lambda list** of this after-method: (new-slot-value (ele tnode)). When this after-method is invoked for execution, the first parameter, new-slot-value, is instantiated to the new value of the slot event-time. [Recall, it is the change in the value of this slot to whatever will be instantiated to new-slot-value that causes this after-method to be invoked in the first place.] The second parameter in the lambda list is the symbol ele, short for element, which has a specialization constraint placed on it. This specialization constraint, implied by the form (ele tnode), says that the parameter ele can only be bound to an object of class tnode. In other words, the after-method is only defined for track-level nodes on the blackboard. As is evident from its definition, this

* The reader who is already somewhat familiar with RTBB may be puzzled by this ***defmethod*** since it ***creates*** a track level goal node from a change in the track level on the data panel. Usually, a track level goal node will be created by the addition of a segment on the **data** panel, the **purpose** of the goal being to either merge the segment with one of the existing tracks or to start a new track with the segment. However, RTBB also needs facilities for creating track level goals directly from changes in the tracks **because** of the need for verification and possible subgoaling if the track is a threat, meaning if the average velocity vector representing a track is aimed directly at the origin of the coordinate system. The verification consists of making sure that **all** the segments are similar in the polynomial sense discussed in Section 4. When a **track** fails verification, **subgoals** must be created that check each segment against the average properties of the track, and if a segment is found to be **too** different, it must be released **from** the track and allowed to participate in **or** initiate a new **track.** The ***defmethod*** shown here could lead to the formation of **KSAR's** that could produce these **subgoals.**

** **A** lambda list is a list that specifies the names of the parameters of a function, sometimes loosely called the arguments of a function. Strictly speaking, the arguments are what you provide a function when **you** call it; you name the parameters of a function when you define it [18].

after-method **first** makes an instance of the class ***bbgoal*** and then deposits this goal instance at the track level (how precisely that is done will be explained later). With regard to the values given to the slots when an instance of ***bbgoal*** is created, the ***event-time*** slot takes on the value bound to the symbol ***new-slot-value*** that is the updated value of the slot ***event-time*** in the ***tnode*** object bound to the symbol ***ele.*** The slot ***source*** is set to the name of the ***tnode*** object that invoked the method. The slot ***purpose*** is set to 'change to reflect that the goal was caused by changing the ***event-time*** value, in contrast with, for example, a goal node that might be created by sub-goaling. The slot ***initiating-data-level*** takes the value 'track since the formation of the goal node was caused by a change in a data node at the track level. The slot ***threat*** inherits its value **from** the **tnode** that caused the method to be invoked. The value of the slot ***snode*** is a list of pointers to the **snodes** that support this track node. Finally, the slot ***duration*** is set to 'one-shot; this causes only one attempt to be made for this goal node to be satisfied. The reader should note that in the syntax of a defmethod, function calls such as ***(snode ele)*** are **accessor** functions, in this case a reader function that retrieves the value of the slot ***snode*** from the object bound to the symbol ***ele.***

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This after-method automatically generates a goal node at the track
;; level whenever there is change in the value of the slot event-time
;; of a track-level data node (such goals are needed for the initiation
;; of the threat verification process).
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (setf event-time) :after   (new-slot-value (ele tnode))
    (sendpushgoal
          (make-instance 'bbgoal
                             :source                   ele
                             :purpose                  'change
                             :initiating-data-level 'track
                             :event-time               new-slot-value
                             :threat                    (threat ele)
                             :snode                     (snode ele)
                             :duration                 'one-shot)
          tracks)
    )
```

The sendpushgoal macro used above is a procedure that pushes an instance of the class ***bbgoal*** onto the track level of the goal panel. In other words, this macro creates a new track level goal node. The macro is defined in the following manner:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This macro pushes a goal object into the goal panel at the track level.
;;  Note that left refers to the left side of the BB, the goal panel.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro sendpushgoal  (object level)
  `(setf (left ,level)
    (push ,object (left ,level) )))
```

So the set of goals on the track level of the goal BB is just a stack of these class

instances. As mentioned before, this after-method is invoked **after** a change has been made to the event-time slot of a track node on the data panel. This occurs whenever a track node is updated. The message that triggers this change will look something like *(setf* (event-time *mode*_object) new-event-time)).

When only one or two methods are associated with each node type, it is a simple matter to write one method for each slot. However, as the number of slots associated with each node class on the blackboard increases, this becomes cumbersome. Seth Hutchinson suggested using a macro to generate these automatically and actually wrote a `macro` that did this using flavors [17]. The following version is a modified version of that `macro` designed for a goal driven BB using CLOS.

```
 1  (defmacro newclass
 2   (class goal-level slot-list monitor-list super-list &rest options)
 3    (cons 'progn        ;; progn runs the sequence of programs created by macro
 4     (cons              ;; cons command program into the gigantic lisp program
 5      `(defclass        ;; first construct the defclass
 6         ,class         ;; class name in the  defclass macro
 7         ,super-list    ;; the list of inherited classes or mixing classes
 8         ,(do*          ;; do loop to construct all the accessors and initforms
 9           (
10            (wlyst slot-list (cdr wlyst))      ;; cdr down the slot-list
11            (op (car wlyst) (car wlyst))       ;; op is the next slot to be done
12            (mylyst nil)                       ;; mylyst is list of slot options
13           )
14           ((null wlyst) (return mylyst))      ;; return the slot-specifier list
15           (setq mylyst                        ;; construct each slot-option list
16             (cons                             ;; make defining list for slot
17               `(,op :initarg ,(keywordize op);; put "op" in keyword package
18                     :initform nil             ;; default value for slot is nil
19                     :accessor ,op)            ;; for read-write functions
20               mylyst)))                       ;; stuff this in the slot options
21           ,@options)                          ;; for options like documentation
22     (do*          ;; generate one after-method for each slot named in monitor list
23       (
24         (worklyst monitor-list (cdr worklyst))  ;; cdr down monitor list
25         (op (car worklyst) (car worklyst))      ;; choose next candidate
26         (mlyst nil)                             ;; construct list of methods
27         )                                       ;; return method list
28       ((null worklyst) (return mlyst))          ;; when monitor-list is empty
29       (setq mlyst                               ;; construct list of defmethods
30         (cons `(defmethod                       ;; cons defmethod into list
31                  (setf ,op) :after              ;; make an after-method writer
32                  (new-slot-value (ele ,class))  ;; construct lambda list
33                  (sendpushgoal                  ;; make body of after method
34                    (make-instance 'bbgoal
35                                  :source                   ele
36                                  :purpose                  'change
37                                  :initiating-data-level    (level ele)
38                                  :coord                    (coord ele)
39                                  :number                   (number ele)
40                                  :event-time               (event-time ele)
41                                  :duration                 'one-shot
42                                  )
43                    ,goal-level))                ;; specify level to push goal on
44               mlyst)                            ;; put methods into list
45  )))))
```

In this macro, class instance is created of type *class* with slot;: whose names are supplied in the list slot-list. When this macro is invoked, the parameter super-list is bound to the list of superclasses of class. The first do loop, in lines 8 through 20, repeatedly

executes the code in lines 17-19 and generates slots of form *(slot-name :initarg :slotname :initform nil :accessor slotname)* for each slot name in the list bound to the parameter *slot-list.* Subsequent execution of the *defclass* in line *5* then creates the appropriate class. The do loop in lines 22 through 43 creates an after-method of the type shown previously for each slot name in the list bound to the parameter *monitor-list.* Therefore, whenever the value of each slot named in *monitor-list* is updated, *a* goal node is automatically created and deposited at the *datalevel* of the blackboard. The reader might note in particular that the *newclass* macro uses the macro *keywordize,* a procedure used to intern the name bound to the symbol *op* into the keyword package. Here is an example of how *newclass* is called:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The class snode is used to form segment level data node.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(newclass snode tracks (
                        coord   ; note this is a coordinate list
                        number  ; number of points the the segment
                        cpa     ; closet point of approach a vector
                        linear  ; (position velocity)
                        tnode   ; ptr to a track node
                        threat  ; true or false -- updated by tnode
                        )
          (number) (node)
   )
```

This call to *newclass* will create the subclass *snode* for segment level data nodes on the blackboard (see Fig. 3) and will do so in such a manner that an after-method will be automatically generated for the slot *number.* This after-method will automatically deposit a goal node at the track level any time the value of the slot *number* for an snode is changed. The call to *newclass* recognizes the fact that, in accordance with Fig. 5, the class snode is a subclass of the superclass node. If we did not use the macro *newclass,* we would have to separately define the class snode by using *defclass* and then add explicitly the following after-method:

```
(defmethod (setf number) :after  (new-slot-value (ele snode))
    (sendpushgoal
       (make-instance  'bbgoal
                         :source                ele
                         :purpose               'change
                         :initiating-data-level 'segment
                         :coord                 (coord ele)
                         :number                (number ele)
                         :event-time            (event-time ele)
                         :duration              'one-shot)
       tracks)
    )
```

The *newclass* macro is an illustration of the power of macros and the ease with which one can create an impressive array of methods automatically in a BB shell.

So far in this section we have talked about object classes for representing the nodes at the different levels of the blackboard and about the methods associated with these object classes. We will now focus on the representation of the levels themselves. Each level of the blackboard is itself an instance of the following class:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The different levels of RIBB are instances of the following class.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bblevel ()
  (
    (up :initarg :up :accessor up)
    (left :initarg :left :accessor left)
    (right :initarg :right :accessor right)
    (down :initarg :down :accessor down)
    )
  (:documentation " The bblevel class is used for constructing the bb levels.")
  )
```

The values for the slots, up and down, determine the level on the blackboard. For example, the segment level in Fig. 3 would be created by making an instance of the above class by setting up to tracks, and down to hits. At each level, all the data nodes are stored in a list that is the value of the slot right, and all the goal nodes in a list that is the value of the slot left. This corresponds to the left, right organization of the blackboard shown in Fig. 3. For illustration, the following code fragment creates the segment level of the blackboard:

```
(setq segments
  (make-instance 'bblevel :up tracks :down hits :left nil :right nil))
```

The fact that we can store all the data nodes at each level in a single list that is the value of the right slot for that level proves very convenient if one is trying to apply the same function to all the nodes at that level. For example, if we want to apply the same function to all the data nodes at the segment level, we can simply **mapcar** the function to the list of nodes retrieved via the (right segments) generic reader call.

When slot values are allowed to be lists in the manner explained above, such lists may be used either as queues or stacks for the purpose of deciding which objects should be processed first. Here, we use the word queue in a generic sense and associate with it three components: its arrival process, its queueing discipline and its service mechanism. The arrival process is characterized by an interarrival-time distribution for items stored in the queue. The service mechanism is composed of servers and the service time distribution; note there can be multiple servers (e.g., processors) catering to a queue. The queueing discipline describes how an item is to be selected from those in the queue. Items arriving at a queue may be enqueued (stored) until serviced, or the items may be blocked (discarded) if no server is free at that time. This makes it possible for us to use the generic term queue to mean any queueing system such as a LIFO queue (stack), a FIFO queue, or some prioritized queue. For an extensive discussion on queueing concepts, the reader is referred to [4].

3. REPRESENTATION OF THE ABSTRACTION LEVELS

As shown in Fig. 3, the RTBB consists of two panels, each containing three abstraction levels. The lowest abstraction level in the data panel consists of bnodes for beam nodes, also called hit nodes.* The bnode class is defined as follows:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The class bnode is for beam nodes -- the objects holding info at the hit level
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bnode (node)
  (
   (coord :initarg :coord :accessor coord)
   (number :initarg :number :accessor number)
   )
  (:documentation "The beam node is the lowest data abstraction level on the bb.")
  )
```

where ***coord*** is a list of coordinates **(x,y,z)** of a time-stamped radar return, which consist of a set of echoes received during a single scan of the entire search space. The time stamp of such a set of echoes becomes the value of ***event-time*** slot inherited from the superclass ***node***. The slot ***number*** contains the actual number of distinct returns in the set of echoes. Another slot inherited from the superclass is ***level*** whose value is set to hit for nodes of ***bnode*** class. Such nodes are generated every n-th clock cycle where at present n is set to eight.

We will now show an after-method, defined for the object class bnode, that creates a goal every time a new set of hits is received; in other words a goal node is created for each new beam node. In comparison with the after-methods shown in the preceding section, the one shown below first does some computation before creating the goal node. The goal represented by the goal node seeks to assign the new hits in the beam node to the existing segments, if possible, or to create new segments. The computation that is carried out before the creation of the goal node **determines** the number of hits in the bnode. Here is the after-method:

---

* **For this blackboard, hit nodes and beam nodes** are treated the same. In practice, a beam of information **is** more primitive than a hit **since** the latter **is** a time integrated sequence of **beams.**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This after-method first updates a slot of bnode and
;; then creates a segment level goal node.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 1 (defmethod initialize-instance :after ((ele bnode) &key)
 2   (with-accessors ((num number) (crd coord) (evt event-time)) ele
 3     (setf num (length crd))
 4     (sendpushgoal
 5       (make-instance 'bbgoal
 6                      :source                   ele
 7                      :purpose                  'change
 8                      :initiating-data-level    'hit
 9                      :coord                    crd
10                      :number                   num
11                      :event-time               evt
12                      :duration                 'one-shot
13                      )
14       segments)      ;; the level on which the goal node will be deposited
15   ))
```

This after-method also demonstrates the use of the with-accessors macro in CLOS. From a logical standpoint, it is convenient to think of the with-accessors macro as creating a "handle" into each of the slots named for the object bound, in the case above, to the symbol ele in line 2. The slots named in line 2, are number, coord, and event-time and we may think of the symbols *num*, crd, and *evt* as handles into these three slots, respectively. Each handle may be used for either reading the value of a slot or for writing a new value into it. For example, the form (crd coord) will bind the value of the slot coord to the symbol crd. Note that the call *(setf num* (length crd)) will first calculate the length of the list bound to crd and will subsequently write an updated value into the slot number of the object bound to ele. The reader should have already noted that the method defined above is an after-method to the initialize-instance method, which is native to CLOS. The behavior of this method should become obvious from the fact that the make-instance method has to call initialize-instance in order to create an instance from a class. Therefore, the after-method defined above will be invoked every time an instance of class bnode is created by a call of the form (make-instance *'bnode :coord* coord) where the argument coord is the list of hits with the same time stamp. As the reader can tell from line 5, the goal node created is an instance of class bbgoal introduced in the preceding section.

The alteration of the slot number in a *bnode* by the above after-method may seem at variance with the usual viewpoint that, in an ideal conceptualization of a BB architecture, only KS's should be allowed to alter information in the BB database. Actually, what has been accomplished with the above method is not at a great variance from the ideal because that aspect of the *defmethod* which updated the value of number could have been incorporated in the KS that created the bnode in the first place. One can view this data refinement aspect of methods either as constituting extensions of the KS's or making the KS's more distributed. One advantage of such methods is that they simplify the coding of interfaces between the BB process and the KS's. The goal node slots in the after-

method will be defined when we discuss goal nodes in greater detail.

The next level of abstraction on the data panel is the ***snode,*** which stands for segment nodes. Segments are defined for convenience and represent a small number of hits (a fixed number chosen by the designer) that can be adequately modeled as linear segments. By fitting linear segments to the returns, we reduce the sensitivity of the system to noise spikes. Segments that are approximately collinear are grouped together to form tracks; more on tracks later. A track will not be started unless a segment is longer than a certain minimum number of points, usually two. In addition, if the most recent hit in a segment is older than 10 time units, it is automatically purged from the BB database. If a track consists of only one segment and that segment is purged due to the time recency requirement, the track would also be purged. The definition of the segment node class using the *newclass* macro was presented in the preceding section; we repeat the definition here for convenience:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This class is 'for segment level nodes.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(newclass snode tracks (
                         coord   ; list of time-sequenced hits
                         number  ; number of points in the segment
                         cpa     ; closet point of approach
                         linear  ; the pair (position velocity)
                         tnode   ; ptr to the track node
                         threat  ; true or false -- updated  by tnode
                         )
                (number) (node)
  )
```

When a segment object is created from this class, the slot coord contains a list of the coordinates of the hits that constitute that particular segment. Note in particular that whereas the similarly named slot for bnodes contains a list of hits for the same time stamp, the slot here has a time sequenced list of hits constituting a geometrical segment in space. In other words, the coordinates in the slot coord are grouped on the basis of spatial continuity, as opposed to the temporal continuity used in bnodes. The value of the slot event-time, inherited from the superclass ***node,*** is the list of event-times corresponding to the hits in the slot coord. In order to clarify the access discipline used for processing the lists in the slots coord and event-time, both lists are treated as stacks. The value of the slot cpa is the closest point of approach if the segment were to be extended all the way to the radar site, assumed to be located at the origin of the (x,y,z) space. The value for the slot cpa is calculated by an after-method using the position and velocity information contained in the slot linear, the reference here being to the position and velocity of the target computed from the two most recent hits in the segment. More specifically, the value of the slot cpa is the perpendicular distance from the origin to a straight line that is an extension of the two most recent hits in the segment. The slot threat is set to true if the value of cpa falls within a small region around the origin,

otherwise it is false. The extent of this region is ε times the last-coord [a slot for track level nodes to be discussed later], the comparison threshold being dependent on the distance since greater directional uncertainty goes with with more distant craft [this point will be explained further in under the discussion on the **GETTRACK** KS]. While the computation of the value for cpa occurs when a segment node is first created, determination of whether threat is true or false does not occur until a track level node is updated with the segment.

The highest data abstraction consists of track nodes. As mentioned before, a track node is a grouping of approximately collinear segments. Two segments belong to the same track if the following two conditions are satisfied: First, we must have $\cos^{-1}\theta > 0.9$, where θ is the angle between the velocity vectors for the two segments, the velocity vectors being contained in the slot linear for the segment nodes; and, second, the faster of the two craft must be able to reach the other in one unit time. The second condition is made necessary by the fact we do not wish to group together segments for aircraft flying parallel trajectories that are widely separated. In general, there will only be a single track node for a single formation of aircraft, no matter how large the formation. Of course, if a formation splits into two or more formations, the original track would split into correspondingly as many tracks. The track nodes are defined as follows:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the class for track level nodes.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass tnode (node)
  (
   (last-coord :initarg :last-coord :accessor last-coord)
   (last-velocity :initarg :last-velocity :accessor last-velocity)
   (threat :initarg :threat :initform nil :accessor threat)
   (snode :initarg :snode :initform nil :accessor snode)
   (cpa-bracket :initarg :cpa-bracket :accessor cpa-bracket)
   (check :initarg :check :initform nil :accessor check)
   (checklyst :initarg :checklyst :initform nil :accessor checklyst)
  )
 (:documentation "The tnode class represents objects at the track level.")
 )
```

For the values of the slots inherited from the superclass *node,* the value of level is set to the symbol track for all nodes of this class, and the value of event-time is set to the time stamp of the most recent hit in any of the segments constituting the track. Now the slots local to the class ***tnode***: the slot snode contains a list of pointers to the segment level nodes supporting the track. The slots last-coord and last-velocity are the latest average position and the velocity vector associated with the track; the averaging is performed by taking a mean of the position and velocity vectors associated with all the segments in the track. The value of the slot threat is set to t through an after-method by taking a disjunction of the threat values of all the segments in the track. The value of the slot ***cpa***-bracket is equal to the intervals along **x** and y, each interval a union of the **cpa** intervals associated with the segments in the track. If threat is set to "t", a goal node is deposited

at the track level whose job is to conduct a spline check of each segment in the track to confirm that the grouping of segments is coherent, where coherence is measured by the similarity of polynomial coefficients associated with fitting splines to the segments; this work is done by GETSPLINE KS. If the grouping of the segments is found to be coherent, the value of the slot check is set to t, and if not, it is set to fail. Setting check to fail causes the formation of another track-level goal node at the next update of the **tnode,** this goal node is recognized by the rule-based planner, which deposits a bunch of **subgoals** for an alternative grouping of the segments, possibly into multiple tracks. The value of the slot ***checklyst*** is the list of modes that need to be verified as part of the track. ***Checklyst*** is nil unless check has been set to fail.

The abstraction levels for goal nodes are identical to the abstraction levels for data nodes, as shown in Fig. 3. The foundation of all goal nodes is the following class:

```
(defclass goal ()
  (
   (initiating-data-level :initarg :initiating-data-level :accessor level)
   (event-time :initarg :event-time :reader event-time)
   (purpose :initarg :purpose :accessor purpose)
   ))
```

The slot ***initiating-data-level*** refers to the data panel level that resulted in the formation of this goal node. The slot event-time has different semantics depending on the goal node being created. For example, for a goal node at the segment level, the value of the slot event-time is the time stamp of the data panel bnode which **triggered** the **formation** of the goal node. On the other hand, if a goal node is initiated by an snode, then event-time is set to a list of time-stamps of the hits constituting the snode. If a track level goal node is initiated by a **tnode,** then ***event-time*** is set to a single value, which is the latest time-stamp associated with the track. The value of the slot purpose describes the purpose of the goal. For segment level goal nodes, the value of ***purpose*** is to change the existing segments by extending them with the new hits or to start new segments altogether. This being represented by the symbol change.

We will now separately define what is called a **mixin** class that will then be added to the just defined goal class for creating a class of goal nodes with the desired behavior. The reason for defining a **mixin** class is best explained with the help of a hypothetical class hierarchy shown in Fig. 6. Let's assume that most of the behavior exhibited by classes A and B will either be inherited from their respective superclasses or defined locally at A and B. Let's further assume that we want both A and B to exhibit some additional behavior that will be the same for these two classes. This can best be done by defining a **mixin** class separately with this additional behavior and mixing in this **mixin** class at the time the classes A and B are defined. In addition to inheriting methods from their superclasses, the classes A and B will also inherit methods **defined** for the **mixin** class. Although, since the hierarchy of goal node classes in RTBB is simple, we could have avoided using a **mixin** class at this time, we have nevertheless chosen to use it for

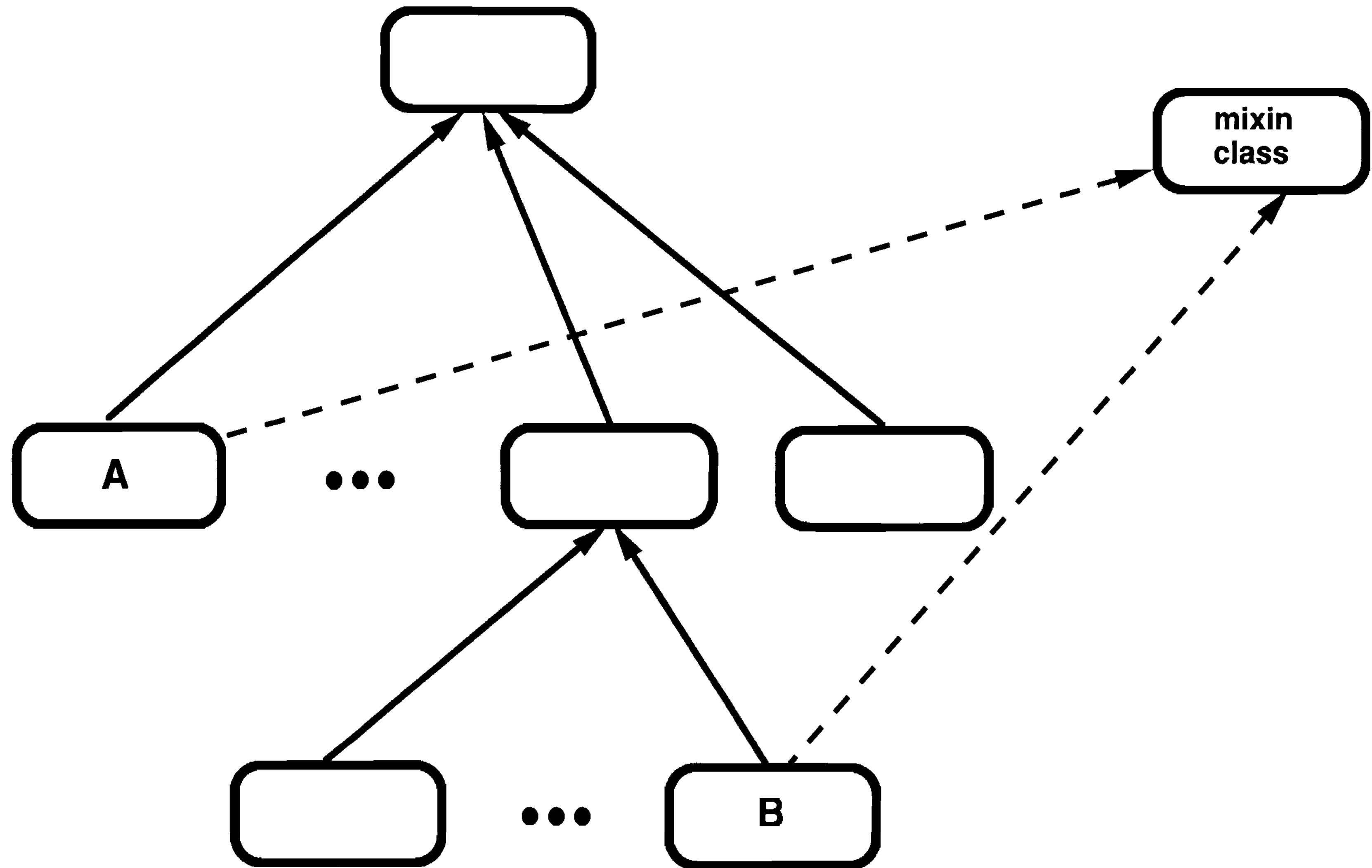


Figure 6 Most of the behavior exhibited by class A and B will either be inherited from their respective superclasses or defined locally at A and B. The use of the mixin class allows us to endow both A and B with some additional but common behavior.

two reasons: one, to make our code more extensible, and, two, to illustrate how a **mixin** class can be used. We use the following **mixin** class:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This mixin class is used for specializing the behavior of the
;;  basic class of goal nodes.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass goal-attributes-mixin ()
  (
   (duration :initarg :duration :accessor duration)
   (source :initarg :source :accessor source :documentation "generating node"))
  (:documentation "This is a goal-attribute-mixin for bbgoal class ")
  )
```

Before explaining the semantics of the slots used here, we will go ahead and define a more useful class of goal nodes that is actually used in the goal panel of the blackboard:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  All goal nodes, regardless of level, are instances of this class.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bbgoal (goal-attributes-mixin goal)  ;; mixing superclass
  (
   (coord :initarg :coord :accessor coord)
   (number :initarg :number :accessor number)
   (threat :initarg :threat :initform nil :accessor threat)
   (snode :initarg :snode :initform nil :accessor snode)
   (ksarptr :initarg :ksarptr :accessor ksarptr)
   )
  (:documentation "This is a subclass of the generic class goal. ")
  )
```

As implied by the definition, the class ***bbgoal*** derives its behavior partly from the **super**-class ***goal*** and partly **from** the **mixin** class ***goal-attribute-mhin.*** We have already explained the semantics of the slots of the class ***goal.*** About the slots inherited from the **mixin** class, the slot ***source*** points to the node on the data panel that resulted in a **particu**-lar goal node. For example, the creation of a bnode is followed immediately by the creation of a segment level goal node whose purpose is to use the hits in the bnode for either extending the existing segments or starting new segments. In this case, the value of the ***source*** slot ***in*** the goal node will be the identity of the bnode that instigated the **formation** of the goal node. For another example, if a track level data is considered to be a **threat,** before the threat is accepted the mode is tested for spatial grouping by applying some tests to each of the segments that constitute the mode. The testing of each segment is carried out by **forming** a separate **subgoal** for that segment. For such subgoals, the value of the slot ***source*** is set to the identity of the mode that failed the grouping test.

The inherited slot ***duration*** has a very important role to play in the control of the blackboard, a fact that will become more obvious in Section 5. The slot ***duration*** refers to the length of time the goal is allowed to stay on the blackboard. For example, a one-shot duration means there is only one opportunity for the planner to test a node against the rules to see if it matches any of the antecedents; if the match fails, the goal node is

discarded. Most goal nodes are of one-shot type; for example, the goal to update a mode with new segments is of one-shot type. Only one KSAR for this goal node, which contains a pointer to the segment that should be used for updating, will ever be formed by the rule-based planner. The goal node is purged as soon as the KSAR is **formed.** Therefore, if this KSAR fails to satisfy the goal node, the goal node will not be there to reattempt updating of the **tnode** with the same segment.

In addition to the one-shot type, RTBB also contains a recurrent goal node. A recurrent goal node is disabled after it satisfies the antecedent of specific rules, and then is reenabled after a KS is *fired* **from** the subsequently generated KSAR. Recurrent goal nodes are never removed from the blackboard. The job of the recurrent goal node that is currently in RTBB is to first locate old segments, these are segments whose most recent returns are between 3 and 10 time units old, and to attempt to join these segments with more recent segments. Suppose the database at the segment level contains an snode composed of the following hits ($h1_1$,.........,hl,) and let's say the time stamp of $h1_1$ is 7, of $h1_2$ 8, and so on. Also, assume that there exists another snode made up of hits ($h2_1$,$h2_2$,$h2_3$) where the time stamp of $h2_3$ is 3. Then the job of the recurrent goal node will be to merge the two segments since the time stamp of $h2_3$ is so close to that of $h1_1$. The actual merging, carried out by the MERGE-SEGMENTS KS, will only take place if the extension of the h2 segment to the time instant corresponding to the beginning of the h1 segment is within an acceptable circle.

About the slots that are defined locally for the class ***bbgoal,*** the value of the slot ***ksarptr*** is set ***to*** nil for one-shot goals; for the recurrent goal, it is set to the internal identity of the KSAR that is generated by the goal node. While the ***ksarptr*** slot maintains this value, the recurrent goal node is inhibited from launching another KSAR. (The instantiation of ***ksarptr*** is reset to nil by the termination of the execution of the MERGE-SEGMENT KS.) For a goal node at the segment level, the value of the slot ***coord*** is set to the list of coordinates of the hits that have to be assigned to segments. When a track level goal node is launched by a mode, then ***coord*** is left uninstantiated. For segment level goal nodes, ***number*** is set to the number of hits in the radar return that are yet to be assigned; for track level goal nodes, it is left uninstantiated. The instantiation for ***threat*** takes place by mechanisms explained earlier; basically this variable is set to t or nil or the list of pointers corresponding to the snodes that compose the track.

The goal nodes at all three levels are created by making instances of the ***bbgoal*** class. Note the important distinction between the data and the goal panels: While on the data side we have a separate class for each abstraction level, on the goal side a single class is used, the reason being that the goal nodes at all the levels form together a database for the rule-based planner and therefore their similarity is a convenience.

Given that the reader is now familiar with the organization of RTBB, the overall method for solution formation is restated, hopefully in a more precise manner. All the

radar returns or hits generated on a scan of the search space are given the same time stamp. The list of hits occurring in one scan is contained in a class instance on the hit level of the data panel shown in Fig. 3. Arrival of a new list of hits triggers the distributed monitor to place a goal node on the segment level of the goal panel. This goal node represents a desire or a request to use the new list of hits to update existing segments. If no existing segment can be found to match a particular hit, a new segment is started with the new hit.

The segment nodes on the data panel are supported by the hit nodes. The segment nodes are, in turn, grouped into track nodes. To drive the segment nodes to a higher abstraction level, that is to push segments into tracks, one needs to express this desire by establishing goal nodes at the track level of the goal panel. These goals point to segment nodes which need to either extend existing tracks or establish new tracks. Tracks are not established from segments unless the segments are at least two points long (actually any length threshold may be chosen since this is a constant parameter). A track may be thought of as a running segment that provides some buffering against spurious noise resulting in false segment starts. However, a track is more than an extended segment; it may represent many segments so that if several craft are in tight formation, these craft would be represented as one track, with the track being characterized by an average position and velocity vector.

To process, say, a track-level goal for extending a track by a new segment, a KSAR would be generated for the goal node at the track level. In this case, the KSAR generation is accomplished by the firing of a rule in the rule-based planner. This rule requires that the value of the initiating-data-level slot of the goal node be "segment", that the goal node have more than one data point, and that the value of the action slot be "change". If all these antecedent conditions are satisfied then the ***create-ksar*** function is called and a KSAR is created to either extend a track by the segment or to use the segment for starting a new track. The function create-ksar uses the information in the goal node to select the correct class instantiation for the KSAR.

In general, a goal can only be satisfied by activating a KS via a KSAR. So a goal node must activate a KS directly via an appropriate KSAR, or indirectly through subgoals generated from the goal. The priority of the KSAR generated by a goal node will determine its position within the KSAR queue, as further discussed in Section 5.

4. KNOWLEDGE SOURCES

There are six KS's that are part of RTBB. Each KS is a specialist solving a small portion of the problem and each concentrates on a blackboard object. The following is a list of these KS's and a short description of their purpose.

HIT GENERATION (GETBEAM)

This KS is written in C and simulates the trajectories for the various craft. Trajectories are generated by using Bezier curves in 3-space [10]. A Bezier curve is specified by a trapezoid formed by four vectors, denoted by $\mathbf{r}_0$, $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{r}_3$ in the following formula in which $\mathbf{r}(t)$ represents the position of an craft at normalized time u:

$$\mathbf{r}(u)=(1-u)^3\mathbf{r}_0+3u(1-u)^2\mathbf{r}_1+3u^2(1-u)\mathbf{r}_2+u^3\mathbf{r}_3$$

where it is assumed that time is normalized such that $0 \leq u \leq 1$ for the entire flight of the craft. Every nth clock cycle (currently n=8) a goal node is placed on the hit level of the goal panel, the goal being to fire the **GETBEAM** KS. A KSAR is then is formed directly from this goal node by the rule-based planner. The scheduler uses the KSAR to invoke the **GETBEAM** KS. For the case when a single craft is being tracked, the KS will create a bnode composed of $\mathbf{r}(u)$ and its associated time stamp and then deposit this bnode on the hit level of the data panel. The step size of the trajectory thus generated is controlled by the step size of u, which is stored as a constant within the C program. When more than one trajectory is desired for simulating the flight of a formation, a separate Bezier curve must be specified by designating its 4x3 parameter matrix for each craft in the formation, and upon each call the **GETBEAM** KS returns the coordinates of all the craft in the formation.

ASSIGNMENT (GETASSIGNMENT)

After a set of radar returns with the same time stamp is received, one of the following actions must be taken:

1. extend an existing segment,
2. start a new segment,
3. merge two existing segments,
4. terminate an existing segment.

The GETASSIGNMENT KS handles the first two cases. Merging is done by a separate KS and termination of atrophied segments handled directly by the rule-based planner.

The problem of assigning hits to .segmentsis akin to the consistent labeling problem in which one seeks to assign a set of labels to a set of objects, each object taking one and only one label. Although, clearly, a metric is needed to compare hits against the segments -- the metric could be a function of how far apart a hit is from a segment spatially and temporally -- assigning hits to segments is made complicated by the fact that after one such assignment has been made, that segment is no longer available for the other hits. Our current solution to this problem uses a branch and bound procedure (a special case of A* search [28]) implemented via a best-first search algorithm; see [16, 31] for

implementation of best-first search. Further discussion on the complexities of the assignment problem, also called the data association problem, can be found in [3].

TRACK FORMATION (GETTRACK)

This KS groups segments or linear fits by average trajectory. More precisely, the KS groups together segments that are close in both coordinate and velocity space; such groups are then represented by "average" trajectories called tracks. "Close" in coordinate space means within one time unit of travel for the fastest craft. That is, if the fastest aircraft turned directly toward the other craft, the former would intersect the latter within one time unit. The velocity vectors are "close" if they are parallel or nearly so (i.e., the cosine of the angle is greater than 0.9) Other conditions may be added to ensure that the velocity vectors are more similar. This KS is written in Lisp and compiled, loaded and saved using dumplisp.

This KS also evaluates the threat of a track to the region near the origin by using a threat assessment algorithm. The two quantities needed for this are the current position, given inthe slot linear, and the value of the slot cpa. (Recall that these two slots are defined for the *snode* class and that a mode points to the snodes that form a track.) The cpa, which stands for "closest point of approach," is computed by extending the velocity vector of the aircraft and then computing the closest distance from the origin to this extended vector. An error vector $\vec{Err} = \varepsilon \cdot (\vec{r} - \vec{cpa})$ is formed, where $\varepsilon = 0.1$ in the current implementation. The magnitudes of the x, y, and z components of the error vector, $|Err_x|$, $|Err_y|$, and $|Err_z|$, are then used to define an uncertainty box centered at the $\vec{cpa}$ point. If any of the coordinate axes passes through this uncertainty box, the craft is considered to be a threat.

SPLINE INTERPOLATION (GETSPLINE)

If the GETTRACK KS determines that a particular track does indeed pose a threat to the origin, a verification of the "soundness" of the track must be immediately carried out, since it is possible for the average parameters associated with a track to give rise to a threat while the actual trajectories within the track are non-threatening or even diverging away from the origin. The GETSPLINE KS does this verification by fitting a spline to each of the trajectories within a track and comparing the trajectories on the basis of the spline coefficients. This KS, written in C, is based on a spline routine in [10, 11] and obtains a polynomial expression for the track between sample points based on the coordinates and time stamps held in the segment nodes.

Periodic verification of threatening tracks are triggered by the rule base control which checks the last time the tnode for the track was spline checked, and initiates a spline check every time this interval exceeds the recheck-period.

MERGE SEGMENTS (MERGE-SEGMENTS)

This KS detects moderate length gaps in the trajectory data and then attempts to extend the older segments to the appropriate current segments, thus creating longer and more established segments. Of course, if a segment stays faded for a long time (in the current implementation, more than 10 clock units), the segment is eventually removed from the BB.

MERGE-SEGMENTS KS goes into action if the time at which a segment was last updated and the beginning time of another segment is greater than 3 clock units and less than 10. For time separations of 3 or less, the GETASSIGNMENT KS is capable of assigning hits to segments directly. If an older segment is eligible for merging on the basis of this time-window criterion, the older segment is extended in time and space and then its predicted position is matched with the more recent segment to see if merging can be carried out successfully.

The MERGE-SEGMENTS KS is implemented within the BB process itself since it requires extensive access to the data nodes on the BB itself. (If shared memory were available on the BB, one could implement the KS as a separate process.) RTBB uses a recurrent goal node, at the segment level, to monitor and schedule the MERGE-SEGMENTS KS, implying that a goal to invoke this KS is placed permanently at the segments level of the goal panel. This goal node is an instance of the class bbgoal; in this instance, the slot purpose is set to 'merge-segments', the duration to 'recurrent', the initiating-data-level to 'segment, and the ***ksarptr*** to nil. When segments satisfy the rule to activate the MERGE-SEGMENTS KS, a flag pointing to the generated KSAR is established in the goal node. As was stated in Section 3, this flag inhibits any further activation of the KS until the end of the execution of the KS. Once the KS has been activated and its execution completed, the flag is removed and the rule base can satisfy the goal again. In this manner, the MERGE-SEGMENTS KS is run on a continuing basis and at a low priority. Example 4 in Appendix A demonstrates how the merge-segments KS works.

THE SEGMENT VERIFY KNOWLEDGE SOURCE (VERIFY)

This KS is used to verify that a segment still matches a particular track after the GETSPLINE KS has failed, indicating that the segments are no longer consistent. Implemented as a part of the main BB process, this KS merely examines each segment composing the current track to determine if it still satisfies the initial formation condition. The manner in which this is done is by subgoaling. One subgoal is generated for each segment node in the track by the rule base, the subgoal being for the BB to verify that the segment belongs to the track. If a segment fails the verification test, the pointer to the segment from the track and the pointer to the track from the segment are removed. The

segments thus released can reform new tracks at a later time.

The test conditions for verifying whether a segment belongs to a track are the same as those needed to form the tracks in the first place. During this verification period, the GETSPLINE KS, which initially detected the improper grouping of segments, is suspended. This is accomplished by marking the track node check slot asfailed and having the GETSPLINE KS check that condition before the KS can be fired.

This ends the introduction to the'various KS's in the system. To put the KS's in a perspective, the GETBEAM KS drives the blackboard with radar return samples. The GETASSIGNMENT KS maps these samples into linear approximations of trajectories and the GETTRACK KS further groups these linear segments into tracks. The GETSPLINE KS checks that the final trajectory grouping makes sense, especially if the average parameters associated with it are such that the track is considered to be threatening. The VERIFY KS is used to break out tracks that fail the GETSPLINE KS test; the segments thus released are allowed to form tracks later. The two KS's, GETSPLINE and VERIFY constitute a backward type of reasoning. And lastly, the MERGE-SEGMENTS KS attempts to maintain track continuity across fades in trajectories.

5. BLACKBOARD CONTROL

Ideally, the control of the blackboard should be opportunistic in nature, i.e., choose the KS that advances the solution the most [32]. However, whether or not control can be exercised in an "optimal" manner depends ultimately on the programmer who presumably has an understanding of the application domain. In RTBB, data events are mapped into goal nodes, which in turn are mapped either into subgoals or KSAR's. The KSAR's are enqueued into a priority queueing system, the queuing discipline determining the order in which the KSAR's appear in their respective queues. The scheduler then cycles through the KSAR queues and selects KS's to fire.

We will now describe various possible approaches to the representation and processing of KSAR queues. Then, at the end of this section, the current RTBB implementation of the KSAR queueing system will be discussed. Ideally, the KSAR priorities should be dynamically determined based on the threat a craft presents to the command and control center presumed to be located at the origin of the coordinate system. For dynamic prioritization, the planner must contain rules for assessing the relative severity and immediacy of a threat. Furthermore, the scheduling of the threatening tnodes must allow the other goal nodes in the system to be serviced often enough so that any future threats would not be ignored. Evidently, designing a planner and scheduler for such dynamic prioritization is a complex task and is not addressed in RTBB. We have chosen

a simpler approach to KSAR prioritization that has the virtue of allowing for the main BB process to activate KS computations while the main process attends to other chores. As we will show later, our approach is an approximation to what may be thought of as a desirable approach to KSAR prioritization in which a separate KSAR queue is used for each KS and then, as shown in Fig. 7, each KSAR queue is visited once in a cyclic fashion, perhaps using a FIFO access discipline.

Before describing the KSAR prioritization scheme actually used in RTBB, we would like to mention that the rule-based planner for mapping goal nodes into **KSAR's** is a forward chaining system. Here is an example of a rule from the planner:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Rule 5 creates a KSAR for invoking the MERGE-SEGMENTS KS if
;;  appropriate conditions are satisfied by the goal node.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq rule5
  '(rule merge-segments
     (if
        (and
          (equal (purpose gnode) 'merge-segments)
          (null (ksarptr gnode))  ; no merge-segment ksar currently active
          (setq rvar1 (find-oldest-segment))
          (setq rvar3 (find-most-recently-started-segment-with-length-gt-y 1))
          (setq rvar2 (abs (- (car (event-time rvar1))
                              (car (last (event-time rvar3)))))))
        (and (> rvar2 3) (<= rvar2 10)) ; is the age within proper range
        ))
    ;; --- rule attempts to patch fades in signal ---
     (then
        (progn                ; this creates ksar and sets ksarptr to that ksar
        (setf (ksarptr gnode)  (create-segment-merging-ksar gnode))
        ))))
```

The goal node, gnode, will be an instance of class ***bbgoal*** defined earlier. This rule states that:

IF • the purpose of the goal node is "merge-segments"
and there are no **KSAR's** fired from this rule
and the difference between the end time of a segment and the start time of another segment is between 3 and 10 time units,

THEN • create a KSAR to merge the two segments.

A condition for this rule to fire is that the value of the slot ksarptr of the object gnode must be nil. Therefore this rule is disabled by the *(setf .....)* statement in the consequent of the rule; this call to ***setf*** invokes the generic writer function and sets the value of the ***ksarptr*** slot of the gnode object to the internal identity of the generated KSAR.

The above rule creates a KSAR by a call to **create-segment-merging-ksar**function which simply first makes an instance of the KSAR class and then pushes this instance

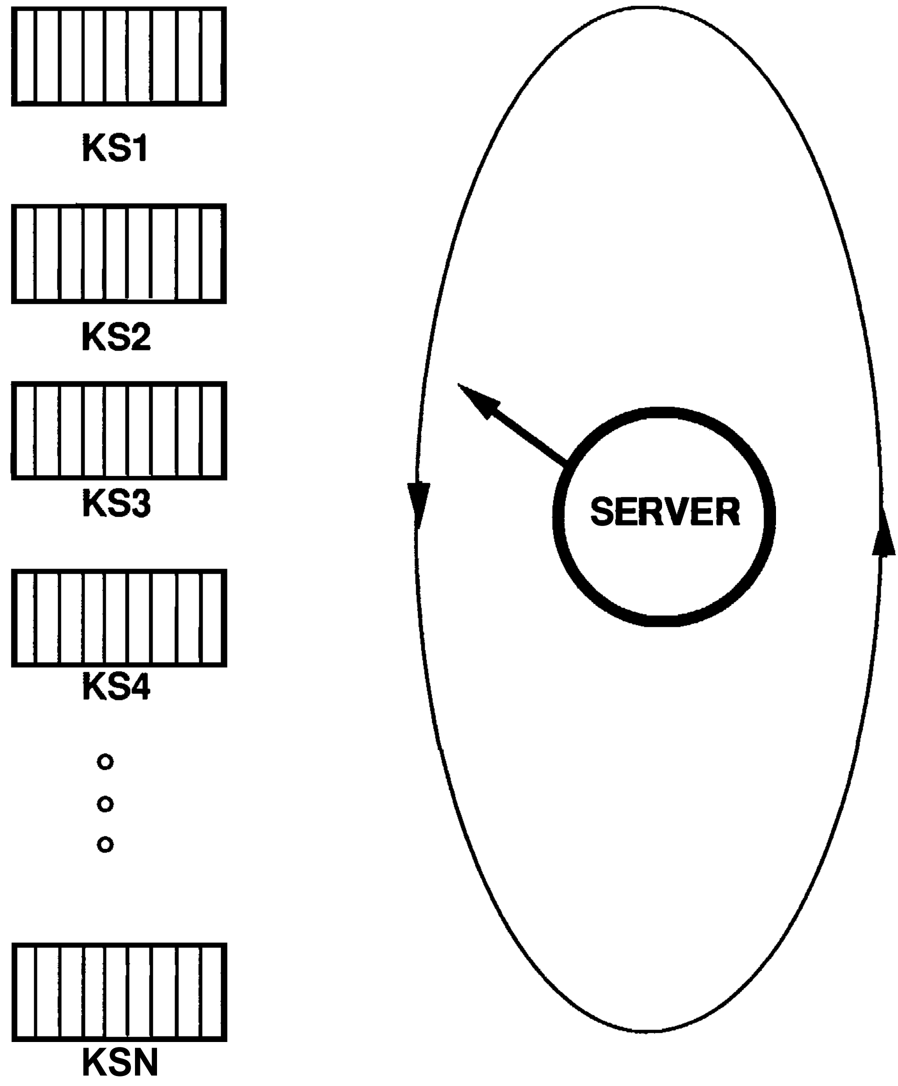


Figure 7. This figure depicts an ideal KSAR queueing system.

into the KSAR queue. This function is fairly easy and is shown below:

```
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function creates the segment merging ksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-segment-merging-ksar (gnode)
  (sendksarpush
    (make-instance 'ksar
                   :priority 1
                   :ksar-id 'merging
                   :postboot '(merge-segments)
                   :command 'merge-segments
                   :cycle clock
                   :context gnode
                   )
    ksarq)
  )
```

The following is an example of a KSAR created by a call to the above function.

;; An example KSAR that seeks to invoke MERGE-SEGMENTS KS

```
#<ksar @ #x6269fe>
is an instance of class #<clos:standard-class ksar @ #x567bce>:
 prelyst      <unbound>
 preboot      <unbound>
 anslyst      <unbound>
 arglyst      <unbound>
 command      merge-segments
 messenger    <unbound>
 channel      nil
 nodeptr      <unbound>
 postboot     (merge-segments)
 context      #<bbgoal @ #x577eee>
 cycle        72
 ksar-id      merging
 priority     1
```

This KSAR is constructed by making an instance of the following class, together with the **mixin** class ks-protocol-mixin whose purpose should become clear when we discuss distributed **KSAR's.**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; When the conditions on the blackboard are right for firing a
;; KS, that fact is stored in a knowledge source activation
;; record (KSAR). Defined here is the class for making KSAR objects.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass ksar (ks-protocol-mixin)
  (
   (priority :initarg :priority :accessor priority)
   (ksar-id :initarg :ksar-id :accessor ksar-id)
   (cycle :initarg :cycle :accessor cycle)
   (context :initarg :context :accessor context)
   (postboot :initarg :postboot :accessor postboot)
   (nodeptr :initarg :nodeptr :accessor nodeptr)
   (channel :initarg :channel :initform nil :accessor channel)
   (messenger :initarg :messenger :accessor messenger)
  )
  (:documentation " The knowledge source activations records ")
  )
```

In the above KSAR class definition, the slot ***priority*** needs some explaining. We said earlier that ideally a separate KSAR queue should be created for each KS. Although this is our goal, it has not been fully achieved yet. At this time in RTBB, we have separate KSAR queues for only the GETBEAM and the GETASSIGNMENT KS's. The queue for the GETBEAM KS is called ***beam-queue*** and the one for the GETASSIGNMENT KS ***segment-queue.*** For reasons that will be explained shortly, the KSAR's corresponding to the GETBEAM and GETASSIGNMENT KS's are said to be of ***distributed*** type. All the other KSAR's will be said to be of ***atomic*** type. All the atomic KSAR's are enqueued separately; this queue is called the ***atomic-queue.*** While the ***beam-queue*** and the ***segment-queue*** are FIFO, as they should be, it would be unreasonable to impose the same queueing discipline on the atomicqueue. The value of the slot ***priority*** reflects the priority that should be accorded to the KSAR shown in the atomicqueue. The ***ksar-id*** slot is used to enqueue the goal node in the proper KS queue.

The slot ***cycle*** is set to the clock time at which the KSAR was created. The slot ***context*** is set to the pertinent aspects of the context at the time of KSAR creation. The value of this slot can be as simple as just the internal identity of the bbgoal that caused the creation of the KSAR; or, in other cases, can also include information such as the latest time associated with an snode, the number of hits of which the snode is composed, etc. The slot ***nodeptr*** is set to the internal identity of the goal node that gave rise to the KSAR. The other slots in the above KSAR will be explained after we define more precisely what we mean by a ***distributed*** KSAR.

When a KSAR of the type shown above (see #<ksar @ #x6269fe>) is selected and its corresponding KS executes, the control resides with the KS until such time when the execution of the KS is over. In other words, the main BB process simply waits for the KS to finish up before focusing on any other activity. **KSAR's** that hand over control to their respective KS's are defined to be of type ***atomic.*** In RTBB, atomic KSAR's have been used for most of the KS's. One advantage of an atomic KSAR is that because it

allows the KS to wrest control away from the main BB process, it implicitly freezes the context. In other words, since the information on the BB can not be altered during the execution of the KS, no intermediate incorrect information can be returned by the KS. Clearly, if the information on the BB was allowed to change during the execution of the KS, it is entirely possible that what is returned by the KS may not be relevant to the new state of the BB.

One major disadvantage of an atomic KSAR is that it does not permit exploitation of parallelism that is inherent to problem solving with blackboards. As we mentioned in the Introduction, one main attraction of using the BB paradigm is that the KS's, if they represent independent modules of domain knowledge, should lend themselves to parallel invocation. Although from the standpoint of enhancing performance, parallel execution of KS's is highly desirable, the reader beware, however. Parallel execution also demands that attention be paid to the elimination of interference between the KS's, in the sense that one KS should not destroy the conditions that must exist on the BB for the results returned by another KS to be relevant. Researchers have proposed methods for dealing with these difficulties; the methods consist of either locking regions of the BB database or tagging different nodes with the identities of the KS's that need them [21]. There is also the opinion that one should not bother with the overhead associated with region locking or data tagging, and should simply let the BB resolve on its own any inconsistencies that might arise due to interference between the KS's.

In addition to atomic KSAR's, in RTBB we also have another type of KSAR's that permits parallel invocation of two of the KS's; we call the latter type distributed KSAR's. The KS's that can be invoked via distributed KSAR's are GETBEAM and GETASSIGNMENT. An instance of a distributed KSAR is made from the same class that is used for an atomic KSAR. A most important characteristic of a distributed KSAR is that it allows the BB database to interact with the KS on a polling basis.

The KS corresponding to a distributed KSAR is executed in three stages. The first stage sends a command to the KS containing all the information needed to execute the KS. The format is just a list which represents a function call with all the information as arguments. The KS then just eval's the list. The second stage occurs when the system first polls the KS port to see if the KS is finished. Note that this polling cannot be accomplished by pressing into service a regular read function, such as the Common Lisp read, since such a function would simply wait for the data to appear or do something unpredictable, but that is not what we want. What we wished was that we be able to poll the KS every few clock cycles, check for whether or not the KS had returned the results, then read the results if available. In the absence of any results from the KS, we wanted the system to move on to other tasks and to return to the KS at a later time. Hence, the use of the Common Lisp function listen for implementing a non-blocking read which takes the KS results and stores them in the KSAR. The third stage occurs when the BB

takes the answer returned from the KS and modifies the BB accordingly. Between the stages, the BB is actively working on other parts of the problem. The result is a speed up due to the parallel processing carried out by the system.

An example of a distributed KSAR which seeks to invoke the GETASSIGNMENT KS follows.

```
#<ksar @ #x66dcee> is an instance of class #<clos:standard-class ksar @ #x567bce>:
   prelyst       ((#<snode @ #x583aa6> #<snode @ #x583ab6> #<snode @ #x583ac6>)
                  ((3 97.68385 2.682486 0.0) (2 1.47 98.4704 0.0)
                  (3 97.68385 2.1825 0.0)) ((4 96.8832 2.88 0.0)
                  (4 96.8832 3.379968 0.0)) ((96.8832 2.88 0.0)
                  (96.8832 3.379968 0.0)) 4)
   preboot       (pre-assign-hits)
   anslyst       nil
   arglyst       ('((3 97.68385 2.682486 0.0) (2 1.47 98.4704 0.0)
                    (3 97.68385 2.1825 0.0)) '((4 96.8832 2.88 0.0)
                    (4 96.8832 3.379968 0.0)))
   command       getassignment
   messenger     #<messenger @ #x577d1e>
   channel       1
   nodeptr       #<bbgoal @ #x659796>
   postboot      (post-assign-hits)
   context       ((event-time nil) (number #<bbgoal @ #x659796>) (coord 4))
   cycle         36
   ksar-id       segment
   priority      1
```

This KSAR is created by making an instance of the ksar class shown earlier. The mixin class ks-protocol-mixin that is a part of the KSAR class definition is presented below:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is a mixin class called ks-protocol-mixin
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass ks-protocol-mixin ()
  (
   (command :initarg :command :accessor command)
   (arglyst :initarg :arglyst :accessor arglyst)
   (anslyst :initarg :anslyst :accessor anslyst)
   (preboot :initarg :preboot :accessor preboot)
   (prelyst :initarg :prelyst :accessor prelyst)
  )
  (:documentation "This is a mixing-class")
  )
```

As will be evident from the following definitions of the slots, the mixin class is only useful for a distributed KSAR. Since all KSAR instances use the mixin, the reader might wonder why use the mixin class ks-protocol-mixin at all; after all, the slots in the mixin could have been incorporated in the definition of the ksar class. The reader should note that even when a mixin is always used for defining objects, its separate definition allows the definitions of objects to be expanded incrementally as the software is being developed. Also, one can take advantage of the fact that mixin associated methods will be invoked in a certain order depending upon the order of appearance of mixins, etc.

We will now explain the nature of the slots in the above distributed KSAR. We have already explained the nature of the slots from *priority* through *nodeptr* in connection with atomic KSAR's. We will now define the other slots. The slot channel takes on a

value **from** the set (2, 1, -1, 0}. When the value is 1 (flow of information outward), the KSAR is in the first phase, meaning that it is ready to send a command to the KS that would initiate the execution of the KS; the command itself is taken off the slot command and its arguments from arglyst. After the command is transmitted to the KS, the value of channel is set to -1 (flow of information inward), which is a signal to the BB process that it should start polling the KS **port** for new results using **non-blocking** read. After the results are read off the KS port, they are deposited in the KSAR at ***anslyst*** and at that time the value of channel is changed to 0. The value 0 (information stays in the BB process) for channel causes the function that is at postboot, in this case the function is **post**-assign-hits, to take the results out of the KSAR and deposit them at the appropriate place in the BB database, at which time the KSAR ceases to exist. It is obvious that channel is being used for sequencing in the **correct** order the initiation, execution, and **results**-reporting phases of KS operation. In the above KSAR example, the slot arglyst already has a value, so KSAR processing can begin in phase 1. While in some cases a value for arglyst can easily be generated at the time the KSAR is formed by the planner -- this is the case when a KSAR is formed for invoking **GETBEAM** since the arglyst here is nil -- in other cases, some computational effort may have to be expended for constructing the arguments. In the latter cases, arglyst is synthesized by adding yet another phase to the three phases we have already mentioned. This additional stage is specified by the setting the value of channel to 2. When the scheduler sees this value, it puts out a function call which constructs the arguments, the function call being held in the slot preboot. In the above example, the value of arglyst was generated by a call to the function **(pre-assign**-hits) during the phase when channel was set to 2. The preboot function, in this case pre-assign-hits, not only synthesizes arguments for the function call to the KS but also puts together, for diagnostic purposes, a **list** of all the BB database items that were used for the arguments. The database items used are stored in the slot prelyst.

A note of explanation is in order for the exact nature of arguments under arglyst in the above example. The function pre-assign-hits examines all the snodes in the BB database and **yanks** out of each snode the most recent hit. This list of these most recent hits is the first of the two arguments in the slot arglyst, the time-stamp corresponding to this argument is 2 or 3. The second argument under arglyst, corresponding to time-stamp 4, is the list of hit nodes that must either be assigned to the segments, or allowed to form new segments. The GETASSIGNMENT KS then tries to assign each new hit to a segment based on the spatial and temporal closeness of the hit to the most recent entry in the segment.

The actual activation of a KS, for both the atomic and distributed **KSAR's,** is carried out by sending a write command to a class which acts as an **Input/Output** (I/O) handler for the BB. The write command is synthesized by the following method that is defined for the ks-protocol-mixin class.

---

*Note that this method is neither an after-method nor a before- method. The method that is shown here is a primary method that is invoked by calling the generic function 'write-ks' with an object,

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This  method writes to the input port of the KS, which is the
;;  same as one of the output ports of the BB process.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod write-ks ((ele ks-protocol-mixin))
  (with-accessors
    ((com command) (alyst arglyst) (mess messenger)) ele
      (format (write-port         ; get output port name from messenger object
                (messenger ele)   ; get messenger name from variable
              ) "~a~%"  (cons com alyst))        ; form function call
      (setf (channel ele)  -1)    ; change state of ksar to read
  ))
```

Essentially it is a complex format statement which finds the correct input port to the KS (which is the same as an output port of the BB process), constructs the command sequence from the slot command and ***arglyst*** in the KSAR and sends the command to the port. Before exiting, the method also changes the state of the KSAR channel to reflect that the command has been sent to start KS execution and that the KSAR is now ready to poll for an answer using the **non-blocking** read.

We have not yet explained the purpose of the slot messenger in the **distributed** KSAR example we showed previously (see **#<ksar @ #x66dcee>).** To understand the function of this slot, note that we need to associate with each KS an I/O handler containing **information** such as the identity of the input and the output ports **associated** with the KS. I/O handlers are created by making instances of the messenger class shown below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the class messenger. Instances of this class are the 1/0
;;  handlers associated with the KS's.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass messenger ()
  (
   (write-port :initarg :write-port :accessor write-port)
   (write-fd :initarg :write-fd :accessor write-fd)
   (read-port :initarg :read-port :accessor read-port)
   (read-fd :initarg :read-fd :accessor read-fd)
   (pid :initarg :pid :accessor pid)
  )
  (:documentation "The messenger class allows us to establish 1/0 with KS's.")
  )
```

The values of the slots write-port and read-port are set to internally generated symbolic names that designate the two ports. Since they are both set to the same bidirectional stream, the read-port and the write-port are really the same in Allegro Common Lisp that we have used for this project. For example, for the **GETBEAM** KS, this is done using a run-shell-command as follows:

---

that will be bound ***to the*** **parameter** ***ele and*** **that must** ***be*** **of** ***class ks-protocol-mixin.***

```
(defun openports ()
  (multiple-value-setq (beam error-beam beam-id)
    (run-shell-command "path" :wait nil
       :input :stream :output :stream
       :error-output :stream))
)
```

The instantiation of the global variable beam is then assigned to both the write-port and the read-ports slots of the message object. The function call **(openports)** produces a bi-directional stream with

```
beam set to #<excl::bidirectional-terminal-stream @ #x7bb96e>
error-beam set to #<excl::input-terminal-stream @ #x7bcbb6>
and beam-id set to 18476, the UNIX process id
```

The slots ***write-fd, read-fd*** and pid are not being used in this version of the blackboard. The slot messenger in the distributed KSAR shown previously is instantiated to the identity of that instance of the messenger class that is associated with the KS that the KSAR seeks to invoke.

We will now address the subject of how the KSAR's are queued in the current implementation of **RTBB**. As mentioned before, to maximize the potential for parallel implementations of the **KS's** the system should construct separate KSAR queues for each KS. However, the current implementation has separate queues for only the **GETBEAM** and GETASSIGNMENT **KS's,** called beam-queue and segment-queue, respectively; all the other KSAR's are enqueued into a single queue called the atomic-queue (Fig. 8). Each KSAR queue is stored in the appropriate slot of an object that is an instance of the following ***bbksarq*** class.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; This class is used for creating ksar queues. The different slots
;;; of the singular object that is made from this class are used for
;;; storing different ksar queues.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bbksarq ()
  (
   (number :initarg :number :initform '() :accessor number )
   (mask :initarg :mask :initform '(1 1 1) :accessor mask)
   (atomic-queue :initarg :atomic-queue :initform '() :accessor atomic-queue)
   (beam-queue :initarg :beam-queue :initform '() :accessor beam-queue )
   (segment-queue :initarg :segment-queue :initform '() :accessor segment-queue )
   (track-queue :initarg :track-queue :initform '() :accessor track-queue )
   (spline-queue :initarg :spline-queue :initform '() :accessor spline-queue )
   (merge-queue :initarg :merge-queue :initform '() :accessor merge-queue )
     )
  (:documentation "Only one object, called ksarq, is made from this class.")
  )
```

The slot number is set to the total number of ksar's held in all the queues. Note that ***mask*** represents the status of the KSAR's that are currently at the head of the queues. The list that is the value of ***mask*** has a status entry for each of the distributed-KSAR queues, and the interpretation to be given to each entry in the list is the same as that given to the values for the slot channel in a distributed KSAR. In the ***defclass,*** the initial mask values have been set to 1 for the head **KSAR's** in both the beam-queue and the segment-queue,

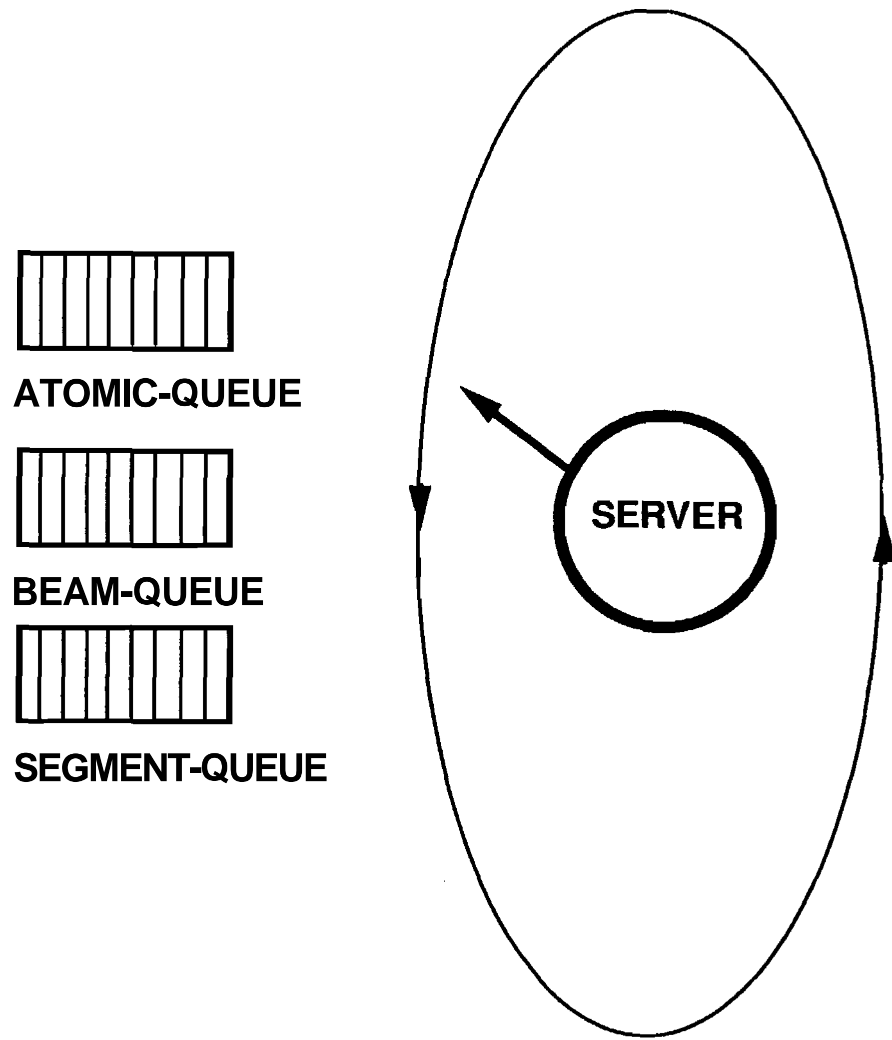


Figure 8. This is the actual KSAR queueing system currently employed in RTBB.

meaning that if any KSAR's are found at the heads of the respective queues, they are in stage 1. Recall that the channel value of 1 corresponds to the write stage in which commands are written out to the KS's.

A single instance of the above class is made and the resulting object is called ksarq. The slot atomic-queue of this object, initially a null list, is set to the list of all the atomic KSAR's, the slot beam-queue to the list of all the distributed KSAR's that seek to invoke the GETBEAM KS, and, finally, the slot segment-queue to the list of all the distributed KSAR's that seek to invoke the GETASSIGNMENT KS. The track, spline, and merge queues are not used at this time, but are included in the definition for anticipated extension of the system.

The RTBB scheduler cycles through the three queues. It looks at the head KSAR in each queue and services it in a manner that depends on whether the KSAR is in the atomic-queue or in one of the other queues. The macro mcpoptart is the utility used in conjunction with the mapcar function for popping the head KSAR's off each non-empty KSAR queue. Before invoking mcpoptart, a list of the names of the non-empty queues is constructed by examining the various slots of the object ksarq. Via the mapping function mapcar, the macro mcpoptart is then applied to this list, the result being the head KSAR's in the various queues.

```
(defmacro mcpoptart (qname)          ;; qname bound to the name of KSAR queue
   `(let*
      ((com (fdefinition ,qname))    ;; com is reader of queue qname
       (comset (fdefinition (concatenate 'list (list 'setf ,qname))))
                                     ;; comset is writer of the same qname
       (xx (funcall com ,'ksarq))    ;; xx is all the KSAR's in the queue
       (y (cdr xx))                  ;; y is the tail of the queue
       (z (car xx)))                 ;; z is the head of the queue
   (funcall comset y ,'ksarq) z ))   ;; the tail remains and head is returned
```

For the atomic-queue the KS is threaded into the BB process before visiting the other queues. On the other hand, for the beam-queue and segment-queue the KS activation is executed in stages as described earlier so that the BB does not wait for the KS to finish executing.

The following is an example of bbksarq during execution.

```
#<bbksarq @ #x56c386> is an instance of class #<clos:standard-class bbksarq @ #x56d8fe>:
 merge-queue       nil
 spline-queue      nil
 track-queue       nil
 segment-queue     nil
 beam-queue        (#<ksar @ #x7b69ee>)
 atomic-queue      (#<ksar @ #x7b138e> #<ksar @ #x7b31ae>)
 mask              (1 -1 nil)
 number            3
```

Here the first item in the list that is the value of the slot mask represents the status of the atomic queue (in the case shown here, the value is set to 1, a meaningless number for atomic queues); the second item, pertaining to the beam queue, is set to -1 and means that the KS is ready to read; and, finally, the third item is set to nil since the segment queue is empty. The slot number is set to the total number of KSAR's held in the

queueing system and is automatically updated after every change by an after-method. As mentioned previously, the track, *spline* and merge queues are not being used at this time.

We will now comment on how the clock is used in the system. Each cycle of the scheduler consists of going through all the three queues. Each cycle of the scheduler is followed by an invocation of the planner, which maps all the previously unattended goals into either **KSAR's** or sub-goals. One cycle of the scheduler followed by one invocation of the planner constitutes one control cycle, and one control cycle constitutes one clock unit. When the BB process is first started, the main control loop first deposits a goal at the hit level; this goal for generating new hits is placed at the hit level every eighth clock unit. The scheduler now looks at all the queues, finding all of them empty except the beum-queue. The scheduler then examines the beam-queue, where it finds a KSAR generated by the planner from the hit-level goal. It services this KSAR according to its stage status value as stored in the *mask* slot of **ksarq.** Finally, the scheduler looks at the **segment-queue,** which it finds in the initialization stage. The process then repeats, as depicted in Fig. 8.

The main control loop that alternately runs the planner and the scheduler is shown below:

```
;;  This is the main control loop for driving RTBB

(defun cloop ()
  (catch 'cloop          ;; throw-catch combination used to break out at right time
    (do ()               ;; put into infinite loop
        (())

        (go-for-it )     ;; limits cycles, throws control back for loop breakout

        (clock-update) ;; update the clock variable and place a goal at the
                       ;; hit level every eighth clock unit.

        (planner)       ;; this maps the goals into KSAR's; it calls planner

        (scheduler)     ;; run the scheduler which cycles through the three
                  ;; KSAR queues held by the object ksarq

        )))
```

6. CONCLUSIONS

We hope we succeeded in conveying to the reader a sense of how Lisp **object-**oriented programming can be used for constructing a blackboard. In practically all the literature we have gone through, we have not encountered much discussion on the programming aspects of a blackboard. We hope this report has rectified that deficiency to **some** extent.

Evidently, our blackboard was meant more as a learning and training exercise. Therefore, our efforts should be judged less from the standpoint of whether we succeeded in designing a system that could actually be used for controlling a radar system and more from the standpoint of whether we succeeded in reducing the problem to manageable proportions, without trivializing it, and whether we succeeded in elucidating adequately the important details of our implementation.

RTBB is an evolving program and many aspects of it could be further refined. For example, one of future goals is to implement a separate queue for each KS; that would enhance a parallel or multi-processor implementation of RTBB. We would also like all the KSAR's to be of distributed type, which would make it necessary that we somehow "split" those KS's that are currently processed via atomic KSAR's into pre, write, read, and post phases. The RTBB rule-based planner is rudimentary at this point. A much more knowledgeable planner could be created to better focus the control.

As was mentioned in the previous section, a clock unit in RTBB consists of the scheduler taking one pass through all the queues and one invocation of the planner. This definition of a clock unit makes the programming easy, but it does make the exercise somewhat artificial. If the blackboard had to run by a real clock, provisions would have to be made to buffer the radar returns; the BB could then take the hits out of the buffer whenever it was allowed to attend to that task by the scheduler. Real time implementation of RTBB remains a future goal.

7. Acknowledgment

Seth Hutchinson's expertise in AI programming was invaluable and our many discussions with him about design decisions provided a sounding board which resulted in a much better product. We also owe thanks to Ann Silva, Computer Scientist at the Naval Undersea Warfare Center Division, for reading the manuscript carefully and providing us with valuable feedback.

8. REFERENCES

[1] K. M. Andress and A. C. Kak, "Evidence Accumulation and Flow of Control in a Hierarchical Spatial Reasoning System," *AI Magazine*, Vol. 9, No. 2, 1988, pp 75 - 94.

[2] K. M. Andress and A. C. Kak, *The PSEIKI report -- Version 3, Evidence Accumulation and Flow of Control in a Hierarchical Spatial Reasoning System,* Technical Report TR-EE 89-35, School of Electrical Engineering, Purdue University, November 1989.

[3] Y. Bar-Shalom and T. E. Fortmann, *Tracking* and *Data Association,* Academic Press, 1987.

[4] R. B. Cooper, *Introduction to Queueing Theory,* Second Edition, North Holland, 1981.

[5] D. D. Corkill, *A Framework for Organizational Self-Design in Distributed Problem Solving Networks,* PhD Thesis, U of Mass, Feb 1983.

[6] D. D. Corkill, et al., *GBB:* A *Generic Blackboard Development System,* AAAI Conference, 1986, Philadelphia.

[7] I. D. Craig, *The Ariadne-1 Blackboard System,* The Computer Journal, Vol. 29, No. 3, 1986, pp. 235-240.

[8] R. Englemore and T. Morgen, Eds., *Blackboard System,* Addison-Wesley, 1988.

[9] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-I1 Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Computing Surveys,* pp. 213-253, 1980.

[10] I. Faux and M. Pratt, *Computational Geometry for Design and Manufacture,* Ellis Horwood Limited, 1979.

[11] G. Forsythe, M. Malcolm and C. Moler, *Computer Methods for Mathematical Computations,* Prentice-Hall, Inc., 1977.

[12] *Allego CL User Guide, Version 4.1 beta* Franz Inc., Aug 1991.

[13] Gabriel, R. P, White, J.L., and Bobrow, D. G., *CLOS: Integrating Object-Oriented and Functional Programming,* Communications of the ACM, Vol 34, No 9, Sept. 91., pp 29-38.

[14] A. R. Hanson and E. M. Riseman, *VISIONS: A Computer System for Interpreting Scenes.* Computer Vision Systems, Hanson and Riseman eds., Academic Press, NY, 1978.

[15] B. Hayes-Roth, "A Blackboard Architecture for Control," *Artificial Intelligence,* 26, 1985, pp. 251-321.

[16] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research,* Holden-Day,1980, Chapter 18.

[17] S. Hutchinson, Personal Communication, Summer 87.

[18] Keene, Sonya E., *Object-Oriented Programming in Common Lisp,* A Programmer's Guide to CLOS, Addison Wesley, 1989.

[19] P. R. Kersten and A. C. Kak, *A Tutorial on Using Lisp Object-Oriented Programming for Blackboard* Computation (Solving the Radar Tracking Problem), *International Journal of Intelligent Systems,* John-Wiley & Sons, Inc., Vol. 8, 1993.

[20] Lawless, J.A. and Miller, M.M. *Understanding CLOS The Common Lisp Object System Digital* Press, 1991.

[21] V. Lesser and R. Fennell, "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II," *IEEE Trans. on Computers,* Vol. C-26, No. 2, Feb. 77, pp 98-143.

[22] V. Lesser and D. Corkill, "Functionally Accurate, Cooperative, Distributed Systems," *IEEE Transactions on Systems, Man, & Cybernetics,* Vol SMC-11, No. 1, Jan. 1981., pp81-96.

[23] V. Lesser and E. Durfee, *Incremental Planning in a Blackboard-based Problem Solver,* AAAI - 86, Philadelphia.

[24] M. Nagao and T. Matsuyama, *A Structural Analysis of Complex Aerial Photographs,* Plenum Press, New York, 1980.

[25] H. P. Nii et al., *Signal-to-Symbol Transformation: HASPISIAP Case Study,* The AI Magazine, Spring 1982, pp 23 - 35.

[26] H. P. Nii, "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures," *AI Magazine,* Summer, 1986, pp 38-53.

[27] H. P. Nii, "Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective," *AI Magazine,* August 1986, pp 82-106.

[28] N. J. Nilsson, *Principles of Artificial Intelligence,* Tioga Publishing Co, Palo Alto, CA., 1980.

[29] Sinclair, K.H., and Moon, D. A., *The Philosophy of LISP,* Communications of the ACM, Vol 34, No 9, Sept. 91., pp 49-57.

[30] M. A. Williams, *Distributed, Cooperating Expert Systems for Signal Understanding,* In *Proceedings of Seminar on AI Applications to Battlefield,* 3.4-1 to 3.4-6, 1985.

[31] P. H. Winston and B. K. P. Horn, LISP, Second Edition, Addison-Wesley, 1984.

[32] R. Worden, *Blackboard Systems,* in *Computer Assisted Decision Making* Edited by G. Mitra, North Holland, 1986, pp. 95-106.

APPENDIX A

We will now present four examples to illustrate the working of RTBB. These examples, at increasing levels of difficulty, start with the case of a single track formed by a single craft in the first example; progress to two stable tracks formed by three separate craft in the second example; further progress to the case where initially three craft form a single track, but eventually form only two tracks as one craft breaks away; and, finally, deal with the problem of fading tracks in the last example.

Example 1

In this example, there is a single craft. Most of the class instances are expanded out to illustrate the &tails involved at each step.

The first example illustrates the BB solution formation for a single trajectory. The data which drives the trajectory is based on Bezier's curve. For this curve the trapezoid which defines the space curve is given by the four points indicated in Fig. 9. Note that the origin is one of the points so the trajectory will go through the origin. The origin in these examples is a special point, in the sense that it represent not only the origin of the coordinate system but also the center of the air space around a hypothetical airport. The starting point of the single trajectory is (100,0,0).

The data nodes on the hit level (the bnodes) are initiated by periodically placing a goal node on the hit level of the goal panel. The goal node causes the generation of a KSAR which causes the hit generation KS GETBEAM to fire. Recall from the BB Control section that this KSAR is of distributed type although no *preboot* function is necessary since the function call is so simple. The KSAR is:

```
#<ksar @ #x7c190e>
is an instance of class #<clos:standard-class ksar @ #x782556>:
 prelyst       <unbound>
 preboot       <unbound>
 anslyst       nil
 arglyst       nil
 command       fire
 messenger     #<messenger @ #x79734e>
 channel       1
 nodeptr       <unbound>
 postboot      (getbeam)
 context       none
 cycle         1
 ksar-id       newhit
 priority      2
```

The *postboot* is the *C* coded GETBEAM KS and its only command is a trigger "fire". The KSAR causes the formation of a data node to be placed on the hit level of the data panel.

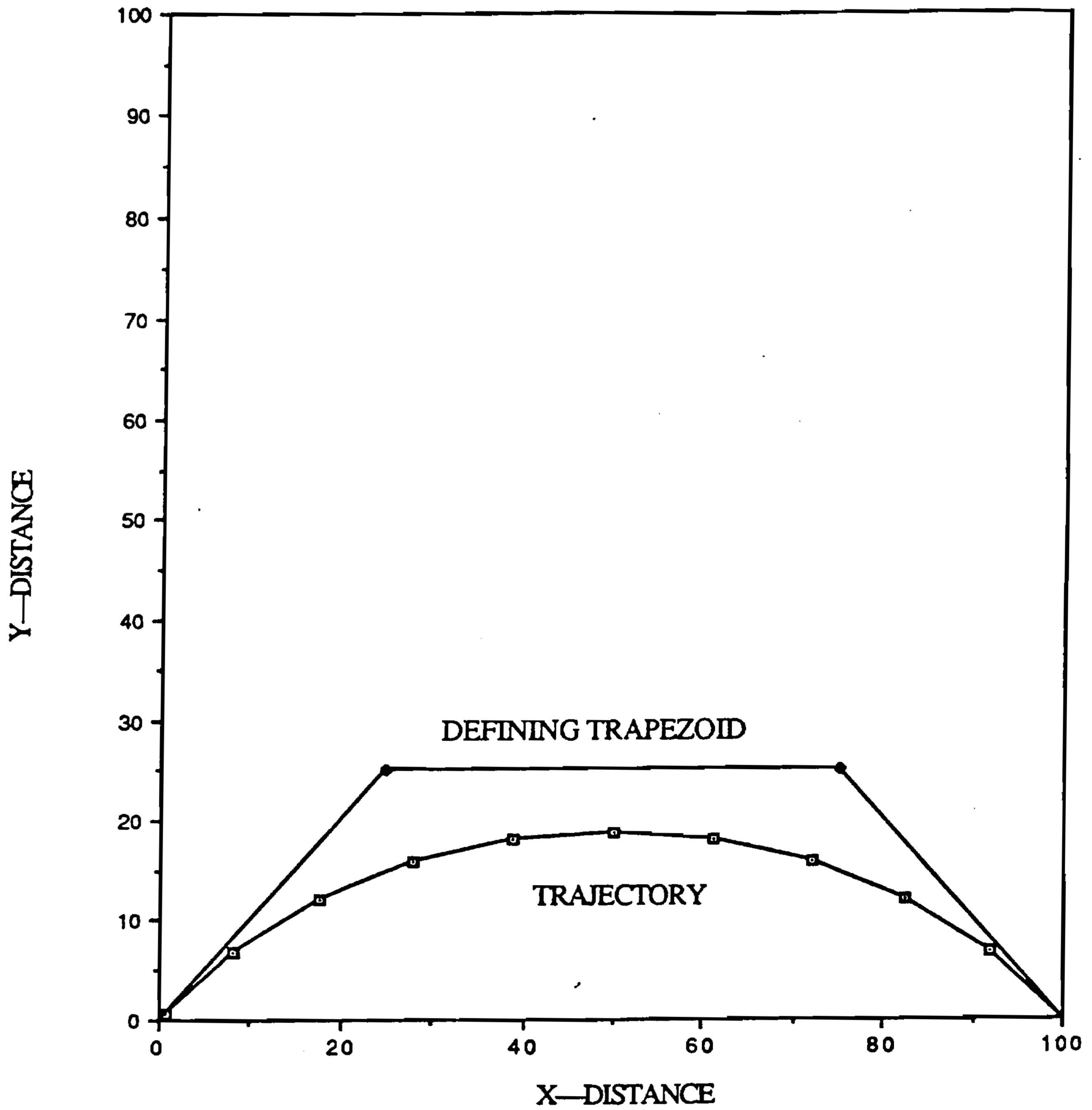


FIGURE 9. Single trajectory generated via Bezier's curve with every tenth point shown. Defining trapezoid shown with curve.

The data node looks as follows:

```
#<bnode @ #x64feae>
is an instance of class #<clos:standard-class bnode @ #x622116>:
 event-time    0
 level         hit
 number        1
 coord         ((100.0 0.0 0.0))
```

Note the return count is given by number and it occurs at event-time 0 at the coordinates coord at level hit.

The placement of this hit node on the data panel causes the placement of the following goal node on the segment level of the goal panel:

```
#<bbgoal @ #x65c736>
is an instance of class #<clos:standard-class bbgoal @ #x6220ce>:
 purpose                      change
 event-time                   0
 initiating-data-level        hit
 source                       #<bnode @ #x64feae>
 duration                     one-shot
 ksarptr                      <unbound>
 snode                        nil
 threat                       nil
 number                       1
 coord                        ((100.0 0.0 0.0))
```

This goal represents the desire to match this data to the nearest segments. The duration of the goal node is one-shot, **i.e.,** the ruler-based planner gets only one pass to satisfy it, and after that the goal node is removed from the goal panel. The slot source contains a pointer to the data node responsible for the creation of the goal node by the dismbuted **moni**tor.

The rule-based planner uses the segment-level goal node to generate a KSAR for matching the hit data to a nearest segment, if there is one. Otherwise, it creates a new segment. The KSAR for this KS is dismbuted with a separate preboot function which forms the arguments for the knowledge source. The ***postboot*** function posts the results of the command to the KS. The command slot holds the main KS call function. Again, the ***ksar-id*** generally describes the driving activity, in this case segment formation. The generated KSAR is:

```
#<ksar @ #x6751fe>
is an instance of class #<clos:standard-class ksar @ #x6220e6>:
 prelyst      nil
 preboot      (pre-assign-hits)
 anslyst      nil
 arglyst      nil
 command      getassignment
 messenger    #<messenger @ #x62559e>
 channel      2
 nodeptr      #<bbgoal @ #x65c736>
 postboot     (post-assign-hits)
 context      ((event-time nil) (number #<bbgoal @ #x65c736>) (coord 0))
 cycle        5
 ksar-id      segment
 priority     1
```

The **GETASSIGNMENT** KS which is fired by this KSAR results in the creation of the following segment-level data node:

```
#<snode @ #x6842ee>
is an instance of class #<clos:standard-class snode @ #x62209e>:
 event-time     (0)
 level          segment
 coord          ((100.0 0.0 0.0))
 number         1
 cpa            nil
 linear         nil
 tnode          nil
 threat         nil
```

Most of the slots are initially *nil* since the segment is not long enough yet; however, the slots do get filled in at a later time when the segment becomes part of a track. In fact, at a later time the snode looks as follows:

```
#<snode @ #x61f616>
is an instance of class #<clos:standard-class snode @ #x56dd36>:
 event-time     (1 0)
 level          segment
 coord          ((99.24255 0.7425 0.0) (100.0 0.0 0.0))
 number         2
 cpa            (49.003643 49.990078 0.0)
 linear         ((99.24255 0.7425 0.0) (-0.7574463 0.7425 0.0))
 tnode          nil
 threat         nil
```

This segment data node, via an after-method from the distributed **monitor,** creates the following track goal node which represents the desire to form a track from the segment:

```
#<bbgoal @ #x68f526>
is an instance of class #<clos:standard-class bbgoal @ #x56dd56>:
 purpose                  change
 event-time               (1 0)
 initiating-data-level    segment
 source                   #<snode @ #x61f616>
 duration                 one-shot
 ksarptr                  <unbound>
 snode                    nil
 threat                   nil
 number                   2
 coord                    ((99.24255 0.7425 0.0) (100.0 0.0 0.0))
```

Note here that a data segment node was the ***source*** of this node and the ***coord*** is the set of two consecutive coordinates which are used to form the segment and the track. The ***event-time*** slot stores the sequence of times which support the formation of the track.

Again, the rule-based creates **from** this track goal node the following KSAR, whose purpose is to form a track.

```
#<ksar @ #x780486>
is an instance of class #<clos:standard-class ksar @ #x567ade>:
 prelyst      <unbound>
 preboot      <unbound>
 anslyst      <unbound>
 arglyst      <unbound>
 command      assign-tracks
 messenger    <unbound>
 channel      1
 nodeptr      #<snode @ #x780b3e>
 postboot     (assign-tracks)
 context      ((event-time nil) (number #<snode @ #x780b3e>) (coord (1 0)))
 cycle        17
 ksar-id      track
 priority     0
```

Note that this KSAR is an atomic KSAR unlike the previous KSAR that assigned hits to segments. The KS places the following track node on the data panel at the track level:

```
#<tnode @ #x7953ce>
is an instance of class #<clos:standard-class tnode @ #x56dd46>:
 event-time       (1)
 level            track
 last-coord       (99.24255 0.7425 0.0)
 last-velocity    (-0.7574463 0.7425 0.0)
 snode            (#<snode @ #x780b3e>)
 threat           nil
 cpa-bracket      ((43.97975 54.027534) (45.065323 54.91484))
 check            nil
 checklyst        nil
```

The data node slot ***event-time*** contains only the current time. Slots ***last-coord*** and ***last-velocity*** correspondingly store the position and the velocity. The ***snode*** contains a list of pointers to the segments that form the logical support for the tracks and the members of the formation. The confidence region which is called ***cpa-bracket*** causes the ***threat*** slot to ***be*** flagged "t" if it includes the origin. For the node above, the track does not appear as a threat -- yet!

The above nodes are the initial formation of the solution track. The solution track structure is a tree with the base of the tree being the track node and the branches being the segments nodes. In this example there is only one branch, so the solution tree is very simple. The track coordinate history contained in the tree expands as the track grows in length. As an example, consider a segment node at a still later time

```
#<snode @ #x583a8e>
is an instance of class #<clos:standard-class snode @ #x56dd36>:
 event-time     ( 4 3 2 1 0 )
 level          segment
 coord          ((96.8832 2.88 0.0) (97.683846 2.1825 0.0)
                 (98.4704 1.47 0.0) (99.24255 0.7425 0.0) (100.0 0.0 0.0))
 number         5
 cpa            (43.2293 49.62189 0.0)
 linear         ((96.8832 2.88 0.0) (-0.8006439 0.6975002 0.0))
 tnode          #<tnode @ #x77e006>
 threat         nil
```

The ***cpa*** has been calculated and the ***linear*** slot contains the current position and velocity so that the segment can be extended forward. The ***threat*** has been evaluated and the track node which this segment node supports is stored in the slot ***tnode.***

After the segment information has been extended to more than fourteen points, the list is truncated by dropping the oldest hits. This is accomplished with an after-method so that after say 20 time units, the snode takes on the following appearance:

```
#<snode @ #x583a8e>
is an instance of class #<clos:standard-class snode @ #x56dd36>:
 event-time     (20 19 18 17 16 15 14 13 12 11 10 9 8 7)
 level          segment
 coord          ((82.4 12.0 0.0) (83.385445 11.5425 0.0) (84.3616 11.07 0.0)
                 (85.328156 10.5825 0.0) (86.2848 10.08 0.0)
                 (87.23125 9.5625 0.0) (88.1672 9.03 0.0)
                 (89.092354 8.4825 0.0) (90.0064 7.92 0.0)
                 (90.90905 7.3425 0.0) (91.8 6.75 0.0)
                 (92.678955 6.1425 0.0) (93.5456 5.52 0.0))
 number         13
 cpa            (19.194233 41.343826 0.0)
 linear         ((82.4 12.0 0.0) (-0.9854431 0.45750046 0.0))
 tnode          #<tnode @ #x5851d6>
 threat         nil
```

In the above snode, the maximum lengths of the ***coord*** and ***event-time*** slots are now only of length fourteen as fixed by a global variable. The truncation length may be set to any fixed value but this threshold is not totally independent of the other parameters. For example, a track may only be generated when the segment length exceeds another fixed parameter. Certainly the truncation length must exceed this minimum length needed to initiate a track, otherwise data will truncated as soon as it is placed on the segment level.

The general sequence of KS calls is outlined in Fig. 10. Here the order of KS calls is numbered to push data nodes to higher levels of abstraction. The order is not exact since several data nodes must be advanced to form a track -- but the general order required to push data through to support a track solution is outlined. The first KS, corresponding to the transition 1 marked in the figure, is the hit generation KS (GET-BEAM); the second KS, corresponding to the transition marked 2, is the GETASSIG-MENT KS; and the third KS is the track formation KS (GETTRACK). Methods from the distributed monitor generate the goal nodes from the data nodes. The simple construction illustrated is essentially data driven with goal nodes being isomorphically mapped to KS's. This example illustrates the operation of a goal driven BB emulating a data driven BB.

Example 2

In this example there are three separate craft being observed. Three craft generate returns, but only two tracks solutions are formed. This example illustrates the track formation process, especially the grouping of segments into tracks.

Fig. 11 shows a plot of the three trajectories. Two of these trajectories are very close and logically form a track. The other trajectory forms a separate track by itself. The plots of Fig. 11 are mirrored in the data structures on the blackboard panels. Since a tree represents a track, one tree will represent two trajectories and the other will represent a single trajectory.

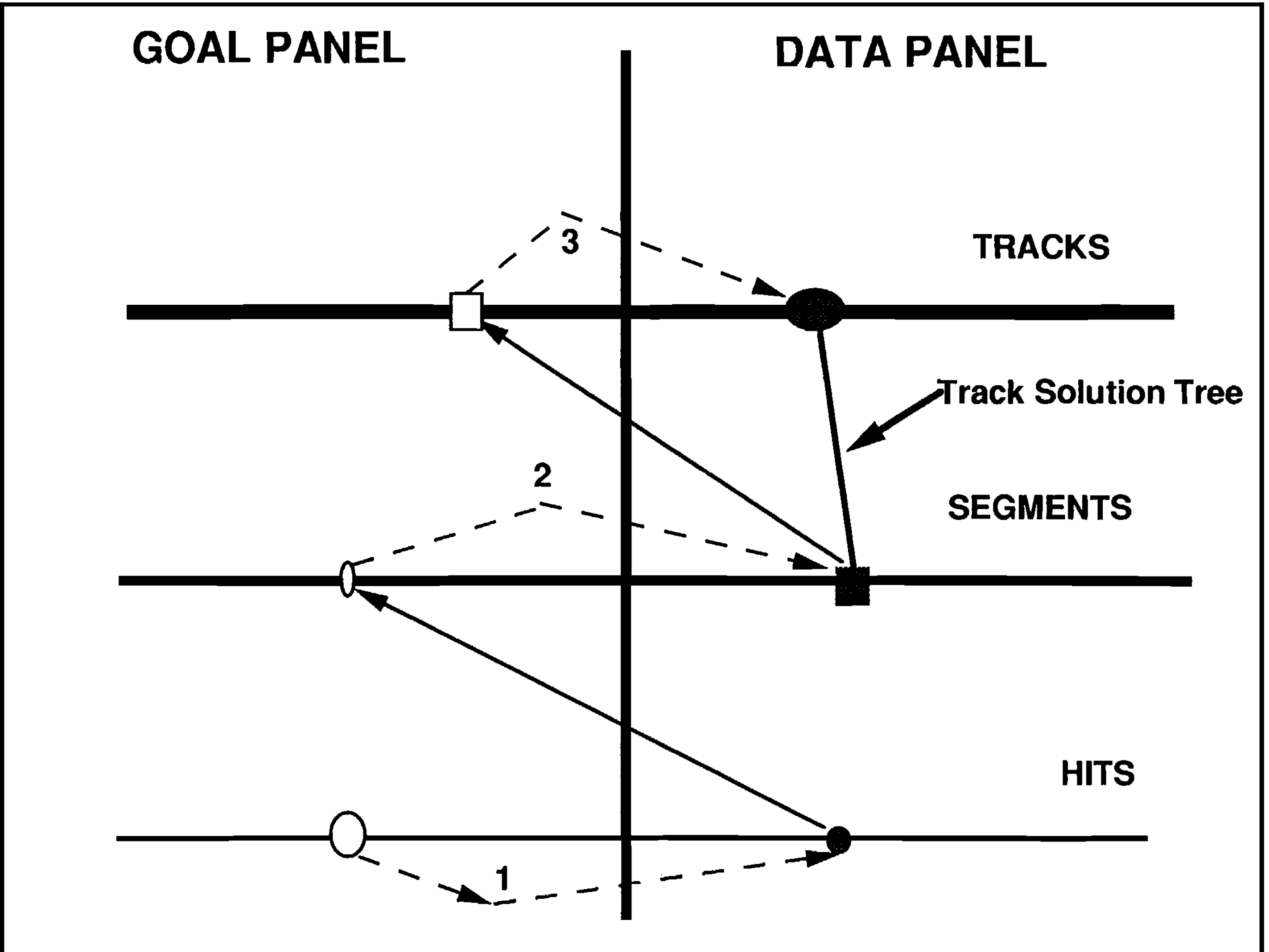


Figure 10. The track formation for a single trajectory is outlined here.

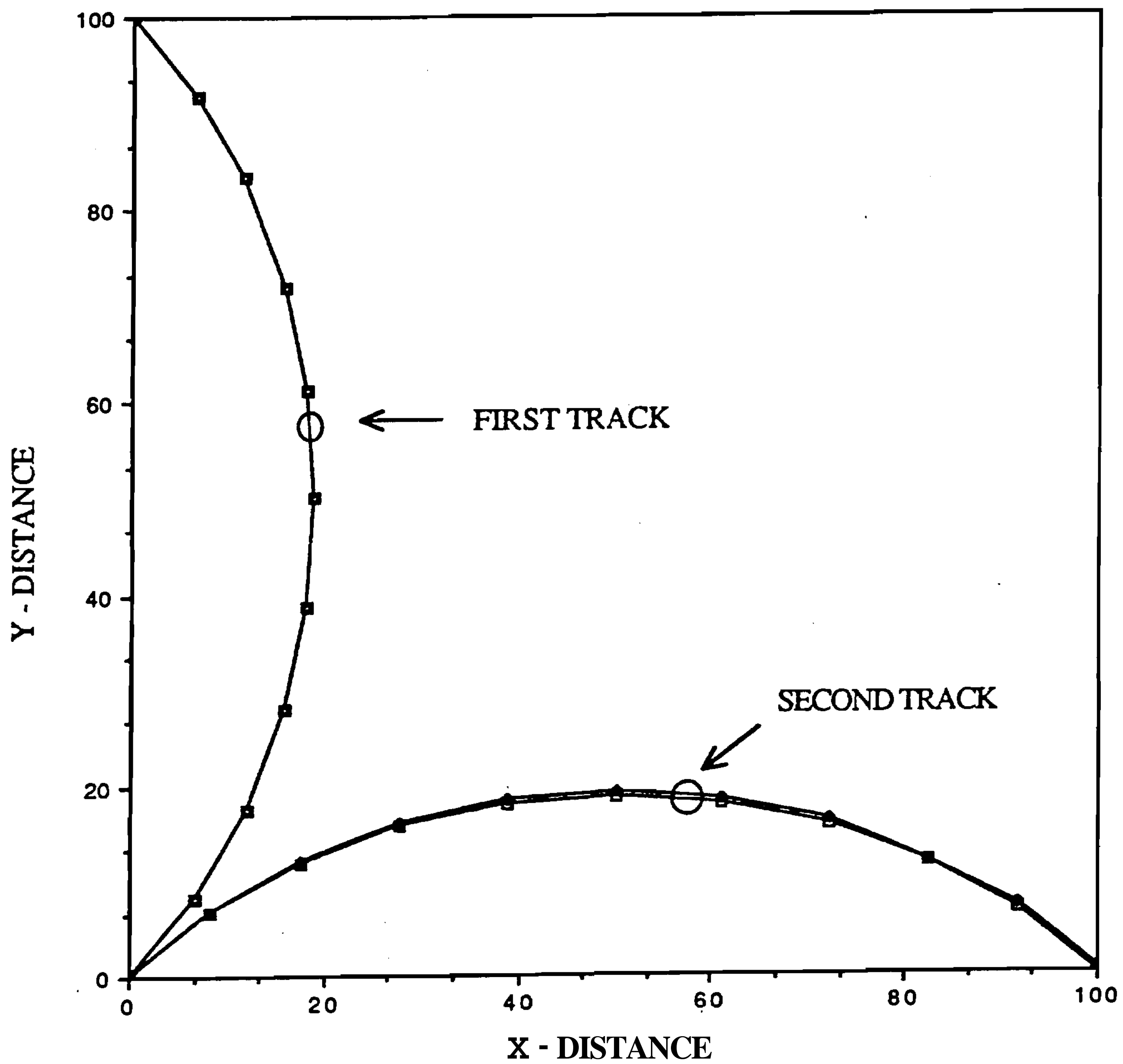


FIGURE 11. Trajectories for three separate craft with two tracks indicated.

Fig. 12 graphically traces the formation of the solution trees on the blackboard. Notice the similarity with the formation of a single track. The overall crisscrossing of the solution path on the blackboard panels from lower levels of abstraction to higher levels is due to the data driven nature of the problem. The presence of the three distinct trajectories in the data causes the formation of three distinct nodes at the track level of the goal panel. Each goal represents the desire to use the segment data node as support for a track node. By support is meant the segment node supports the hypothesis that the track node should contain that segment as part of the group that makes up the track.

Lets looks at some of the data nodes on the BB after the **tracks** are established. The two tracks are represented by the following two modes:

```
#<tnode @ #x584f46>
is an instance of class #<clos:standard-class tnode @ #x56dc76>:
 event-time        (5)
 level             track
 last-coord        (96.07798 3.905423 0.0)
 last-velocity     (-0.8144531 0.6824701 0.0)
 snode             (#<snode @ #x5838a6> #<snode @ #x583886>)
 threat            nil
 cpa-bracket       ((36.18531 54.252422) (44.9477 55.14532))
 check             nil
 checklyst         nil

#<tnode @ #x584f56>
is an instance of class #<clos:standard-class tnode @ #x56dc76>:
 event-time        (5)
 level             track
 last-coord        (3.6225002 96.125755 0.0)
 last-velocity     (0.6824999 -0.8144531 0.0)
 snode             (#<snode @ #x583896>)
 threat            nil
 cpa-bracket       ((44.80412 54.91484) (35.91684 54.027534))
 check             nil
 checklyst         nil
```

Note **#<tnode @ #x584f46>** is the second track in Figs. 11 and 12 with two supporting segment nodes. The slot *snode* contains segment nodes instances that form the branches of the solution tree and the logical support for the track hypothesis. The other track node **#<tnode @ #x584f56>** has only one pointer which simply means only one branch and one supporting segment node. Neither track is presently a threat to the origin, although the trajectory plot indicates that this will not be true in the future.

The snodes contain parent pointers to the which track they support. The snodes are:

```
#<snode @ #x583886>
is an instance of class #<clos:standard-class snode @ #x56dc66>:
 event-time    (5 4 3 2 1 0)
 level         segment
 coord         ((96.06875 4.062438 0.0) (96.8832 3.379968 0.0)
           (97.683846 2.682486 0.0) (98.4704 1.969996 0.0)
           (99.24255 1.242499 0.0) (100.0 0.5 0.0))
 number        6
 cpa           (41.62926 49.679947 0.0)
 linear        ((96.06875 4.062438 0.0) (-0.8144531 0.6824701 0.0))
 tnode         #<tnode @ #x584f46>
 threat        nil
```

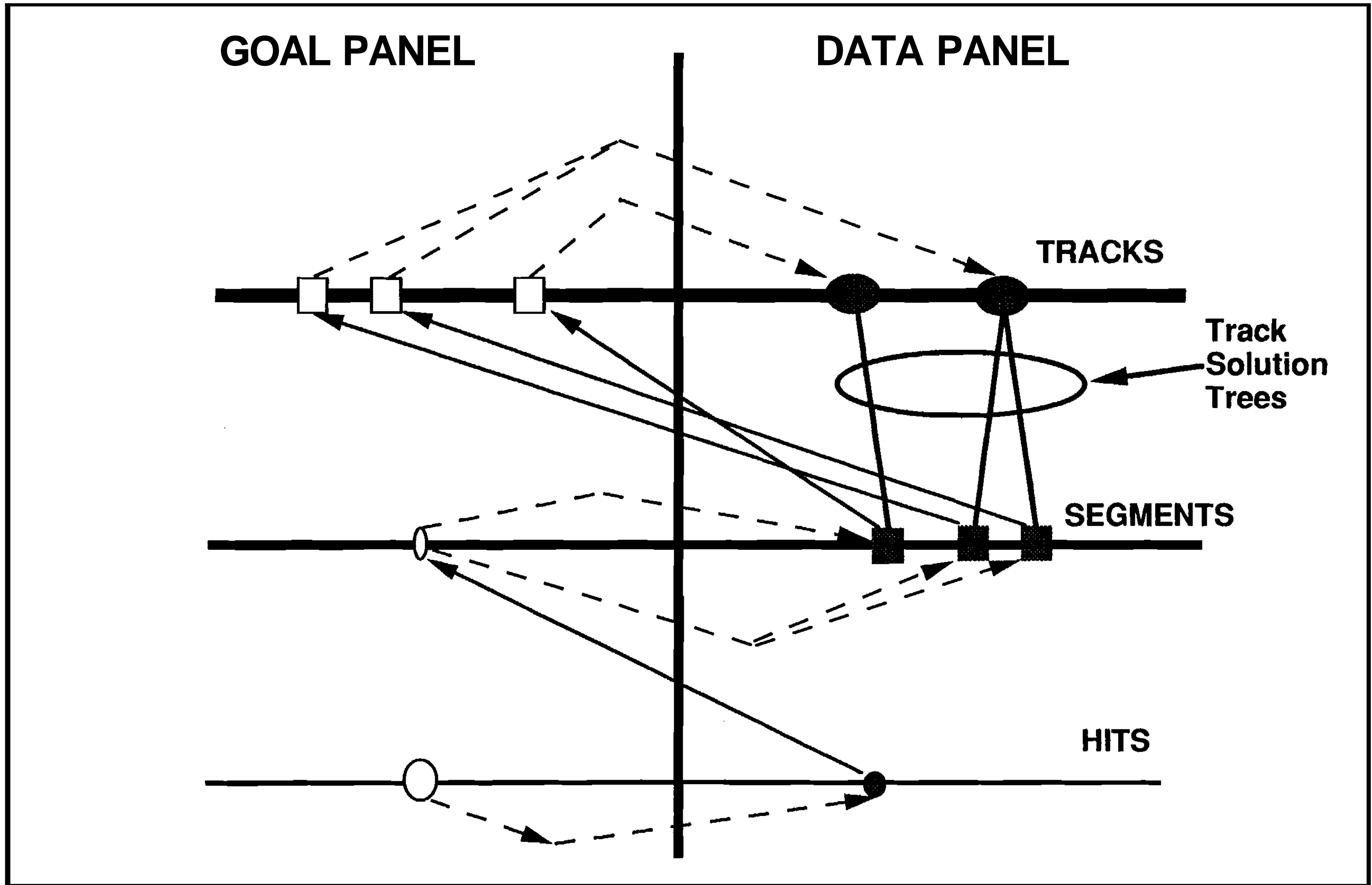


Figure 12. The nodes on the BB for the 3-trajectory, 2-track example. Note the two solution trees.

```
#<snode @ #x583896>
is an instance of class #<clos:standard-class snode @ #x56dc66>:
 event-time     (5 4 3 2 1 0)
 level          segment
 coord          ((3.5625 96.06875 0.0) (2.88 96.8832 0.0)
            (2.1825 97.683846 0.0) (1.47 98.4704 0.0)
            (0.7425 99.24255 0.0) (0.0 100.0 0.0))
 number         6
 cpa            (49.38652 41.385197 0.0)
 linear         ((3.5625 96.06875 0.0) (0.6824999 -0.8144531 0.0))
 tnode          #<tnode @ #x584f56>
 threat         nil

#<snode @ #x5838a6>
is an instance of class #<clos:standard-class snode @ #x56dc66>:
 event-time     (5 4 3 2 1 0)
 level          segment
 coord          ((96.06875 3.5625 0.0) (96.8832 2.88 0.0)
            (97.683846 2.1825 0.0) (98.4704 1.47 0.0)
            (99.24255 0.7425 0.0) (100.0 0.0 0.0))
 number         6
 cpa            (41.385197 49.38652 0.0)
 linear         ((96.06875 3.5625 0.0) (-0.8144531 0.6824999 0.0))
 tnode          #<tnode @ #x584f46>
 threat         nil
```

At a much later time both tracks are classified as threats. In fact at time 41 the track nodes look as follows:

```
#<tnode @ #x584f46>
is an instance of class #<clos:standard-class tnode @ #x56dc76>:
 event-time        (41)
 level             track
 last-coord        (60.09016 18.42884 0.0)
 last-velocity     (-1.1114502 0.14003944 0.0)
 snode             (#<snode @ #x5838a6> #<snode @ #x583886>)
 threat            t
 cpa-bracket       ((-2.437229 54.252422) (25.053728 55.14532))
 check             36
 checklyst         nil

#<tnode @ #x584f56>
is an instance of class #<clos:standard-class tnode @ #x56dc76>:
 event-time        (40)
 level             track
 last-coord        (18.0 61.2 0.0)
 last-velocity     (0.15749931 -1.108448 0.0)
 snode             (#<snode @ #x583896>)
 threat            t
 cpa-bracket       ((25.350838 54.91484) (-2.030042 54.027534))
 check             nil
 checklyst         nil
```

By now both tracks represent threats to the origin and so the threat slot holds the flag for **true.** Note that the confidence region contained in cpa-bracket has one coordinate which straddles the origin. Although this threat detection scheme is arbitrary and probably not a sharp criterion, it does illustrate the detection via the rule-based planner.

Example 3

In this example there are three separate craft being observed. Initially these three craft form one track. Subsequently, one craft breaks away from the established track. This example illustrates the detection of the break away and the subgoaling needed to establish two tracks.

Fig. 13 shows three trajectories for the three different craft generating radar returns. All of these trajectories are initially very close and form a single track. However, as the track evolves in time, one of the segments supporting the track formation obviously departs from the track itself. By departs is meant that if the track grouping were to be reformed, two tracks instead of one track would be formed. A backchaining algorithm fitting splines to tracks is designed to detect if the grouping of segments into a track is still logically valid.

One way to solve the problem of regrouping the tracks is simply to dissolve the track node and keep the segment nodes on the data level after removing their parent pointers to a track. The track formation algorithm would then pick up these "uncommitted" segments and regroup the segments into tracks. This solution is acceptable but not as desirable as maintaining the track history and forming a new track from a subset of the segments of the original track. This is implemented by subgoaling - an important technique which allows knowledge sources to become more specialized in their competence and makes it easier to incorporate more complex relationships between goals.

The nodes or solution tree should reflect the history of the trajectories. Indeed, Fig. 14 shows the correspondences between the physical trajectories and data structures which represent these trajectories. First a tree will form on the blackboard which has only one root - i.e. one trajectory with three branches representing the three distinct craft. Once the track is established and determined to be a threat, the track grouping will be checked via the spline KS. When the track grouping is not verified by the spline KS, subgoals for each segment are created and placed on the goal segment level. Each goal represents the desire to determine if that segment is in the same formation as the average track representing the root of the track. If the segment does not satisfy the grouping criterion against the track, it is spun off as a segment with no parent pointers. This means the BB will establish this segment as a separate track. The following paragraphs will show the state of the nodes in this sequence of events.

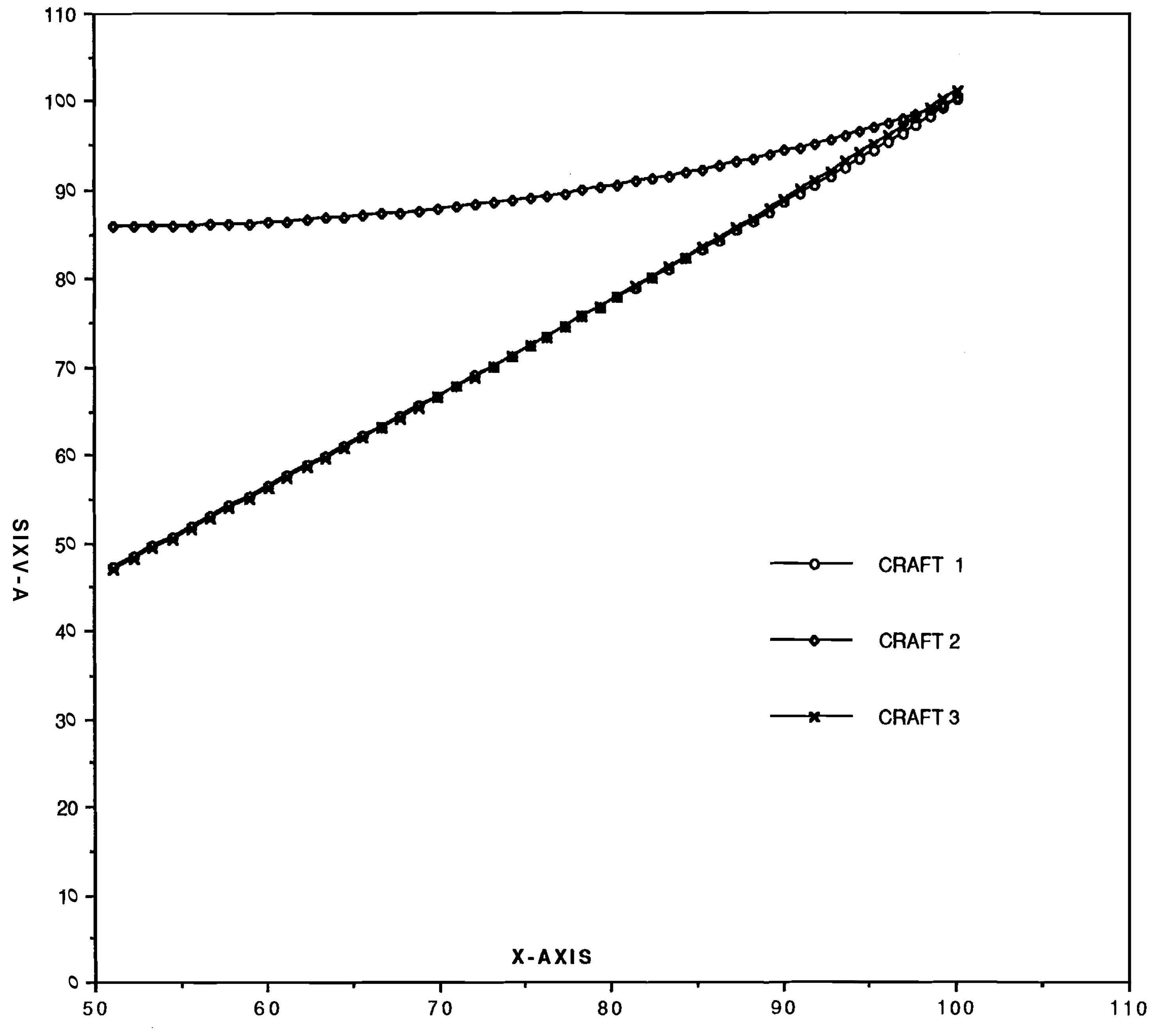


Figure 13. Shows breakaway craft 2 from the formation

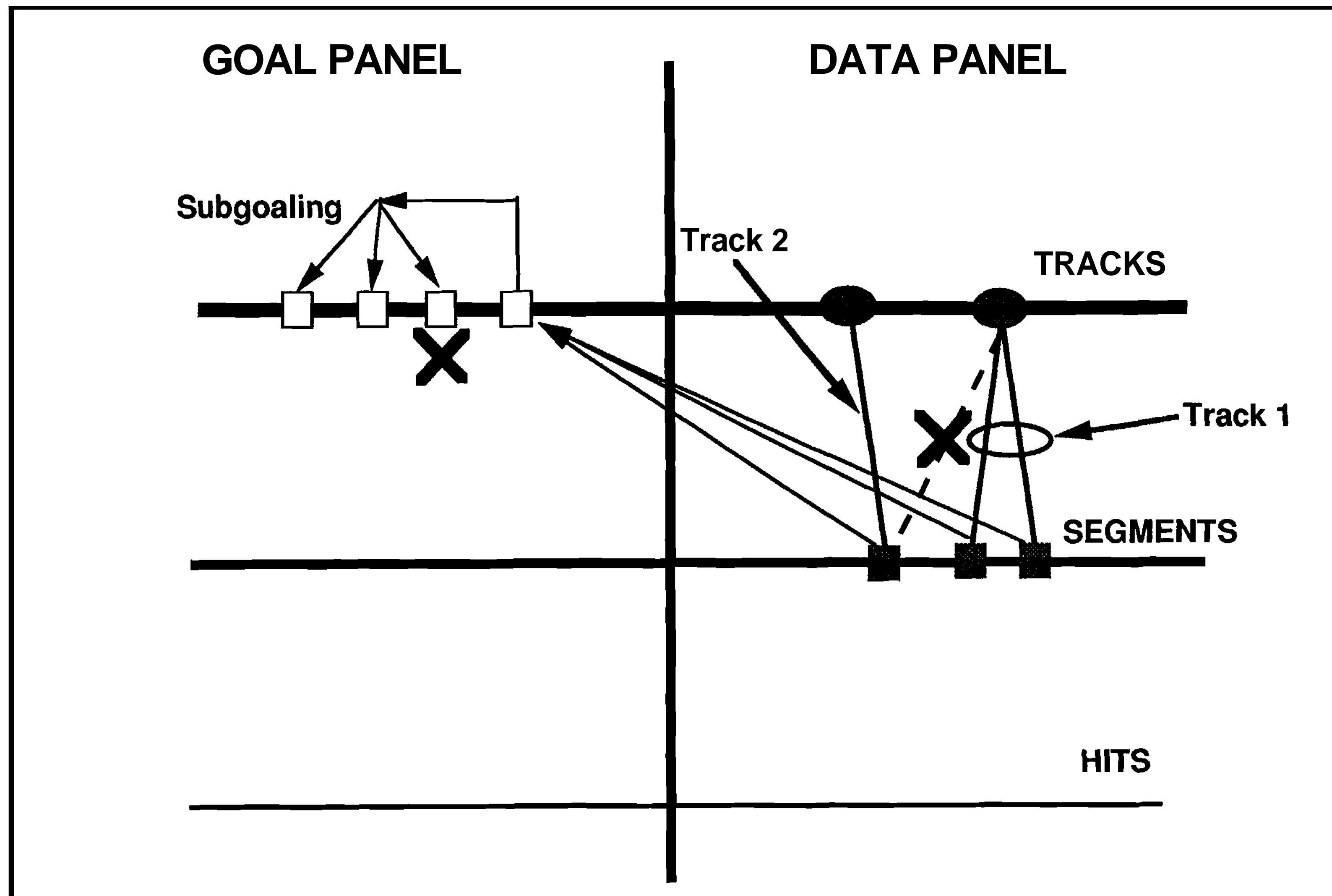


Figure 14. This example illustrates the cancellation of the segment node support of the track1 hypothesis. Subgoaling triggered by the failure of the spline test is illustrated in the goal panel.

Initially, the track node formed from the three segments is:

```
#<tnode @ #x7d55ae>
is an instance of class #<clos:standard-class tnode @ #x56daa6>:
 event-time        (1)
 level             track
 last-coord        (99.34056 99.531265 0.0)
 last-velocity     (-0.65943915 -0.8020681 0.0)
 snode             (#<snode @ #x78104e> #<snode @ #x781036> #<snode @ #x78101e>)
 threat            t
 cpa-bracket       ((-1.523449 21.468126) (-21.077364 3.9739904))
 check             nil
 checklyst         nil
```

Observe that there are three snodes or branches supporting this track. The three segments supporting the trajectory are given below. Note that the track node pointers in these nodes are really the parent pointers or the edges of the graph pointing to the root of the tree which represents the track.

```
#<snode @ #x78101e>
is an instance of class #<clos:standard-class snode @ #x56da96>:
 event-time     (1 0)
 level          segment
 coord          ((99.24255 100.03494 0.0) (100.0 101.0 0.0))
 number         2
 cpa            (12.826523 -10.067154 0.0)
 linear         ((99.24255 100.03494 0.0) (-0.7574463 -0.9650574 0.0))
 tnode          #<tnode @ #x7d55ae>
 threat         nil

#<snode @ #x781036>
is an instance of class #<clos:standard-class snode @ #x56da96>:
 event-time     (1 0)
 level          segment
 cwrd           ((99.24255 99.09405 0.0) (100.0 100.0 0.0))
 number         2
 cpa            (9.648041 -8.066505 0.0)
 linear         ((99.24255 99.09405 0.0) (-0.7574463 -0.90595245 0.0))
 tnode          #<tnode @ #x7d55ae>
 threat         nil

#<snode @ #x78104e>
is an instance of class #<clos:standard-class snode @ #x56da96>:
 event-time     (1 0)
 level          segment
 coord          ((99.536575 99.464806 0.0) (100.0 100.0 0.0))
 number         2
 cpa            (7.663826 -6.636101 0.0)
 linear         ((99.536575 99.464806 0.0) (-0.46342468 -0.5351944 0.0))
 tnode          #<tnode @ #x7d55ae>
 threat         nil
```

This solution tree structure is the initial state of the track prior to the discovery that the trajectory is a threat and prior to the departure of one of the craft from the formation.

Almost immediately, at time 1, the track is determined to be a threat to the origin and the spline KS (GETSPLINE) will now begin to check to see if the composition of the track still makes sense. The following track node illustrates the track node state just after it has been determined it is a threat.

```
#<tnode @ #x7d55ae>
is an instance of class #<clos:standard-class tnode @ #x56daa6>:
 event-time        (1)
 level             track
 last-coord        (99.34056 99.531265 0.0)
 last-velocity     (-0.65943915 -0.8020681 0.0)
 snode             (#<snode @ #x78104e> #<snode @ #x781036> #<snode @ #x78101e>)
 threat            t
 cpa-bracket       ((-1.523449 21.468126) (-21.077364 3.9739904))
 check             nil
 checklyst         nil
```

After the spline test detects the break away of a track, it marks the track node check variable as failed. A failed spline test automatically disables further spline tests for that track until a track verification KS can be run. The rule-base planner will detect a failed track in the goal blackboard, and then generate a subgoal for each segment which supports the track. Each goal expresses the desire to re-evaluate the track formation grouping criterion of each segment against the averaged track. The following are the subgoals generated by the rule base.

```
#<bbgoal @ #x7ccb7e>
is an instance of class #<clos:standard-class bbgoal @ #x56dab6>:
 purpose                 verify-track
 event-time              (4)
 initiating-data-level   track
 source                  #<tnode @ #x584e96>
 duration                one-shot
 ksarptr                 <unbound>
 snode                   #<snode @ #x583896>
 threat                  nil
 number                  <unbound>
 coord                   ((96.767204 97.039505 0.0) (-0.8006439 -0.99399567 0.0))

#<bbgoal @ #x7cc18e>
is an instance of class #<clos:standard-class bbgoal @ #x56dab6>:
 purpose                 verify-track
 event-time              (4)
 initiating-data-level   track
 source                  #<tnode @ #x584e96>
 duration                one-shot
 ksarptr                 <unbound>
 snode                   #<snode @ #x5838a6>
 threat                  nil
 number                  <unbound>
 coord                   ((96.767204 97.039505 0.0) (-0.8006439 -0.99399567 0.0))

#<bbgoal @ #x7cb5e6>
is an instance of class #<clos:standard-class bbgoal @ #x56dab6>:
 purpose                 verify-track
 event-time              (4)
 initiating-data-level   track
 source                  #<tnode @ #x584e96>
 duration                one-shot
 ksarptr                 <unbound>
 snode                   #<snode @ #x58323e>
 threat                  nil
 number                  <unbound>
 coord                   ((96.767204 97.039505 0.0) (-0.8006439 -0.99399567 0.0))
```

Each of these subgoals points to the parent track as the source and the supporting segment node as the snode. The KSAR generated from each of these subgoals will activate

the **VERIFY** KS. This KS is part of the blackboard process - **i.e.** it is not spun off as a separate process. If the segment is re-verified to be in the same track grouping , then nothing is done, except to record the verification result by removing the nude from the ***checklyst.*** If not, then the KS does three things. First KS removes the segment pointers in the track node - **i.e.** the pointer to this sibling or branch of the tree. Then it removes the parent pointer in the segment node or the pointer to the root of the tree representing the track. And lastly, it removes the pointer from the ***checklyst*** from the track node.

The snode that becomes orphaned by the **VERIFY** KS is the following segment node.

```
#<snode @ #x583896>
is an instance of class #<clos:standard-class snode @ #x56da96>:
 event-time     (8 7 6 5 4 3 2 1 0)
 level          segment
 coord          ((95.57696 95.99027 0.0) (96.215935 96.45725 0.0)
           (96.83128 96.9341 0.0) (97.42249 97.42075 0.0)
           (97.98912 97.91718 0.0) (98.530655 98.42336 0.0)
           (98.4704 99.05997 0.0) (99.24255 100.03494 0.0)
           (100.0 101.0 0.0))
 number         9
 cpa            (-12.452843 17.039467 0.0)
 linear         ((95.57696 95.99027 0.0) (-0.63897705 -0.46697998 0 0))
 tnode          #<tnode @ #x585fce>
 threat         nil
```

After the blackboard detects the unmatched segment node, it constructs a distinct track for this segment and the resulting solution consists of the two track nodes given below. The first track node is the newly created node from the unmatched segment nude. The second track node is the old established track node which now contains only two supporting segment nodes. The solution of the tracking problem is now two trees (and in general a forest of trees) representing two separate tracks. The tnode corresponding to Track 1 of Fig. **14** is:

```
#<tnode @ #x585fce>
is an instance of class #<clos:standard-class tnode @ #x56daa6>:
 event-time       (8)
 level            track
 last-coord       (95.64931 95.96082 0.0)
 last-velocity    (-0.63897705 -0.46697998 0.0)
 snode            (#<snode @ #x583896>)
 threat           nil
 cpa-bracket      ((-23.255823 4.3589487) (-2.2320776 24.93455))
 check            nil
 checklyst        nil
```

The tnode corresponding to Track 2 of Fig. **14** is:

```
#<tnode @ #x585fbe>
is an instance of class #<clos:standard-class tnode @ #x56daa6>:
 event-time       (8)
 level            track
 last-coord       (93.55575 92.813065 0.0)
 last-velocity    (-0.8540497 -1.0287323 0.0)
 snode            (#<snode @ #x58323e> #<snode @ #x5838a6>)
 threat           t
 cpa-bracket      ((-27.750824 107.72498) (-21.077364 33.016785))
 check            5
 checklyst        nil
```

Note that the slot check now has a integer value. This value is another way to set up a periodic process in the blackboard. The antecedent of the rule that checks the validity of the track formation is considered verified if the track is a threat and if the check slot is either nil or an integer. The integer is the last time the track was checked so that when the current time exceeds the last time checked by ***recheck-interval,*** the antecedent is considered satisfied. Thus any threatening track is rechecked periodically.

Example 4

In this example there are three craft, one of which has a signal which fades for a short period. This example illustrates how faded segments may be matched up with established segments. It also illustrates a different type of goal to activate the MERGE-SEGMENTS KS.

Recall from the KS section, that the MERGE-SEGMENTS KS is activated by a recurrent goal. This means that once a KSAR has been created and scheduled, the goal is inhibited from creating another KSAR until the **KS** finishes its attempt to extend atrophied segments. In this example there are three trajectories as illustrated in Fig. 15. Only one of these trajectories fades. Its segment node is given by:

```
#<snode @ #x583ab6>
is an instance of class #<clos:standard-class snode @ #x56ddbe>:
 event-time      (2 1 0)
 level           segment
 coord           ((1.47 98.4704 0.0) (0.7425 99.24255 0.0) (0.0 100 0 0.0))
 number          3
 cpa             (49.926693 47.039314 0.0)
 linear          ((1.47 98.4704 0.0) (0.7275 -0.77215576 0.0))
 tnode           #<tnode @ #x780026>
 threat          nil
```

This is the initial part of the trajectory, which results in a track node being formed. However, after the time 2 the trajectory input fades resulting in a time gap for the input values. The path does not return until the time 8. At this time the following segment node is generated on the BB. Note that this is now a new segment.

```
#<snode @ #x620dee> is an instance of class #<clos:standard-class snode @ #x56ddbe>:
 The following slots have :INSTANCE allocation:
 event-time      (8 7)
 level           segment
 coord           ((5.52 93.5456 0.0) (4.8825 94.39965 0.0))
 number          2
 cpa             (48.386578 36.11784 0.0)
 linear          ((5.52 93.5456 0.0) (0.6374998 -0.8540497 0.0))
 tnode           #<tnode @ #x65f9d6>
 threat          nil
```

Both of the above segments point to separate track nodes. At this time, these tracks are different since the second segment has not yet been determined to be an extension of the first one.

After the MERGE KS has run, the above segment is recognized as an extension of the first segment. So the latest track and segment nodes are retained and the older

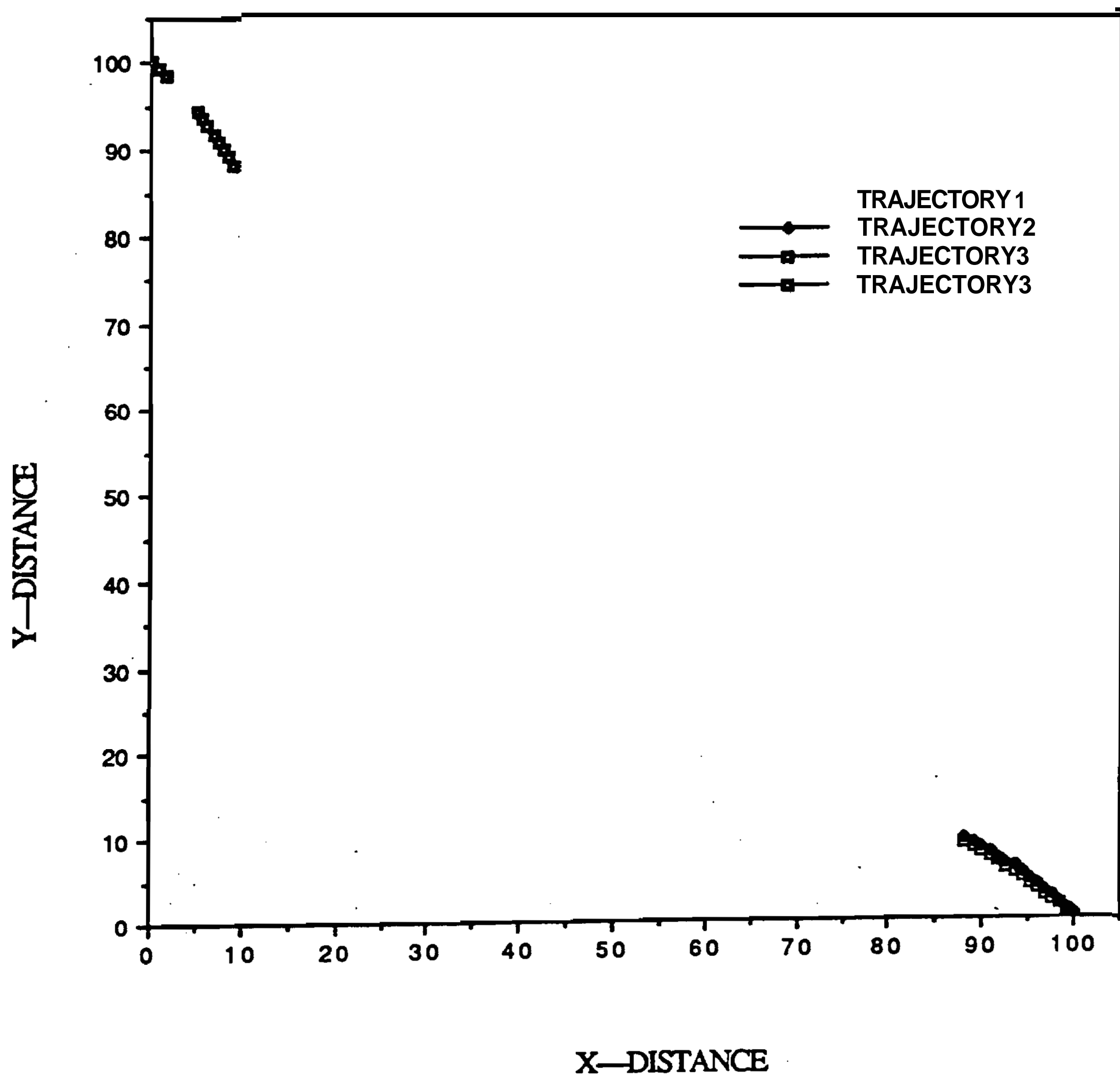


FIGURE 15. Merging of the two segments of a track is illustrated.

segment is removed from the BB and the sibling pointer from the older track to the oldest segment node is deleted. If that is the only segment supporting that older track, then the entire track is removed by removing the track node from the BB. The surviving track node that was established for the reappearing track now represents both the current track and the merged track. The track node looks like:

```
#<tnode @ #x65f9d6>
is an instance of class #<clos:standard-class tnode @ #x56ddce>:
 event-time        (8)
 level             track
 last-coord        (5.52 93.5456 0.0)
 last-velocity     (0.6374998 -0.8540497 0.0)
 snode             (#<snode @ #x620dee>)
 threat            nil
 cpa-bracket       ((44.09992 52.673237) (30.375065 41.860615))
 check             nil
 checklyst         nil
```

So the old segment has been patched to the new track although the old segment data has not been appended to the new track. No history of this older track has been included in the current track since the segment nodes and hit nodes are removed from the BB as soon as possible. However, a short history trail could be easily added to the track node.

#"#

APPENDIX B

RTBB CODE

Some variable names in this listing are different from those shown in the examples in the text part of this report. This is primarily a result of an **ungrade** in the Allegro Common Lisp by Franz. Some of the slot names used by the code prior to the upgrade became reserved words. An example is the slot values number which had to be changed to ***numbor***. Another is the word time which was changed to event-time. These are minor changes. The code runs on SUN SPARC Allegro CL 4.1 [SPAR; R1] (11/10/92 15:35).

Be advised though that it is essential to disable the *print-pretty* option which appears to be the default on the previous version. This is done by the (setq *print-pretty* nil), otherwise commands sent to the Knowledge Sources, compiled as separate executable~will be truncated; the resulting **errors** can be very difficult to trace. This command is embedded in the ***ggclass.cl*** file.

Its advisable to load the source files in a **separate** directory to avoid name conflicts and the destruction of similarly named files used by the RTBB. To build the four executable Knowledge Sources one needs to:

1. **csh makepath**

   This command will construct the executable called path and this will then serve as the data driver.

2. **csh makespline**

   This command will construct the executable called spline and this will then serve as the spline KS.

3. To construct the LISP executables, **first** use the
   (compile-file "nassign")
   (compile-file "testtrack.cl")
   commands when in the LISP environment -- this producers two fasl files called ***nassign.fasl*** and ***testtrack.fasl***. Now first load ***nassign.fasl*** into the LISP environment and then do a ***(dumplisp :name "test")*** which will return a nil when finished. Then do an (exit). This should create a executable file called test, which is about 12 megs in size. Check it first before continuing.

4. After you **fire** up **LISP** again, load ***testtrack.fasl*** and follow that with a ***(dumplisp :name track)*** which should return a nil, then do an ***(exit).*** This should generate another 12 meg executable file called ***track.***

5. The system is loaded by entering the **LISP** environment, and then doing a load of the file ***bbc*** which downloads all the files. A ***cloop*** command is used to watch the system **run** and ***fcloop*** allows you to **dump** the output into a file called ***out.*** Make **sure** that ***our*** is not a file you are using for other purposes.

***************** end of readme ******************************

Next two files have no commend lines in them at all. The files are
the **makepath** and the **makespline** files - each of length two lines.

****

```
cc singlepath.c -lm
mv a.out path

cc testspline.c -lm
mv a.out spline
```

****

```
/*
                FILENAME IS singlepath.c
*/

#include <stdio.h>
#include <strings.h>
static int bnum=0;
static double scale=100e0;
/*
  each row of this matrix is a vector in space forming
  a trapozoid associated with generating the bezier curves
  The orginal program for this was bezier.c                    i
*/
static double r[4][3] = {  {100, 0, 0 },
                    {75, 25, 0 },
                    {25, 25, 0 },
                    { 0,  0, 0 } };


main()
{
int i,c,j,ntracks=1;
double coord[3];
char line[80],*s;
char *gets();
setlinebuf(stdout);
s="(fire)";
for(i=0; (i < 250) && (strcmp(gets(line),s) == 0);i++)
  {
  printf("((%d) ",ntracks);
  printf("%d ",bnum);
  /*
  Could insert as many or possibly a random number of calls
  to track generation programs.
  */
    printf("(");
    beamone(coord,r);
    printf("(%lf %lf %lf)",coord[0],coord[1],coord[2]);
    printf("))0);

/* advance step */
bnum++;

  /*
  Note that setlinebuf avoids using the fflush program to
  initiate emptying of the line buffer to the coorperating lisp
  program.
  fflush(stdout);
  */
  }
}

int beamone(coord,r)
double coord[3];
double r[4][3];

int i;
double  u,v;
double *rzero,*rone,*rtwo,*rthree;
double pow();

/* Initialize the vectors for Bezier's curve. */
rzero = r[0];
rone = r[1];
rtwo = r[2];
rthree = r[3];
/*
for (i=0; i<3 ; i++)
{
```

```
printf( "the vector printed is %e 0, rzero[i]);
printf( "the vector printed is %e 0, rone[i]);
printf( "the vector printed is %e 0, rtwo[i]);
printf( "the vector printed is %e 0, rthree[i]);
}
•/
u = bnum/scale;
v = 1.e0 - u;
/•
if (u < 0.e0 || u > 1.e0)
  printf( " parameter of space curve out of range %e0,u);
else
  printf( " value if u is %eOvalue bnum is %dO,u,bnum);
•/
for(i=0; i < 3 ; i++)
{
coord[i] = 0.e0;
/*
printf(" the value of coord[i] before update %e0,coord[i]);
printf(" the power term is %e0,pow(v,3.e0));
printf(" the rzero[i] term is %e0,rzero[i]);
printf(" product pow and rzero is %e0,pow(v,3.e0)*rzero[i]);
•/
coord[i] += pow(v,3.e0)•(rzero[i]);
/* printf(" the value of coord[i] is %d  %e0,i,coord[i]); */
coord[i] += pow(u,3.e0)*rthree[i];
coord[i] += 3*u*pow(v,2.e0)*rone[i];
coord[i] += 3*v*pow(u,2.e0)*rtwo[i];
/* printf(" the value of i and coord is %d  %e 0,i,coord[i]); */
}
     return(0);
}
```

```
;;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   FILE NAME IS nassign
;;;;;;;;;;;;;;;;;;;;;file        name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq *print-pretty* nil)  ;; Oct 92 to eliminate mismatch from old version

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; An upper value which is the global value needed in branch and bound.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar upper 1000) ; global variable for least upper bound
(defvar upath nil)  ; path corresponding to upper

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This functions tears apart an a-list to get the value or the path.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun getvalue (lyst) (cadr (assoc 'value lyst)))
(defun getpath (lyst) (cadr (assoc 'path lyst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This are test matrices to work on the branch-and-bound
;;; or the best-first search
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq r1 '((1)(2)(3)(4)(5)))
(setq r2 '((5)(3)(1)))
(setq r3 '((1100 3 4)(1 3 100 4)(1 2 98 4)))
(setq r4 '((0 100 1 2)(0 2 99 3)))

(setq test5by4 '((9 5 4 5)
                 (4 3 5 6)
                 (3 1 3 2)
                 (2 4 2 6)
                 (0 1 4 5)))

(setq testmatrix '((9 5 4 5)
                   (4 3 5 6)
                   (3 1 3 2)
                   (2 4 2 6)))
(setq testmatrixl '((1 5 4 5)
                    (4 1 5 6)
                    (3 2 1 2)
                    (2 4 2 1)))

;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function is for accessing an element of the array.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun getelement (i j matrix) (nth j (nth i matrix)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  This function gets the minimum element of the first column of the matrix.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun min-col-element (matrix) (apply #'min (mapcar #'car matrix)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function gets the minimum element of the ith column.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun min-ith-col-element (i matrix)
  (apply #'min
     (mapcar       ; this returns the ith column
      #' (lambda (x) (nth i x))  ; as a list
      matrix)
```

```
      )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Function closerp is then just the minimum of the two
;;; values obtained from the a-lists.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun closerp (a b) (< (getvalue a) (getvalue b)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Function gets min of column col excluding rows in path from matrix.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun min-col (path col matrix)
  (do
   (
   (vmin 1000)
   (test (length matrix) (1-test))
   (k 0 (1+ k))
   )
   ((zerop test) (return vmin))
   (setq x (getelement k col matrix))
   (cond
    ((member k path)) ; is in excluded row - ignore
    ((>= x vmin))     ; is not less of equal - ignore
    (t (setq vmin x)) ; if less, then record minimum
    )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Function  finds the f value of the best-first search procedure.
;;;  Note that it works with the reverse path since the bestfirst
;;;  procedure uses the reverse path in its procedure.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun f (path matrix)
  (do
   (
    (worklyst (reverse path) (cdr worklyst))
    (k 0 (1+ k))
    (sum 0)
    (count (length (car matrix)) (1-count))
    )
   ((zerop count) (return sum))  ; returns best est lower than actual
   (cond
    (worklyst   ; trace down the know path so far
     (setq sum (+ (getelement (car worklyst) k matrix)
                 sum )
        ))
    (t (setq sum (+ sum  ; look for min excluding rows chosen
                   (min-col path k matrix))))
    )
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Fvalue returns a feasible path choosing the minimum
;;; element in each column excluding those rows not
;;; choosen first.  Path returned is in reverse order.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun fvalue (matrix)
  (do (
       (col 0 (1+col))
       (value 0)  ;initialize value
       (path nil) ; initialize path
       ;; FIRST CHANGE
       (test (length (car matrix))) ; test now uses number of cols
       )
```

```
      ((equal col test)    ; when column number = max row number - stop
       (setq upper value upath path)
       (return (list value path))) ; returns value and path
      ; find min value and index for next element in path
      (setq x (min-col-index path (length path) matrix))
      ; update value for minimum
      (setq value (+ value (cadr x))) ; add min value of col
      ; update the path - path in reverse order
      (setq path (cons (car x) path)) ; cons index of row
      )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This function returns
;;  1.  the minimum value of column col
;;      in the array called matrix but over the row
;;      elements NOT in path
;;  AND
;;  2.  the index of the element
;;      e.g. (index vmin).
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun min-col-index (path col matrix)
  (do
   (
   (vmin 1000) ; bind a initial value to vmin
   (test (length matrix) (1- test)) ; path length based on rows
   (k 0 (1+ k)) ; search down column, k row index
   )
   ((zerop test) (return (list index vmin)))
   (setq x (getelement k col matrix))
   ;; find minimum, x test element
   (cond
    ((member k path)) ; if element is on path row, don't look here
    ((>= x vmin)) ; if value exceeds current min, ignore
    (t (setq vmin x index k)) ; if better value, update index and vmin
    )
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This fills the list with 0 1 ... n-1.
;;  Function is needed to form the complement of a set.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun numset (n)
  (numsetl (- n 1) nil)) ; call with n-1 to initialize recursion

(defun numsetl (n lyst)
  (cond
   ((< n 0) lyst) ; n less than 0, set to nil
   (t
    (cons n (numsetl (- n 1) lyst)))
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Expand2 gives the children of a node given the present path.
;;; It uses the path to the node as a set, complements it using
;;; the difference and this then returns the children of the node.
;;; To generalize this I think the length must be changed to
;;; length of the matrix which is the number of rows or the number of hits.
;;; Presently rows >= cols or the thing won't work.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defun expand2 (path matrix)
  (mapcar #'(lambda (child) (cons child path) )
```

```
;;       (our-set-difference
           (set-difference
          (numset (length matrix))
         path  ; this is the row set already used in the path
         )
        )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Expandl gives the f value for the path and constructs
;;;  the association list.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expandl (path matrix)
  (feasible path matrix) ; extends partial path and lowers upper bdd
  (list  ; this constructs the association list for the best first
   (list 'value  (f path matrix)) ; first the value and then
   (list 'path path)              ; the path
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Expand just applies expandl and 2 to all the children,
;;; fathoms those paths which have a lower bound exceeding known
;;; feasible paths.  Same as pruning subtrees from best-first.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expand  (path matrix)
  (remove-if          ; removes paths which exceed upper bound
   #'(lambda(x)  (>= (getvalue x) upper)) ; this is same a fathom
  (mapcar #'(lambda  (x)  (expandl x matrix)) ; in the B and B
       (expand2 path matrix)))
  )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function is used as a mask for a list removing
;;; those elements which are nil for the mask.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun my-filter  (test element)
  (if test nil  (list element))
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is only a limited version of replace since
;;; it works only on a single list whereas the other
;;; version built in is virtually equivalent to a mapcar.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun my-remove-if  (func lyst)
  (apply #'append            ; remove element by replacing by nil
       (mapcar #'my-filter (mapcar func lyst) ; then appending out
            lyst)
       )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Given a partial path, this function extends it
;;; to a feasible solution and that solution then is used
;;; to lower the least upper bound if it is smaller.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun feasible  (path matrix)
  (do
   (
    (worklyst  (reverse path)  (cdr worklyst))
    (k 0 (1+ k))
    (newlyst path)
    (sum 0)
    (count  (length  (car matrix))  (1- count))
    )
```

```
    ((zerop count)  ; update upper if return value smaller than upper
     (if (< sum upper) (setq upper sum upath newlyst) nil)
     (return (list sum newlyst)))
    (cond
     (worklyst  ; follow the partial path until it ends
      (setq sum (+ (getelement (car worklyst) k matrix)
                sum )))
     (t     ; extend partial path
       (setq x (min-col-index newlyst (length newlyst) matrix))
       (setq sum (+ sum (cadr x)))
       (setq newlyst (cons (car x) newlyst))))
   )
  )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the main routine -- first the initialization and then
;;; the call to the recursive version of best first search.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun best (matrix)
  (fvalue matrix) ; this initializes the upper bound to feasible value
  (setq queue (expand nil matrix))
  (sort queue 'closerp)
  (bestl queue matrix)
  )

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the recursive version of the solution.
;;; Queue is the ordered queue which is used to choose the next
;;; child for expansion.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defun bestl (queue matrix)
  (cond
   ((null queue) (list upper (reverse upath)))
   (t
    (best1 (sort (append (expand (getpath (car queue)) matrix)
                  (cdr queue))
            'closerp)
         matrix)
    )))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Square is just a utility for the metric.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defun square (x ) (* x x))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Recursive version returns the euclidian distance.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defun euclid (coordl coord2)
  (sqrt (euclidl coordl coord2))
  )

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  This is the recursive part of euclid.
;;;   Actually this will work for n dimensions.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defun euclidl (coordl coord2)
  (cond
   ((null coordl) 0) ; initialize as zero at bottom of recursion
```

```
    (t
     (+ (square (- (car coordl) (car coord2))) ; keep adding
        (euclidl (cdr coordl) (cdr coord2))
        )
     )))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  This takes two vectors and constructs the outer product.
;;;  lystl >< lyst2 means lystl forms the rows and lyst2 the
;;;  forms the columns. The (i,j) element of the matrix
;;;  is lystl(i)*lyst2(j) and "*" operation
;;;  represents the distance between the two position vectors.
;;;  In application, rows are hits and cols are segments.
;;;  The children of the best first search are then segments.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defun form-matrix (lystl lyst2)
  (do   ; this is the first do loop cdring down the first lyst
    (
     (rlist lystl (cdr rlist))
     (form nil)
     (form2 nil)
     (row (length lystl) (1- row)) ; outer loop counter
     )
    ((zerop row)
;;     (format t "~% matrix is ~% ~a ~% " form)
     (return form))
    (setq form2 (do   ; this is the second do loop cdring down lyst2
            (
            (clist lyst2 (cdr clist))
               (form2 nil)
               (col (length lyst2) (1- col))  ;NOTE CHANGE
            )
    ((zerop col) (return form2))
     (setq form2 (append form2         ; calculates dist for each col
                  (list (euclid   ;stuff in list using append
                         (car rlist)
                         (car clist)))))
       )
    (setq form (append form (list form2)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  This is the interface function to the KSAR called getassignment
;;;;  returning a list of the optimum assignment. Best
;;;;  returns the value of the minimum path.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun getassignment (lystl lyst2)
  ;; (let ()
     ;;(lystl (read))  ; gets the assignment coord
     ;;(lyst2 (read))  ; gets the hit coordinates
     ;;)
  (cadr (best (form-matrix lystl lyst2))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  For a read-evaluate-write loop so that one can
;;;;  drive the functions in this process.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defun rewl ()
  (do () (nil )
     (format t "~a~%" (eval (read)))))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Initialize the read-evaluate-write loop.
```

```
;;;; This is commented out for debugging purposes.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
 (rewl)
;;
;;
;;;;;;;;;;end    of nassign file ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
/*
                  FILENAME IS testspline.c
*/

#define MAXPTS 4
/*
#define DEBUG 1
*/

#define MAXCALL 100
#include <stdio.h>
#include <strings.h>
#include <math.h>
static double h[MAXPTS],diag[MAXPTS],d[MAXPTS],b[MAXPTS],c[MAXPTS];
static double delta[MAXPTS],alpha[MAXPTS],beta[MAXPTS],sigma[MAXPTS];
static double rhs[MAXPTS];

main()
{
int i,j,n=MAXPTS;
int nbound;
double x[MAXPTS],y[MAXPTS],z[MAXPTS],s[MAXPTS],t[MAXPTS];
double u,scale,pow(),espline();
char *sptr ,line[180],workstring[180];
;setlinebuf(stdout);

/*
The coefficients b(i),c(i),d(i) i=0,1,...,n-2 for each
of the segments to yield the curve

   f(s) = y(i)+ b(i)(s-s(i)) + c(i)(s-s(i))**2 + d(i)(s-s(i))**3

*/

for(nbound=0;nbound < MAXCALL;nbound++)

{
sptr = gets(line);
/* position sptr at first coord */
for(;*sptr == '(' || *sptr == ')' || *sptr == ' ';sptr++);
for(i=0;*sptr != ')';sptr++,i++)
    workstring[i] = *sptr;
workstring[i] = ' ';
sscanf(workstring, "%lf %lf %lf %lf", &t[3],&t[2],ht[1],&t[0]);

printf("(%e %e %e %e)0,t[3],t[2],t[1],t[0]);

sptr = gets(line);

for(j=0;j<MAXPTS;j++)

/* position sptr at first coord */
for(;*sptr == '(' || *sptr == ')' || *sptr == ' ';sptr++);
for(i=0;*sptr != ')';sptr++,i++)
    workstring[i] = *sptr;
workstring[i] = ' ';
sscanf(workstring, "%lf %lf %lf", &x[j],&y[j],&z[j]);
}

for(j=0;j<MAXPTS;j++)
  printf("(%e %e %e %e)0,t[j],x[j],y[j],z[j]);


  /*
for (i=0; i < (MAXPTS-1) ; i++)
  {
  printf(" y and s are given by %e %e 0,y[i],t[i]);
}
```

```
*/

spline(n,t,x,b,c,d);
printparameters(x,b,c,d);

spline(n,t,y,b,c,d);
printparameters(y,b,c,d);

spline(n,t,z,b,c,d);
printparameters(z,b,c,d);

  /* end condition goes here */

  printf("()0);
  /*
for( ;u != -1;)
{
printf(" ENTER a value of u, u = -1 stops loop 0);
scanf("%e",&u);
printf(" the spline value is %e0,espline(n,u,t,y,b,c,d));
}
*/
}
}

int spline(n,s,y,b,c,d)
int n;
double s[],y[],b[],c[],d[];

{
/* Calculate tridiagonal elements. */
double temp0,temp1,temp2;
double pow();
int i ;

/* Initialize all the variables used in this function. */
for(i=0; i < n ; i++)
{
b[i]=0.e0; c[i]=0.e0; d[i]=0.e0;
diag[i]=0.e0; h[i]=0.e0; delta[i]=0.e0; rhs[i]=0.e0;
alpha[i]=0.e0; beta[i]=0.e0; sigma[i]=0.e0;
}


h[0] = s[1]-s[0];  diag[0] = -h[0]; i=0;
delta[i] = y[i+1]-y[i];

iifdef DEBUG
printf("i is %d  h is %e delta is %e 0,i, h[i],delta[i]);
#endif

for(i=1; i< n-1 ; i++)
{
h[i] = s[i+1]-s[i];
diag[i] = 2*(h[i]+h[i-1]);
delta[i] = y[i+1]-y[i];
rhs[i] = delta[i]-delta[i-1];

#ifdef DEBUG
printf("i is %d  h is %e 0,i, h[i]);
printf("diag is %e  delta is %e rhs is %e0,diag[i], delta[i],rhs[i]);
#endif

}
diag[n-1] = h[n-2];

/* The rhs derivative

put in error trap here
```

```
*/
/* Construct the first and last vector elements of the rhs. */

temp0 = (delta[1]-delta[0])/(s[2]-s[0]);
templ = (delta[2]-delta[1])/(s[3]-s[1]);
temp2 = (templ - temp0)/(s[3]-s[0]);
rhs[0] = h[0]*h[0]*temp2;

/* Do the last element of rhs vector. */

temp0 = (delta[n-3]-delta[n-4])/(s[n-2]-s[n-4]);
templ = (delta[n-2]-delta[n-3])/(s[n-1]-s[n-3]);
temp2 = (templ - temp0)/(s[n-1]-s[n-4]);
rhs[n-1]= -pow(h[n-2],2.e0)*temp2;

/* Now compute the alphas.  */

alpha[0]= -h[0]; i=0;

tifdef DEBUG
printf(" i is %d and alpha is %e 0,i,alpha[i]);
#endif

for(i=1; i< n-1 ;i++)
{
alpha[i] = 2*(h[i-1]+h[i]) - pow(h[i-1],2.e0)/alpha[i-1];

#ifdef DEBUG
printf(" i is %d and alpha is %e0,i,alpha[i]);
#endif

}
alpha[n-1] = -h[n-2] - h[n-2]*h[n-2]/alpha[n-2];
i=n-1;

tifdef DEBUG
printf(" i is %d and alpha is %e0,i,alpha[i]);
#endif

/* Now compute the betas. */

beta[0] = rhs[0]; i=0;

tifdef DEBUG
printf(" i is %d and beta is %e0,i,beta[i]);
#endif

for(i=1; i < n-1 ; i++)
{
beta[i] = (delta[i]-delta[i-1]) - h[i-1]*beta[i-1]/alpha[i-1];

tifdef DEBUG
printf(" i is %d and beta is %e0,i,beta[i]);
#endif
}
i=n-1;
beta[n-1] = rhs[n-1] - h[n-2]*beta[n-2]/alpha[n-2];
#ifdef DEBUG
printf(" i is %d and beta is %e0,i,beta[i]);
#endif

/* Compute sigmas. */
i=n-1;
sigma[n-1] = beta[n-1]/alpha[n-1];

#ifdef DEBUG
printf(" i is %d and sigma is %e 0,i,sigma[i]);
#endif

for(i=n-2; i>= 0 ; i--)
```

```
{
sigma[i] = (beta[i] - h[i]*sigma[i+1])/alpha[i];

#ifdef DEBUG
printf(" i is %d and sigma is %e 0,i,sigma[i]);
#endif
}

/* Compute coefficient. */

for(i=0; i< n-1 ; i++)
{
b[i] = (y[i+1] - y[i])/h[i] - h[i]*(sigma[i+1] + 2.e0*sigma[i]);
c[i] = 3*sigma[i];
d[i] = (sigma[i+1] - sigma[i])/h[i];
}


double
espline(n,u,t,y,b,c,d)
int n;
double u;
double t[],y[],b[],c[],d[];
{
double temp, dx;
int i;
/*
for(i-0; i < n ; i++)
  printf(" input values to evaluate for i and y %d %e 0,i,y[i]);
*/

for(i=0; !(u>=t[i] && u<t[i+1]) 66 i<4 ; i++);

/*
printf(" the interval evaluate spline found is %d 0,i);
*/

if (i == 4)
{
  printf(" u is being used for prediction 0);
  i=3; b[i]=b[i-1]; c[i]=0; d[i]=0;
  }

/* Calculate change from t[i] and then use Horners rule. */

dx = u - t[i];
temp = y[i] + dx*(b[i] + dx*(c[i] + dx*d[i]));
/*
printf( " the value of u and spine poly is %e %e 0, u , temp);
*/
return(temp);
}

int printparameters (a,b,c,d)
double a[],b[],c[],d[];
{
int i;
for (i=0; i < (MAXPTS-1) ; i++)
  printf("(%d%e %e %e )0, i,a[i],b[i],c[i],d[i]);
}
```

```
;;;;;;;;;;;;;;;;;file          name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    THIS FILE IS          testtrack.cl
;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq *print-pretty* nil) ;; Oct 92 to eliminate mismatch between versions

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    This function evaluates the threat via confidence
;;    interval and records the confidence interval.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar epsilon 0.1)

(defun gettrack (xcpa xr)
  (let*
   (
    (cpa xcpa)
    (r   xr)
    (dr (vector-difference cpa r)) ; vector difference between cpa and r
    (edr (scale-vector epsilon dr)); error in estimation fixed at epsilon
    (dx (sort (list    ; for x confidence interval
       (- (car cpa) (car edr))
       (+  (car cpa)(car edr)))
           '< ))
    (dy (sort (list    ; form y confidence interval
       (- (cadr cpa)(cadr edr))
       (+  (cadr cpa)(cadr edr)))
           '< ))
    (threat (or (minusp (apply #'* dx))  ; see if intervals split origin
           (minusp (apply #'* dy)))))
;;   (format t "~a~%" (list dx dy))  ; for pair of confidence intervals
;;   (format t "~a~%" threat)
 (list (list dx dy) threat)
    ))      ; true if origin in confidence region

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Vector sum and difference formulas.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vector-difference (u v)
  (cond
   ((null u) nil)
   (t
    (cons (- (car u) (car v))
        (vector-difference (cdr u) (cdr v))))
   )
  )

(defun vector-sum (u v)
  (cond
   ((null u) nil)
   (t
    (cons (+ (car u) (car v))
        (vector-sum (cdr u) (cdr v))))
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Function scales a vector.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun scale-vector (alpha v)
  (mapcar #'(lambda (x) (* alpha x)) v))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  For a read-evaluate-write loop so that one can
;;;;  drive the functions in this process.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
(defun rewl ()
  (do () (nil )
      (format t "~a~%" (eval (read)))))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Initialize the read evaluate write loop.
;;;; This is commented out for debugging purposes.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
 (rewl)
;;
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;end of testtrack.cl;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;       filename is bbc -- used to load BB
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Compile instructions are in the readme file
;;  note that track* is the compilation of testtrack
;;            test*  is the compilation of nassign
;;            path* is the compilation of path generation
;;            spline* is the compilation of testspline
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq lyst
  '(
    vil.cl
    ggsetops.cl
    ggutil.cl
    ggmacro.cl
    ggclass.cl
    ggksar.cl
    ggnode.cl
    ggplan.cl
    ggports.cl
    ggrule.cl
    ggrloop.cl
    ggmess.cl
    ggmerge.cl
    gginit.cl
             ))
(defun loadm (lyst)
  (dolist (var lyst)
      (load var)))
(loadm lyst)
```

```
;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; THIS FILE IS ggsetops.cl
;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Our-union is the Winston set function on page 34 of Winston's book.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-union (x y)
  (cond ((null x) y)
      ((member (car x) y) (our-union (cdr x) y))
      (t (cons (car x) (our-union (cdr x) y)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Our-intersection is the Winston set function page 344.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-intersection (x y)
  (cond ((null x) nil)
      ((member (car x) y)
       (cons (car x) (our-intersection (cdr x) y)))
      (t (our-intersection (cdr x) y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Our-set-difference is the Winston set function page 344.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-set-difference (in out) ; in the larger set
  (cond ((null in) nil)
      ((member (car in) out) (our-set-difference (cdr in) out))
      (t (cons (car in) (our-set-difference (cdr in) out)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Same-set determines if two sets are equal.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun samesetp (a b) (and (our-subsetp a b) (our-subsetp b a)))

(defun our-subsetp (a b)
  (cond ((null a) t)
      ((member (car a) b) (our-subsetp (cdr a) b))
      (t nil)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function fills the list with n-1 ... n and
;; is needed to form the complement of a set.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun numset (n)
  (numsetl (- n 1) nil)) ; call with n-1 to initialize recursion

(defun numsetl (n lyst)
  (cond
   ((< n 0) lyst) ; n less than 0, set to nil
   (t
    (cons n (numsetl (- n 1) lyst)))
   )
  )
```

```
;;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;       THIS FILES IS ggutil.cl
;;;;;;;;;;;;;;;;;;;;;file     name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the vector magnitude.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vector-magnitude (vector)
  (sqrt (vector-magnitude1 vector)))

(defun vector-magnitude1 (vector)
  (cond
   ((null vector) 0)
   (t
    (+ (expt (car vector) 2)
       (vector-magnitude1 (cdr vector))))
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the unit vector given a vector.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-unit-vector (vector)
  (let ((mag (vector-magnitude vector)))
       (cond
      ((zerop mag) vector)
      (t
       (mapcar #'(lambda(x) (/ (float x) mag))
            vector)))
       )
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns a scaled vector.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun scale-vector (alpha v)
  (mapcar #'(lambda (x) (* alpha x)) v))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the dot product of two vectors.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun dot-product (u v)
  (cond
   ((null u) 0)
   (t
    (+
     (* (car u)  (car v))
     (dot-product (cdr u)(cdr v))
     )
    ))
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the vector difference of two vectors.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vector-difference (u v)
  (cond
   ((null u) nil)
   (t
    (cons (- (car u) (car v))
        (vector-difference (cdr u) (cdr v))))
   )
```

```
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the vector sum of two vectors.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vector-sum (u v)
  (cond
   ((null u) nil)
   (t
    (cons (+ (car u) (car v))
        (vector-sum (cdr u) (cdr v))))
   )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the velocity from the distance vector.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-velocity (vector deltat)
  (mapcar #'(lambda (x) (/ (float x) deltat))
        vector))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the cpa fromthe position vector.
;;   This function finds the minimum distance from the origin
;;   to the projected path and returns the vector of that distance.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-cpa (coord time)
  (let* (
      (r (car coord))
      (u (vector-difference (car coord)
                      (nth (1- prediction-threshold) coord)))
       (v (find-unit-vector u))
       (scale (- (dot-product r v)))
       (vtemp (scale-vector scale v)))
      (vector-sum r vtemp)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the cpa from the position vector.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-linear-model (coord time)
(format t "***** ENTERED FIND-LINEAR-MODEL *****")
(format t " coord is ~a~%" coord)
(format t " time is ~a~%" time)
  (let* ((r (car coord))
      (u (vector-difference
          (car coord) (nth (1- prediction-threshold) coord)))
      (dt (- (car time)(nth (1- prediction-threshold) time)))
      (v (find-velocity u dt))
      )
(format t "dist vec r is ~a~% " r)
(format t "vec diff u is ~a~% " u)
(format t "dt time diff is ~a~% " dt)
(format t "v velocity is  ~a~% " v)
       (list r v)
       )
;;(break "~% ABOUT TO EXIT LINEAR MODEL ~%")
  )



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the average and vector average.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun average (lyst)
```

```
  (let* (
       (asum (apply '+ lyst))
       (len  (length lyst))
       )
      (/ asum len)))

(defun convex-average (alpha x y)
  (+ (* alpha x)
       (* (- 1.e0 alpha) y)
       ))

(defun vector-average (lyst)
  (do*
   (
    (scale (/ 1.e0 (length lyst)))
    (vlyst (mapcar #'(lambda (x) (scale-vector scale x)) lyst))
    (worklyst vlyst (cdr worklyst))
    (element (car worklyst) (car worklyst))
    (result (car vlyst) (vector-sum result element))
    )
   ((zerop (1- (length worklyst))) (return result))
   ))

(defun convex-vector-average (alpha x y)
  (vector-sum (scale-vector alpha x)
            (scale-vector (- 1.e0 alpha) y)
            ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the range of the intervals.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun union-intervals (lyst)
(format t "Inside UNION-INTERVALS input lyst is ~a~% " lyst)
  (let (
      (low (apply 'min (mapcar 'car lyst)))
      (high (apply 'max (mapcar 'cadr lyst)))
      )
        (list low high)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function returns the cosine between two vectors.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vector-angle-cosine (u v)
  (dot-product
   (find-unit-vector u)
   (find-unit-vector v)
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the absolute difference between two numbers.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun absolute-difference (x y)
  (abs (- x y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the sum of the absolute diff between two
;; vectors using the above function.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vector-absolute-difference (u v)
  (apply '+
       (mapcar 'absolute-difference
            u v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Gaussian density function.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun gauss (x)
  (let ((const (/ 1.eO (sqrt 3.141592653589793))))
       (* const (exp (- (* x x))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  CDF -- exponential function given mean, returns 1-exp(-x/m).
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun exp-cdf (x mean)
  (- 1.e0 (exp (- (/ (float x) mean)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;end        of ggutil.cl file;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;file          name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   File is ggmacro.cl.
;;   This a file of macros and &unctions used  in the BB.
;;   Most of the routines act as accessors and modifiers for the
;;   objects on the BB.
;;
;;;;;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   This macro pushs objects from the queue instance variable
;;   of the objects acting like queues -- e.g. ksarq.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro mypush (object stack)
  `(cons ,object ,stack))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Macro push-value-onto-node-at-attribute which
;;   pushes value on the variable attribute of object node.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro push-value-onto-node-at-attribute (value node attribute)
  `(with-accessors ((x ,attribute)) ,node
    (setf x (cons ,value x ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This macro pushes a goal object onto the goal BB at said level.
;;   Note left refers to the LHS of the BB which is the goal BB.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro sendpushgoal (object level)
  `(setf (left ,level)
       (mypush ,object (left ,level) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This macro pushes an object onto the pqueue queue.
;;   This macro has to be rewritten
;;   since set-queue triggers a method which inserts the
;;   object in the proper queue.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro sendksarpush (object pqueue)
  `(set-queue ,pqueue ,object))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This macro pushes an object at said level to the data BB.
;;   The right refers to the RHS of the BB and the accessor method.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendpushlevel (object level)
  `(setf (right ,level)
       (mypush ,object (right ,level))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This removes the last member of a queue on the data BB.
;;   Right method referes to the data side of the BB.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro fifodequeue (level)
  `(setf (right ,level)
       (reverse (cdr (reverse (right ,level))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   This function is an ordering function.
```

```
;;  Porder stands for priority order function used in the sort function
;;  to order the ksar's.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun porder (kx ky)
  (<= (priority kx )
      (priority ky )
      ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Macro get-nodes returns a list of all the data objects on
;; the level namedf -- levels are hits, segments, or tracks.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro get-nodes (level)
  `(right ,level))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This is a display function which shows the KSAR entries in
;;  in the various sub-queues.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun showq ()
;;  (format t " ENTER QUEUES to DISPLAY ~%")
;;  (format t " e for eventq, w for workq and k for ksar queue ~%")
;;  (format t " place these in a lyst ~%")
(format t "~% ========= BEGIN QUEUES ========== CLOCK is ~a =====~%" clock)
  (do*
   ((wlyst '(atomic-queue beam-queue segment-queue)(cdr wlyst))
    (plyst (car wlyst) (car wlyst))
    (var ksarq)
    )
   ((null wlyst)
(format t " ========= END   QUEUES ========== CLOCK is ~a =====~%" clock)
)
(format t " ************  ~a  ************************~%" plyst)
     (format t " * ~a~%" (funcall plyst var))
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function expands the KSAR's in the subqueues of the KSAR queue.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expandq ()
  (dolist
   (var (cons 'atomic-queue KSQUEUES))
  (format t "~%***********~a*queue expansion*************" var)
  (dolist (ele (funcall var ksarq))
   (describe ele))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;*  This a display function to display the levels of the blackboard.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun showl ()
(format t "~% ======== START OF BB LEVELS ======== CLOCK is 'a ====== ~%" clock)
  (do* (
      (worklyst level-lyst (cdr worklyst)) ; level-lyst global setq in
      (var (car worklyst) (car worklyst))  ; in ggoalbb.cl
      (varl '(tracks segments hits) (cdr varl))    ; labels for levels
      )
       ((null worklyst)
(format t " ========= END OF BB LEVELS ==== CLOCK is ~a ========== ~%~%" clock))
   (format t " ------------------  ~a  data------------------ ~%" (car varl))
       (format t "  ~a~%" (right var))
   (format t " ------------------  ~a  goal------------------ ~%" (car varl))
```

```
        (format t "  ~a~%" (left var))
        ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This display function expands the objects on a level.
;;  Expandl expands the DATA side and expandg expands GOAL side.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expandl (level)
  (dolist
   (var (right level))
   (describe var)))

(defun expandg (level)
  (dolist
   (var (left level))
   (describe var)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro removes a GOAL object specified by x from
;;  level y.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro remove-goal-x-from-level-y (node level)
  `(let*
    ( (temp (left ,level))  ;copy goal list
    (tlyst (delete ,node temp)) ) ; delete node from temp list
    (setf (left ,level) tlyst)
    (format t "~% GOAL NODE~a  has been removed from level ~a" ,node ,level)
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro removes a DATA node specified by x from
;;  level y.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro remove-data-x-from-level-y (node level)
  `(let*
    ( (temp (right ,level))  ;copy goal list
    (tlyst (delete ,node temp)) ) ; delete node from temp list
    (setf (right ,level) tlyst)
    (format t "~% DATA NODE~a  has been removed from level ~a" ,node ,level)
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function gets the coord associated with the segments,
;; and in addition, inserts the time coordinate as well to
;; include the time parameter in the matching process.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-segments-coord-with-time ()
        (mapcar
         #'(lambda (x)
                (cons
                (car (event-time x))
                (car (coord x))))
          (get-nodes segments)
        ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function gets the coord associated with the segments,
;; specified by y (a list of segment nodes)
;; and in addition puts in the time coordinate as well to
;; include the time parameter in the matching process.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-segments-coord-with-time-for-nodes-y (y)
  (format t "~% input to get-segments is ~a " y)
        (mapcar
         #'(lamb&  (x)
```

```
                (cons
                (car (event-time x))
                (car (coord x))))
          y))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function gets the coords from the oldest hits object.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (defun get-hits-coord ()
     (coord (car (last (right hits)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function gets the time from the oldest hits object.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (defun get-hits-time ()
     (event-time (car (last (right hits)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function gets the coord and time for last object of hits level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-hits-coord-with-time ()
  (mapcar #'(lambda(x) (cons (get-hits-time) x))
                    (get-hits-coord)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function gets the number of objects enqueued on the
;; data blackboard at the specified level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-number-on-level (level)
  (length (right level)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This macro invokes the ACCESSOR function for all the data track
;; objects in the list tracklyst.
;; The get-track-x-intervals function returns the cpa bracket
;; estimated from the tracks.  The get-track-y-intervals is the
;; corresponding function for the y-axis.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmacro get-track (operation tracklyst)
  `(mapcar #'(lambda (x) (,operation x)) ,tracklyst))

(defun get-track-x-intervals (trklyst)
  (mapcar #'(lambda (x) (car (cpa-bracket x))) trklyst))

(defun get-track-y-intervals (trklyst)
  (mapcar #'(lambda (x) (cadr (cpa-bracket x))) trklyst))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function removes a list of objects from the track level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun remove-nodes-from-level (level tnodelyst)
  (do*
   (
    (twork tnodelyst (cdr twork)) ; cdr down the nodelyst
    (tn (car tnodelyst) (car twork)) ; this is the head of lyst
    (lyst (right level))  ; this all the nodes on the level
    )
   ((null twork) (setf (right level) lyst))
    (setq lyst (remove tn lyst))  ; remove the unwanted nodes one-by-one
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; This function constructs a lyst from the first n elements of lyst.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun first-n-elements (n lyst)
  (if (< (length lyst) n)
      (format t " ERROR attempting take too many elements in lyst ~%")
     nil)
  (do*
   (
    (index n (1- index))
    (outlyst nil (cons (nth index lyst) outlyst))
    )
   ((zerop index) (return outlyst))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Get-recent-segments gives the segments which are within
;; "recent-time" i.e. < oldage -- which is a global variable
;; Becareful, time is now a system function associated with LISP
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-recent-segments (etime)
  (do*
   (
    (slyst (reverse (right segments)) (cdr slyst))
    (clyst (reverse (get-segments-coord-with-time) (cdr clyst))
    (newlyst nil)
    )
   ((null slyst) (return newlyst))
   (if (>= (- etime (caar clyst)) oldage) ; oldage global set at 3
       nil
       (setq newlyst (cons (car slyst) newlyst))
       )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the latest data point and returns the
;; time of that data point (hit or segment).
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-time-of-last-data-point ()
  ;; look at the segment and the hits data bb level
  ;; find the maximum point
  ;; if both are empty or either one return value of zero
  (let*
   ((hlyst (right hits))
    (hmlyst (mapcar #'(lambda (x) (event-time x)) hlyst))
    (slyst (right segments))
    (smlyst (mapcar #'(lambda (x) (car (event-time x))) slyst))
    (testlyst (append hmlyst smlyst)))
   (if testlyst (apply 'max testlyst) 0)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the oldest or (min clock time) of
;; a segment entry.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-oldest-segment ()
  ;; look at the segments on the data bb
  ;; find the minimum clock time of latest data point
  ;; return the node to test for atrophied data segments
  (do* (
    (slyst (right segments) (cdr slyst))
    (node nil)
    (smlyst (mapcar #'(lambda (x)
                        (car (event-time x))) slyst)
     (cdr smlyst))
    (work (car smlyst)))
    ((null slyst) (return node)) ; returns nil if no segments
```

```
(cond
 ( (< (car smlyst) work)
   (setq work (car smlyst)) ; update current minimum
   (setq node (car slyst))) ; update node corresponding to min
 (t nil)
 )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This selector copies from level y those data nodes that satisfy f.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun copy-data-nodes-at-level-y-satisfying-predicate-f (y f)
  (do (
      (temp (reverse (right y)) (cdr temp))
      (y-temp nil)
      )
       ((null temp) (return y-temp))
       (if (funcall f (car temp))
         (setq y-temp (cons (car temp) y-temp))
         )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro gets the goal lyst to be processed by the forward
;;  based chaining system.
;;  Note it can remove all the goals which have only one shot at
;;  getting processed.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro get-goals-from-level-x-of-duration-y (level lifespan)
  '(let*  (
        (temp (left ,level))
        (keep (remove-if
             #'(lambda (x) (equal (duration x) ',lifespan))
             temp))
        )
    (setf (left ,level) keep)
    temp
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro queue-flavor-onto-node-at-attribute
;;  pushes the object into the object called node using the
;;  accessor called attribute.
;;  Note that the object is merged with the existing list of objects
;;  using the forder function to order the list.  This is for pushing
;;  ksars into a priority queue.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro queue-flavor-onto-node-at-attribute (object node attribute)
  '(with-accessors ((x ,attribute)) ,node
(format t "~% the x variable will return ~a " x)
     (setf x (merge 'list (list ,object) x 'forder)
     )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Forder is the function which orders the objects or ksars.
;;
;; Order the queue with lower value priority first, highest value
;; priority last.
;; Note that with equal priority, they should be FIFO within the
;; priority.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun forder (x y)
  (<= (priority x) (priority y)))  ;; changed 2 jan 92
;;  (< (priority x) (priority y)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This macro pops object off the queue at node. Returns popped object.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro pop-flavor-at-node-at-queue (node qname)
'(format t "Inside pop-flavor-at-node-at-queue ~a 'a"  ,node ,q)
  `(with-accessors ((q ,qname)) ,node
    (let* (
        (y (cdr q))
        (z (car q)))
      (setf q y) z)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro is only good with interperative debugging. It fails
;;  for some reason when it is used with mapcar.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro poptart (node qname)
  `(with-accessors ((x ,qname)) ,node
    (let* ((y (cdr x)) (z (car x)))
      (setf x y) z)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro is good with the function mapcar.
;;  Qname must be quoted.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmacro mcpoptart (node qname)  ;; mc stands for mapcar
    `(let*
     (                           ;; this gets the fn def of accessor
      (com (fdefinition ,qname))  ;; comset is fn of (setf accessor)
      (comset (fdefinition (concatenate 'list (list 'setf ,qname))))
      (xx (funcall com ,node)) ;; xx is all the queue entries
      (y (cdr xx))  ;; y is the rest of the stack below top
      (z (car xx))) ;; z is the top of the stack
 (funcall comset y ,node) z ))  ;; this sets the function to what is left


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function truncates a list to length n removing the entries from
;;  the end of the list.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun truncate-lyst (lyst n)
  (cond
    ((null lyst) lyst)
    ((<= (length lyst) n)  lyst)
    (t (shorten-to-n lyst n)
    )
))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function shortens the lyst to length n provided that list is
;; long enough.  The previous function checks this.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun shorten-to-n (lyst n)
  (reverse
  (nthcdr (- (length lyst) n )
  (reverse lyst)
  ))
  )

;;;;;;;;;;;;; end of ggmacro.cl file ;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;file name ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This is ggclass.cl file -- for the CLOS definitions.
;; Most class definitions are contained in this file.
;;
;;;;;;;;;;;;;;;;;;;;;;;; file name ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


      (defvar clock 0)
      (defvar prediction-threshold 2)
      (defvar group-threshold 5)
      (defvar ggoal nil)
      (defvar rvarl 0)
      (defvar rvar2 0)
      (defvar rvar3 0)
      (defvar window-scale 3)
      (defvar pi 3.141592653589793)
      (defvar cloop-count 0)
      (defvar cloop-display t)
      (defvar junkheap nil)
      (defvar oldage   3)
      (defvar recheck-interval  5 )
      (defvar max-segment-length 13)
        (defvar KSQUEUES (list 'beam-queue  'segment-queue
                  ;; 'track-queue 'spline-queue  'merge-queue
                  ))
      (setq *print-pretty* nil) ;; to remove mismatch with old version Oct 92

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is a mixin object called ks-protocol-mixin.
;;  The mixin contains the functions and arguments to fire the
;;  KS's in phases.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass ks-protocol-mixin ()
  (
  (command :initarg :command :accessor command)
  (arglyst :initarg :arglyst :accessor arglyst)
  (anslyst :initarg :anslyst :accessor anslyst)
  (preboot :initarg :preboot :accessor preboot)
  (prelyst :initarg :prelyst :accessor prelyst)
  )
  (:documentation "This is a mixin-class")
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This  method writes to the output port.
;;  The method constructs the call to the output port from the
;;  slots in the ks-protocol-mixin mixin class.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmethod write-ks ((ele ks-protocol-mixin))
  (with-accessors
   ((com command) (alyst arglyst) (mess messenger)) ele
  (format t "COMMAND sent to ks ~a~%"  (cons com alyst))
 ;; (break " before-write-getassignment command executed above ~%")
  (format (write-port  ; get output port name from messenger object
         (messenger ele)  ; get messenger name from variable
         ) "~a~%"  (cons com alyst)) ; form function call
  (setf (channel ele)  -1)  ; change state of ksar to read
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;; The Knowledge Source Activation Records are the control structures,
;; which are used to store the requirement that a knowledge source
;; is to be run.  These form the elements of the priority queueing
;; system needed to run the BB.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass ksar (ks-protocol-mixin)
  (
  (priority :initarg :priority :accessor priority)
  (ksar-id :initarg :ksar-id :accessor ksar-id)
  (cycle :initarg :cycle :accessor cycle)
  (context :initarg :context :accessor context)
  (postboot :initarg :postboot :accessor postboot)
  (nodeptr :initarg :nodeptr :accessor nodeptr)
  (channel :initarg :channel :initform nil :accessor channel)
  (messenger :initarg :messenger :accessor messenger)
  )
  (:documentation " The knowledge source activiations records ")
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; This is the event class -- actually it is a queueing system
;;; with several subqueues.  The variable number represents the
;;; total number in all the subqueues and the mask represents
;;; the state of each of the subqueues.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defclass bbksarq ()
(
  (number :initarg :number :initform '() :accessor number )
  (mask :initarg :mask :initform '(1 1 1 1 1) :accessor mask)
  (atomic-queue :initarg :atomic-queue :initform '() :accessor atomic-queue)
  (beam-queue :initarg :beam-queue :initform '() :accessor beam-queue )
  (segment-queue :initarg :segment-queue :initform '() :accessor segment-queue )
  (track-queue :initarg :track-queue :initform '() :accessor track-queue )
  (spline-queue :initarg :spline-queue :initform '() :accessor spline-queue )
  (merge-queue :initarg :merge-queue :initform '() :accessor merge-queue )
          )
  (:documentation " This is the class of the ksar queue")
    )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Defmethod for constructing the parallel queues.
;;  The set-queue method is definitely terminology left over from
;;  flavors.  It takes a ksar and enqueues it in the proper subqueue
;;  of the ksar-queue.  Each of these queues is a priority queue
;;  so that they are enqueued according to a priority queueing
;;  discipline contained in the function queue-flavor...
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod set-queue ((ele bbksarq) (value ksar))
  (let*
   (
   (temp value)
   (test (ksar-id temp))
   )
(format t " inside method temp is ~a ~% " temp)
(format t " inside method test is ~a ~% " test)
  (cond
   ((equal test 'newhit)
```

```
    (queue-flavor-onto-node-at-attribute temp ele beam-queue) )
  ((equal test 'segment)
    (queue-flavor-onto-node-at-attribute temp ele segment-queue) )
   (t
    (queue-flavor-onto-node-at-attribute temp ele atomic-queue) )
   )
     ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This defmethod recalculates the slot numbor and updates the bbksarq.
;; Note this runs before each access of the slot numbor so that it
;; automatically updates the queue length information before each
;; access of that information.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmethod  numbor :before ((ele bbksarq))
(format t "~% ------ ENTERED DEFMETHOD BEFORE NUMBER KSARQ ------ ")
(format t "~% Total numbor in queue prior to set ~a ~%"
   (slot-value ele 'numbor))
;;  (setf (slot-value ele 'numbor)
  (setf (numbor ele)
  (apply '+ (mapcar #'(lambda (x) (length (funcall x ele)))
                 (cons 'atomic-queue KSQUEUES))))
(format t "~% ------ EXITING DEFMETHOD EVENT ------ ")
(format t "~% Total numbor in queue has been set to ~a ~%"
  (slot-value ele 'numbor))
      )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Defmethod mask is for constructing the status mask.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmethod mask :before ((ele bbksarq))
(format t "--- ENTERED defmethod MASK which is ~% ---")
(format t " The ksarq is ~%")
;; (describe ele)
  (let (
(xxx (format t "~% atomic-queue is ~a~%" (slot-value ele 'atomic-queue)))
(yyy (format t "~% beam-queue is ~a~%" (slot-value ele 'beam-queue)))
      (temp
       (mapcar #'(lambda (x)  ;; gets channel status
              (if (null x) nil (slot-value (car x) 'channel)))
         (mapcar #'(lambda(y) (slot-value ele y)) ;get lists
      (cons 'atomic-queue KSQUEUES))
              )));; limit mask size for now

       (setf (mask ele) temp)  ;; set the value of the bask
      (format t " Mask ele before exiting before method ~a~%" temp)
      (format t "----- EXITING DEFMETHOD before mask -----~%")
))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This puts a keyword into the keyworkd package so it can be used
;;  with an object.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro keywordize (sym) '(intern (symbol-name ,sym) *keyword-package*))
(defmacro my-keywordize (sym) '(intern ,sym *keyword-package*))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  These are the classes needed to define bbgoal instances.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; This is a superclass containing general information used as mixin class.

(defclass goal ()
 ((initiating-data-level :initarg :initiating-data-level :accessor level)
  (event-time :initarg :event-time :reader event-time)
  (purpose :initarg :purpose :accessor purpose)))

;; This is a superclass containing general information used as mixin class.

(defclass goal-attributes-mixin ()
 ((duration :initarg :duration :accessor duration)
  (source :initarg :source :accessor source :documentation "generating node"))
(:documentation "This is a goal-attribute-mixin for bbgoal class ")
  )

;; This is the main goal class with its mixin superclass.

(defclass bbgoal (goal-attributes-mixin goal)
  (
  (coord :initarg :coord :accessor coord)
  (number :initarg :number :accessor number)
  (threat :initarg :threat :initform nil :accessor threat)
  (snode :initarg :snode :initform nil :accessor snode)
  (ksarptr :initarg :ksarptr :accessor ksarptr)
   )
  (:documentation " This is a local data in the goal object ")
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This is the class object which makes up the levels of the
;; blackboard hierarchy.  Goals are to the left and data to the right.
;; The slot "up" is for high abstraction, "down" for lower data abstraction.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defclass bblevel ()
  (
(up :initarg :up :accessor up)
(left :initarg :left :accessor left)
(right :initarg :right :accessor right)
(down :initarg :down :accessor down)
 )
(:documentation " The bblevel constructs the levels of the bb")
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Node is basic node -- the class is superclass to all data nodes.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass node ()
  (
  (level :initarg :level :accessor level)
  (event-time :initarg :event-time :accessor event-time)
  )
  (:documentation "The node is superclass to all the data nodes")
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Bnode is for beam node -- the object holding info at the hit level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bnode (node)
  (
  (coord :initarg :coord :accessor coord)
  (numbor :initarg :numbor :accessor numbor)
  )
  (:documentation "The beam node is first level data panel object ")
  )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Defmethod which updates the slot numbor and generates a goal to update.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod initialize-instance :after ((ele bnode) &key)
  ;; (break "At init bnode to push goal on segment goal level ~%")
  (format t " '%Entered bnode after initial to put goal in segment level-%")
  (with-accessors ((num numbor) (crd coord) (evt event-time)) ele
  (setf num (length crd))
  (sendpushgoal
   (make-instance 'bbgoal
              :source     ele
              :purpose     'change
              :initiating-data-level    'hit
              :coord      crd
              :numbor     num
              :event-time evt
              :duration   'one-shot
              )
   segments)
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Unnode is partial beam node -- for the unmatched part of a hit node.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass unnode (node)
  (
  (coord :initarg :coord :accessor coord)
  (numbor :initarg :numbor :accessor numbor)
  )
  (:documentation "The beam node is first level data panel object ")
  )


(defmethod initialize-instance :after ((ele unnode) &key)
 (format t " '%Entered the initial queue'%'%")
 (format t " ~%message to eventq here ~%~%")
  (setf (numbor ele) (length (coord ele)))
  (sendpushgoal
   (make-instance 'bbgoal
              :source     ele
              :purpose    'unmatched
              :initiating-data-level      'hit
              :coord      (coord ele)
              :numbor     (numbor ele)
              :event-time  (event-time ele)
              :duration   'one-shot
              )
   segments)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Tnode = track node -- the class object holding info at the track level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass tnode (node)
  (
  (checklyst :initarg :checklyst :initform nil :accessor checklyst)
  (check :initarg :check :initform nil :accessor check)
  (cpa-bracket :initarg :cpa-bracket :accessor cpa-bracket)
  (threat :initarg :threat :initform nil :accessor threat)
  (snode :initarg :snode :initform nil :accessor snode)
  (last-velocity :initarg :last-velocity :accessor last-velocity)
  (last-coord :initarg :last-coord :accessor last-coord)
   )
  (:documentation "The tnode is class object on track level of data panel")
```

```
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This defmethod update the snodes hanging off a track level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (setf threat) :after  (new-slot-value (ele tnode))
  (mapcar #'(lambda (x) (setf (threat x) new-slot-value)) (snode ele)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This defmethod generates a goal node on the track level.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (setf event-time) :after  (new-slot-value (ele tnode))
  (format t "Entered the defmethod ~%")
    (sendpushgoal
         (make-instance 'bbgoal
          :source ele
          :purpose  'change
          :initiating-data-level    'track
          :event-time   new-slot-value    ;;(event-time ele)
          :threat  (threat ele)
          :snode  (snode ele)
          :duration 'one-shot)
        tracks)
       )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This defmethod activates the spline check after tracks have been
;; broken off from the main track.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (setf checklyst) :after  (new-slot-value (ele tnode))
  (format t "Inside defmethod of tnode to reset CHECKLYST ~%")
;;  (format t "  checklyst is ~a~%" (slot-value ele 'checklyst))
  (format t "  checklyst is ~a~%" new-slot-value)
  (if (null new-slot-value) ;; if all paths have been checked and modified
      (remove-nodes-from-level tracks (list ele))
      nil))   ;; otherwise dont reactivate spline tests



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This macro newclass generates a class and the appropriate after-
;;  methods forming a distributed monitor for the blackboard database.
;;  Class is the class name.
;;  Datalevel is the level the after-method must push the class node into.
;;  Slot-list is the list of slots to be included in the class.
;;  Monitor-list is the slots that when changed, are to generate goal nodes.
;;  Super-list is the list of inherited superclasses.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro newclass
 (class datalevel slot-list monitor-list super-list &rest options)
  (cons 'progn   ;; progn runs the sequence of programs created my macro
     (cons    ;; cons command program into the gigantic lisp program
  '(defclass    ;; first list form to construct is  the defclass
    ,class      ;; class name in the  defclass macro
    ,super-list ;; the list of inherited classes or mixin classes
    ,(do*       ;; do loop to construct all the accessors and initforms
      (
       (wlyst slot-list (cdr wlyst))  ;; cdr down the slot-list
         (op (car wlyst) (car wlyst))   ;; op is the next slot to be done
       (mylyst nil)                   ;; mylyst is list of slot-specifiers
       )                              ;; when slot-list is exmpty then
      ((null wlyst) (return mylyst))    ;; return the slot-specifier list
      (setq mylyst                      ;; construct each slot-specifier
```

```
        (cons                                ;; cons the slot name and initarg
         '(,op :initarg ,(keywordize op) ;; to make op ":op" op is put in
              :initform nil                   ;; keyword package, put default nil
              :accessor ,op)                  ;; make accessor function same as op
          mylyst)))                           ;; stuff this in the slot-specifiers
     ,@options)  ;; other options like documentation could be include
   (do*     ;; generate one method for each triggering report
    (
     (worklyst monitor-list (cdr worklyst)) ;; cdr down monitor list
     (op (car worklyst) (car worklyst))     ;; choose next candidate
     (mlyst nil)                            ;; construct list of methods
     )                                      ;; return method list
    ((null worklyst) (return mlyst))        ;; when monitor-list is empty
(setq mlyst                                 ;; construct list of defmethods
(cons  '(defmethod                          ;; cons defmethod into list
        (setf ,op) :after             ;; make an after method writer
             (new-slot-value (ele ,class))     ;; construct lambda list
             (sendpushgoal             ;; make body of after method
                 (make-instance 'bbgoal   ;; make-instance and push
                           :source ele     ;; initialize each value
                           :purpose 'change
                           :initiating-data-level  (level ele)
;;                         :variable ',op
                           :coord   (coord ele)
                           :numbor (numbor ele)
                           :event-time    (event-time ele)
                           :duration 'one-shot
                           )
             ,datalevel))    ;; specify level to push goal on
             mlyst)          ;; put methods into list
)))))


;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Snode is for segment node -- the object class holding info at the
;; segment level
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(newclass snode tracks (
                 coord   ; note this is a coordinate list
                 numbor  ; number of points the the segment
                 cpa    ; closet point of approach a vector
                 linear ; (position velocity)
                 tnode  ; ptr to a track node
                 threat ; true or false - updated  by tnode
                 )
           (numbor) (node)
  )



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This defmethod updates the attribute numbor.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmethod initialize-instance .:after  ((ele snode) &key)
(with-accessors ((num numbor) (lin linear) (c cpa) (evt event-time)
            (crd coord)) ele
  (setf (numbor ele) (length (coord ele)))
(format t " the numbor is ~a ~%" num)
(format t " the prediction threshold is ~a ~%" prediction-threshold)
  (cond
   ((< num prediction-threshold))
   ( t
   (format t " Started to calculated cpa and linear ~%" )
   (setf c  (find-cpa crd evt))
   (setf lin (find-linear-model crd evt))
   )
```

```
 )
 (format t " ABOUT TO EXIT DEMETHOD TO SET LINEAR 0)
 (describe ele)
 ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This defmethod truncates the coordinate and event-time lists
;; and also updates the cpa and linear slots.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (setf coord) :after (value (ele snode))
(with-accessors ((num number) (lin linear) (c cpa) (evt event-time)
           (crd coord)) ele
  (cond
   ((> (length crd) max-segment-length)
    (setq crd (truncate_lyst crd max-segment-length))
    (setq evt (truncate_lyst evt max-segment-length)))
   (t nil))
  (setq num (length crd))
(format t " the number is 'a ~%" num)
(format t " the prediction threshold is 'a ~%" prediction-threshold)
  (cond
   ((< num prediction-threshold))
   ( t
   (format t " Started to calculated cpa and linear ~%" )
   (setf c (find-cpa crd evt))
   (setf lin (find-linear-model crd evt))
   )
   )
   (format t " ABOUT TO EXIT DEMETHOD TO SET LINEAR 0)
   (describe ele)
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Initialize two levels on the blackboard and then connect them.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (setq hits
   (make-instance 'bblevel
            :down nil
            :left nil
            :right nil))

 ;;;

 (setq segments
   (make-instance 'bblevel
            :down nil
            :left nil
            :right nil))

 ;;;

 (setq tracks
   (make-instance 'bblevel
            :up nil
            :left nil
            :right nil
            )
   )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;     Link the blackboard levels hits <-> segments <-> tracks.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setf (down tracks) segments)  ; links top level to bottom level
(setf (up segments) tracks) ; links bottom level to top level
(setf (down segments) hits)  ; links top level to bottom level
```

```
(setf (up hits) segments) ; links bottom level to top level
(setq level-lyst (list tracks segments hits))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;     Initialize an instance of the ksar queue.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq ksarq (make-instance 'bbksarq))
(setq queue-lyst (list ksarq ))
;;
;;
;; End of ggclass.cl file.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;file-name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  File is ggksar.cl
;;;;;;;;;;;;;;;;;;;;file-name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Create-subgoals to break track.
;;  Spline check of the track has failed so the formation should
;;  be broken up and reformed.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-subgoals-to-break-track (trnode)
  (let
   (
    (snode (snode trnode))
    (pos   (last-coord trnode))
    (time  (event-time trnode))
    (velocity (last-velocity trnode))
    )
   (dolist (var snode)
         (sendpushgoal
          (make-instance 'bbgoal
                  :source trnode
                  :purpose 'verify-track
                  :initiating-data-level   'track
                  :event-time time
                  :coord (list pos velocity)
                  :duration 'one-shot
                  :snode var
                  )
          tracks)
         (setf (checklyst trnode) (snode trnode))
         )
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Create ksar to verify track test.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-verify-track-ksar (trnode segnode coords)
  (let
   (
    (tnode trnode)  ;; I don't know why I created this var
    (pos (car coords))
    (vel (cadr coords))
    (snode segnode)    ;; I don't know why I created this var
    )
   (sendksarpush
    (make-instance 'ksar
            :priority 0
            :ksar-id 'verify-track
            :postboot '(verify)
            :command 'verify   ;;; changed 21 Jan 92
            :nodeptr trnode
            :cycle clock
            :context (list tnode snode)
            )
    ksarq)
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Function verify is KS that checks the segment path against the
;;  average track trajectory using the angle and the distance between
;;  track members.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun verify (ksarptr)
  (let*
   (
    (trknode (nth 0 (context ksarptr)))
    (segnode (nth 1 (context ksarptr)))
    (tvec  (last-coord trknode))
    (tvel  (last-velocity trknode))
    (ttime (car (event-time trknode)))
    (svec  (car (coord segnode)))
    (stime (car (event-time segnode)))
    (dstime (- (car (event-time segnode))
             (cadr (event-time segnode))))
    (dsvec  (vector-difference (car (coord segnode))
             (cadr (coord segnode))))
    (svel  (scale-vector
          (/ 1.0 dstime) dsvec))
    (lystl (list tvec tvel ttime))
    (lyst2 (list svec svel stime))
    )
;;   (format outverify "~a~%" '(verify ',lyst ',lyst2))
(format t "~% INSIDE VERIFY INSIDE VERIFY INSIDE VERIFY ~%")
(format t " the STIME is 'a  and TTIME is ~a ~%" stime ttime)
(format t " the svec is 'a  and tvec is 'a  ~%" svec tvec)
(format t " the svel is 'a  and tvel is 'a  ~%" svel tvel)
   (cond
    ((ksverify lystl lyst2)
(format t " the snode is ~a is REMAIN PART tnode is ~a ~%" segnode trknode)
     nil) ; if the tracks are paired correctly
    (t ;; if track should be broken - rip out segments
(format t " the snode is 'a  is RIPPED from tnode is 'a  ~%" segnode trknode)
      (setf (tnode segnode) nil) ;; removes pointer to track-node
      (setf (snode trknode)
           (remove segnode (snode trknode)))  ;; remove ptr to snode
      (setf (checklyst trknode)  ;; forces an and of children:
           (remove segnode (checklyst trknode)))
       ))
(format t "~% END VERIFY END VERIFY END VERIFY ~%")
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Returns true only if the paths craft are within one unit time travel
;; and the cosine of the angle between the velocity vectors > 0.9
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun ksverify (lystl lyst2)
  (let*
   (
    (tvec (nth 0 lystl))
    (tvel (nth 1 lystl))
    (ttime (nth 2 lystl))
    (svec (nth 0 lyst2))
    (svel (nth 1 lyst2))
    (stime (nth 2 lyst2))
    (tmax (max ttime stime))
    (tnewvec (vector-sum tvec (scale-vector (- tmax ttime) tvel)))
    (snewvec (vector-sum svec (scale-vector (- tmax stime) svel)))
    (deldist (vector-magnitude (vector-difference tnewvec snewvec)))
    (maxvel  (max (vector-magnitude svel) (vector-magnitude tvel)))
    (cosangle (vector-angle-cosine svel tvel))
    )
   (format t " ~% INSIDE KS VERIFY  INSIDE KS VERIFY  INSIDE KS VERIFY ~%")
   (format t " dist end points 'a  , max dis in 1 unit ~a~%" deldist maxvel)
   (format t " cosangle between velocity vectors 'a  ~%" cosangle)
   (and
    (> maxvel deldist)
    (> cosangle 0.9))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,
;;  The lyst argument consists of the following terms :
;;  0. bbnode id which goes into the trigger node variable.
;;  1. The ks which is to be invoked.
;;  2. The type of node or level it came from.
;;  3. Entry identifies the entities that follow.
;;  4. List of the values of the variable designed by 3
;;  5. Number of entities or length of queue of entities.
;;  6. Time stamp.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,
;;
;;  This function constructs the ksar and pushes it into the ksar-queue
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,

(defun create-update-segments-ksar (lyst)
;; (break  "ENTERED CREATE-UPDATE-KSAR ~%")
  (sendksarpush
  (make-instance 'ksar
             :priority 1
             :ksar-id 'segment
             :postboot '(post-assign-hits)
             :nodeptr (car (last lyst))
             :cycle clock
             :context (list
                  (list 'event-time (nth 6 lyst))
                  (list 'numbor (nth 5 lyst))
                  (list 'coord (nth 4 lyst)))
             :channel 2
             :command 'getassignment
             :arglyst '()
             :anslyst '()
             :messenger assignmsg
             :preboot '(pre-assign-hits)
             :prelyst '()
             )
   ksarq)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  This function create-check-track-ksar which checks if the track
;;;  is a threat and, if so, marks the object accordingly.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-check-track-ksar (lyst)
  (sendksarpush
  (make-instance 'ksar
             :priority 0
             :ksar-id 'spline
             :postboot '(assign-threat)
             :command 'assign-threat   ;; changed 21 Jan 92
             :nodeptr (car (last lyst))
             :cycle clock
             :context (list
                  (list 'snodes (nth 4 lyst))
                  (list 'event-time (nth 2 lyst))
                  (list 'tnode (nth 5 lyst)))
             :channel 1
             )
   ksarq)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,
;;  Function create-update-tracks-ksar pushes a goal onto goal BB
;;  to update the track.
;;  This causes information to percolate up from
;;  the supporting segments.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,

(defun create-update-tracks-ksar (lyst)
```

```
;;(break "***** BREAK - JUST ENTERED UPDATE-TRACKS-KSAR ****** ~%")
  (sendksarpush
  (make-instance 'ksar
            :priority 0
            :ksar-id 'track
            :postboot '(assign-tracks)
            :command 'assign-tracks   ;; changed 21 Jan 92
            :nodeptr (car (last lyst))
            :cycle clock
            :context (list
                 (list 'event-time (nth 6 lyst))
                 (list 'numbor (nth 5 lyst))
                 (list 'coord (nth 4 lyst)))
            :channel 1
            )
   ksarq)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Create-newhit-ksar function creates a goal to get another
;; set of radar returns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-newhit-ksar (lyst)
  (sendksarpush
  (make-instance 'ksar
            :priority 2
            :ksar-id 'newhit
            :postboot '(getbeam)
            :cycle clock
            :context 'none
            :channel 1
            :command 'fire
            :arglyst '()
            :anslyst nil
            :messenger beammsg
            )
   ksarq)
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function creates all the different type of ksars.
;;  Note that this argument lyst can be extended considerably to aid
;;  in the mapping process for later expansion.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-ksar (lyst)
  (format t "just entered create-ksar ~%")
  (format t " the ksar lyst upon entry to create-ksar is ~a~%" lyst)
  ;; (break "Just entered the create-ksar and argument of call given above ~%")
  (cond
   (
    (equal (nth 0 lyst) 'newhit)
    ;(format t "about to create a newhit ksar ~%")
    (create-newhit-ksar lyst))
   ((equal (nth 0 lyst) 'change)
   ;(format t "about to create a change ksar ~%")
    (cond
     ((equal (nth 1 lyst) 'hit)
      (create-update-segments-ksar lyst))
     ((equal (nth 1 lyst) 'segment)
      (create-update-tracks-ksar lyst))
     ((equal (nth 1 lyst) 'track)
      (create-check-track-ksar lyst))
     (t
      (format t "++++++++ ERROR - UNKNOWN CHANGE KSAR TYPE +++++++++~%"))
     ))
   (t
      (format t "++++++++ ERROR - UNKNOWN KSAR TYPE +++++++++~%"))
```

```
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function gets the assignments of data to segments
;;  when given two lists of coordinates by passing the problem
;;  off to the assignment KS.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun getassignment (lystl lyst2)
;; (break "***** BREAK - JUST ENTERED GETASSIGNMENT ****** ~%")
(format t "read assign ~a~%" (read assign))
; note this is read the prompt of rewl supplied by franz
  (format t "COMMAND SENT ~a~%" '(getassignment ',lystl ',lyst2))
  (format assign "~a~%" `(getassignment ',lystl ',lyst2))
  ;;(format assign "~a~%" '(getassignment))
  ;;(format assign "~a~%" lystl)
  ;;(format assign "~a~%" lyst2)
 ;; (read assign)  ; reads the answer a list like (0 2 1) to compare to (0 1 2)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function assigns the incoming hits to existing segments.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Note that in using KS getassignment, the first argument is the row
;; of the distance array and the second is the columns of the distance
;; Two cases: 1. More segments than hits -- segments are rows
;;            2. More hits than segments -- hits make up rows
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun assign-hits (ksarptr)
  (cond
   ((zerop (get-number-on-level segments))
    (cond (     ; if there are no segments - initialize one for each hit
     (> (get-number-on-level hits))
    (dolist (var (get-hits-coord))
     (sendpushlevel (make-instance 'snode
                          :level 'segment
                          :coord (list var)
                          :event-time (list (get-hits-time)))
               segments))
     (fifodequeue hits))
    (t (format t "ERROR in the nodes on the hit level"))))
   (t (cond ((zerop (get-number-on-level hits))
        (format t "ERROR in the nodes on the hit level"))
     (t (let* ;otherwise match hits to the segments by using  b&b algorithm
       (
     (time (get-hits-time))
     ;; test statement to remove updates to segments older than 3
     ;; time units
     (snodelyst (get-recent-segments time))
     (lystl (get-segments-coord-with-time-for-nodes- snodelyst))
     (lyst2 (get-hits-coord-with-time)) ; forms list of (t x y z)
     (lyst3 (get-hits-coord)) ;forms list (x y z)
     (temp
      (if (>= (length lystl) (length lyst2))
            (getassignment lystl lyst2)
            (getassignment lyst2 lystl)
            )))

     (format t "~% +++++++ order of getassignment +++clock is~a   +++" clock)
     (format t "~%  the lystl is~a   " lystl)
     (format t "~% the order info in temp is~a   " temp)
     (setq junkheap (list 'lystl lystl 'temp temp
                      'lyst2 lyst2 'snodelyst  snodelyst))

     (update-segment-coord-and-time temp snodelyst lystl lyst3 time)

;;  here is where one must take the set difference from the new data
;;  points in order to insert a goal which accounts for unmatched
```

```
;;  data

      (if (< (length lystl) (length lyst2))
     (create-goal-for-unmatched-hit-&ta  temp lyst2)
     nil)


      (fifodequeue hits)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Function gettrack interfaces with the KS -- writes command to gettrack
;; KS giving the cpa and the position to the KS and the KS returns the
;; confidence interval and assessment of the threat.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun gettrack (cpa vector)
;;  (read track)
(format t "~%TRACK  TRACK  TRACK  TRACK  TRACK  TRACK  ~% ")
(format t "~a~%" `(gettrack ,cpa ,vector))
(format t "~%TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT~%")
;; (break  "** Here is the place to check for prompt **")  ;; changed 30 Dec 91
  (format track "~a~%" `(gettrack ',cpa ',vector))
;; (format outtrack "~a~%" '(gettrack))
;;  (format outtrack "~a~%" cpa)
;;   (format outtrack "~a~%" vector)
;;  (progl (list (read track)(read track)) (read track))
  (read track)
  )
;;
;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function puts in the precondition material to freeze the context
;;  and hold the information in the KSAR.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun pre-assign-hits (ksarptr)
  (cond
     ((zerop (get-number-on-level segments))
     (cond (   ; if there are no segments - initialize one for each hit
     (> (get-number-on-level hits))
     (dolist (var (get-hits-coord))
     (sendpushlevel (make-instance 'snode
                         :level 'segment
                         :coord (list var)
                         :event-time (list (get-hits-time)))
               segments))
      (fifodequeue hits)
     (poptart ksarq segment-queue) )
      (t (format t "ERROR in the nodes on the hit level"))))

   (t (cond ((zerop (get-number-on-level hits))
             (format t "ERROR in the nodes on the hit level"))
         (t (let* ;otherwise match hits to segments by b&b algorithm
               (
             (time (get-hits-time))
      ;; test statement to remove updates to segments older than 3
      ;; time units
             (snodelyst (get-recent-segments time))
             (lystl (get-segments-coord-with-time-for-nodes-ysnodelyst))
             (lyst2 (get-hits-coord-with-time)) ; forms list of (t x y z)
             (lyst3 (get-hits-coord)) ;forms list (x y z)
             (temp (if (>= (length lystl) (length lyst2))
              `(',lystl ',lyst2) `(',lyst2 ',lystl)))
             )
;;     (format t "Inside pre-assignment function - arglyst is ~a~%" temp)
         (setf (arglyst ksarptr) temp)
```

```
          (setf (prelyst ksarptr)
             (list snodelyst lystl lyst2 lyst3 time))
             ))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This is the post condition for assign-hits which will be fired from the
;;  normal postboot slot of the KSAR with arglyst.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun post-assign-hits (ksarptr)
(format t "~% *********** ENTERED POST-ASSIGN-HITS ****************~%")
  (let*
   (
     (temp (anslyst ksarptr))
     (lyst (prelyst ksarptr))
     (snodelyst (nth 0 lyst))
     (lystl (nth 1 lyst))
     (lyst2 (nth 2 lyst))
     (lyst3 (nth 3 lyst))
     (time  (nth 4 lyst))
     )
;;   (format t "~% +++++++ order of getassignment +++clock is ~a +++" clock)
;;   (format t "~% POST ASSIGN  POST ASSIGN
;;        POST ASSIGN  clock is ~a +++" clock)
;;   (format t "~% the snodelyst is 'a  " snodelyst)
;;   (format t "~% the lystl is ~a " lystl)
;;   (format t "~% the lyst2 is ~a " lyst2)
;;   (format t "~% the lyst3 is 'a  " lyst3)
;;   (format t "~% the time is 'a  " time)
;;   (format t "~% the order info in temp is ~a " temp)
     (setq junkheap (list 'lystl lystl 'temp temp
                    'lyst2 lyst2 'snodelyst snodelyst))

     (update-segment-coord-and-time temp snodelyst lystl lyst3 time)

;;  here is where we must take the set difference from the new data
;;  points inorder to insert a goal which accounts for unmatched
;;  data

       (if (< (length lystl) (length lyst2))
     (create-goal-for-unmatched-hit-data temp lyst2)
     nil)


       (infodequeue hits)
       ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function interfaces with the spline KS written in C.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun getspline (snodeptr)
;;  (dribble "outpour")
(format t "~% ************* ENTERED GETSPLINE *********** ~%")
  (let*
   (
   (lystt (event-time snodeptr))  ;; this gets time list
   (lystl (coord snodeptr)) ;; this gets coord list
   (lystt (first-n-elements 4 lystt)) ;; get last 4 time pts
;;   (xx (format t "lystt is ~a~%" lystt))
   (lystl (first-n-elements 4 lystl)) ;; get last 4 pos pts
;;   (xx (format t "lystl is ~a~%" lystl))
   )
;;  (break "~% about to send the first list ~%")
;;   (format t "~a~%" lystt)
  (format spline "~a~%" lystt) ;; send ks time instances
  ;; (break "~% about to receive the first list back from spline~%")
  (format t "~a~%" (read spline))
```

```
;;  (break "~% about to send the second list ~%")
;;   (format t "second list is ~a~%" lystl)
  (format spline "~a~%" lystl) ;; send ks pos. instances
;; (break "~% about to rcv the second list from spline process ~%")
  (do (
       (work (read spline) (read spline)) ;; read coefficients
       (lyst nil (cons work lyst)) ;; continue reading coef x y z
      )
      ((null work) (return
                (cddddr (reverse lyst)))) ;; reverse to get in order
  ))
;;  (break "~% about to exit getspline ~%")
;;  (dribble)
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function compares two spline representations.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun compare-spline-models (sptrone sptrtwo )
  (format t " IN COMPARE-SPLINE-MODELS ~a~a~% " sptrone sptrtwo)
  ;; (break "******** COMPARE-SPLINE-MODELS *********~%")
  (let*
   (
    (lystone (getspline sptrone )) ; this gets the spline representation
    (lysttwo (getspline sptrtwo )) ; for both nodes
;;
;;   (format t "~% ============================================~%")
;;   (format t " SPLINE  SPLINE  SPLINE  SPLINE  SPLINE  SPLINE ~%")
;;   (format t "first set of coefficients is ~% ~a ~%" lystone)
;;  (format t "second set of coefficients is ~% ~a ~%" lysttwo)
     (dist (apply +   ; this just sums the abs diff of coef for alltimes
       (mapcar 'vector-absolute-difference
            lystone lysttwo))))
   (format t " ERROR between pairs is 'a'%  " dist)
   (format t " Group threshold is ~a~% " group-threshold)
   (format t "~% END SPLINE END SPLINE END SPLINE END SPLINE ~%")
   (format t " =========================================~%")
    dist
  ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function creates a segment for an unmatched hit.
;; It does no checking since it has already been confirmed that
;; there are no matching segments.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-unmatched-hit-ksar (bnodeptr)
  (let*
   (
   (coordinates (coord bnodeptr))  ; copy the coords
   (time (event-time bnodeptr))   ; remove time part of the coordinate
   (numbor (length coordinates))
   )
     (sendpushlevel (make-instance 'snode
                         :level 'segment
                         :coord coordinates
                         :numbor numbor
                         :event-time  (list time)   )
              segments)
   (remove-data-x-from-level-y bnodeptr hits)
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function grabs a sample information from the KSAR
;; after getbeam KS has been run.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun getbeam (ksarptr)
;;  (format beam "(fire)~%")
  (let* (
    (temp (anslyst ksarptr))  ; read in the answer list
    (count (nth 0 temp))  ; first entry of the list
    (timestamp (nth 1 temp)) ; second timestamp of list
    (coord (nth 2 temp))     ; third entry of list
    (xnode (make-instance 'bnode :coord coord
       :level 'hit :numbor count :event-time timestamp)))

   (format t "Entered getbeam and coord is ~a~%" coord)
   (format  t "About to enter drawtrack ~%")
   ;;(break)
;;   (mapcar #'(lambda (x) (drawtrack (*  window-scale (car x))
;;                           (*  window-scale (cadr  x)))) coord)

;;
;;  Enter the data point in the data file for plotting
;;
;;    (dolist (ele coord)
;;         (format outdata "'a   'a   "
;;            .    (nth 0 ele)
;;                 (nth 1 ele)))
;;    (format outdata "~%")
   (cond
    (count
;;     (format t " count is ~a~% " count)
;;     (format t " timestamp is 'a'%   " timestamp)
;;     (format t " coord is ~a ~% " coord)
;;     (format t " beam node is ~a ~% " xnode)
      (setf (right hits)
         (cons xnode (right hits))))
    (t nil))))
```

```
;;;;;;;;;;;;;;;;;file        name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;* File is ggnode.cl .
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function puts a true or false in those nodes that
;; are threats and have more than one segment associated with tnode.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun assign-threat (ksarptr)
  (let*
   ((nodelyst (cadar (context ksarptr))) ; get seg set list
    (testlyst (find-closest-set nodelyst))  ; find spline fit set
    (test (samesetp nodelyst testlyst)) ; compare track lists
    )
   (format t "~% ASSIGN THREAT ASSIGN THREAT ASSIGN THREAT ~%")
   (format t "Original set from context is ~a~%" nodelyst)
   (format t "Test verified set from splines is 'a'%" testlyst)
   (format t " Result of sameset predicate ~a~% " test)
   (format t " ------------------------------------------ ~%")
   (setf (check (nodeptr ksarptr))
;; (if test (car (event-time (nodeptr ksarptr))) 'failed)  changed 12 Oct 92
   (if test (car (event-time (nodeptr ksarptr))) 'failed) ;;changeback 25 Oct 92
;;   (if test t 'failed) ;; changed back 25 Oct 92
      ))) ; set check


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Update-segment-coord-and-time function is the goal bb version of
;; update-segment-coord and it handles both cases when
;; 1. # segments >= # hits
;; 2. # segments < # hits.
;; It also assigns the coordinates to the proper segments after
;; the assignment problem has been solved and stores the solution
;; in the permutation vector.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun update-segment-coord-and-time (order snodelyst segcoord hitcoord time)
  (cond
   ( (>= (length segcoord) (length hitcoord)) ;more segs than hits
     (do*
      (
       (nlyst order (cdr nlyst))  ; permutation of segments
       (flyst snodelyst) ; flavor lyst of segments
       (clyst hitcoord (cdr clyst))  ; coordinate lyst
       )
      ((null nlyst)
       (format t "~% Segments are updated, TIME IS ~a " clock))
      (let
       (
      (snode (nth (car nlyst) flyst))
      (value (car clyst))
      )
      ;; update the time and then the coord values of snode
      ;; time for the attribute must be event-time now prk 29 Dec 91
       (push-value-onto-node-at-attribute time snode event-time)
       (setf (coord snode)
           (mypush value (coord snode)))
      )))
   ;;;;;;;      more hits than segments
   (t
    (do*
     (
      (nlyst order (cdr nlyst))  ; permutation of hits
      (flyst snodelyst (cdr flyst))
      (clyst hitcoord)
      )
```

```
      ((null nlyst)
        (format t "~% Segments are updated, TIME IS 'a " clock))
      (let
       (
        (value (nth (car nlyst) clyst))
        (snode (car flyst))
        )
       ;; update the time and then the coord values of snode
       ;; must replace attribute time with event-time prk Dec 29, 1991
        (push-value-onto-node-at-attribute time snode event-time)
        (setf (coord snode)
            (mypush value (coord snode)))
       )))
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function generates a hit goal to account for those
;; data points that are not matched to the current segments.
;; Lystx is list of coordinates with time in the original data hit
;; and intset is the assignment of segments to data.
;; This function should only be applied when there are more hits
;; then current segments to match to the hits.
;; Numset generates integer set (n-1,n-2,...,0) for input n.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-goal-for-unmatched-hit-data (intset lystx)
  (format t "~% intset and lyst2 ~% ~a ~% ~a" intset lystx)
  (let*
   ((nset (set-difference (numset (length lystx)) intset))
    (time (caar lystx))          ; copy time from one coord
    (number (length nset))       ; number of coord's unmatched
    (wlyst (mapcar 'cdr lystx))  ; removes the time from all coord
    (lyst nil)                   ; initialize the lyst
    )
   ;; note the sort is used keep the order of the wlyst
   ;; unaltered
   (dolist (var (sort nset '>) lyst)
           (setq lyst (cons (nth var wlyst) lyst )))
  (sendpushlevel                 ;pushes node onto data side
   (make-instance 'unnode
             :level      'unmatched
             :coord      lyst
             :numbor     number
             :event-time  time
             ) hits)
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Creates-new-track-node is a function which generates the track node
;; from the segment node when there is no tracks on the track level.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-new-track-node (segnode)
(format t "****** ENTER CREATE-NEW-TRACK-NODE ********~%")
;; (break "Time to switch to dribble ~%")
  (let*
   (
    (temp (gettrack (cpa segnode) (car (linear segnode))))
    (if (null temp) (format t "~% null sequence returned from gettrack~% ")
    (format t "~% gettrac returns NON NULL SEQUENCE~% "))
    (format t "gettrack returns ~a~%" temp)
    (intervals (car temp))
    (format t "intervals is ~a~%" intervals)
    (threat (cadr temp))
    (format t "threat is ~a~%" threat)
    (trknode
     (make-instance 'tnode
              :level 'track
```

```
                :event-time (list (car (event-time segnode)))
                :snode (list segnode)
                :cpa-bracket intervals
                :threat threat
                :last-coord (car (linear segnode))
                :last-velocity (cadr (linear segnode))
                )
      )
     )
     (format t "gettrack returns ~a~%" temp)
     (format t "intervals is ~a~%" intervals)
     (format t "threat is ~a~%" threat)
    (setf (tnode segnode) trknode)  ; establish forward ptr
    ;; changed 15 Feb 92 added back in on Oct 92
 ;;  (setf (threat segnode) threat)  ; establish threat contribution
    (setf (threat segnode) threat)  ; establish threat contribution
    (format t " About to Push a Node on the Track File ~%")
    (sendpushlevel trknode tracks)
    ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Assign-tracks is the postboot to the knowledge source which refines
;; the segment into a new or established track.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun assign-tracks (ksarptr)
    ;;
    ;; first if there are no tracks - create the one for one with
    ;; the segments so far
    ;;
  (cond
   ((zerop (get-number-on-level tracks))
    (cond
     ((plusp (get-number-on-level segments))
      (mapcar I' create-new-track-node (right segments))
      (merge-tracks) )
     (t
      (format t "ERROR no segment to construct tracks ~%"))))

;;
;;  if the segment already associated with track node
;;

    ((tnode (nodeptr ksarptr))
     (update-track ksarptr)
     (format t "Updated TRACK NODE ~% "))

;;
;; segment node is not associated with track node - so it
;; the tracks will have to be reformed
;;

   ( (copy-data-nodes-at-level-y-satisfying-predicate-f
     segments I' (lambda (x) (null (tnode x))))
    (format t "~% KSAR CREATING NODE follows: ")
    (describe ksarptr)                         ;(*break t 'look)
      (mapcar #'create-new-track-node      ;put these on the blackboard
    (copy-data-nodes-at-level-y-satisfying-predicate-f
     segments #'(lambda (x) (null (tnode x)))))
    (merge-tracks))
   (t (format t "~% ERROR - Fallen through inside assign-tracks"))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Find-formation function detects when two tracks are "close"
;; to each other.  "Close" is defined to mean traversing almost parallel
;; tracks and traversing within time unit of distances using the max
;; speed of both craft.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun find-formation (tnode1 tnode2)
  (let*
   (
   (pos1 (last-coord tnode1)) ;pos of the first track
   (pos2 (last-coord tnode2)) ;pos of the second track
   (vel1 (last-velocity tnode1)) ;vel of the first track
   (vel2 (last-velocity tnode2)) ;vel of the second track
   (mag1 (vector-magnitude vel1))   ; mag of velocity 1
   (mag2 (vector-magnitude vel2))   ;; mag of velocity 2
   (mag  (max mag1 mag2))
   (dp (vector-magnitude (vector-difference pos1 pos2)))
   (cosangle (vector-angle-cosine pos1 pos2))
   )
; is the angle between tracks small ??
 (format t "~% INSIDE FIND-FORMATION  INSIDE FIND-FORMATION  ~%")
 (format t " distance is 'a  and the mag of velocity is 'a  ~%" dp mag)
 (format t " cosine angle is 'a  %  " cosangle)
  (cond
   ((and (> cosangle 0.9) ;; if angle small enough
      (> mag dp)) t)  ;;  and the distance is within 1 unit of travel
   (t nil)    ;; otherwise do not associate this pair
   )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Find-nearby-tracks function returns a list of nearby tracks
;; to the given craft from the list of other craft.
;; Note that a given craft always is a nearby track of itself.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-nearby-tracks (ele lyst)
  (cond
   ((null lyst) (list ele)) ;; return element of lyst is empty
   (t
  (cons ele  ;;include itself
     (mapcan   ;;; test each track in against lyst to see if it is
      #'(lambda (x) (if (find-formation ele x) ; it is close enough
                 (list x) nil))
      lyst)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The following function returns a list of subsets of nearby tracks.
;; The function creates sets of tracks using the function
;; find-nearby-tracks to construct sets.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-track-subsets ()
  ;(format t "~% ENTERED find-track-subsets ~%")
  (do*
   (
    (tlyst (right tracks)  ; gets all the tracks
         (our-set-difference tlyst (car endlyst))) ;remove ones matched
    (tindex (car tlyst) (car tlyst))  ; start with first one and find ones
    (endlyst nil (if tindex          ; in the same equivalence class
                (cons (find-nearby-tracks tindex (cdr tlyst)) endlyst)
               endlyst))
    )
   ((null tlyst) (return endlyst)))) ;endlyst is a list of lists
                              ; each list is the eqivalence class

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The function make-merged-nodes merges tracks
;; and replaces the track objects with the newly grouped or
;; associated set of tracks.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-merged-nodes (tsets)
  (format t "ENTERING MAKE-MERGED-NODES ----------~%")
;; (break "Inside make-merged-nodes about to construct another tnode ~%")
  (format t "The set of tsets is 'a  ~%" tsets)
```

```
(do*
 ((ttwork tsets (cdr ttwork))
  (twork (car ttwork) (car ttwork)))
 ((null twork) (format t "MERGED TRACK NODE CREATED~%"))
 (format t "~% the set of sets is tsets = ~a ~%" tsets)
 (format t "~% the set of nodes being merged is ~a ~%" twork)
 (let
  (
   (temp
    (make-instance 'tnode
               :level 'track
               :event-time (list
                    (average ;; get-track event-time 31 Dec 1991
                     (mapcar 'car (get-track event-time twork))))
               :snode
               (apply 'append (get-track snode twork))
               :cpa-bracket (list
                      (union-intervals
                          (get-track-x-intervals twork))
                      (union-intervals
                          (get-track-y-intervals twork))
                      )
               :threat (cond
                 ( (format t "~a~%" (get-track threat twork))
                 (null (get-track threat twork)) nil )
                 (t (eval (cons 'or
                          (get-track threat twork)))))
               :threat (apply 'or (get-track threat twork))
               :last-velocity (vector-average
                      (get-track last-velocity twork))
               :last-coord (vector-average
                      (get-track last-coord twork))
               )))
   ;; (mapcar #'(lambda (x) (setf (tnode (car (snode x))) temp)) twork)
   ;; going to do all the snode pointers, 6 Jan 92 ,replaced by following
;; (format t " ------- NEW NODE CREATED exam new node ------ ~%")
;; (format t "clock is = ~a ~%" clock)
;; (format t " twork is given by ~a~%" twork)
;;  (format t " temp is given by ~a~%" temp)
;;  (describe temp)

   (dolist (elyst (snode temp))
;; (format t "Setting the parent pointers for snode = ~a ~%" elyst)
;; (format t "Parent pointer = ~a ~%" twork)
    (setf (tnode elyst) temp)
;; (format t "Check what pointers are there in snode ~%" )
;; (describe elyst)
     )
   ;; end replacement
 ;;   (break "INSIDE MAKE-MERGED NODES before removal of old tracks ~%")
   (remove-nodes-from-level tracks twork) ; remove tracks grouped
   (sendpushlevel temp tracks) ; replace with new group
   )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Merge-tracks function is the top-level track formation program
;; that first groups the segments into formations and then
;; replaces the track objects with the consolidated tracks.
-------------------------------------------------------------------

(defun merge-tracks ()
  (let*
   ( (tsets (find-track-subsets)) );creates the groups
   ;; (break "Here is where to evaluate tsets")
   (format t "+++++ INSIDE merge-tracks +++++++++ ~%")
   (format t " tsets are ~a~%" tsets)
;;   (mapcar #'(lambda(x) (describe x)) (car tesets))
   (make-merged-nodes tsets)    ;creates an equivalence node
  (format t "TRACKS MERGED ~%")
  ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The update-track function updates the tracks from the segment nodes
;; and allows the updated information to percolate up the hierarchy.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun update-track (ksarptr)
  (let•
   (
    (snodeptr (nodeptr ksarptr)) ; gets the snode class for ksar
    (tnodeptr (tnode snodeptr))  ; get the tnode class from snode
    (lyst (linear snodeptr))     ; reads in the linear model
    (segp (get-position lyst))      ; tears out the position info fm linear
    (segv (get-velocity lyst))      ; tears out the velocity info fm linear
    (trkpos (last-coord tnodeptr))    ; get track position from track node
    (trkvel (last-velocity tnodeptr)) ;obtain track velocity fm trk node
    (dt (- (car (event-time snodeptr))
           (car (event-time tnodeptr))
           )) ;determine the time difference between snode and tnode
    (n (length (snode tnodeptr))) ; number of snodes in a track
    (s1 (- 1.e0 (/ 1.e0 n)))        ; convex weight 1/number in tracks
    (newtime (car (event-time snodeptr))) ; want the latest time on track
    (temp (gettrack (cpa snodeptr) ; want the threat and intervals
                (car (linear snodeptr))))
    (intervals (car temp))  ;;tears out the interval information
    (newthreat (cadr temp)) ;;tears out the threat information
    (txint (get-track-x-intervals (list tnodeptr))) ;get snode x-interval
    (tyint (get-track-y-intervals (list tnodeptr))) ;get snode y-interval
    (txint (cons (car intervals) txint)) ; collect the x intervals
    (tyint (cons (cadr intervals) tyint)) ; collect the y intervals
    )
   (if (minusp dt)
       (format t "~% ERROR INSIDE UPDATE-TRACK TIME DIFF IS NEG "))
;   (format t "~% Inside update-track ")
;   (format t "~% segp and trkpos are ~a ~a " segp trkpos)
;   (format t "~% length n is 'a  and s1 is ~a " n s1)
;   (format t "~% dt is ~a and trkvel is ~a " dt trkvel)
   (cond
    ((zerop dt) (setf (last-coord tnodeptr)  ; just average positions
                  (convex-vector-average s1 trkpos segp)))
    (t
     (setf (last-coord tnodeptr)    ; just average updated positions
           (convex-vector-average s1 segp
                     (vector-sum trkpos (scale-vector dt trkvel))))
     (setf (last-velocity tnodeptr) segv)))

    (setf (event-time tnodeptr) (list newtime))   ; update the time

   (setf (cpa-bracket tnodeptr)    ; update the cpa intervals
      (list (union-intervals txint)
            (union-intervals tyint)))

   (setf (threat tnodeptr)      ; update the threat accessment
      (eval (cons 'or
            (list newthreat
                  (car
                   (mapcar I' (lambda (x) (threat x))
                        (our-set-difference
                      (snode tnodeptr)
                      (list snodeptr))
                       ))))))
   ))

(defun get-velocity (p1) (cadr p1))
(defun get-position (p1) (car p1))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Find-closest-pairs function returns the tracks that are close enough
;;  to be grouped with the segment represented by snode.
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-closest-pairs (snode lyst) ;assumes snode in lyst
  (format t " in find-closest-pairs of ~a in set 'a'%  - - " snode lyst)
  (do*
   (                                          ;compare to others in group
    (worklyst (our-set-difference lyst (list snode))
           (cdr worklyst)) ; do it one at time
    (xxx (format t "~% DIFFERENCE set (should exclude snode) is ~a~%" worklyst))
    (dlyst nil) ; cummulate those tracks are close enough in dlyst
    )
   ((null worklyst) (return dlyst)) ; returns lyst with snode included

    (format t "CLOSEST PAIRS for snode 'a and first ele in worklyst  ~a ~%"
         snode (car worklyst))
   (if (<  (compare-spline-models snode (car worklyst)) group-threshold)
       (setq dlyst (cons (car worklyst) dlyst))
      nil )
    (format t " the dlyst as it builds  ~a~%" dlyst)

   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  The function find-closest-set returns the group check of the tracks.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-closest-set (lyst)  ; returns the lyst of grouped tracks
  (format t " --- inside - find-closest-set of the list  ~a~% ----" lyst)
  (let ((ulyst nil))  ; initialize union lyst as nil
       (dolist (ele lyst ulyst)
       (setq ulyst (our-union (find-closest-pairs ele lyst) ulyst))
  (format t " Solution as it builds up, ele is single track ~%")
  (format t " ele lyst and ulyst are ~a~% ~a~% ~a~% " ele lyst ulyst)
           )))
;;
;;
;;;;;;;;;;;;;;;;;;;end        of ggnode.cl file ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;; file name ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  File is ggplan.cl
;;;;;;;;;;;;;;;;;;;;; file name ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function separates the if and then parts then
;; tests the if part and if true evaluates the then part.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun runrule (node rule)
;,  (format t "~% -------  entered runrule ----------"]
;;  (format t "~% node value is 'a  " node)
;;  (format t "~% rule is 'a  " rule)
 (setq*ggoal node) ;; need global variable to construct code via subst
  (let
   (
    (ifs   (subst 'ggoal 'gnode  (car(cdaddr rule))))
    (thens (subst 'ggoal 'gnode  (cadr(cadddr rule))))
    )
;;    (format t "~% node is now 'a  ~%" node)
;;    (format t "~% ifs is now 'a  ~%" ifs)
;;    (format t "~% eval of ifs is 'a  ~%" (eval ifs))
;;    (cond
;;     ((eval ifs)
;;    (format t "~% thens is now 'a  " thens)
;;    (format t "~% eval of thens is 'a  " (eval thens)))
;;     (t (format t "~% NO GOALS SATISFY RULE ~%")))
   (if (eval ifs)
       (progn (eval thens) t)
      nil)
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This is the start of monitor.
;;  It just determines which nodes will be placed
;;  back on the blackboard.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun mini-monitor (gnode)
  (let
   ((tvalue (duration gnode)))
   (cond
    ((equal tvalue 'one-shot)
     (remove-goal-x-from-level-y gnode level))
  . ((equal tvalue 'continuous) nil)
    (t nil)
    )
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Try-all-rules function tries all the rules on each
;;;  given node which is input to the procedure.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun try-all-rules (node)
  (do
   ( (rules-to-try rules (cdr rules-to-try))
     (record nil) )    ; suppose to hold all rule fired on this  node
    ((null rules-to-try)
     (format t "~%#### fini try-all-rules ")
     (format t " applied -- 'a  -- ####" record)
     )
    (cond
     ( (runrule node (car rules-to-try)) ; try each rule
      (setq record (cons (cadar rules-to-try) record)))
     (t nil)
     )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Given a "lyst" of goal nodes, try each rule on each
;;;  of these nodes.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun map-the-goals (lyst level)
;  (format t "~% just entered map-the-goals, lyst is ~a   ~%" lyst)
  (do*
   (
   (wlyst lyst (cdr wlyst))
   (node (car wlyst) (car wlyst))

  ((null wlyst)
;    (format t "~% *** finished mapping goals to ksars *** ~%")
;    (format t "~% ***** clock is now~a   ****** ~%" clock)

  (try-all-rules node)
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Plan-goals gets the goals and maps them into KSAR's.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


  (defun plan-goals ()
  (do*
   (
   (llyst level-lyst (cdr llyst))
   (goallyst
    (get-goals-from-level-x-of-duration-y(car llyst) one-shot)
          (if llyst
    (get-goals-from-level-x-of-duration-y (car llyst) one-shot)
            ))
   )
   ((null llyst))
   (format t "~% goallyst is before map  ~a " goallyst)
   (format t "~% llyst is before map  ~a " llyst)
   (cond
    (goallyst                   ;(mapcar 'describe goallyst)
     (format t "~% to enter map-the-goals - lyst - ~a ~%" goallyst)
;   note this is going through the rule list by level
     (map-the-goals goallyst (car llyst)))
    (t nil))))

;;;;;;;;;;;;;; end of file ggplan.cl ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  File is ggports.cl
;;;;;;;;;;;;;;file   name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Read-header may not be needed depending on header in LISP.
;; Reads header off of lisp so that you can communicate with
;; compiled lisp KS.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun read-header (stream)
(dotimes (i 2) (read-line stream))
(read stream))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function opens an in port called inbeam for the path
;;; process which represents the beam generation program. The out
;;; port is outbeam.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun openports ()
(multiple-value-setq (beam error-beam beam-id)
  (run-shell-command "path" :wait nil
     :input :stream :output :stream
     :error-output :stream))
;; (read-header beam)
)

;;
;; now open ports to keep this process active
;;

(openports)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This part opens two-way assignment ports for the executable
;;  called "test."
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun open-assignment-ports ()
(multiple-value-setq (assign error-assign assign-id)
  (run-shell-command "test" :wait nil
     :input :stream :output :stream
     :error-output :stream))
 )

;;
(open-assignment-ports)
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function opens track ports.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun open-track-ports ()
(multiple-value-setq (track error-track track-id)
  (run-shell-command "track" :wait nil
     :input :stream :output :stream
     :error-output :stream) )
;; (read-header track)
 )

;;;
 (open-track-ports)
;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function opens ports for spline fucntion.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun opensplineports ()
(multiple-value-setq (spline error-spline spline-id)
  (run-shell-command "spline" :wait nil
     :input :stream :output :stream
     :error-output :stream))
;;  (read-header spline)
  )

;;
 (opensplineports)
;;;

;;;;;;;;;;;;;; end of ggports.cl file ;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;file-name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  this file is ggrule.cl
;;;;;;;;;;;file-name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This is the initialization for a recurrent goal.  A recurrent goal is a
;;  goal which, when satisfied, generates a ksar and then is disabled from
;;  generating anymore ksar's until the KS is activated. Such a goal is
;;  enabled even if a KS fails for some reason. This goal creation
;;  is for rule5.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(sendpushgoal
  (make-instance 'bbgoal
            :purpose 'merge-segments
                 :initiating-data-level 'segments
            :ksarptr nil
            :duration 'recurrent
           )
  segments)


;;
;; This file contains the rules.
;; These rules use bbgoal class objects as facts
;; in the antecedents.
;;

(setq rule0
  '(rule clocked-data-arrival
       (if
        (equal (purpose gnode) 'clock))
       ;; --- then place data node on the hit data ---
       (then
        (progn
        (create-ksar
         (list 'newhit 'add 'hit 'unknown 'unknown clock))
         (format t "~% $$$$$  CLOCK RULE FIRED  CLOCK IS 'a  $$$$$ " clock))
        )))

;;

(setq rule1
  '(rule create-segments-from-hits
       (if
        (and
         (equal (level gnode) 'hit)  ; is goal a hit node
        ;; ;(null (right segments))  ; no segments
         (equal (purpose gnode) 'change)  ; is it for change
         ))
       ;; --- then assign hit update to the segments ---
        (then
         (progn (create-ksar     ; this is call to create a KSAR
           (list 'change 'hit (coord gnode)
               (number gnode) (event-time gnode) gnode))
   (format t " ~%$$$$$111 rule assign hits that arrived fired 111$$$$$$")
          ))))

;;
;; Rule to process unmatched hits.
;;

(setq rule1a
  '(rule process-unmatched-hits
       (if
        (and
        (equal (purpose gnode) 'unmatched) ; is purpose unmatched
```

```
        (setq rvarl (source gnode)) ; what is hit node
        ))
        ;; -------- for now just remove these two nodes ----
        (then
         (progn
        (create-unmatched-hit-ksar rvarl) ; note ksars created directly
        (remove-goal-x-from-level-y gnode segments)
        (format t " ~%$$$$$ rule removes UNMATCHED nodes fired $$$$$$")
        )
        )))

;;

(setq rule2
  '(rule spline-check-of-tracks
       (if
        (and
         (equal (level gnode) 'track)
         (equal (threat gnode) t)
         (or
           (null  (check (source gnode)))
           (if (numberp (check (source gnode)))
             (> (- (car (event-time gnode)) (check (source gnode)))
                recheck-interval
                )
             nil)
          )

         (and (> (length (snode gnode)) 1)
            (>= (apply 'min
                    (mapcar
                     #'(lambda (y) (numbor y))
                     (snode gnode)
                     )) 4))))
       ;;-----------------------------------------------------
        (then
         (progn
;;        (setf (check (source gnode)) (car (event-time gnode))) ;; added
          (create-ksar
           (list 'change 'track  (event-time gnode)
                  (threat gnode) (snode gnode)
                  (source gnode)))
          (format t "~%$$$$$ this is spline-check-of-tracks $$$$$$")
          ))))

;;
;;  This rule generates the subgoals needed to check tracks
;;

(setq rule2a
  '(rule spline-check-failed-generate-subgoals
       (if
        (and
         (equal (level gnode) 'track)
         (equal (threat gnode) t)
         (equal (check (source gnode)) 'failed)))
       ;; ---------- generate subgoals ····················
        (then
         (progn
          (create-subgoals-to-break-track(source gnode))
          (format t "~% 2A 2a 2a 2a 2a 2a 2a 2a 2a FIRED ")
 (format t "~% $$$$ rule spline-check-failed==> generate subgoals $$$$")
 )
         )))

;;
;;
;;

(setq rule3
```

```
  '(rule create-tracks-from-segments
       (if
        (and
         (equal (level gnode) 'segment)
         (> (number gnode) 1)  ; are the number pts in a segment > 1
         (equal (purpose gnode) 'change)
         ))
        ;;--- construct the tracks from the segments ----
        (then
         (progn
         (create-ksar
           (list 'change 'segment (coord gnode)
                  (number gnode) (event-time gnode)
                  (source gnode)))
    (format t "~%RULE 33333 just executed rule 3, create tracks RULE 3333")
;;        (break "~%***** STOPPED RULE THREE FROM FIRING *****~%")
         ))))

;;
;;  rule 4 purges old objects from the goal BB
;;

(setq rule4
  '(rule purge-old-segment-nodes
      (if
      (and
        (equal (purpose gnode) 'purge-segments) ; is it a purge node
        (setq rvar1 (find-oldest-segment))
        (> (abs (- (car (event-time rvar1) )
                   (find-time-of-last-data-point)
                   ))
             10  ); 10 is age afterwhich is purged from the list
       ))
        ;; ---delete the goal node and its supporting data---
      (then
        (progn
      ; is the number of snodes supporting track <= 1
        (if (and (tnode rvarl)
               (<= (length (snode (tnode rvar1))) 1) )
               ; then delete both track and segment nodes
            (progn (setq rvar2 (tnode rvarl))
                 (remove-data-x-from-level-y rvar2 tracks)
                 (remove-data-x-from-level-y rvar1 segments))    nil)
         (format t "~%$$$$$ rule purge-old-segment-nodes fired $$$$$")
        ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Rule 5 is to merge-segments when the appropriate conditions exits.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(setq rule5
  '(rule merge-segments
       (if
        (and
       (equal (purpose gnode) 'merge-segments) ; is it a purge node
       (null (ksarptr gnode)).; no extend segment ksar active
       (setq rvar1 (find-oldest-segment))
       (setq rvar3 (find-most-recently-started-segment-with-length-gt-y 1))
       (setq rvar2 (abs (- (car (event-time rvarl))
                       (car (last (event-time rvar3))))))
       (and (> rvar2 3) (<= rvar2 10)) ; is age of proper range
       ))
        ;; --- rule attempts to patch fades in signal ---
      (then
       (progn  ; this creates ksar and sets ksarptr to that ksar
;;     (break "INSIDE RULE 5 - attemp to extend old segments ~%")
        (setf (ksarptr gnode) (create-segment-merging-ksar gnode))
        (format t "~% 5555555 CLOCK ~a 5555555555555555555 " clock)
```

```
        (format t "~%$$$$$$ rule 5--- merge-segments --- fired $$$$$")
        ))))

;;;
;;  This rule verifies the track composition
;;

(setq rule6
  '(rule verify-track-composition
       (if
        (and
         (equal (level gnode) 'track)
         (equal (purpose gnode) 'verify-track)
         ))
;;  ---------------  rule verify track composition ---------
       (then
        (progn
         (create-verify-track-ksar (source gnode)
                                   (snode gnode)
                                   (coord gnode))
   (format t "~% 6666  RULE6 verify-track-composition fired RULE6 6666 ~%")
;;(break "~%   ***** STOPPED RULE SIX - verify-track-comp -FROM FIRING *****~%")
         ))
       ))

;;
;;
(setq rules (list rule0 rule1 rule1a rule2 rule2a rule3 rule4 rule5 rule6))
;;;;;
;;;;;;;;;;;; end of ggrule.cl ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   FILE NAME IS grloop.cl
;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   This function is used to update the global variable called clock
;;   and to
;;     1. push a clock goal into the goal BB
;;     2. push periodic goals into the goal BB.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun clock-update ()
  (cond
   ((zerop (mod clock 8))       ;; was 4 is now 8 should be reset
   (sendpushgoal                  ;; push a clock goal onto BB
    (make-instance 'bbgoal
              :purpose 'clock
              :initiating-data-level   'hit
              :duration 'one-shot)
          hits)
   (sendpushgoal                  ;; push a purge-segment goal onto BB
    (make-instance 'bbgoal
              :purpose 'purge-segments
              :initiating-data-level   'hit
              :duration 'one-shot)
          hits)
    )
   (t))
  (setq clock (1+clock))
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  The function goon steps through the control loop.  It is used
;;  primarily for debugging purposes.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun goon ()
  (format t "Do you wish to go on ? ~%")
  (format t "Answer nil for no, and anything else for yes ~%")
  (cond
   ((null (read)) (reset))
   (t t) )
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This the bootstrap function which constructs commands contained
;; in the KSAR's for invoking the KS's.
;; There are two separate procedures used for the KSAR's depending
;; on whether they are in an atom queue or any other queue.
;; KSAR's from the atom queue invoke the KS and then wait for the
;; response.
;; In contrast, KSAR's from the beam-queue and segment-queue execute
;; the KSAR is three phases.
;;   1. Write the command to the KS  Channel state = 1.
;;   2. Read the command to the KS  Channel state = -1.
;;   3. Post the KS results onto BB. Channel state = 0.
;;   4. Preconditions must be calculated Channel state = 2.
;; The four possibilities are handled in four separate sections.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun bootstrap ()
  (cond
   ((equal (number ksarq) 0)
 (format t "~% ******* ENTERED BOOTSTRAP - EMPTY KSAR QUEUE ****~%")
 (format t "number in ksarq  is 'a  - first test of ksarq contents ~%"
      (number ksarq))
```

```
        nil)
     (t (format t "~% ******* ENTERED BOOTSTRAP - KSARQ QUEUE HAS ENTRIES ****~%")
        (cond ((> (length (atomic-queue ksarq)) 0)  ; if atom not empty
        (format t "~% ATOM-QUEUE IS NOT EMPTY - EXECUTE COMMAND ~%")
        (let* (                    ; pull ksar out of queue
          (temp  (poptart ksarq atomic-queue))
          (xxx (format t "~%AFTER poptart, KSAR is~%"temp))
          ;;  (ctemp (car (postboot temp)))  ; pull command out of postboot
          (ctemp (command temp))  ;; changed 21 Jan 92 boot fm command
          (ctemp '(,ctemp ',temp)) ; insert arg of ksarq ptr
          )
          (format t "+++~% BOOTSTRAPING COMMAND 'a COMMAND FIRING +++~%" ctemp)
          (eval ctemp) ; this fires the command held in postboot
          (format t " BOOT COMMAND COMPLETE -- continuing on~% ")
          ))
            (t (format t "Number of Entries in Garbage Queue< 0 ~%"))
            )

        ;;
        ;; now finish off ksars which need only to put data on bb
        ;;                 channel is = 0
        ;;
        ;;

    (format t "TEST IF MASK = 0, PROCESS RESULTS AND PUT DATA ON BB ~%")
      (cond   ; take care of the reads, are any mask elements = 0
       ((and (> (numbor ksarq) 0)
       (eval (cons 'or (mapcar #'(lambda (x) (if (equal x 0) t nil))
                  (cdr (mask ksarq))))))
         (format t 'SOME MASK ELEMENTS = 0, at the let statement-%")
         (let*(
         ;;    (xxx (break  "break-at-finish-work'%))
          (mtemp (cdr (mask ksarq)))
            ;; at this point mtemp will be the mask
            ;; the zero values correspond finishing ksars
         (ntemp (mapcan #'(lambda (x y) (if (equal x 0) (list y) nil))
                  mtemp  KSQUEUES )) ; this a list of KS's to read
          (xxx (format t "mask is 'a and ksqueues are 'a  ~%" mtemp ntemp))
          (ftemp (mapcar
;;              #'(lambda (x) (pop-flavor-at-node-at-queue ksarq x))
;; inserted 25 Dec 91 to try to fix this problem
                #'(lambda (x) (mcpoptart ksarq x))
                    ntemp)) ;; this is a list of instances
            (xxx (format t "instances is ~a ~%" ftemp ))
        (btemp (mapcar #'(lambda (x) (car (postboot x))) ftemp))
              (xxx (format t "command is ~a  and flavor is ~a ~%" btemp ftemp))
         (rtemp (mapcar #'list btemp ftemp)) ; list of functions to fire
         )
    (format t " MASK is ~a and KSQUEUES with MASK=0 is ~a~%" mtemp ntemp)
    (format t " COMMAND is 'a and FLAVOR with MASK=0 is ~a~%" btemp ftemp)
     ;; (break "Just before firing the functions in bootstrap ")
     (mapcar #'(lambda (x y) (funcall x y)) btemp ftemp)
          ;; (mapcar #'eval rtemp)
           ))  ; this fires all the functions
       (t
       (format t "~%NO MASK ELEMENTS ARE ZERO ~%"))
            )


        ;;
        ;; now take care of reads when channel is -1
        ;;

   (format t "TEST IF MASK = -1 ==> READ KS ~%")
      (cond   ; take care of the reads, are any mask elements = -1?
       ((eval (cons 'or (mapcar #'(lambda (x) (if (equal x -1) t nil))
                  (cdr (mask ksarq)))))
   (format t 'SOME MASK ELE = -1 ==> START READ KS ~%")
        (let*(
        (ktemp (poll-reads KSSOURCES)) ;ktemp is read ready ports
        (xxx (format t " ktemp = ~a ~%" ktemp))
```

```
        (mtemp
        (mapcar #' (lambda (x y) (* x (if (null y) 0 y)))
             ktemp (cdr (mask ksarq))))
       (xxx (format t " mtemp = 'a ~%" mtemp))
          ;; at this point mtemp will have 0,1,-1 and the -1 correspond
          ;; to those ks's that need to be read from
        (rtemp (mapcan #' (lambda (x y) (if (equal x -1) (list y) nil))
                 mtemp KSQUEUES )); this a list of KS's to read
          ) ;; now read each of these KS's and place result in
    (format t " READY PORTS are 'a NEED TO READS is ~a and KS's TO READ is 'a ~%"
         ktemp mtemp rtemp)
    (format t " the call (funcall 'a 'a) ~%" rtemp ksarq)
          ;; (break "Break ~ inside bootstrap about to read-ks %")
        (mapcar #' (lambda (x) (read-ks
                        (car (funcall x ksarq)) ;; new Jan 90
                        )) rtemp)
          ;; (break "Break - inside bootstrap just attempted to read-ks %")
          ));; message flavor
       (t
       (format t 'NO MASK ELE = -1, NO KS's READY TO READ ~%")))

     ;;
     ;; now take care of the writes -- put this code in later
     ;;            channel is 1
     ;;

  (format t "TEST MASK = 1 ==> WRITES TO KS'S, mask ele of 1 ~%")
     (cond  ; take care of the writes, are any mask elements = 1?
       ((eval (cons 'or (mapcar #' (lambda (x) (if (equal x 1) t nil))
                   (cdr (mask ksarq)))))
     (format t 'SOME MASK = 1, START WRITE SEQUENCE ~%")

          ;; (break " about to bootstrap-write-ks~%")

        (let*(
;;     (ktemp (poll-writes KSSOURCES)) ;ktemp is write ready ports
;;          (ktemp (mapcar #' (lambda (x) 1) KSSOURCES))
         (mtemp
        (mapcar #' (lambda(x) (if (equal x 1) 1 0))
               (cdr (mask ksarq))))
    (xxx (format t "mtemp after testing the (mask ksarq) ~a~%" mtemp))
          ;; at this point mtemp will have 0.1. -1 and the 1 correspond
          ;; to those ks's that need to be written to
        (rtemp (mapcan #' (lambda (x y) (if (equal x 1) (list y) nil))
                 mtemp KSQUEUES )); this a list of KS's to read
  (xxx (format t "rtemp contains matches between writes and queues ~a~%" rtemp))
          ) ;; now read each of these KS's and place result in
    (format t "~% **** COMMAND **** is (-a 'a) ~%" mtemp rtemp)
        (mapcar #' (lambda (x) (write-ks
                        (car (funcall x ksarq)) ;; Jan 90
                        )) rtemp)
          )) ;; message flavor
       (t
       (format t "NO KS's HAVE MASK = 1 ** NO WRITE COMMANDS ~%")))


     ;;
     ;; now use the preboot to establish the precondtions and
     ;; freeze the local context
     ;; channel is 2
     ;;

   (format t "TEST MASK = 2, ==> ESTABLISH PRECONDITIONS ~%")
     (cond  ; take care of the reads, are any mask elements = 0
       ((and (> (number ksarq) 0)
       (eval (cons 'or (mapcar #' (lambda (x) (if (equal x 2) t nil))
                   (cdr (mask ksarq))))))
        (format t 'SOME MASK ELEMENTS = 2, NOW FREEZE CONTEXT ~%")
        (let*(
         (mtemp (cdr (mask ksarq)))
```

```
        ;; at this point mtemp will be the mask
        ;; the zero values correspond finishing ksars
       (ntemp (mapcan #'(lambda (x y) (if (equal x 2) (list y) nil))
                   mtemp KSQUEUES)) ; this a list of KS's to read
      (xxx (format t "mask is 'a and KS FLAVORS is 'a ~%" mtemp ntemp))
        (ftemp (mapcar #'(lambda (x) (car
                              (funcall x ksarq ))) ;; Aug 90
                    ntemp)) ; this is a list of flavor instances
;;    (xxx (format t "ftemp is 'a ~%" ftemp ))
      (btemp (mapcar #' (lambda (x) (car (preboot x))) ftemp))
      (xxx (format t "preboot command is 'a and flavor ~a ~%" btemp ftemp))
        (rtemp (mapcar 'list btemp ftemp))) ; list of functions to fire
    (format t "~%MASK  is 'a,  KS FREEZABLE is 'a  ~%" mtemp ntemp)
    (format t "~%FLAVOR is -a, COMMAND is 'a  ~%" ftemp btemp)
    (format t "~%LISP EXPRESSION is ~a~%" rtemp )
       ;; (break "bootstrap-fire-pre-assign-hits-before-mar ~%")
     (mapcar #'(lambda (x y) (funcall x y)) btemp ftemp)
     ;;    (mapcar 'eval rtemp)  ; this fires all the functions
        ;;    (*break  t 'bootstrap-fire-pre-assign-hits)
         (mapcar #'(lambda(x) (setf (channel x) 1)) ftemp)
         ))
    (t (format t " MASK NOT 2, NO PRECONDITIONS TO EXECUTE ~%")))

    )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  The function cloop -- for control loop and it
;;  is the main loop for driving the BB.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun cloop ()
 (catch 'cloop    ;; throw-catch combo used to break out at right time
  (do ()     ;put into infinite loop
      (())
  (format t "CLOCK UPDATE -- TIME IS ~a -- CLOCK UPDATE~%"clock)
  (go-for-it )
  (clock-update) ; update the clock variable and place on event q

  (cond
   (cloop-display ; if global variable set for display then
  (showq)     ;; print out the ksar queueing system
  (expandq)  ;; expand out the entries in the ksar queueing system
  (showl)     ;; expand out the levels in the blackboard
  (expandl tracks)(expandl segments)(expandl hits)
 ;; (expandg tracks)(expandg segments)(expandg hits)
   ) ;;display queues and levels
   (t nil))

 ;(goon)          ; wait for signal to continue
 ;(go-for-it )

  (format t "You about to map events to ksars, the ksarq is follows ~%")

  (plan-goals)   ; this maps the goals into ksars; it calls planner

  (format t "~%BEFORE BOOTSTRAP KSARQUEUE EXPAND ~%")
  (expandq)    ;; write out the queueing system in gory detail

  (bootstrap) ;; fire off the next sequence of ksars

 (format t "~%AFTER BOOTSTRAP KSARQUEUE EXPAND ")
 (expandq)     ;; write out the queueing system in gory detail

 (showq) (showl) ;; display queues and levels

 ;(goon)
```

```
  )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Function go-for-it allows you to set the number of loops you want.
;;  It breaks out of the cloop by throwing to the catch function
;;  in the cloop function.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun go-for-it ()
;; (format t "~% cloop-count is at top  'a  " cloop-count)
  (cond
   ((plusp cloop-count) (setq cloop-count (1- cloop-count))
 ;;    (format t "~% cloop-count reduced by 1, it is now  'a  " cloop-count)
    t)
   ((<= cloop-count 0)
    (format t "~% Current Clock is 'a  .  How many control loop steps do
        you want?  Zero means STOP.  N means N steps. " clock)
    (format t "~%  Enter number NOW and hit return !! ")
    (let ((reply nil)
         (answer (read)))
;;      (format t " Answer you entered ~a ~%" answer)
       (cond
        ((and (numberp answer) (plusp answer))
           (setq cloop-count (1- answer))
           (format t "~% Do you want queues and levels displayed????")
           (setq cloop-display (y-or-n-p)))
        (t (throw 'cloop (format t "BROKE OUT of CLOOP - CLOCK = 'a"  clock)))
        )))
  ;;  (format t "~% cloop-count it is at bottom  ~a " cloop-count)
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  The function my-redirect is used redirect the screen io to
;;  the file out.
;;  The companion function my-direct resets the stream io to the screen.
;;
;;  Written for use
;;  at home where baud rate of transmission does not allow program
;;  to run as fast.  Good for tracing bugs, like the function dribble but
;;  without the writing to the screen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun my-redirect ()
(setq myoutfile (open  "out"
            :direction :io
            :if-exists :append
            :if-does-not-exist :create
      ))
  ;; now reset the std io files
  (setq savestdio *standard-output*)
  (setq *standard-output* myoutfile)
  )

(defun my-direct ()
  ;; now reset the std io files to orginal setting
  (setq *standard-output* savestdio )
  (close myoutfile)
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function expandall expands the levels and the
;;  queues to get a snapshot of the state of the system.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expandall ()
  (format t " %  First expand the LEVELS to see the data BB ~%")
  (showl)
  (expandl tracks)
```

```
  (expandl segments)
  (format t " ~% SECOND expand the QUEUES to see the data BB ~%")
  (expandg tracks)
  (expandg segments)
  (format t " ~% THIRD expand the QUEUES to see the data BB ~%")
  (showq)
  (expandq)
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This is the main loop for driving the BB for file output.
;;  This is the redirected file output form of the cloop.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun fcloop ()
 (catch 'fcloop
  (my-redirect)  ;; redirect the output to file out
  (do ()     ;put into infinite loop
      (())
  (format t "CLOCK UPDATE -- TIME IS 'a  -- CLOCK UPDATE~%" clock)
  (file-go-for-it )
  (clock-update) ; update the clock variable and place on event q

  (cond
   (cloop-display  ; if global variable set for display then
  (showq)
  (expandq)
  (showl)
  (expandl tracks)(expandl segments)(expandl hits)
  (expandg tracks)(expandg segments)(expandg hits)
    )  ;display queues and levels
   (t nil))

  ;(goon)           ; wait for signal to continue
  ;(file-go-for-it )

  (format t "You about to mapped the events, the ksarq is follows ~%")

  (plan-goals)   ; this maps the goals into ksars; it calls planner

  (format t "~%BEFORE BOOTSTRAP KSARQUEUE EXPAND ~%")
  (expandq) (describe ksarq)

  (bootstrap)

 (format t "AFTER BOOTSTRAP KSARQUEUE EXPAND ~%")
 (expandq)

 (showq) (showl) ; display queues and levels

  ;(goon)

  )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Function file-go-for-it allows you to set the number of loops you want.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun file-go-for-it ()
  (let ((e *error-output*))
;; (format e "~% cloop-count is at top  'a  " cloop-count)
  (cond
   ((plusp cloop-count) (setq cloop-count (1- cloop-count))
 ;;    (format e "~% cloop-count reduced by 1, it is now  ~a " cloop-count)
    t)
   ((<= cloop-count 0)
    (format e "~% Current Clock is ~a .  How many control loop steps do
       you want?  Zero means STOP.  N means N steps. " clock)
```

```
    (format e "~%  Enter number NOW and hit return !! ")
    (let ((reply nil)
         (answer (read)))
;;     (format e " Answer you entered ~a ~%" answer)
       (cond
        ((and (numberp answer) (plusp answer))
           (setq cloop-count (1- answer))
           (format e "~% Do you want queues and levels displayed????")
           (setq cloop-display (y-or-n-p)))
        (t (throw 'fcloop
                (progn
              (format e "BROKE OUT of CMOP - CLOCK = 'a"  clock)
              (my-direct))   ;; return the stream to standard io
                ))
        )))
  ;;  (format e "~% cloop-count it is at bottom 'a  " cloop-count)
   )))

;;;;;;;;;;;;;;; end of ggrloop.cl ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;file        name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This is file ggmess.cl, the file which establishes the
;; communication classes.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This defclass defines the object messenger.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass messenger ()
  (
  (write-port :initarg :write-port :accessor write-port)
  (write-fd :initarg :write-fd :accessor write-fd)
  (read-port :initarg :read-port :accessor read-port)
  (read-fd :initarg :read-fd :accessor read-fd)
  (pid :initarg :pid :accessor pid)
  )
  (:documentation "The messenger class contains the 1/0 to KS ")
  )



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; These two statements define the beam and assignment messengers.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq beammsg (make-instance 'messenger ))
(setq assignmsg (make-instance 'messenger ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This method reads from the output port and places it in the
;; answer lyst in the ksar object.quit
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod read-ks ((ele ks-protocol-mixin))
       (format t "~% READKS READKS READKS READKS READKS READKS ~%")
       (format t "INSIDE DEFMETHOD READ-KS clock is ~a ~% " clock)
;;       (describe ele)
       ;; (*break (> clock 8) 'stop-to-test-read)
  (let* (
     (mtemp (messenger ele))
     (ptemp (read-port mtemp))
     (xxx (format t "messenger is ~a and port is ~a %" mtemp ptemp))
     (temp (read ptemp))
      )
       (format t "The address of messenger was ~a~%" mtemp)
       (format t "The port read was ~a~%" ptemp)
       (format t "The message received back from ks is ~a~%" temp)
       (setf (anslyst ele) temp)
       (setf (channel ele) 0)
      ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This new commonlisp predicate can test a port to see if
;; there is input to read -- note a port is a stream in commonlisp.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun readp (port)
 (force-output port)
 ;; nil if nothing; else, t
 (if (listen port) 1 0) ;; return 1 if data at port, else 0
 )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Function poll-reads polls KS's to see which are ready to read.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun poll-reads (kslyst)
  (mapcar I'readp    ; check to see which ports are ready to read
       (mapcar I' (lambda(x) (read-port x)) kslyst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,,
;;  Function poll-writes polls KS's to see which are ready to receive.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun poll-writes (kslyst)
  (mapcar I'readp   ; check to see which ports are ready to write
       (mapcar #' (lambda(x) (write-port x)) kslyst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,
;;  Define slots for messenger objects ports.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;,


(setf (write-port beammsg) beam)
(setf (read-port beammsg) beam)
(setf (write-port assignmsg) assign)
(setf (read-port assignmsg) assign)
(defvar KSSOURCES (list beammsg assignmsg))
;;
;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This file is ggmerge.cl.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;file name;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function merges segments.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun merge-segments (ksarnode)
  (let*
   (
   (wlyst (find-segment-pair))  ; finds best pair of segments to extend
   (s1 (car wlyst))             ; first segment .
   (s2 (cadr wlyst))            ; second segment
   (value (car (last wlyst)))   ; value of the cost
   )
;;   (break  "~% INSIDE merge-segments for extending atrophied paths ~%")
   (merge-segment-x-to-segment-y s1 s2)  ; change the nodes
   (setf (ksarptr (context ksarnode)) nil) ; reset goal node
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function merges two segments which have been determined to be
;; the same.  It needs to remove the track that is associated with segment
;; s1 if it is only supported by s1.  If more than one
;; segment support the track hypothesis, then the snode must be removed
;; from the support list.  All these cases are included under the conds
;; statement.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun merge-segment-x-to-segment-y (s1 s2)
  (format t "ENTERED merge-segment-x-to-segment-y ~%")
  (setf (numbor s2)
      (1+ (- (car (event-time s2))
               (car (last (event-time s1))))))
  (format t "~% New numbor is ~a~%" (numbor s2))
  (let*
   (
   (t1  (tnode s1))
   (xxx (format t " The tnode is now ~a~%" t1))
   (snodelyst (if t1 (snode t1) nil))
   (xxx (format t " snode lyst that tnode points to is ~a~%" snodelyst))
   )
   (cond
    ((null t1))  ; if there is no track established just remove s1
    ((and t1 (equal (length snodelyst) 1)) ; if only one supporting
   (format t " tnode has only one snode - remove both ~%" )
     (remove-data-x-from-level-y t1 tracks)) ;snode, remove track node
;; (remove-goal-x-from-level-y t1 tracks)) ; changed 5 Jan 92
    ((and t1 (> (length snodelyst) 1)) ; if more than 1 snode supports
   (format t " tnode >1 snode support - remove only snode ~%" )
     (setf (snode t1) (remove s1 snodelyst))) ; remove pointer fm t1
    (t
     (format t "~% and ERROR in logic inside merge-segments in gksar ")
     ))
  (remove-data-x-from-level-y s1 segments)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function creates the segment merging ksar.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-segment-merging-ksar (gnode)
  (sendksarpush
  (make-instance 'ksar
           :priority 1
           :ksar-id 'merging
```

```
            :postboot '(merge-segments)
            :command 'merge-segments   ;;  added 21 Sep 92
            :cycle clock
            :context gnode
            )
   ksarq)
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the equivalence class which contains the first
;; element in the larger given set minus the element itself.
;; In some texts this would be given by [a]-a.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun equivalence-class-of-a-minus-a (a lyst rab rba)
  (do
   (                         ;compare to others in group
    (worklyst (our-set-difference lyst (list a))
          (cdr worklyst))  ; do it one at time
    (dlyst nil) ; accumulate the member of equivalence class
    )
   ('(null worklyst) (return dlyst)) ; return [a]-a
   (if (and (rab a (car worklyst))   ;relation of Rab
         (rba a (car worklyst)))  ; relation of Rba
       (setq dlyst (cons (car worklyst) dlyst))
      nil )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Finds just the RAT matches ie reflexive, antisymmetric and transitive
;;  matches FORWARD from s1 to s2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-forward-candidates (a lyst rab)
  (do
   (                         ;compare to others in group
    (worklyst (our-set-difference lyst (list a))
          (cdr worklyst))  ; do it one at time
    (dlyst nil) ; accumulate the possible candidates of forward merging
    )
   ((null worklyst) (return dlyst)) ; return [a]-a
   ;; (if (rab a (car worklyst))  ; altered 4 Jan 92
   (if (funcall rab a (car worklyst))  ;relation of Rab, possible link
       (setq dlyst (cons (car worklyst) dlyst))  ; record if true
      nil )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Predicate to check proper ordering of sequence time.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun time-ordering (snode1 snode2)
  (let
   ((t1 (event-time snode1))  ; time sequence of snode1
    (t2 (event-time snode2))  ; time sequence of snode2
    )
   (or (< (car t1) (car (last t2)))  ; proper forward order
      (> (car (last t1)) (car t2)) ; proper backward order
      )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Predicate to check if sequence is properly forward ordered.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun forward-time-ordering (snode1 snode2)
  (let
   ((t1 (event-time snode1))  ; time sequence of node1
```

```
    (t2 (event-time snode2)) ; time sequence of node2
    )
    (< (car t1) (car (last t2))))) ; proper forward time ordering


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Computes the cost of extending a path forward over the time gap
;; to the possible merging candidate.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun cost (s1 s2) ; from s1 to s2 where these are segment nodes
  (let*(
   (deltat (- (car (last (event-time s2))) (car (event-time s1))))
   (model  (linear s1)) (estimate (vector-sum (car model)
                (scale-vector deltat (cadr model))))
   (dx     (vector-difference (car (last (coord s2))) estimate))
   (d      (vector-magnitude dx))
   (mean   (/ (vector-magnitude (cadr (linear s1))) 2.e0))
   (prob   (exp-cdf d mean))
   (cprob  (- 1.e0 prob)))
(format t "~% time difference is 'a " deltat)
(format t "~% model is given by 'a " model)
(format t "~% estimate of new position is 'a " estimate)
(format t "~% actual position is ~a " (car (last (coord s2))))
(format t "~% vector difference of these last two values is 'a " dx)
(format t "~% magnitude of this difference is 'a " d)
(format t "~% the mean value is ~a " mean)
(format t "~% the probability of being <= this 'a " prob)
(format t "~% the probabilitiy of > 'a " (- 1.e0 prob))
(cond            ; temp criterion, if 1/cprob > 3 then eliminate
  ((< cprob 0.33333333e0) 101.e0)
  (t (+ deltat (/ (- 1.eO prob)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the best pair of eligible segments for merging.
;; Note the problem may not be commutative.
;; That is, it matters which pair is extended first.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-segment-pair ()
  (do*(
   (slyst (right segments)(cdr slyst)) ; original segment lyst
   (olyst (sort slyst #'(lambda (x y) (< (car (event-time x))
                (car (event-time y))))))
   (s1 (car olyst) (car olyst)) ; first trial candidate
   (clyst nil nil) ; candidate list
   (dlyst nil nil) ; distance list corresponding to the segment s1
   (answer nil) ) ;answer from internal loop passed to outer loop
  ((or answer (not s1)) answer)
  (setq clyst (find-forward-candidates s1 slyst 'forward-time-ordering))
  (setq dlyst (mapcar #'(lambda (x) (cost s1 x)) clyst))
  (do ( (cwlyst clyst (cdr cwlyst)) ; list of candidate segments
       (dwlyst dlyst (cdr dwlyst)) ; list of distance value
       (bestseg (car clyst))       ; the segment with lowest dist
       (bestvalue (car dlyst)))    ; the distance
      ((null cwlyst) ; out of candidate to share
    (setq answer (if (< bestvalue 100.e0)
               (list s1 bestseg bestvalue) nil)))
      (cond ((< (car dwlyst) bestvalue) ; less than current minimum
      (setq bestvalue (car dwlyst)) ; if so reset minimum
      (setq bestseg (car cwlyst))) ; also reset the segment flavor
        (t nil)))) ; end of if statment - returns answer is not nil

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the start time of the most recently initiated
;;  segment.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun find-latest-segment-start-time ()
  (let*
   ((slyst (right segments))
    (tlyst (mapcar
          #'(lambda (x)
                  (car (last (event-time x)))) slyst))
    )
   (apply 'max tlyst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  This function finds the object representing
;;  the most recently started segment.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-most-recently-started-segment ()
  (let*
   ((slyst (right segments))
    (tlyst (if slyst (mapcar
          #'(lambda (x)
                  (car (last (event-time x)))) slyst) nil))
    (tmax (apply 'max  tlyst))
    (smax (if slyst
     ;; (my-remove-if   changed to below 4 Jan 92
     (remove-if
     #'(lambda (x) (< (car (last (event-time x))) tmax)) slyst) nil))
    )
   (if smax (car smax) nil)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function finds the most recently started
;; segment with length greater than y.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-most-recently-started-segment-with-length-gt-y (y)
  (let*
   ((slyst (right segments))
    (tlyst (if slyst (mapcar
          #'(lambda (x)
                  (car (last (event-time x)))) slyst) nil))
    (tmax (apply 'max  tlyst))
    (smax (if slyst
     ;;(my-remove-if   changed to below 4 Jan 92
     (remove-if
     #'(lambda (x) (< (car (last (event-time x))) tmax)) slyst) nil))
    (ymax (if smax
     ;;(my-remove-if-not   changed to below 4 Jan 92
     (remove-if-not
           #'(lambda (x) (> (length (event-time x)) y)) smax)))
    )
   (if ymax (car ymax) nil)))


;;;;;;;;;;;;; end of ggmerge.cl file ;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Drain the 1/0 Ports of Header messages
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (read-header assign)
 (read-header track)
```