Purdue University
Purdue e-Pubs

Department of Electrical and Computer Engineering Technical Reports | Department of Electrical and Computer Engineering

7-1-1989

Tutorial on Lisp Object- Oriented Programming for Blackboard Computation (Solving the Radar Tracking Problem)

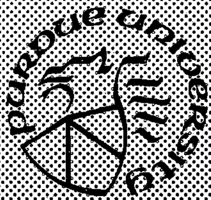
P. R. Kersten
Purdue University

A. C. Kak
Purdue University

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

Kersten, P. R. and Kak, A. C., "Tutorial on Lisp Object- Oriented Programming for Blackboard Computation (Solving the Radar Tracking Problem)" (1989). *Department of Electrical and Computer Engineering Technical Reports.* Paper 643.
https://docs.lib.purdue.edu/ecetr/643

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 89-11
July, 1989

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

A TUTORIAL ON LISP OBJECT-ORIENTED PROGRAMMING FOR BLACKBOARD COMPUTATION

(SOLVING THE RADAR TRACKING PROBLEM)

P. R. Kersten and A. C. Kak

Robot Vision Lab
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907

ABSTRACT

This exposition is a tutorial on how object-oriented programming in Lisp can be used for programming a blackboard. Since we have used Franz Lisp and since object-oriented programming in Franz is carried out via flavors, the exposition demonstrates how flavors 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 flavors.

We have used the radar tracking problem as a 'medium' for explaining the concepts underlying blackboard programming. The blackboard database is constructed solely of flavors which act as data structures as well as method-bearing objects. Flavor instances form the nodes and the levels of the blackboard. The methods associated with these flavors 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.

TABLE OF CONTENTS

Page

1. Introduction ... 3

2. The Representational Problem -- Flavors ... 8

3. Representation of Abstraction Levels ... 16

4 Knowledge Sources ... 24

5. The Blackboard Control ... 28

6. Conclusions ... 42

7. Acknowledgement ... 42

8. References ... 43

Appendix A (Examples) ... 45

Appendix B (Source Code) ... 68

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, 13, 20], planning [14, 19], high-level signal processing [21], general problem solving [6], etc. 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 knowledge sources 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 [22], there is a great difference between understanding the concept of a blackboard model and its implementation, the latter being 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 various items, such as the data on the blackboard, etc.; the nature of interaction between the data on the blackboard and the knowledge sources must then be made explicit; the designer must also make explicit how the high-level goals get decomposed into lower-level goals during problem solving; etc. etc. The purpose of this tutorial exposition is to rectify this deficiency in the literature -- if only from the standpoint of getting a beginner 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 in Lisp can be used to set up the flow of control required for blackboard-based problem solving. The radar tracking problem (RTP) is defined as follows: Given the radar returns, find the best partition of these returns into disjoint time sequences which represent the trajectories of craft or other moving objects. 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 from a formation, any single track can lead to multiple 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 on numerous systems. In fact, a blackboard solution of the radar tracking problem may already exist, although it is probably proprietary. In fact, there is indication [23] that TRICERO has a radar tracker embedded in it, although it is not clear from the system description that a separate BB was used for the tracker [25].

On the basis of the criteria advanced by Nii [23], 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 SNR in uncluttered backgrounds but may also be noisy, cluttered, and weak. Obviously you design for the worst case which includes *noisy and unreliable data.* Noise and clutter induce track anomalies such as fades, splits, merges, etc. Track formation in a noisy environment requires not only significant signal processing but, in general, will also require forward and backward reasoning at a symbolic level. For example, backward reasoning can verify a track by a hypothesis-and-test scheme which 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 *mutiple 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 optimal solution under these conditions. It is generally believed these days that radar 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 in. 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.*

In addition to using the above rationlization 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 the 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 Lisp with Knowledge Sources (KS's) written either in Lisp or in C. The overall organization of RTBB is shown in Fig. 1. 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. On the goal side, 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 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 a flavor*

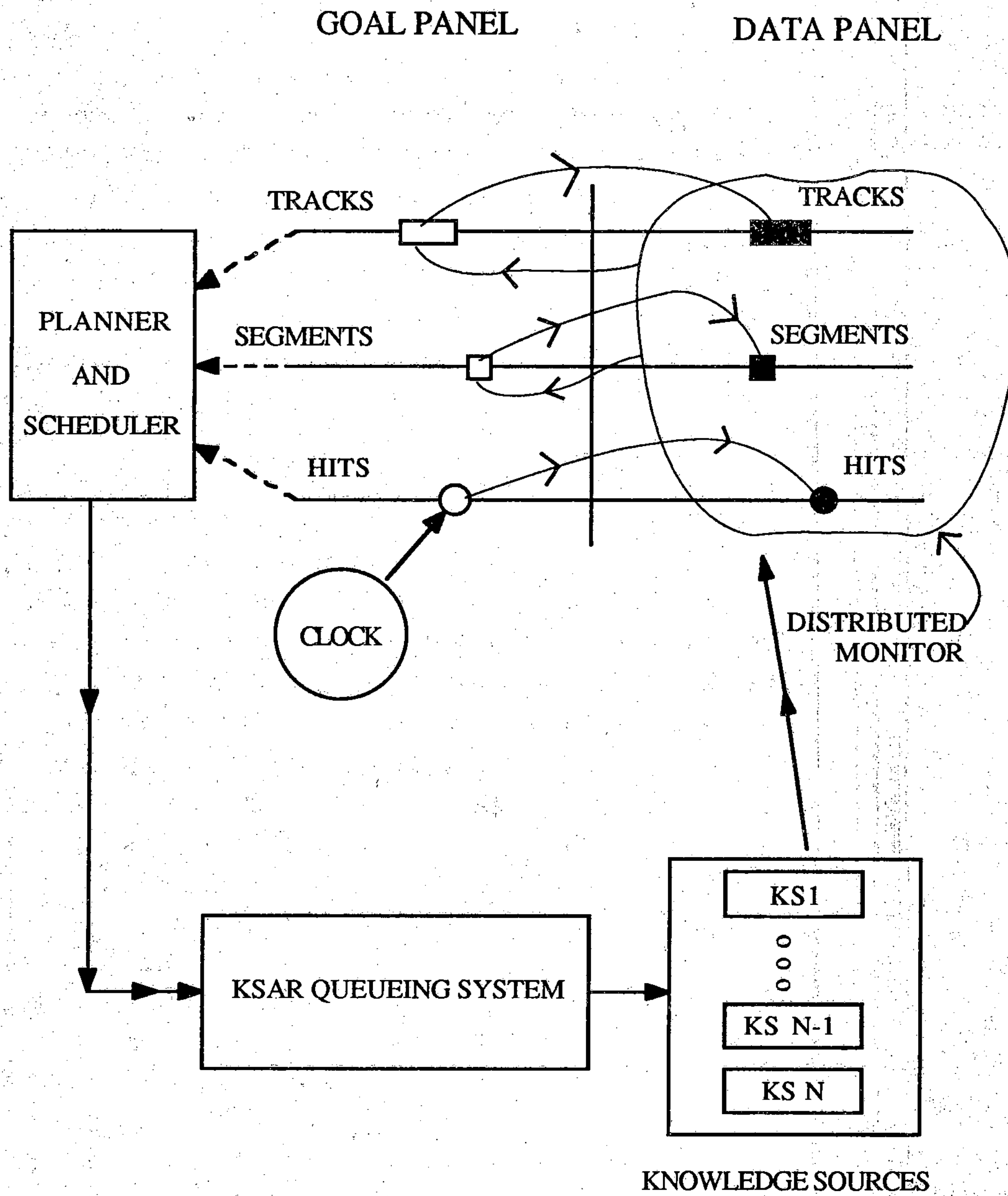


FIGURE 1. Architecture of Radar Tracking Blackboard (BB).

instantiation. These flavor instantiations are method-bearing data structures which are part of Franz Lisp. A method is a procedure which is invoked by a message to a flavor instance. The method triggered depends upon the message sent and the type of flavor receiving the message. For convenience, *flavor instances* are referred to as *flavors* or *instances* or *nodes* interchangeably. 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 enter the changes on the goal BB. This implementation of the monitor using flavors 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 flavors. 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 Franz should find this tutorial useful; such a reader may want to browse through Section 2, if only to become familiar with what is the main data structure 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.

2. THE REPRESENTATION PROBLEM -- FLAVORS

The representation problem is central to problem solving in general. Implementation of a chosen representation requires suitable data structures. In a blackboard, each level can have its own representation and there need not be any consistency between levels. However in this implementation, flavors have been used to represent the data at each level of the blackboard and to implement the blackboard itself. There are several reasons for this decision. Flavors are versatile data structures which are easily initialized and have built-in constructors, selectors, and mutators. In addition, flavors are method bearing objects, and their methods can be used to monitor and to update the blackboard itself. The following paragraphs expand on the properties of flavors which have proved useful in this blackboard implementation.

Flavors are method bearing objects with instance variables which may be easily redefined [12]. These instance variables or just variables may be instantiated to numerical values, symbolic values, lists, or symbolic expressions (s-expressions). Thus the flavor composition is appropriately tailored to the abstraction level in the BB. In addition, there are very sophisticated ways of mixing flavors, i.e. constructing flavors which are built up of other flavors. We will extensively use *before-methods* and *after-methods* that one can attach with a flavor, the former types of methods initiate procedures before altering a flavor instantiation and the latter type after. The reader may note that it possible to invoke methods in sequences that are more complex than what is implied by the names before-methods and after-methods. Only a small portion of the flavors' potential has been tapped for this project. Some of the key properties of flavors are described in this section.

Flavor creation is achieved via the *defflavor* function which defines the name and the characteristics of the s-expressions in the structure. Consider a track node, the highest abstraction on the BB.

```
;;    This node resides at the track level of the data panel.

(defflavor tnode (
    (type 'track) ; the type is track
    (time 1234)  ; the last time stamp in the track
    last-coord   ; latest position of the track
    last-velocity ; latest velocity of the track
    threat       ; true if interval straddles zero
    snode        ; backward pointer list to snode
    cpa-bracket  ; bracket about x and y
    check        ; spline check of segment group
    checklyst )  ; and list that must be checked to verify track
  () ; other flavors included must be placed inside this parentheses
```

```
:gettable-instance-variables  ; allows send to ask for current value
:settable-instance-variables  ; allows send to set the variables
:inittable-instance-variables ; allows variables to be set at creation
)
```

The *defflavor* (define flavor) function is followed by a list of variables which may be initialized in more than one way. Following these is a place to mix in other flavors, and finally there are several options listed to apply to the variables. Flavor instances may be initialized via two simultaneous avenues. First, one can specify a default initialization for instances of a flavor during the defflavoring function. Second, one can initialize instance variables when the flavor instance is created, the latter may supplement or override the default initializations. In this example, the flavor called *tnode* contains eight variables. Two of these variables take default instantiations that will occur in every instance of the tnode flavor. The *time* variable will be automatically set to 1234 and the *type* will be set to 'track. However, this default initialization can be overridden by the inittiable variable which was set as a flavor option :inittable-instance-variable. Thus for example if the creation statement was:

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

then the initial value of *time* would now be 2222 and the initial value of *threat* 'true. In the latter case the unspecified default value of nil is changed to 'true and in the former case the default value of *time*, 1234, is overridden to be 2222. In the creation of abstract data objects, data constructors, such as make-instance shown here, are procedures which make data objects [26]. So the flavors have a simple yet powerful set of constructors.

Flavors also have natural selectors and mutators built into their instantiations. Selectors extract information from data objects and mutators alter information in the data objects. Both of these mechanisms are embedded in the send operator which sends the flavors (objects) messages to perform operations. Flavor operations are optional and are declared in our example by specifying the *gettable* and the *settable* options in the instantiations. The selector gets information by sending a message to the object with the variable name. For example, *(send track :time)* will return the current instantiation of *time* in the tnode flavor instance called track. The mutator uses the same format except now the variable name has ":set-" prepended to it, so *(send track :set-threat 'false)*, alters *threat* variable of the track instantiation to 'false. The object oriented nature of the flavors is a real advantage for obtaining and altering the contents of the BB nodes.

As we mentioned before, a most useful feature of flavors is that they are method-bearing objects, each method being invoked by sending a suitable message to an object. The operation specified in the message and the object combine to uniquely define the procedure that is be used to execute the method. *Before-* and *after-methods* are executed

before and after specified operations such as :set- or :init. Usually, these two types of methods are useful for massaging the data received, and storing information in other instance variables. Since an after-method may be invoked after initializing or altering critical variables in a flavor instance, 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 flavors 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 flavor 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 a method which places a node in the goal panel after the time variable has been set by an after-method.* The *time* variable is updated as new return information percolates up to the track level. In this new goal node, the variable *source* will be set to the name of the flavor instantiation which invoked the method, which in this case is the internal identity of the tnode whose *time* change caused the method to be invoked. The variable *action* will be set to 'change to reflect that the goal was caused by changing the time value, in contrast with, for example, a goal node that might be created by sub-goaling. The variable *type* takes the value 'track for obvious reasons and the variable *time* inherits the updated time value. The variable *threat* inherits its value from the tnode that caused the method to be invoked. The variable *snode* is a pointer to the snode that supports this track node.** Finally, the variable

* 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.

** As will be explained in Section 3, radar returns, in the form of hits, are first grouped into segments, which in turn are then grouped into tracks. In RTBB, each segment is represented by a

duration will be instantiated to 'one-shot; this instantiation 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, the symbolic name following each variable, such as *:snode*, is the name of a variable from the flavor to which the method is attached, unless the symbolic name is quoted, in which case that symbolic name is used directly.

```
;;
;; this defmethod pushes a node into the track level of goal panel
;;

(defmethod (tnode :after :set-time) (value)
  (sendpushgoal
        (make-instance 'bbevent
         :source self
         :action 'change
         :type 'track
         :time time
         :threat threat
         :snode snode
         :duration 'one-shot)
        tracks))
```

The flavor instance is placed on the goal side of the blackboard at the track level by the macro called sendpushgoal which pushes a goal onto the track level using a send message. The sendpushgoal macro is just a procedure which pushes an instance of the *bbevent* flavor onto the track level of the goal panel. It looks as follows:

```
;; this macro pushes an object onto the level on the goal BB

(defmacro sendpushgoal (object level)
  `(send ,level :set-left
       (push ,object (send ,level :left))))
```

So the set of goals on the track level of the goal BB is just a stack of these flavor instances. This method is invoked after a change has been made to the *time* variable of the track node on the data panel. This occurs whenever the tracknode is updated and the message which triggers this change will look something like *(send tnodeptr :set-time (list newtime))*.

When only one or two methods are associated with each node type then it's a simple matter to write one method for each variable. However, as the number of variables associated with each node on the blackboard increases, this becomes cumbersome. Seth

node called *snode*.

Hutchinson suggested using a macro to generate these automatically and actually wrote a macro which did this. The version which follows is a modified version of that macro designed for a goal driven BB [16].

```
;;
;; This macro generates a flavor and the corresponding after-demons
;; which report to the goal panel any changes to the bnode, snode or
;; tnode levels on the data panel. The defmethods of the type embedded
;; in the macro constitute a distributed monitor.
;;

(defmacro newflavor (flav level var-list var-sub inher-list &rest options)
 (cons 'progn
      (cons
 `(defflavor ,flav ,var-list ,inher-list ,@options)
 (do*
  (
   (worklyst var-sub (cdr worklyst))
   (op (car worklyst) (car worklyst))
   (mlyst nil)
   )
  ((null worklyst) (return mlyst))
(setq mlyst
(cons `(defmethod (,flav :after ,(keywordize (concat :set- op)))
            (value)
         (sendpushgoal
             (make-instance 'bbevent
                       :source self
                       :action 'change
                       :type  type
                       :variable ',op
                       :coord  coord
                       :number number
                       :time  time
                       :duration 'one-shot
                       )
          ,level))
          mlyst)
)))))
;;
;;;
```

In this macro, a flavor is created of type *flav* with variables *var-list* and inheritance list *inher-list* i.e., with all the variables and options normally available with any flavor. In addition the variables contained in the *var-sub* list will have update methods automatically generated by the macro code. So if any of these variables is altered, the automatically constructed *after methods* push goal nodes onto the proper levels of the goal panel.

To construct these methods, the macro generates a program which returns the list of methods created in the *do** loop. Once the macro is finished it executes the *progn* statement it has constructed which includes the creation of the flavor and the associated methods which report changes to the goal panel. Note the macro *keywordize* is a procedure which is used to intern the *:set-op* name into the keyword package so that the flavors features of Franz recognize this operation. Here is an example of *newflavor's* use.

```
;;
;;snode is a segment level node
;;

(newflavor snode tracks (
             type   ; is segment
               time   ; this is the time of last coord
               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)
 ()
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables)
;;
```

Note here the segment data node has been set up so that when the *number* of points in the segment is changed, a goal node is pushed onto the goal side of the BB at the track level. To generate equivalent code without this macro, one would first need to define a flavor using the *defflavor* function and then add to it the following method.

```
(defmethod (snode :after :set-number) (value)
 (sendpushgoal
  (make-instance 'bbevent
             :source self
             :action 'change
             :type type
             :variable 'number
             :coord coord
             :number number
             :time time
             :duration 'one-shot)
  tracks))
```

The reader might have noticed that some of the variables, such as *source, action*, etc., do not appear in the *newflavor* call, while they do in the *defmethod* call. The reason for this

is that the definition of *newflavor* automatically sets these variables to certain fixed values. The *newflavors* macro is an illustration of the power of macros and the ease with which one can create an impressive array of methods in a BB shell.

At each level of the blackboard, the nodes are flavor instantiations. Each level is itself also a flavor instantiation. For example:

```
;; This flavor serves as a template for constructing the
;; abstraction level of the blackboard.

(defflavor bblevel (
                up   ;; for higher level in BB hierarchy
                left ;; for the goal BB panel
                right ;; for the data BB panel
                down) ;; for the lower level in BB hierarchy
 ()
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables)
```

The following program statements create the segment level of the blackboard and then set the pointers held in the *up* and *down* variables to link the levels to one another.

```
(setq segments
  (make-instance 'bblevel :down nil :left nil :right nil))
;;;
;;;
(send segments :set-up tracks)     ;; links segment level to track level
(send segments :set-down hits)     ;; links segment level to hit level
```

The *right* variable is a data panel level since this variable will be instantiated to a list of data segment nodes. The *left* variable allows us to refer to the corresponding levels on the goal panel. Since the variables are allowed list instantiations, both the *right* and the *left* variables serve as storage sites for the data and goal nodes, respectively, at different levels of the BB -- at the segment level in the above example. In effect the lists that become instantiations for the *right* and *left* variables act as queues or stacks of flavors depending upon their queueing discipline. This is convenient when one wishes to apply some function on the entire set of nodes since one may mapcar the function onto the list simply obtained via *(send segments :right)* message. Fig. 1 illustrates the *left*, *right* organization of the BB, the right panel storing the data at different abstraction levels, and the left panel storing goals, again at different abstractions, for the purpose of control. Further advantages of flavors will become more evident in the description of the blackboard itself.

Although RTBB is constructed entirely of flavors, as mentioned before, the variables in the flavors may be instantiated to any s-expressions, such as lists. Any list may be used as a queue or a stack. 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 ABSTRACTION LEVELS

As shown in Fig. 1, 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 or hits for hit nodes. A bnode is defined as follows:

```
;; definition of hit node or beam node

(newflavor bnode nil (
     type   ; type is hit
     time   ; time stamp associated with coordinates
     coord  ; list of the coordinates assoc with time
     number ; number points in the list
          )
          ()
     ()
     :gettable-instance-variables
     :settable-instance-variables
     :inittable-instance-variables)
```

Note that the *newflavor* macro has been used to define the bnodes although in this case no methods will be automatically generated since the fourth argument is an empty list. For the same reason, the second argument is set to nil. *Coord* is a list of coordinates (x,y,z) of a radar return. The variable *time* is the time-stamp of the return which is the integer representing the number of clock units since the system started. Unit time intervals are usually chosen to normalize out the actual system parameters. The variable *number* is the actual number of distinct returns received at the time instance corresponding to the time stamp. The node *type* is "hit" and specifies the abstraction level. 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. Hit nodes are generated every n-th clock cycle where at present n is set to four.

A method may be used to refine the data *before* reporting changes. For example, before reporting the change to the goal blackboard, the following method first calculates the number of radar returns, enters that in the bnode and only then reports the change. Alternately, *number* could have been set directly. This method is a trivial illustration of the data modification capabilities of the monitor methods.

```
;;
;; an after-method which first updates the number
;; of returns and then reports to the goal panel
;;
```

```
(defmethod (bnode :after :init) (value)
 (setq number (length coord))
 (sendpushgoal
  (make-instance 'bbevent
                 :source   self
                 :action   'change
                 :type     'hit
                 :variable 'coord
                 :coord    coord
                 :number   number
                 :time     time
                 :duration 'one-shot
                 )
  segments)
 )
```

This method would be triggered after the creation of a hit or beam node -- for example *(make-instance 'bnode :coord coord)*. Here the variable *coord* is instantiated via the built-in methods specified by the *inittable* option in the flavor definition of bnode. Note that the inclusion of ':init' in the first line of the *defmethod* ensures that the method would be executed upon the creation (or, initiation) of an instance of bnode on the data panel and upon intialization of any of the variables in that bnode. The goal created by making an instance of the flavor *bbevent* represents the desire to extend existing segments using the data in the bnode.

A most interesting aspect of the above method is that it alters the bnode whose creation causes the execution of the method. The bnode is altered because the variable *number* now has an instantiation which is equal to the length of *coord*. This 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 we have 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 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 consisted of only one segment and that segment was purged due to the time recency requirement, the track would also be purged. Segment nodes are defined as follows:

```
(newflavor snode tracks (
        type    ; is segment
        time    ; this is the time of last coord
        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); the variables that trigger a report
        ()
        :gettable-instance-variables
        :settable-instance-variables
        :inittable-instance-variables)
```

Note that the variable *coord* is a list of coordinates associated with a given segment, and not with a given time instance as in the beam nodes. That is, the coordinates are grouped via spatial continuity, vice temporal continuity as in bnodes. The variable *time* refers to the sequence of times corresponding to the coordinate points. So both *time* and *coord* are stacks implemented as lists. The *cpa* is the closest point of approach calculated by an *after-method* via the position and velocity information contained in *linear*. The variable *linear* is instantiated to a pair whose entries consist of the position and the velocity computed from the two most recent hits in the segment. Note that the variable *cpa* is instantiated to the perpendicular distance from the origin to a straight line that is an extension of the two most recent hits in the segment. The variable *threat* is true if the instantiation 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*, the comparison threshold being dependent upon 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). For a given segment, while the computation of a value for *cpa* occurs when the segment node is initiated, 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 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 instantiation of the variable *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 groups 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:

```
;;
;; tnode is of form track node -- the flavor holding info on the track level
;;

(defflavor tnode (
            type        ; the type is track
              time        ; the last time stamp making the track
              last-coord  ; latest position of the track
              last-velocity ; latest velocity of the track
              threat      ; interval straddles zero
              snode       ; backward pointer list to segment node
              cpa-bracket ; bracket about x and y
              check       ; spline check of segment group
              checklyst   ; and list for track verify and break
              )
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
```

The variable *type* will always be instantiated to the atom track. The variable *time* is instantiated to the time-stamp of the most recent hit in any of the segments composing the track. The variable *snode* is a list of pointers to the segment level nodes supporting the track. The variables *last-coord* and *last-velocity* are the latest average position and the velocity vector associated 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 position and velocity vectors for each segment are contained in the instantiation of *linear*). The variable *threat* is instantiated to t through an *after-method* by taking a disjunction of the *threat* values of all the segments in the track. The variable *cpa-bracket* is equal to the intervals along x and y, each interval being the union of the cpa intervals associated with the segments in the track. If *threat* is set to yes, 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 variable *check* in the tnode in the database 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 alternative grouping of the segments into possibly multiple tracks.

The abstractions for the goal nodes are identical to the abstractions for the data nodes, as shown in Fig. 1. The goal nodes are defined as flavor instances built up as mixtures of two flavors. The main flavor is *bbevent* and mixed with this is the flavor *goal-attributes* which contains duration and position attributes for the goal nodes. The *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 tnode with new segments. 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 re-attempt 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 re-enabled after a KS is fired from the subsequently generated KSAR. Recurrent goal nodes are never removed from the blackboard. So they act much like synapses having a latency period before they may be fired again. 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 bnodes ($b1_1$,.........,$b1_n$) and let's say the time stamp of $b1_1$ is 7, of $b1_2$ 8, and so on. Also, assume that there exists another snode made up of ($b2_1$,$b2_2$,$b2_3$) where the time stamp of $b2_3$ is 3. Then the job of the recurrent goal node will be to merge the two segments since the time stamp of $b2_3$ is so close to that of $b1_1$. The actual merging, carried out by the MERGE-SEGMENTS KS, will only take place if the extension of the b2 segment to the time instant corresponding to the beginning of the b1 segment is within an acceptable circle.*

The goal nodes at all three levels are created by making instances of the following bbevent flavor mixed in with the goal-attribute flavor. Note the important distinction between the data and the goal panels: While on the data side we have a separate flavor for each abstraction level, on the goal side a single flavor is used, the reason being that the goal nodes at all the levels form together a database for the rule-based planner and

*In our explanation here, an snode was shown as a list of bnodes. In actual practice, an snode is list of coordinates and associated time stamps of the bnodes that form the snode. The actual bnodes are discarded as soon as their returns are assigned to prevent them from overwhelming the BB database.

therefore their similarity is a convenience.

```
;; The goal node flavor called bbevent which is the basic
;; goal blackboard node.

(defflavor bbevent (
            source     ; generating node
            action     ; level this event affects
            type       ; hit or track etc
            variable   ; this is variable triggering event
            time       ; may be list or number
            coord      ; list of coordinates
            number     ; number of coordinates
            threat     ; for tnodes
            snode      ; pointers to snodes
            pattern    ; this is list used for pattern match
            )
 (goal-attributes)         ; mixed in flavor
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables)


;;
;; The mixed in flavor representing the goal node attributes.
;;

(defflavor
 goal-attributes (
           duration   ; time latency of the goal node
           position   ; position relative to coord
           goalptr    ; pointer to other goal nodes
           conditions ; preconditions to fire
           ksarptr    ; pointer to ksar which is queued
           )
 ()
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables)
```

For those goal nodes which are created by *after-methods* executed in response to new entries on the data panel, the variable *source* is instantiated to the internal identity of the data panel node. On the other hand, when a goal node is created by the subgoaling process, *source* is instantiated to the internal identity of the tnode which caused subgoaling to take place. The variable *action* is usually instantiated to 'change, as can be seen in the definition of *newflavor*, to reflect the fact that a goal node was created by a change in the data panel. The variable *type* is set to the level at which the goal node is created,

meaning that it is instantiated to either hit, segment or track. The variable *variable* is instantiated to the name of the variable for which *newflavor* creates an *after-method* for reporting to the goal panel; this can be seen in the definition of *newflavor* in Section 2. The variable *time* is the time-stamp of the data panel node which triggered the formation of the goal node; if a goal node is intiated by an snode, then *time* would be instantiated to a list of time-stamps of the hits constituting the snode. If a track level goal node is initiated by a tnode, then *time* is set to a single value which is the latest time-stamp associated with the track. For a goal node at the segment level, the variable *coord* is instantiated 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 tnode, 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 before; basically this variable would be set to t or nil or the list of pointers corresponding to the snodes that compose the track. The variable *pattern* is not used at this time.

In the flavor *goal-attributes*, the variable *duration* indicates whether the goal node is one-shot or recurrent; its instantiations are therefore 'one-shot or 'recurrent. The next three variables are not being used at this time and have been included for possible future use. The variable *ksarptr* is left uninstantiated for one-shot goals; for the recurrent goal, it is instantiated to the internal identity of the KSAR that is generated by the goal node. While *ksarptr* maintains this instantiation, 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.

Given that the reader is now familiar with the organization of RTBB, we will now reiterate, hopefully in a more precise manner, the overall method for solution formation. 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 flavor on the hit level of the data panel shown in Fig. 1. 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. In order to drive the segment nodes to a higher abstraction level, meaning 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 an extended segment with the segments providing 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 a track goal, a KSAR is generated via the nodes on the track level of the goal panel. The KSAR generation is accomplished via the rule base when rule3 is fired. This rule requires that the node be of type "segment", have more than one data point and have an action variable instantiated to "change". If all these antecedents are satisfied then the *create-ksar* function is called and a KSAR is created to extend a segment into a track or extend and update an existing track. The function *create-ksar* uses the information in the goal node to select the correct flavor instantiation for the KSAR.

In general, a goal can only be achieved 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 knowledge sources (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 aircraft. Trajectories are generated by using Bezier curves in 3-space. 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 aircraft 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 $0 \le u \le 1$ for the entire flight of the craft. Every nth clock cycle (currently n=4) 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 must spit out 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 segments is 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 approach implemented via a best first-search algorithm; see [23, 25] for implementation of best-first search. Further discussion on the complexities of the assignment 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 which 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 using Liszt

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 and the cpa of the craft, both variables defined in the tnode flavor. 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 is formed from the difference between the cpa and the current position, that is, by using $\varepsilon(\overrightarrow{cpa}-\vec{r})$ where ε=0.1. The use of ε allows a confidence region to be formed at the cpa point for each coordinate axis separately. If for any of the axes, the confidence region includes the origin, then 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 on 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.

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 flavor bbevent with goal-attributes mixin; in this instance, the variable *type* is set to 'merge-segments', the variable *duration* to 'recurrent' and the rest of the variables 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 the appendix 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 the track indicating the segments are no longer consistent.

This KS, implemented as a part of the main BB process, 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. The current position of the segment must be within one clock unit of the maximum velocity craft. In addition, the angle θ between the trajectories must have cosine$^{-1}\theta > 0.9$. 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 variable as *failed* and having the GETSPLINE KS check that condition before the KS can be fired.

This ends our 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 which advances the solution the most. However, the design of the "optimum" choice is ultimately the product of 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 scheduler then cycles through the KSAR queues and selects KS's to fire.

We will now briefly describe various possible approaches to the representation and processing of KSAR queues. Then, at the end of this section, we will discuss the current implementation in RTBB of the KSAR queueing system. Ideally, the KSAR priorities should be dynamically determined based upon the threat the craft present to the air space represented by 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 which has the virtue of allowing for the main BB process to fork off KS computations while the main process attends to other chores. The KSAR prioritization in RTBB is accomplished by making a separate KSAR queue for each KS and then visiting each queue, implemented with a FIFO access discipline, in a cyclic fashion, as shown in Fig. 2. The main consequence of this prioritization is that every goal node gets equal priority through its KSAR. In other words, the priority accorded a goal node does not depend upon its abstraction level, as is the case with some other systems. This may appear excessively simplistic, but we felt that at this time there was not enough knowledge available about the radar tracking problem to permit a more sophisticated approach.

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 (send gnode :type) 'merge-segments)
```

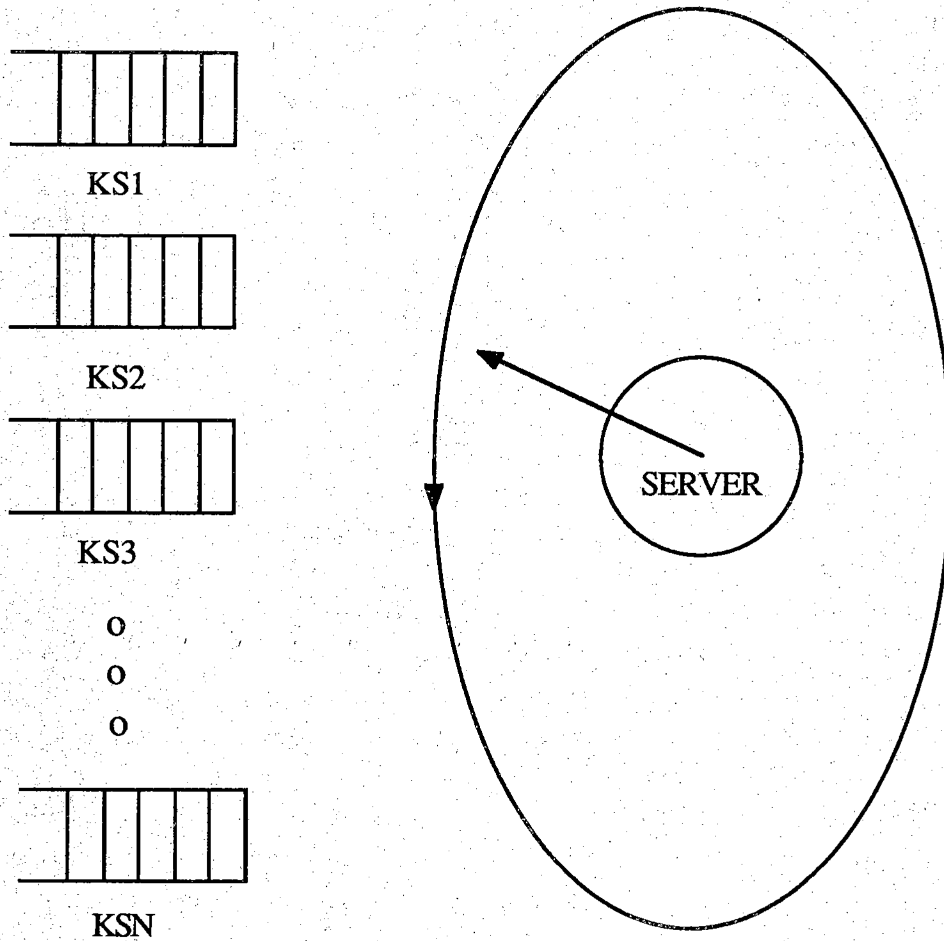


FIGURE 2. This figure depicts an ideal KSAR queueing system.

```
(null (send gnode :ksarptr))      ; no merge-segment ksar active
(setq rvar1 (find-oldest-segment))
(setq rvar3 (find-most-recently-started-segment-with-length-gt-y 1))
(setq rvar2 (abs (diff (car (send rvar1 :time))
                       (car (last (send rvar3 :time))))))
(and (> rvar2 3) (<= rvar2 10))  ; is time separation right?
))
 ;; --- rule attempts to patch fades in signal ---
(then
 (progn  ; this creates ksar and sets ksarptr to that ksar
  (send gnode :set-ksarptr (create-segment-merge-ksar gnode))
  (format t "~% 5555555  CLOCK ~a  55555555555555555555 " clock)
  (format t "~%$$$$$ rule 5--- MERGE-SEGMENTS --- fired $$$$$")
  ))))
```

This rule states that:

IF - the goal node is of type "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.

Note that that this rule is disabled by the *send gnode* statement in the consequent of the rule by assigning the *ksarptr* to point to the generated KSAR, since to fire the rule the *ksarptr* must be nil. The format statements are merely for the purpose of printing out a history file on a BB run. The first format statement will print out a line like "555555 CLOCK 7 5555555555" indicating that rule 5 was fired at clock time 7. The second format statement would similarly print out on a new line "$$$$$ rule 5 -- MERGE-SEGMENTS -- fired $$$$$$"

The above rule creates a KSAR by a call to create-merge-segment-ksar function which simply first makes an instance of the KSAR flavor and then pushes this instance into the KSAR queue. This function is fairly easy and is shown below:

```
;;
;; This function creates the merge-segments ksar
;;
(defun create-merge-segment-ksar (gnode)
 (sendksarpush
 (make-instance 'ksar
            :priority 1
            :ksar-id 'merge
            :ks nil
            :boot '(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 1074948> is an instance of flavor ksar with instance variables:
      priority:      1
      ksar-id:       extension
      ks:            nil
      cycle:         40
      trigger"       nil
      context:       <bbevent 1071572>
      preconditions: nil
      boot:          (merge-segments)
      nodeptr:       nil
      channel:       nil
      messenger:     nil
      command:       nil
      arglyst:       nil
      anslyst:       nil
      preboot:       nil
      prelyst:       nil
```

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

```
(defflavor ksar  (
                  priority      ;; static priority now
                  ksar-id       ;; used at present
                  ks            ;; ks to be fired
                  cycle         ;; cycle created
                  trigger
                  context       ;; arguments to the function boot
                  preconditions ;; undefined for now
                  boot          ;; the function call for the ks
                  nodeptr       ;; can point to any node
                  channel       ;; nil no transmission, -1 ready-to-read,
                                ;; 1 ready-to-write
```

```
        messenger    ;; the i/o handler for this ksar
        )
(ks-protocol)
:gettable-instance-variables
:settable-instance-variables
:inittable-instance-variables)
```

In the above KSAR, the variable *priority* needs some explaining. We said earlier that a separate KSAR queue is created for each KS. This is our goal in this research project and has not been fully achieved yet. At this time in RTBB, we have separate KSAR queues for only the distributed KSAR's, these corresponding to the GETBEAM and GETASSIGNMENT KS's. The queue for the GETBEAM KS is called beam-queue and the one for the GETASSIGNMENT KS assign-queue. All the atomic KSAR's are enqueued separately; this queue is called the atomic-queue. While the beam-queue and the assign-queue are FIFO, as they should be, it would be unreasonable to impose the same queueing discipline on the atomic-queue. The instantiation of the variable *priority* reflects the priority that should be accorded to the KSAR shown in the atomic-queue.

The variable *ksar-id* is instantiated to a symbol that reflects the general activity of the KS that will be invoked by the KSAR -- in this case the activity is 'merge'. The variable *ks* will usually be instantiated to the previous activity that resulted in a data node whose addition to the data panel gave rise to the goal node which led to the present KSAR. The reader should note that both these variables are not important to the processing of KSAR's and have not been used in a consistent manner.

The variable *cycle* is instantiated to clock time at which the KSAR was created. The variable *trigger* also is not important to KSAR processing and should be ignored. The variable *context* is instantiated to the pertinent aspects of the context at the time of the KSAR creation. The instantiation for this variable can be as simple as just the internal identity of the bbevent 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 variable *preconditions* is not used at this time. Perhaps at a future date this variable could be used to ascertain that at the time the KS is fired the conditions that gave rise the KSAR are still valid. For such usage, *preconditions* would be set to the minimum conditions that must hold for the KS to be fired for satisfying the goal node that gave rise to the KSAR.

The variable *boot* is important and is instantiated to the name of the KS which the KSAR must invoke. The actual execution of the KS takes place by making a function call composed of the name of the KS followed by appropriate arguments. The variable *nodeptr* is set to the internal identity of the goal node that gave rise to the KSAR. The other variables in the above KSAR will be explained after we introduce the notion of a

distributed KSAR.

When a KSAR of the type shown above is selected and its corresponding KS executed, then during the KS processing the control resides completely in the KS. In other words, the main BB process simply waits for the KS to finish up before focussing on any other activity. We will call such KSAR's *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 alter during the execution of the KS, you don't have to worry about the inapplicability of what might 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 usually associated with blackboard problem solving. 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 executions of KS's are 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 to deal 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 [13]. 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 interfernce between the KS's.

In addition to *atomic* KSAR's, in RTBB we also have another type of KSAR's that permit 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 flavor 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 does a non-blocking read of the KS port to see if the KS is finished. A non-blocking read first checks the port to see if there is data available before actually reading the data. If we had used a regular read, as for example via *read* or *tyipeek* functions, and there was no

data available at the port, the used function may either wait indefinitely 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 has returned the results, then read the results if available. In the absence the results, we wanted the system to move on to other tasks, to return to the KS at a later time. Hence, the reason for non-blocking read. The non-blocking read function 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 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 1074284> is an instance of flavor ksar with instance variables:
priority:         1
ksar-id:          segment
ks:               hit
cycle:            40
trigger:          change
context:          ((time nil) (number <bbevent 1075036>) &)
preconditions:    empty
boot:             (post-assign-hits)
nodeptr:          <bbevent 1075036>
stage:            1
messenger:        <messenger 1072204>
command:          getassignment
arglyst:          ('((8 93.54559999999999 6.019744 0.0)
                     (8 5.52 93.54559999999999 0.0)
                     (8 93.54559999999999 5.52 0.0))
                   '((9 92.67895 6.1425 0.0)
                     (9 6.1425 92.67895 0.0)
                     (9 92.67895 6.642136 0.0)))
anslyst:          nil
preboot:          (pre-assign-hits)
prelyst:          ((<snode 1073184> <snode 1073144>
                         <snode 1072948>) & & & 9)
```

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

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

```
(defflavor ks-protocol
 (
 command    ; the input function
 arglyst   ; the argument list
 anslyst   ; answer list
 preboot    ; command to start up function after read
 prelyst   ; argument list for reboot after read
 )

 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables
)
```

As will be evident from the following definitions of the variables, the mixin flavor is only useful for a distributed KSAR. Since all KSAR instances use the mixin, the reader might wonder why use the mixing ks-protocol at all; after all, the variables in the mixin could have been incorporated in the definition of the ksar flavor. 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 variables in the above distributed KSAR. We have already explained the nature of the variables from *priority* through *nodeptr* in connection with atomic KSAR's. We will define the other variables. The variable *stage* is instantiated to either 2, 1, -1, or 0. When the instantiation is 1, the KSAR is in the first stage, meaning that it is ready to send a command to the KS that would initiate the execution of the KS; the command is taken off the variable *command* and its arguments from *arglyst*. After the command is transmitted to the KS, the instantiation of *stage* is set to -1, 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 instantiation of *stage* is changed to 0. The instantiation of 0 for *stage* causes the function that is at *boot*, 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 *stage* 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 variable

*The usage of *boot* in a distributed KSAR is different from that in an atomic KSAR. In the latter, *boot* held the function name for invoking the KS, a job now carried out by the instantiation of *command*. This inconsistency in the usage of *boot* and some other variables is owing to the manner in which RTBB has evolved. The authors apologize to the readers for any resulting confusion.

arglyst already has an instantiation, so KSAR processing can begin in stage 1. While in some cases *arglyst* instantiation can be generated easily 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 stage to the three stages we have already mentioned. This additional stage is specified by instantiating *stage* to 2. When the scheduler sees this instantiation, it puts out a function call which constructs the arguments, the function call being held in the variable *preboot*. In the above example, the instantiation of *arglyst* was generated by a call to the function (pre-assign-hits) during the stage when *stage* is 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 variable *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 under the variable *arglyst*, the time-stamp corresponding to this argument is 8. The second argument under *arglyst*, corresponding to time-stamp 9, 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 flavor which acts as an I/O handler for the BB. The write command is synthesized by the following method that is defined for the ks-protocol flavor.*

```
;; 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 (ks-protocol :write-ks) ()
 (format t "COMMAND sent to ks ~a~%" (cons command arglyst))
 (format (send ; get output port name from messenger flavor
         (send self :messenger) ; get messenger name from variable
         :write-port) "~a~%"
        (cons command arglyst)) ; form function call
```

*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 sending the ':write-ks' message to the ks-protocol flavor.

```
(send self :set-stage -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 variables *command* and *arglyst* in the KSAR and sends the command to the port. Before exiting, the method also changes the state of the KSAR *stage* 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 variable *messenger* in the distributed KSAR example we showed above. To understand the function of this variable, note that we need to associate with each KS an I/O handler; the handler should contain 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 flavor shown below.

```
;; This is the flavor messenger

(defflavor messenger   ;; these should be named after ks's
 (
      write-port   ; the output port to the process
      write-fd     ; the output port file descriptor
      read-port    ; the input port to the process
      read-fd      ; the input port file descriptor
      pid          ; the process identity
      )
      ()
      :gettable-instance-variables
      :settable-instance-variables
      :inittable-instance-variables
)
```

The variables *write-port* and *read-port* are instantiated to internally generated symbolic names that designate the two ports; the symbolic names are returned by a '*process' call like

```
(*process 'path t t)
```

This call will return

```
(#<port from-process> #<port to-process> 13067)
```

where the symbolic name #<port from-process> is the output port of the unix process whose processor id is 13067, the unix process in this case being 'path'. Similarly, #<port to-process> is the symbolic name of the input port of the process, which calls up the unix process representing the KS. A reader not too familiar with the lisp-unix interface might want to know that a call like (*process 'path) would actually run the unix process 'path'. The variables *write-fd* and *read-fd* are instantiated to the file descriptors for the two ports;

these file variables were good for diagnostics and are not being currently used for anything. The variable *pid* is instantiated to the process id of the process; this variable also is not used at this time. The variable *messenger* in the distributed KSAR shown previously is instantiated to the identity of that instance of the messenger flavor 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 assign-queue, respectively; all the other KSAR's are enqueued into a single queue called the atomic-queue (Fig. 3). Each KSAR queue is an instance of the following event flavor.

```
(defflavor event (
          number
          (mask '(1 1 ))
          (atomic-queue '())
          (beam-queue '())
         (assign-queue '())
          )
 ()
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables)
```

Note the mask represents the status of the KSAR's that are currently at the head of the queues. The list that is the instantiation of *mask* has a status value for each of the distributed-KSAR queues, and the interpretation to be given to each value in the list is the same as that given to the instantiations for the variable *stage* in a distributed KSAR. In the *defflavor*, the initial mask values have been set to 1 for head KSAR's in both the beam-queue and the assign-queue, meaning that if any KSAR's are found at the heads of the respective queues, they are in stage 1. Recall that stage value of 1 corresponds to write stage in which commands are written to the KS's.

A single instance of the above flavor is made, the resulting object being called ksarq. The variable *atomic-queue* of this object, initially a null list, is instantiated to the list of all the atomic KSAR's, the variable *beam-queue* to the list of all the distributed KSAR's that seek to invoke the GETBEAM KS, and, finally, the variable *assign-queue* to the list of all the distributed KSAR's that seek to invoke the GETASSIGNMENT KS.

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 upon whether the KSAR is in the atomic-queue or one of the other queues. For the atomic-queue the KS is threaded into

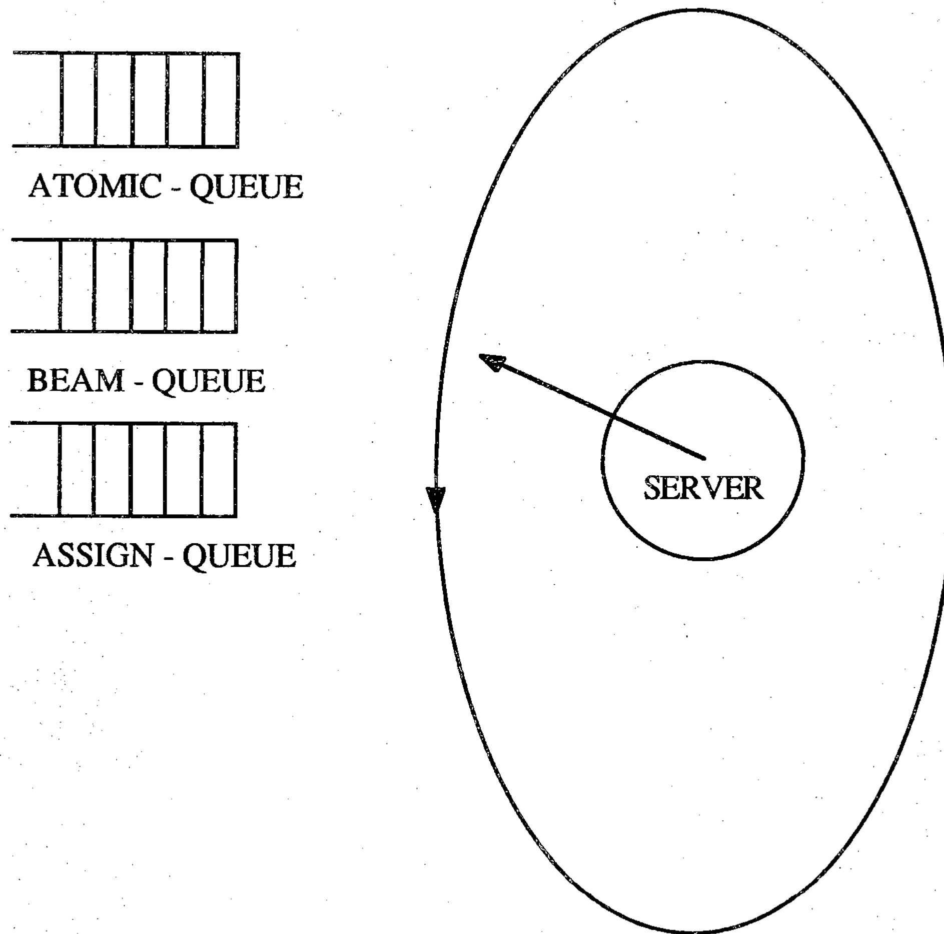


FIGURE 3. This is the actual KSAR queueing system currently employed in the RTBB.

the BB process before visiting the other queues. On the other hand, for the beam-queue and assign-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 ksarq during execution.

```
<event 1071376> is an instance of flavor event with instance variables:
number:          3
mask:            (1 nil 2)
atomic-queue:    (<ksar 1074140> <ksar 1074212>)
beam-queue:      nil
assign-queue:    (<ksar 1074284>)
```

Here the mask represents the status of the atomic-queue as 1 which does not mean anything for this queue, nil for the beam queue which is empty and 2 for the assign-queue which means that the KSAR is ready for the preboot function to be run. The variable *number* is instantiated to the total number of ksar held in the queueing system and is updated at every change by a defmethod.

We will now make comments about 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 fourth clock unit. The scheduler now looks at all the queues, first examining the atomic-queue, which it finds empty. 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* variable of ksarq. Finally, the scheduler looks at the assign-queue, which it finds empty also. The process then repeats, as depicted in Fig. 3.

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this is the main loop for driving the BB
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun cloop ()
```

```
(do ()   ;put into infinite loop
   (())
(go-for-it )  ; allows you to choose the number of control cycles
(clock-update) ; updates the clock variable and place a goal at the
             ; hit level every fourth clock unit. It also places
             ; a purge-segments goal at the hit level every fourth
             ; cycle.
(plan-goals)  ; maps the goals into ksars, it calls the rule-based
             ; planner
(scheduler)   ; runs the scheduler which cycles through the three
             ; KSAR queues in ksarq.
))
```

The comments explain the nature of each function in the main control loop.

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 programmings aspects of a blackboard. We hope this report has rectified that deficiency, if only to a slight 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.

Although, the RTBB as discussed in this report does work, 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. ACKNOWLEDGEMENT

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.

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 2,* Technical Report TR-EE 88-9 School of Electrical Engineering, Purdue University, 1988.

[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 Systems,* 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 Systems,* Addison-Wesley, 1988.

[9] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-II 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] *Franz Lisp Reference Manual,* ECN No. 750, Purdue University, March, 1987.

[13] 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.

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

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

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

[17] 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.

[18] 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.

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

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

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

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

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

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

[25] 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.

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

[27] 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 workings of the RTB blackboard. The examples are at increasing levels of difficulty, starting with the case of a single track formed by a single craft in the first example; progressing to two stable tracks formed by three separate craft in the second example; further progressing to the case where initially three craft form a single track, but eventually form only two tracks as one craft breaks away; and, finally, dealing with the problem of fading tracks in the last example.

Example 1

In this example, there is a single craft. Most of the flavors are expanded out to illustrate the detail of 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 trapazoid which defines the space curve is given by the four points indicated in Figure 4. 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, it represent not only the origin of the coordinate system but also 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 distributed although no *preboot* function is necessary since the function call is so simple. The *preboot* may be considered as a precondition type of function which freezes the context and extracts the variables needed by the KS. The *preconditions* variable is never used, a misnomer arising out of development process. The KSAR looks as follows:

```
<ksar 1072368> is an instance of flavor ksar with instance variables:
priority        2
ksar-id:        newhit
ks:             add
cycle: 1
trigger:        clock
context:        none
preconditions:      empty
boot:           (getbeam)
nodeptr:        nil
```

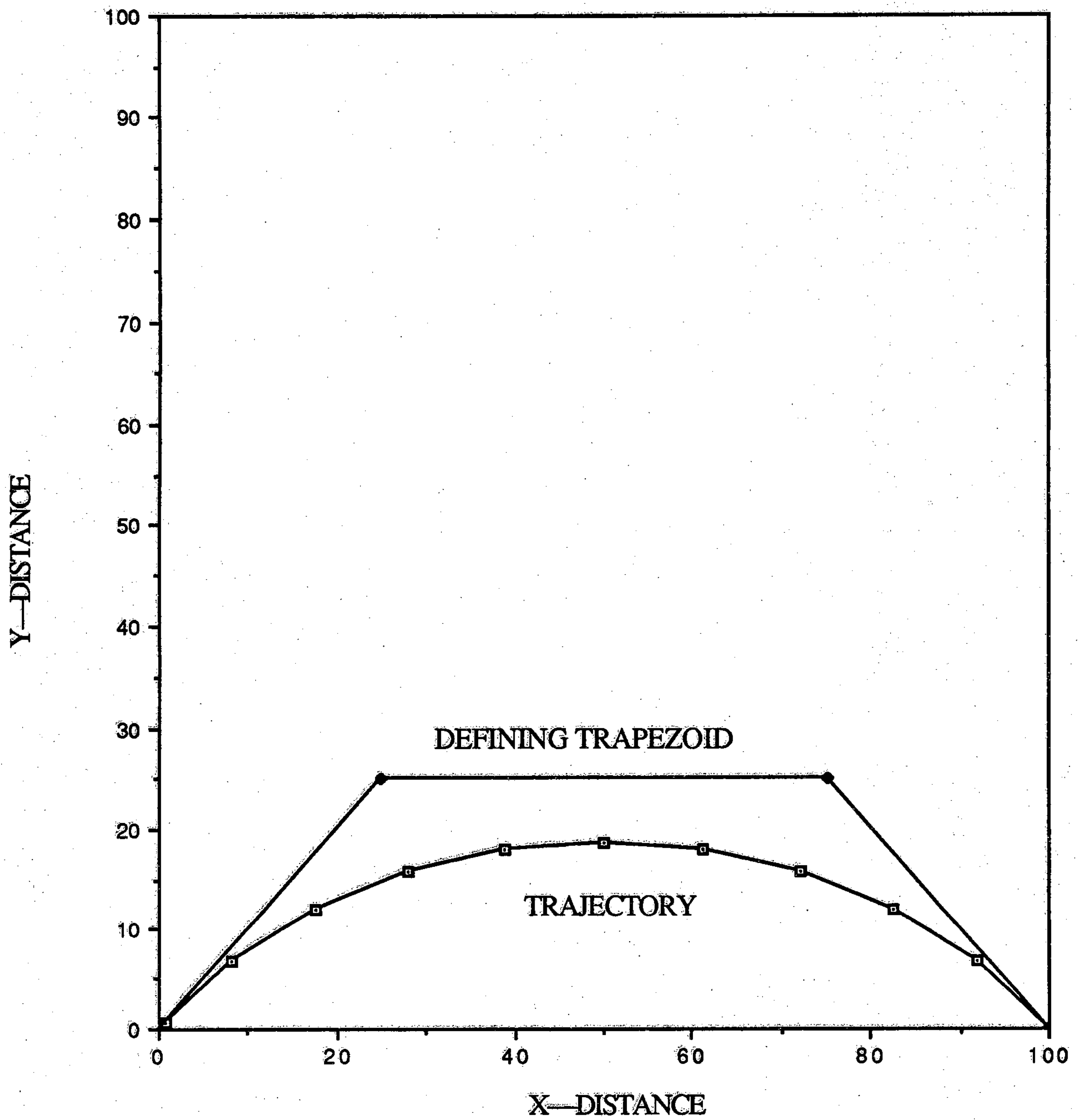


FIGURE 4. Single trajectory generated via Bezier's curve with every tenth point shown. Defining trapezoid shown with curve.

channel: 1
messenger: <messenger 1072044>
command: fire
arglyst: nil
anslyst: nil
preboot: nil
prelyst: nil

The *boot* is the C coded KS called GETBEAM 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. The data node looks as follows:

<bnode 1072668> is an instance of flavor bnode with instance variables:
type: hit
time: 0
coord: ((100.0 0.0 0.0))
number: 1

Note the return count is given by *number* and it occurs at *time* 0 at the coordinates *coord*. The node type is specified by *type*.

The placement of this hit node on the data panel causes the placement of a goal node on the segment level of the goal panel. This goal node looks as follows:

<bbevent 1072808> is an instance of flavor bbevent with instance variables:
source: <bnode 1072668>
action: change
type: hit
variable: coord
time: 0
coord: ((100.0 0.0 0.0))
number: 1
threat: nil
snode: nil
pattern: nil
duration: one-shot
position: nil
goalptr: nil
conditions: nil
ksarptr: nil

This goal represent the desire to match this data to the nearest segments. The *duration* of the goal node is one-shot meaning the rule base gets only one pass to satisfy it, otherwise the goal node is removed from the goal panel. The *source* is a pointer to the data node responsible for the creation of the goal node by the distributed monitor.

The rule base causes the segment goal node to generate a KSAR to match the hit data to a nearest segment, if there is one. Otherwise, it creates a new segment. Again, the *ksar-id* generally describes the driving activity, i.e. segment formation. The KSAR for this KS is distributed with a separate *preboot* function which forms the arguments for the blackboard formation. The *boot* function is really the "postboot" function that is

improperly named since its use changed while the blackboard system evolved - and its name is too embedded to easily change. The *command* variable carries the main KS call function. It looks as follows:

```
<ksar 1072876> is an instance of flavor ksar with instance variables:
priority:      1
ksar-id:       segment
ks:    hit
cycle: 4
trigger:       change
context:       ((time nil)(number <bbevent 1072808>) &)
preconditions:    empty
boot:          (post-assign-hits)
nodeptr:       <bbevent 1072808>
channel"       2
messenger:  <messenger 1072204>
command:    getassignment
arglyst:       nil
anslyst:       nil
preboot:       (pre-assign-hits)
prelyst:       nil
```

The GETASSIGNMENT KS which is fired by this KSAR matches the segments to the data and the *post-assign-hits* function places the nodes on the data panel at the segments level. This segment node looks as follows:

```
<snode 1072948> is an instance of flavor snode with instance variables:
type:  segment
time:  (0)
coord:         ((100.0 0.0 0.0))
number:        nil
cpa:   nil
linear:        nil
tnode:         nil
threat:        nil
```

Most of the variables are initially *nil* since the segment is not long enough; however, this will fill in at a later time when the segment becomes part of a track. In fact, two time units later the snode looks as follows:

```
<snode 1072948> is an instance of flavor snode with instance variables;
type:  segment
time:  (1,0)
coord:         ((99.24254999999999 0.7425 0.0) (100.0 0.0 0.0 ))
number:     2
cpa:   (49.00339911339883 49.99006688005953 0.0)
linear:        ((99.24254999999999 0.7425 0.0 )
        (-0.7574500000000057 0.7425 0.0 ))
trode: nil
threat:        nil
```

At the formation of a segment data node, a demon from the distributed monitor creates a

track goal node which represents the desire to form a track from the segments. The goal node looks as follows:

```
<bbevent 1073816> is an instance of flavor bbevent with instance variables:
source:          <snode 1072948>
action:          change
type:       segment
variable:   number
time:       (1 0)
coord:           ((99.24254999999999 0.7425 0.0)
            (100.0 0.0 0.0))
number:          2
threat:          nil
snode:           nil
pattern:    nil
duration:   one-shot
position:   nil
goalptr:    nil
conditions: nil
ksarptr:    nil
```

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

Again, the rule base creates from this track goal node the following KSAR, whose purpose is to form tracks from the segments.

```
<ksar 1074640> is an instance of flavor ksar with instance variables:
priority:   0
ksar-id:    track
ks:         segment
cycle:      16
trigger:    change
context:    ((time nil)(number <snode 1072948>) &)
preconditions:   empty
boot:       (assign-tracks)
nodeptr:    <snode 1072948>
channel:    1
messenger:  nil
command:    nil
arglyst:    nil
analyst:    nil
preboot:    nil
prelyst:    nil
```

Note that this KSAR is an atomic KSAR unlike the previous KSAR on the assignment of hits. The KS places a track node on the data panel at the track level. This node looks as follows:

```
<tnode 1074228> is an instance of flavor tnode with instance variables:
type:       track
```

```
time:        (1)
last-coord:  (99.24254999999999 0.7425 0.0)
last-velocity:       (-0.574500000000057 0.7425 0.0)
threat:      nil
snode:               (<snode 1072948>)
cpa-bracket:((43.97948402473871 54.02731429295895)
             (45.063561019205357 54.91482356806549))
check:               nil
checklyst:   nil
```

This data node *time* variable contains only the current time. The *last-coord* and *last-velocity* are the corresponding position and velocity. The variable snode contains a list of pointers to the segments which form the support for the tracks. The confidence region which is called *cpa-bracket* causes the *threat* variable to be marked 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 later time

```
<snode 1072984> is an instance of flavor snode with instance variables:
type:  segment
time:  (4 3 2 1 0)
coord:        ((96.8832 2.88 0.0)
       (97.68385000000001 2.1825 0.0)
       (98.4704 1.47 0.0)
       (99.24254999999999 0.7425 0.0)
       (100.0 0.0 0.0))
number:       5
cpa:   (43.2289227832382 49.621934519913962 0.0)
linear:       ((96.88322.88 0.0)
       (-0.800650000000045 0.6974999999999998 0.0))
tnode:        <tnode 1074228>
threat:       nil
```

Note that the *cpa* has been calculated and the *linear* variable 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 contained in tnode. Information has been sent to this node either by demons associated with these nodes or by the KS's themselves.

After the segment information has been extended to more than thirteen points, the list is truncated. This is accomplished with after-method so that after say 20 time units, the snode looks as follows:

```
<snode 1072948> is an instance of flavor snode with instance variables:
type:  segment
time:  (20 19 18 17 16 15 14 13 12 11 10 9 8)
```

```
coord:      ((82.40000000000001 12.0 0.0)
        (83.38545000000001 11.5425 0.0) (84.3616 11.07 0.0)
        (85.32814999999999 10.5825 0.0) (86.2848 10.08 0.0)
        (87.23125 9.5625 0.0)
        (88.16719999999999 9.029999999999999 0.0)
        (89.09235 8.4825 0.0) (90.0064 7.92 0.0)
        (90.90904999999999 7.3425 0.0) (91.8 6.75 0.0)
        (92.67895 6.1425 0.0) (93.54559999999999 5.52 0.0))
number:     13
cpa:  (19.19401128389733 41.34369053489974 0.0)
linear:     ((82.40000000000001 12.0 0.0)
        (-0.9854500000000002 0.4574999999999996 0.0))
tnode:      <tnode 1074228>
threat:     nil
```

In the above snode the maximum length of the *coord* and *time* variables are now only of length thirteen as fixed by a global variable. The truncation length may set to any fixed value but this variable 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. So the truncation length must be chosen carefully, otherwise, the entire program could be comfounded.

The general sequence of KS calls is outlined in Figure 5. 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 is the hit generation KS (GETBEAM), the second KS is the GETASSIGMENT KS and the third KS is the track formation KS (GETTRACK). Demons from the distributed monitor generate the goal nodes from the data nodes. The simple construction illustrated is essentually data driven with goal nodes being isomorphically mapped to KS. 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 and and especially the grouping of segments into tracks.

In Figure 6, there is a plot of 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 Figure 6 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. So each tree groups the time sequences forming a track and the set of all tracks formed from the data is a forest.

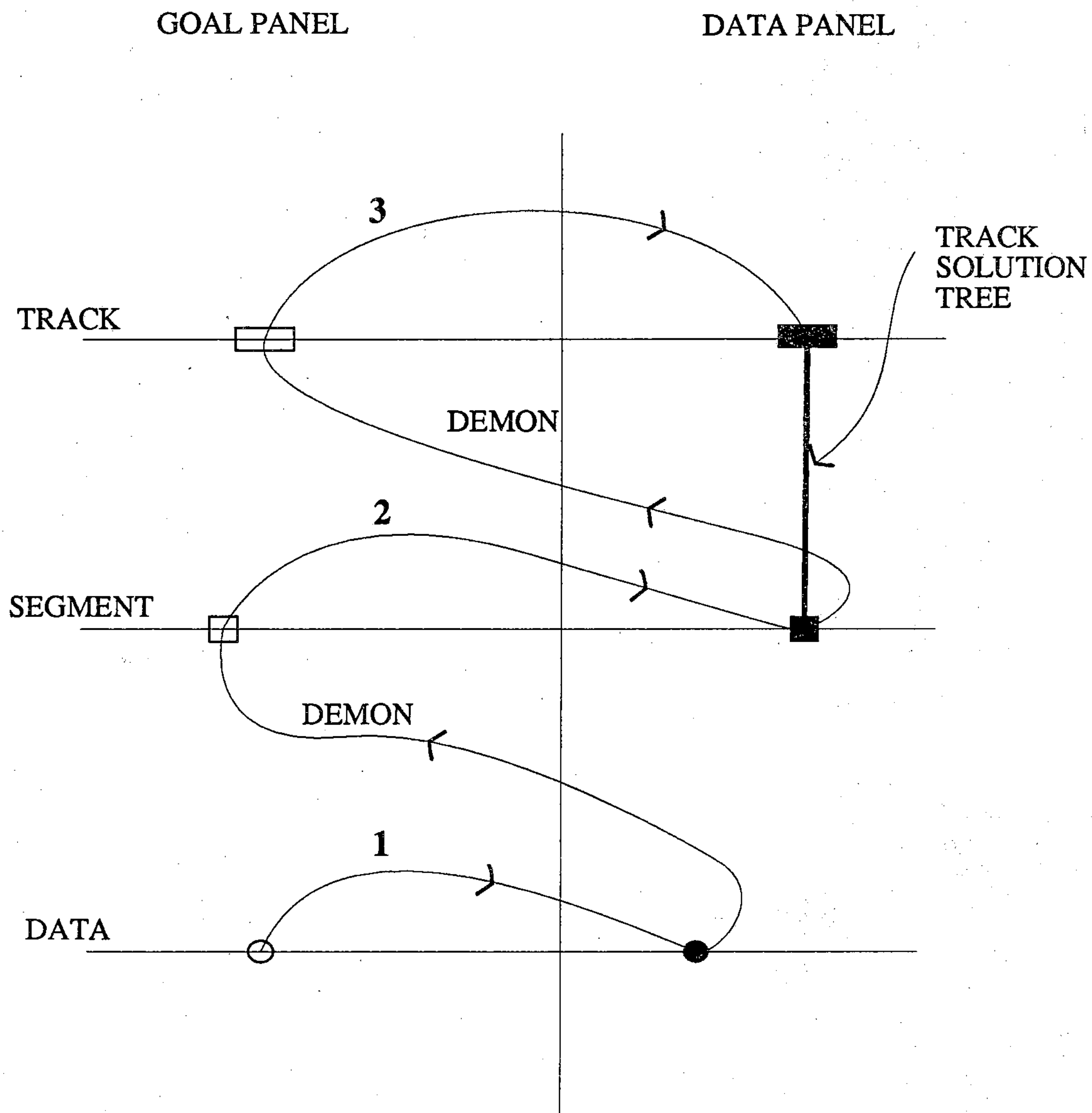


FIGURE 5. The track formation for a single trajectory is outlined here.

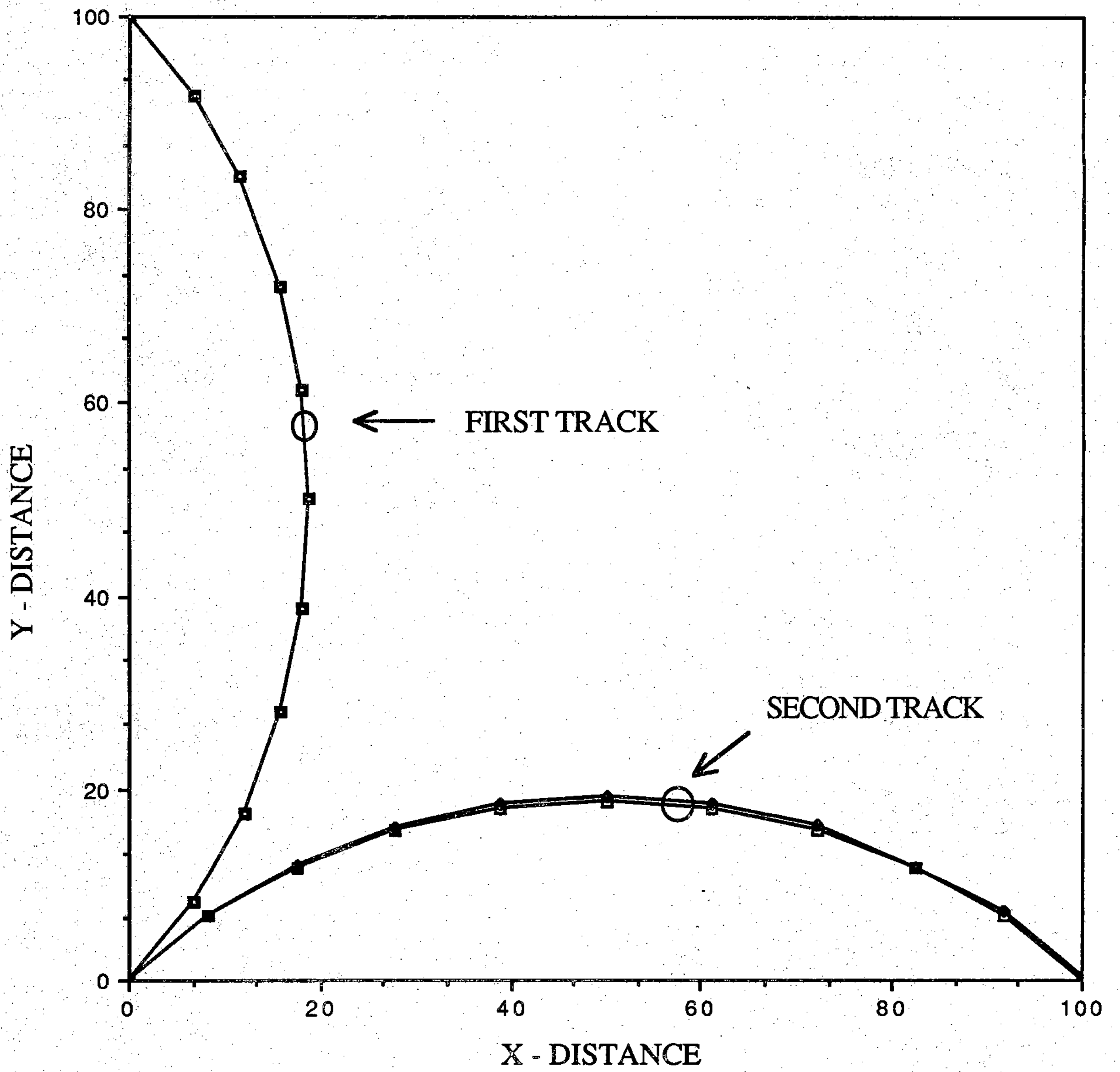


FIGURE 6. Trajectories for three separate craft with two tracks indicated.

Figure 7 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 on 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 look as follows:

```
<tnode 1074720> is an instance of flavor tnode with instance variables:
Type: track
time: (5)
last-coord:  (96.07797734375001 3.905422931640625 0.0)
last-velocity:      (-0.8144500000000079 0.6824700000000004 0.0)
threat:      nil
snode:       (<snode 1072948> <snode 1073184>)
cpa-bracket:((36.18550040607643 54.2522089149583)
        (44.94766067922566 55.14530386696201))
check:       nil
checklyst:   nil

<tnode 1074676> is an instance of flavor tnode with instance variables:
type:  track
time:  (5)
last-coord:  (3.6225 96.12575 0.0)
last-velocity:      (0.6825000000000001 -0.8144500000000079 0.0)
threat:      nil
snode:       (<snode 1073144>)
cpa-bracket:((44.80414791166273 54.91482356806549)
        (35.91704278661875 54.02731420205895))
check:       nil
checklyst:   nil
```

Note <tnode 1074720> is the second track in Figures 6 and 7 with two supporting segment nodes. *Snode* is the list of pointers to segment nodes. The pointers contained in *snode* are the branches of the solution tree and the supports for the track hypothesis. The other track node <tnode 1074676> 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 its plot of the trajectories indicates that this will not be true in the future.

The snodes in this case also contain parent pointers which establish which track they support. These nodes look as follows:

```
<snode 1073184> is an instance of flavor snode with instance variables:
type:  segment
time:  (5 4 3 2 1 0)
```

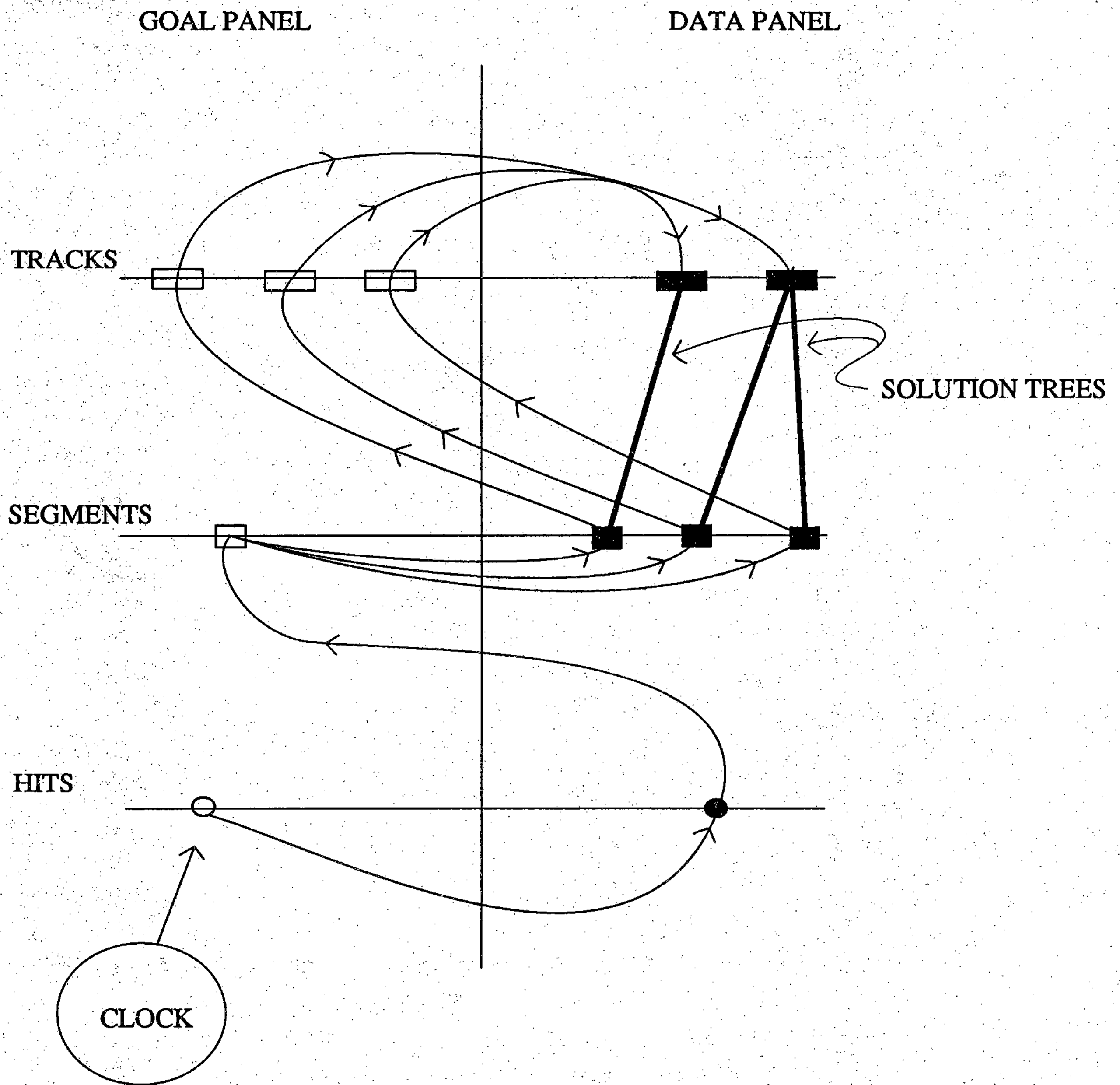


FIGURE 7. The nodes on the BB for the 3-trajectory, 2-track example. Shown here are two solution trees.

coord: ((96.06874999999999 4.062438 0.0)
(96.8832 3.379968 0.0)
(97.68385000000001 2.682486 0.
number: 6
cpa: (41.62943218734221 49.67997281196416 0.0)
linear: ((96.06874999999999 4.062438 0.0)
(-0.8144500000000079 0.6824700000000004 0.0))
tnode: <tnode 1074720>
threat: nil

<snode 1073144> is an instance of flavor snode with instance variables:
type: segment
time: (5 4 3 2 1 0)
coord: ((3.5625 96.06874999999999 0.0) (2.88 96.8832 0.0)
(2.1825 97.68385000000001 0.0) (1.47 98.4704 0.0)
(0.7425 99.24254999999999 0.0) (0.0 100.0 0.0))
number: 6
cpa: (49.38655323518081 41.38537980601705 0.0)
linear: ((3.5625 96.06874999999999 0.0)
(0.6825000000000001 -0.8144500000000079 0.0))
tnode: <tnode 1074676>
threat: nil

<snode 1072948> is an instance of flavor snode with instance variables:
type: segment
time: (5 4 3 2 1 0)
coord: ((96.06874999999999 3.5625 0.0)
(96.8832 2.88 0.0)(97.68385000000001 2.1825 0.0)
(98.4704 1.47 0.0) (99.24254999999999 0.7425 0.0)
(100.0 0.0 0.0))
number: 6
cpa: (41.38537980601705 49.38655323518081 0.0)
linear: ((96.06874999999999 3.5625 0.0)
(-0.8144500000000079 0.6825000000000001 0.0))
tnode: <tnode 1074720>
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 1074720> is an instance of flavor tnode with instance variables:
type: track
time: (41)
last-coord: (60.0895833333333418.3031817537037 0.0)
last-velocity: (-1.111450000000005 0.1400399999999991 0.0)
threat: t
snode: (<snode 1072948> <snode 1073184>)
cpa-bracket:((-2.437209695431297 54.2522089149583)
(24.69991086768157 55.14530386696201))
check: t
checklyst: nil

```
<tnode 1074676> is an instance of flavor tnode with instance variables:
type:  track
time:  (41)
last-coord:  (18.7425 60.44255 0.0)
last-velocity:     (0.1424999999999983 -1.111450000000005 0.0)
threat:       t
snode:        (<snode 1073144>)
cpa-bracket:((24.69991086768157 54.91482356806549)
        (-2.422621460220589 54.02731420205895))
check:        nil
checklyst:  nil
```

By now both tracks represent threats to the origin and so the threat variable is instantiated as true. Note that the confidence region represented by the *cpa-bracket* has one coordinate which straddles the origin. Although this condition is an arbitrary and probably not a sharp criterion, the point is to illustrate the detection via the rule based system.

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.

In Figure 8, there is a plot of three trajectories. All of these trajectories are initially very close and logically form a 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. The spline test is a backchaining algorithm which 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 finer granularity in KS's and easier implementation of more complex goal interrelationships.

The nodes or solution tree should reflect the history of this trajectory. Indeed, Figure 9 shows the parallel 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

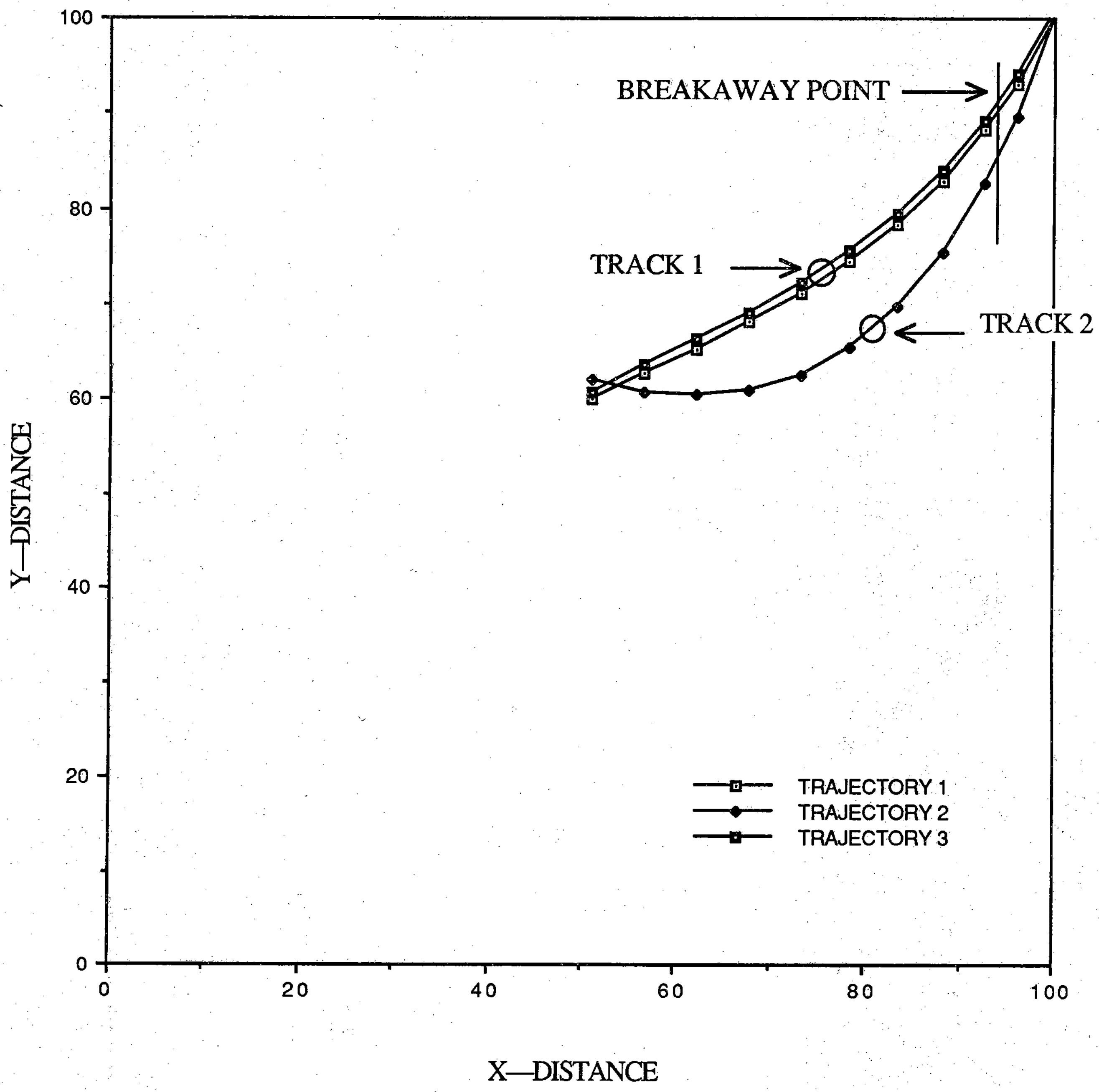


FIGURE 8. The three-aircraft problem where one track breaks away.

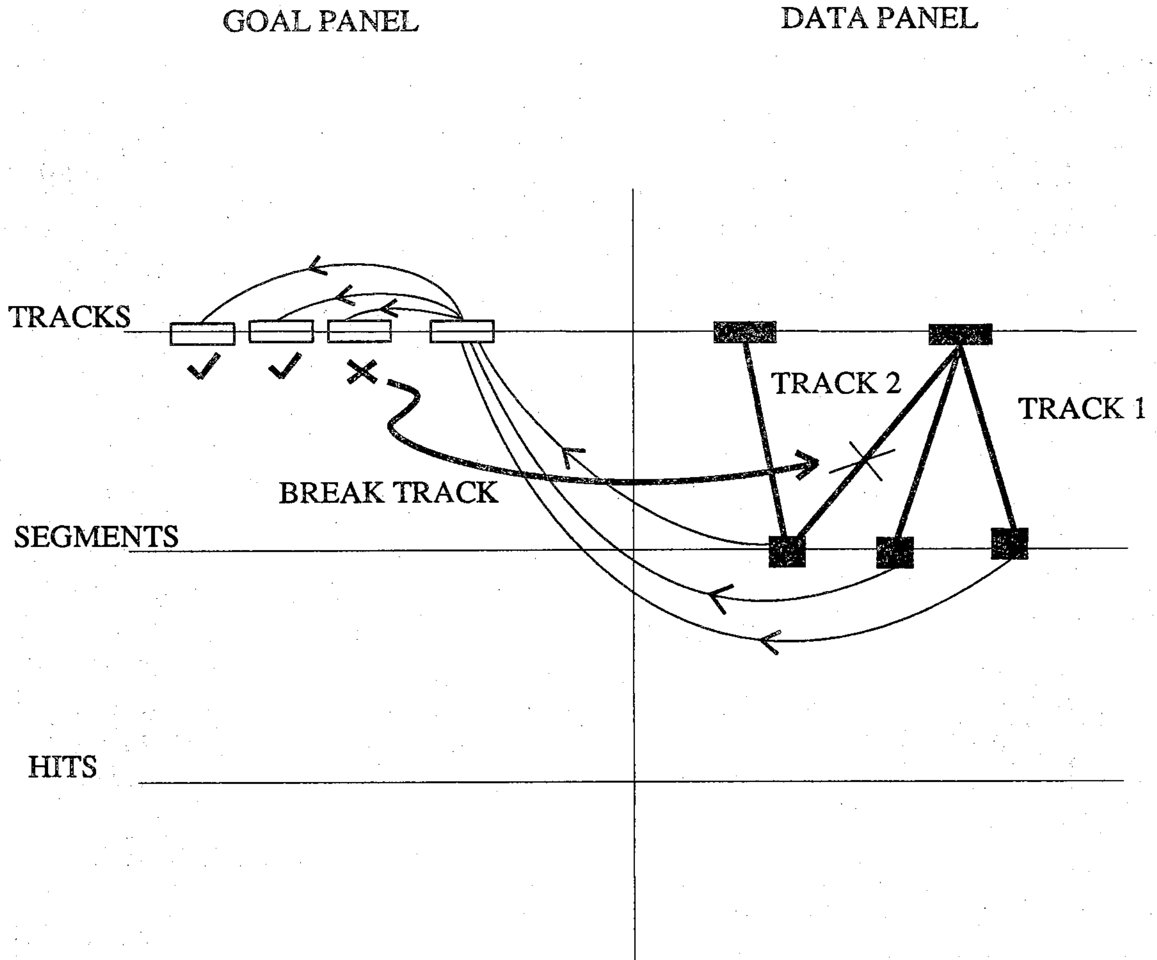


FIGURE 9. This example illustrates the cancellation of the segment node support of the track 1 hypotheses. This node eventually supports a new track hypothesis. Subgoaling triggered by the failure of the spline test is illustrated in the goal panel.

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 upon the goal segment level. Each goal represents the desire to determine if that segment is in the same equivalence class 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.

Initially, the track node formed from the three segments looks as follows:

```
<tnode 1074676> is an instance of flavor tnode with instance variables:
type:  track
time:  (1)
last-coord:   (99.24254999999999 98.61066633333331 0.0)
last-velocity:        (-0.7574500000000057 -1.722667000000001 0.0)
threat:       nil
snode:        (<snode 1072948> <snode 1073144>
      <snode 1073184>)
cpa-bracket:((32.08760233000798 62.90600256325121)
      (-31.94628530341367 -7.667384301036718))
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 are really the parent pointers or the edges of the graph pointing to the root of the tree which represents the track.

```
<snode 1073184> is an instance of flavor snode with instance variables:
type:  segment
time:  (1 0)
coord:        ((99.24254999999999  99.522324 0.0)
      (100.0 101.0 0.0))
number:      2
cpa:  (38.19259757273452 -19.5773518900408 0.0)
linear:       ((99.24254999999999  99.522324 0.0)
      (-0.7574500000000057 -1.477676000000002 0.0))
tnode:        <tnode 1074676>
threat:       nil

<snode 1073144> is an instance of flavor snode with instance variables:
type:  segment
time:  (1 0)
coord:        ((99.24254999999999  98.522325 0.0)
      (100.0 100.0 0.0))
number:      2
cpa:  (38.59849373098595 -19.78542580508939 0.0)
linear:       ((99.24254999999999  98.522325 0.0)
      (-0.7574500000000057 -1.477675000000005 0.0))
tnode:        <tnode 1074676>
```

```
threat:     nil

<snode 1072948> is an instance of flavor snode with instance variables:
      type:  segment
time:  (1 0)
coord:      ((99.24254999999999 97.78735 0.0)
      (100.0 100.0 0.0))
number:     2
      cpa:  (58.86860840361245-20.15231845764879 0.0)
linear:     ((99.24254999999999  97.78735 0.0)
      (-0.7574500000000057 -2.212649999999996 0.0))
tnode:      <tnode 1074676>
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.

At the time stamp of 11, the track is determined to be a threat to the origin and the spline KS (GETSPLINE) will now begin to check to see if 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 1074676> is an instance of flavor tnode with instance variables:
type:  track
time:  (11)
last-coord:  (90.91376606802292 86.24495864263466 0.0)
last-velocity:     (-0.8909500000000037 -1.085756000000003 0.0)
threat:     t
snode:      (<snode 1072948> <snode 1073144> <snode 1073184>)
cpa-bracket:((3.746220505494785 62.90600256325121)
      (-32.05750115050004 0.08018509395760631))
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 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.

```
<bbevent 1074788> is an instance of flavor bbevent with instance variables:
source:     <tnode 1074676>
action:     verify-track
type:  track
variable:   nil
time:  (12)
coord:      ((90.00946578449609 82.81725806347009 0.0)
```

```
        (-0.90264999999999942 -1.052371999999991 0.0))
number:     nil
threat:     nil
snode:      <snode 1073184>
pattern:    nil
duration:   one-shot
position:   nil
goalptr:    nil
conditions: nil
ksarptr:    nil

<bbevent 1074720> is an instance of flavor bbevent with instance variables:
source:     <tnode 1074676>
action:     verify-track
type:  track
variable:   nil
time:  (12)
coord:      ((90.00946578449609 82.81725806347009 0.0)
        (-0.90264999999999942 -1.052371999999991 0.0))
number:     nil
threat:     nil
snode:      <snode 1073144>
pattern:    nil
duration:   one-shot
position:   nil
goalptr:    nil
conditions: nil
ksarptr:    nil

<bbevent 1075076> is an instance of flavor bbevent with instance variables:
source:     <tnode 1074676>
action:     verify-track
type:  track
variable:   nil
time:  (12)
coord:      ((90.00946578449609 82.81725806347009 0.0)
        (-0.90264999999999942 -1.052371999999991 0.0))
number:     nil
threat:     nil
snode:      <snode 1072948>
pattern:    nil
duration:   one-shot
position:   nil
goalptr:    nil
conditions: nil
ksarptr:    nil
```

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 node 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 which is orphaned by the VERIFY KS is the following segment node.

```
<snode 1073184> is an instance of flavor snode with instance variables:
type:  segment
time:  (13 12 11 10 9 8 7 6 5 4 3 2 1 0)
coord:       ((89.09235 84.915828 0.0) (90.0064 85.935872 0.0)
         (90.90904999999999 86.98824399999999 0.0)
         (91.8 88.074 0.0) (92.67895 89.19419600000001 0.0)
        (93.54559999999999 90.34988800000001 0.0)
        (94.39964999999999 91.542132 0.0)
        (95.24079999999999 92.771984 0.0)
        (96.06874999999999 94.04049999999999 0.0)
        (96.8832 95.348736 0.0) (97.68385000000001 96.697748 0.0)
        (98.4704 98.08859200000001 0.0)
        (99.24254999999999 99.522324 0.0) (100.0 101.0 0.0))
number:    14
cpa:  (7.210431076363022 -6.461186503081834 0.0)
linear:      ((89.09235 84.915828 0.0)
        (-0.9140500000000031 -1.020043999999999 0.0))
tnode:       nil
threat:      nil
```

After the blackboard detects the unmatched segment node, it constructs a distinct track for this segment and the resulting solution is the two track nodes given below. The first track node is the newly created node from the unmatched segment node. 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.

Track 1 of Figure 9.

```
<tnode 1074280> is an instance of flavor tnode with instance variables:
type:  track
time:  (13)
last-coord:  (89.09235 76.75794999999999 0.0)
last-velocity:     (-0.9140500000000031 -1.382850000000005 0.0)
threat:      nil
snode:       (<snode 1072948>)
cpa-bracket:((20.45362766455384 32.93339536190769)
        (-27.08435313612137 -8.203934384099306))
check:       nil
checklyst:   nil
```

Track 2 of Figure 9.

```
<tnode 1074676> is an instance of flavor tnode with instance variables:
```

type: track
time: (13)
last-coord: (89.09235 84.4169265 0.0)
last-velocity: (-0.9140500000000031 -1.019809500000001 0.0)
threat: t
snode: (<snode 1073144> <snode 1073184>)
cpa-bracket:((-0.977760816000675 15.82484430129828)
(-15.96939768539792 2.67651494722635))
check: nil
checklyst: nil

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 extend-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 Figure 10. Only one of these trajectories fades. Its segment node is given by:

<snode 1073144> is an instance of flavor snode with instance variables:
type: segment
time: (2 1 0)
coord: ((1.47 98.4704 0.0)
(0.7425 99.24254999999999 0.0)
(0.0 100.0 0.0))
number: 3
cpa: (49.92671489395662 47.0396750441671 0.0)
linear: ((1.47 98.4704 0.0)
(0.7274999999999999 -0.7721499999999963 0.0))
tnode: <tnode 1074676>
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 a new segment node is generated on the BB which represents a "newly found segment." It looks as follows:

<snode 1075068> is an instance of flavor snode with instance variables:
type: segment
time: (8 7)
coord: ((5.52 93.54559999999999 0.0)
(4.8825 94.39964999999999 0.0))
number: 2
cpa: (48.38657378899811 36.11783945961739 0.0)

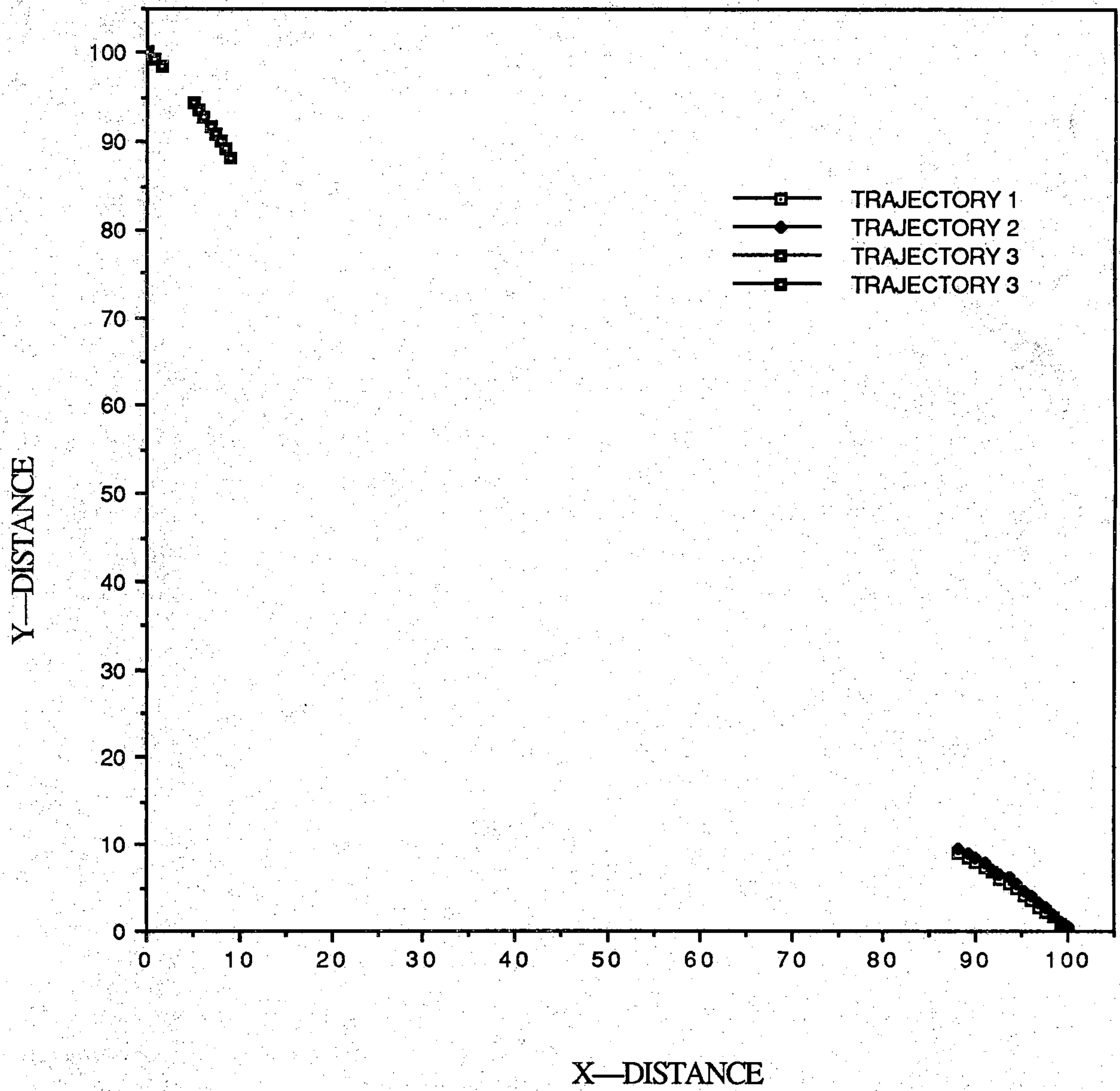


FIGURE 10. Merging of the two segments of a track is illustrated.

```
linear:         ((5.52 93.54559999999999 0.0)
        (0.6374999999999993 -0.8540500000000009 0.0))
tnode:          <tnode 1074172>
threat:         nil
```

Note that both of these segments point to a track node, so that a track exists for each one. Moreover, these tracks are different since the second segment was not determined to be an extension of the first one - yet.

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 segment is removed from the BB and the pointer from the track to that segment node is deleted. If that is the only segment supporting that track, then the entire track is removed by removing that node from the BB. This is the case here.

The KSAR which initiates this KS looks as follows:

```
<bbevent 1071572> is an instance of flavor bbevent with instance variables:
source:       nil
action:       nil
type:  extend-segments
variable:     nil
time: nil
coord:        nil
number:       nil
threat:       nil
snode:        nil
pattern:      nil
duration:     recurrent
position:     nil
goalptr:      nil
conditions:   nil
ksarptr:      <ksar 1073224>
```

It has a recurrent duration and contains a pointer to the KSAR which initiates merge-segments.

The resulting track node which was established for the reappearing track now represents both the current track and the merged track. The older segment and its track have been removed and is now represented by this track node as well. The track node looks like:

```
<tnode 1074632> is an instance of flavor tnode with instance variables:
type:  track
time:  (8)
last-coord:  (5.52 93.54559999999999 0.0)
last-velocity:      (0.6374999999999993 -0.8540500000000009 0.0)
threat:        nil
snode:         (<snode 1075068>)
cpa-bracket:((44.0999164100983 52.67323116789792)
        (30.37506340557913 41.86061551365565))
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

THE BLACKBOARD CODE

The code is contained in files which are functionally organized. The organization is not perfect. What follows is a brief description of the contents of each file and its purpose. This description is a only a preliminary guide. The code follows the description.

THE BLACKBOARD DATABASE FILES

Most of the structures which define the blackboard are contained in the file *ggoalbb.l*. Here the flavors that make up the BB levels and nodes are defined along with their associated defmethods. Many of the global variables and constants are also defined in this file. A second file containing the communication structures and methods is *ggmess.l*. These structures contain the information needed to interface the KS's with the BB database, e.g. the I/O ports. The KS protocol information and the non-blocking read is also contained in this file. So most of the BB database is contained in these two files.

THE CONTROL FILES

The main control loop is contained in the file *ggrloop.l* along with several display utilities for observing the flavors on the blackboard. In addition, the bootstrap function which kicks off the KS's from the KSAR's is contained in this file. The rule base control is run by the functions contained in *ggplan.l*. Here the goals that form the data of the rule base are tested against the rules and the resulting goals are placed on the BB or the generated KSAR's are queued. The rules composing the rulebase are found in the file *ggrule.l*. Many of the KSAR creation functions which support the control are located in the files *ggnode.l* and *ggksar.l*. These above five files form the control of the BB.

KNOWLEDGE SOURCE FILES

There are six knowledge sources, four of which run as subprocesses. The HIT GENERATION program is a C program which is spun off as a subprocess by the *process command. Actually there are many different HIT GENERATION programs, each representing a different set of tracks. The C program included here is for a single track called not surprisingly *singlepath.c*. The ASSIGNMENT KS resides in the file *nassign*, a LISP file which is also run as a subprocess. The TRACK FORMATION KS is another LISP file run as a subprocess and found in the file *testtrack.l*. The SPLINE INTERPOLATION KS file is called *testspline.c* and is run as a subprocess. The file *smerge* contains the MERGE SEGMENTS KS which is run as part of the blackboard process itself, because of the extensive access needed to the BB nodes. The SEGMENT VERIFY KS is

also part of the BB process and is found in the file *ggksar.l.* This file is also part of the control so that the exception proves the rule here. So each KS is located in a separate file and each of these files, except for *ggksar.l,* is solely dedicated to that KS.

GENERAL SUPPORT FILES

Several files contain supporting functions which are more general in nature. The first is the *setoperations* file which contains the functions that operate on sets like set differences. Another file is the *testutilities* file which contains many vector subroutines like dot-product and even some functions that mimic those found in COMMON LISP. Further there are several functions which are cfasl'd in on loading. The first file *read.c* is for the non-blocking read function and the second file *getfd.c* finds the file descriptor number of the subprocess.

```
/* Note that the lisp function (setq port (infile ...))
 * of (setq port (car (*process ...))) makes that lisp
 * variable port be bound to a pointer to a file pointer.
 * that is, what comes into this function is actually the
 * address of the pointer to the FILE thing. thus, we use
 * **port and set up fd to be fd=*port at the beginning
 * of the routine. In this way, we can use fd as expected.
 */

#include <stdio.h>

getfd(port)
FILE *(*port);

{

FILE *FILE_ptr;
int fd;

        FILE_ptr = *port;

        fd = fileno(FILE_ptr);
        return(fd);
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;; file name ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this is ggoalbb.l file
;;
;;;;;;;;;;;;;;;;;;;;;;;; file name ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;      (include /usr/franz/lib/flavors)

        (defvar clock 0)
        (defvar prediction-threshold 2)
        (defvar group-threshold 5)
        (defvar rvar1 0)
        (defvar rvar2 0)
        (defvar rvar3 0)
        (defvar pi 3.141592653589793)
        (defvar cloop-count 0)
        (defvar cloop-display t)
        (defvar junkheap nil)
        (defvar oldage   3)
        (defvar max-segment-length   13)
        (defvar KSQUEUES (list 'beam-queue  'assign-queue
                       ;; 'track-queue 'spline-queue 'merge-queue
                       ))
;; This value set in ggmess.l since instances must be created first
;;        (defvar KSSOURCES (list beammsg  assignmsg
;;                        ))


;;in the file...i.e. load the flavor stuff *before* invoking
;;defflavor.  Must be some problem with autoloading inside
;;our funny little macro.
;;;
;;; this is the event queue
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defflavor event (
                  number
                 (mask '(1 1 1 1 1))
                 (atomic-queue '())
                 (beam-queue '())
                 (assign-queue '())
                 (track-queue '())
                 (spline-queue '())
                 (merge-queue '())
                 )
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

  ;;


;;
;; (defflavor event (
;;                (queue '())
;;                (num-in-queue 0)
;;                (beam-queue '())
;;             ;;  (assign-queue '())
;;             ;;  (track-queue '())
;;             ;;  (spline-queue '())
;;             ;;  (merge-queue '())
;;                )
;;  ()
;;  :gettable-instance-variables
;;  :settable-instance-variables
;;  :inittable-instance-variables)
;;
;;  ;;
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Defmethod for constructing the parallel queues
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defmethod (event :set-queue) (value)
  (let*
   (
;;   (temp (car value))
;;   (test (send temp :ksar-id))
   (temp value)
   (test (send temp :ksar-id))
   )
  (cond
   ((equal test 'newhit)
    (queue-flavor-onto-node-at-attribute temp self beam-queue) )
  ((equal test 'segment)
    (queue-flavor-onto-node-at-attribute temp self assign-queue) )
   (t
    (queue-flavor-onto-node-at-attribute temp self atomic-queue) )
  )

;; (send self :set-number
;;   (apply 'sum (mapcar 'length '(,atomic-queue ,beam-queue))))
;;      (format t " inside method temp is ~a ~% " temp)
;;      (format t " inside method number is ~a ~% " (send self :number))

        ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; this defmethod recalculates the number and places it event
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmethod (event :number) ()
  (send self :set-number
  (apply 'sum (mapcar '(lambda (x) (length  (eval x)))
                      (cons 'atomic-queue KSQUEUES)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Defmethod for constructing the parallel queues
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (defmethod (event :after :queue) ()
;;   (let*
;;    (temp (car (send self :queue)))
;;    (test (send temp :ksar-id))
;;    (cond
;;     ((equal test 'newhit)
;;      (queue-flavor-onto-node-at-attribute temp self beam-queue))
;;     ((equal test 'segment)
```

```
;;      (queue-flavor-onto-node-at-attribute temp self assign-queue))
;;     ((equal test 'track)
;;      (queue-flavor-onto-node-at-attribute temp self track-queue))
;;     ((equal test 'spline)
;;      (queue-flavor-onto-node-at-attribute temp self spline-queue))
;;     ((equal test 'extension)
;;      (queue-flavor-onto-node-at-attribute temp self merge-queue))
;;    (t (format "~a is not a ksar-id, inside defmethod ~%" test))
;;    )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Defmethod for constructing the status mask
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmethod (event :before :mask) ()
  (let (
        (temp
         (mapcar '(lambda (x)
                        (if (null (eval x))
                            nil
                            (send
                             (car (eval x) )
                             :stage)))
                  (cons 'atomic-queue KSQUEUES  ;; limit mask size for now
                        ))))
       (send self :set-mask temp)
;;       (format t " inside defmethod mask which is ~a~%" temp)
))


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

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

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  these are the flavors for the event queue
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defflavor bbevent (
                     source     ; generating node
                     action     ; level this event affects
                     type       ;  hit or track etc
                     variable   ; this is variable triggering event
                     time       ; may be list or number
                     coord      ; list of coordinates
                     number     ; number of coordinates
                     threat     ; for tnodes
                     snode      ; pointers to snodes
                     pattern    ; this is list used for pattern match
                     )
  (goal-attributes)
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

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

(defflavor
  goal-attributes (
                   duration   ; time latency of the goal node
                   position   ; position relative to coord
                   goalptr    ; pointer to other goal nodes
                   conditions ; preconditions to fire
                   ksarptr    ; pointer to ksar which is queued
                   )
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)


;;

(defmethod (bbevent :before :pattern) ()
  (cond
   ((eq type 'track)
  (setq pattern
        (list source action type time threat snode)))
   ((eq type 'clock)
    (setq pattern
          (list type)))
   (t
  (setq pattern
        (list source action type variable coord number time)))
   ))

;;
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This macro generates a flavor and the corresponding after
;;  deamons which which report changes of a bnode, tnode, or snode
;;  to the event queue.  It is in effect part of a distributed
;;  monitor.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;;
(defmacro newflavor (flav level var-list var-sub inher-list &rest options)
  (cons 'progn
        (cons
  '(defflavor ,flav ,var-list ,inher-list ,@options)
  (do*
   (
    (worklyst var-sub (cdr worklyst))
    (op (car worklyst) (car worklyst))
    (mlyst nil)
    )
   ((null worklyst) (return mlyst))
(setq mlyst
(cons  '(defmethod (,flav :after ,(keywordize (concat :set- op)))
                     (value)
                (sendpushgoal
```

```
                    (make-instance 'bbevent
                                   :source self
                                   :action 'change
                                   :type   type
                                   :variable ',op
                                   :coord  coord
                                   :number number
                                   :time   time
                                   :duration 'one-shot
                                   )
              ,level))
              mlyst)
)))))
;;
;;;
;;
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; this is the flavor which makes up the levels of the
;; blackboard hierarchy
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defflavor bblevel (
                up  ;; next level up in hierarchy
                left  ;; the goal panel of the BB
                right  ;; the data panel of the BB
                down)  ;; the lower level of the BB
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

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

(defflavor ksar      (
                   priority  ;; statis priority now
                   ksar-id ;; used at present
                   ks      ;; ks to be fired
                   cycle   ;; cycle created
                   trigger
                   context  ;; arguments to the function boot
                   preconditions  ;; undefined for now
                   boot      ;; the function call for the ks
                   nodeptr  ;; can point to any node
                   stage  ;; nil no transmission, -1 ready-to-read,
                           ;; 1 ready-to-write
                   messenger  ;; the i/o handler for this ksar
                   )
  (ks-protocol)
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  bnode is form beam node - the flavor holding info and the hit level
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(newflavor bnode segments (
                   type    ; type is hit
                   time    ; time stamp for coordinates
                   coord   ; list of the coordinates assoc with time
                   number  ; number points in the list
                   )
              ()
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

;;
;; try designing a defmethod which updates the number
;;
(defmethod (bnode :after :init) (value)
  ;;  (format t " ~%message to eventq here ~%~%")
  (setq number (length coord))
  (sendpushgoal
   (make-instance 'bbevent
                  :source    self
                  :action    'change
                  :type      'hit
                  :variable  'coord
                  :coord     coord
                  :number    number
                  :time      time
                  :duration  'one-shot
                  )
   segments)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  unnode is partial beam node - for the unmatched part of hit node
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defflavor unnode (
                   type    ; type is hit
                   time    ; time stamp for coordinates
                   coord   ; list of the coordinates assoc with time
                   number  ; number points in the list
                   )
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)


(defmethod (unnode :after :init) (value)
  ;;  (format t " ~%message to eventq here ~%~%")
  (setq number (length coord))
  (sendpushgoal
   (make-instance 'bbevent
                  :source    self
                  :action    'change
                  :type      'unmatched
                  :variable  'coord
                  :coord     coord
                  :number    number
                  :time      time
                  :duration  'one-shot
                  )
   segments)
  )

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

```
;;  tnode is form track node - the flavor holding info on the track level
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defflavor tnode (
                      type           ; the time is track
                      time           ; the last timestamp making the track
                      last-coord     ; latest position of the track
                      last-velocity  ; latest velocity of the track
                      threat         ; interval straddles zero
                      snode          ; backward pointer list to segment node
                      cpa-bracket    ; bracket about x and y
                      check          ; spline check of segment group
                      checklyst      ; and list for track verify and break
                      )
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

;;
;; this defflavor generates an entry into the event queue
;;

(defmethod (tnode :after :set-time) (value)
    (sendpushgoal
          (make-instance  'bbevent
           :source self
           :action 'change
           :type  'track
           :time   time
           :threat threat
           :snode  snode
           :duration 'one-shot)
          tracks)
        )
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; this defmethod activates the spline check after tracks have been
;; broke out from the main track
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (tnode :after :set-checklyst) (value)
  (format t "Inside defmethod of tnode to reset CHECKLYST ~%")
  (format t "  checklyst is ~a~%" checklyst)
  (if (null checklyst) ;; if all paths have been checked and modified
      (remove-nodes-from-level tracks (list self))
   ;;  (send self :set-check nil)  ;; then all spline tests to continue
      nil))   ;; otherwise dont reactivate spline tests
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;snode is form segment node - the flavor holding info on the segment level
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(newflavor snode tracks (
                      type    ; is segment
                      time    ; this is the time of last coord
                      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)
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
;;
;; try designing a defmethod which updates the number
;;

(defmethod (snode :after :set-coord) (value)
  ;(setq number (length coord)) ; note this does not trigger after number
  (send self :set-number (length coord))
  (cond
   ( (> (length coord) max-segment-length)
     (send self :set-coord (truncate_lyst
                             (send self :coord)
                             max-segment-length))
     (send self :set-time (truncate_lyst
                             (send self :time)
                             max-segment-length))
     )
   (t nil)
   )
  (cond
   ((< number prediction-threshold))
   ( t
   (setq cpa (find-cpa coord time))
   (setq linear (find-linear-model coord time))
   )
   )
  )

;;
;;
;;
;; this function is an accessor function which gets the
;; list of all id of the nodes on a level
;;;

(defun getid ( level )
  (mapcar '(lambda (x) (send x :node-id)) (send level :right)))

;;
;; this function is an accessor function which gets the
;; list of all props of the nodes on a level
;;;

(defun getprop ( level property)
  (mapcar '(lambda (x) (send x property)) (send level :right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  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
                 )
  )

;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;     instantiate the queues - which are flavors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;(setq eventq  (make-instance 'event))
(setq ksarq     (make-instance 'event))
;;(setq workq     (make-instance 'event))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;     link the black board levels hits to segments to tracks
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(send tracks :set-down segments)  ; links top level to bottom level
(send segments :set-up tracks) ; links bottom level to top level
(send segments :set-down hits)  ; links top level to bottom level
(send hits :set-up segments) ; links bottom level to top level
(setq level-lyst (list tracks segments hits))
(setq queue-lyst (list ksarq ))

;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  File is ggksar.l (goal gksar) part of partition of gksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  create-subgoals to break track
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-subgoals-to-break-track (trnode)
  (let
   (
    (snode (send trnode :snode))
    (pos   (send trnode :last-coord))
    (time  (send trnode :time))
    (velocity (send trnode :last-velocity))
    )
   (dolist (var snode)
           (sendpushgoal
            (make-instance 'bbevent
                           :source trnode
                           :action 'verify-track
                           :type 'track
                           :time time
                           :coord (list pos velocity)
                           :duration 'one-shot
                           :snode var
                           )
            tracks)
           (send trnode :set-checklyst (send trnode :snode))
           )
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  create ksar for verify track test
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-verify-track-ksar (trnode segnode coords)
  (let
   (
    (tnode trnode)
    (pos (car coords))
    (vel (cadr coords))
    (snode segnode)
    )
   (sendksarpush
    (make-instance 'ksar
                   :priority 0
                   :ksar-id 'verify-track
                   :ks 'verify-track
                   :boot '(verify)
                   :nodeptr trnode
                   :cycle clock
                   :preconditions 'empty
                   :context (list tnode snode)
                  )
   ksarq)
  ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  function verify is KS to check the segment path against the
;;  average track trajectory using the angle and the distance
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun verify (ksarptr)
  (let*
   (
    (tnode (nth 0 (send ksarptr :context)))
    (snode (nth 1 (send ksarptr :context)))
    (tvec  (send tnode :last-coord))
    (tvel  (send tnode :last-velocity))
    (ttime (car (send tnode :time)))
    (svec  (car (send snode :coord)))
    (stime (car (send snode :time)))
    (dstime (diff (car (send snode :time))
                  (cadr (send snode :time))))
    (dsvec  (vector-difference (car (send snode :coord))
                  (cadr (send snode :coord))))
    (svel  (scale-vector
            (quotient 1.0 dstime) dsvec))
    (lyst1 (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)
(format t "~% END VERIFY END VERIFY END VERIFY ~%")
   (cond
    ((ksverify lyst1 lyst2) nil) ; if the tracks are paired correctly
    (t ;; if track should be broken - rip out segments
       (send snode :set-tnode nil) ;; removes pointer to track-node
       (send tnode :set-snode
             (remove snode (send tnode :snode)))  ;; remove ptr to snode
       (send tnode :set-checklyst   ;; forces an and of children:
             (remove snode (send tnode :checklyst)))
       ))
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the program will eventually be apart of the knowledge graph
;;  Returns true only if the paths craft are within one seconds travel
;;  and the angle between the velocity vectors is greater than 0.9
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun ksverify (lyst1 lyst2)
  (let*
   (
    (tvec (nth 0 lyst1))
    (tvel (nth 1 lyst1))
    (ttime (nth 2 lyst1))
    (svec (nth 0 lyst2))
    (svel (nth 1 lyst2))
    (stime (nth 2 lyst2))
```

```
    (tmax (max ttime stime))
    (tnewvec (vector-sum tvec (scale-vector (diff tmax ttime) tvel)))
    (snewvec (vector-sum svec (scale-vector (diff 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 consists of the following terms
;;  0. bbnode id which goes into the trigger node variable
;;  1. the ks which is to be invoked
;;  2. is the type of node or level it came from
;;  3. identifies the entities which follows
;;  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 constructs the ksar and stacks it on the ksar queue
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-update-segments-ksar (lyst)
  (sendksarpush
  (make-instance 'ksar
               :priority 1
               :ksar-id  'segment
               :ks (nth 1 lyst)
               :boot '(post-assign-hits)
               :nodeptr (car (last lyst))
               :cycle clock
               :trigger (nth 0 lyst)
               :preconditions 'empty
               :context (list
                      (list 'time (nth 6 lyst))
                      (list 'number (nth 5 lyst))
                      (list 'coord (nth 4 lyst)))
               :stage 2
               :command 'getassignment
               :arglyst '()
               :anslyst '()
               :messenger assignmsg
               :preboot '(pre-assign-hits)
               :prelyst '()
               )
   ksarq)
  )

;;;
;;;
;;;

(defun create-check-track-ksar (lyst)
  (sendksarpush
  (make-instance 'ksar
               :priority 0
               :ksar-id  'spline
               :ks (nth 1 lyst)
               :boot '(assign-threat)
               :nodeptr (car (last lyst))
               :cycle clock
               :trigger (nth 0 lyst)
               :preconditions 'empty
               :context (list
                      (list 'snodes (nth 4 lyst))
                      (list 'time (nth 2 lyst))
                      (list 'tnode (nth 5 lyst)))
               :stage 1
               )
   ksarq)
  )

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

(defun create-update-tracks-ksar (lyst)
  (sendksarpush
  (make-instance 'ksar
               :priority 0
               :ksar-id  'track
               :ks (nth 1 lyst)
               :boot '(assign-tracks)
               :nodeptr (car (last lyst))
               :cycle clock
               :trigger (nth 0 lyst)
               :preconditions 'empty
               :context (list
                      (list 'time (nth 6 lyst))
                      (list 'number (nth 5 lyst))
                      (list 'coord (nth 4 lyst)))
               :stage 1
               )
   ksarq)
  )

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

(defun create-newhit-ksar (lyst)
  (sendksarpush
  (make-instance 'ksar
               :priority 2
               :ksar-id  'newhit
               :ks (nth 1 lyst)
               :boot '(getbeam)
               :cycle clock
               :trigger 'clock
               :preconditions 'empty
               :context 'none
               :stage 1
               :command 'fire
               :arglyst '()
               :anslyst  nil
               :messenger beammsg
               )
   ksarq)
  )

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

```
;;
;;  creates all the different type of ksars
;;  note that this list must be extended considerably to aid
;;  in the mapping process
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defun create-ksar (lyst)
;  (format t "just entered create-ksar ~%")
;  (format t " the lyst is ~a~%" lyst)
;  (*break t " stopped before cond ")
  (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 when given two lists
;;  of coordinates by passing the problem off to the assignment
;;  process.  Note that the lisp file of testassign.l must be
;;  compiled and placed in a file called test for the ports to be
;;  assigned correctly
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun getassignment (lyst1 lyst2)
;;  (read inassign)  ; note this is read the prompt of rewl supplied by franz
  ;;(format t "~a~%" '(getassignment ',lyst1 ',lyst2))
  (format outassign "~a~%" '(getassignment ',lyst1 ',lyst2))
  ;;(format outassign "~a~%" '(getassignment))
  ;;(format outassign "~a~%" lyst1)
  ;;(format outassign "~a~%" lyst2)
  (read inassign)  ; reads the answer a list like (0 2 1) to compare to (0 1 2)
  )

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the main function which assigns the incoming hits to
;;  existing tracks.  It assumes that the number of tracks and hits
;;  match directly and that there is a one-to-one correspondence.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
(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
                                   :type 'segment
                                   :coord (list var)
                                   :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))
         (lyst1 (get-segments-coord-with-time-for-nodes-y 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 lyst1) (length lyst2))
                  (getassignment lyst1 lyst2)
                  (getassignment lyst2 lyst1)
                  )))

         (format t "~% +++++++ order of getassignment +++clock is ~a +++" clock)
         (format t "~% the lyst1 is ~a " lyst1)
         (format t "~% the order info in temp is ~a " temp)
         (setq junkheap (list 'lyst1 lyst1 'temp temp
                               'lyst2 lyst2 'snodelyst snodelyst))

         (update-segment-coord-and-time temp snodelyst lyst1 lyst3 time)

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

          (if (< (length lyst1) (length lyst2))
         (create-goal-for-unmatched-hit-data temp lyst2)
         nil)

;;        (update-segment-time)
;;        (update-segment-coord temp lyst3)

        (fifodequeue hits)))))))
;;
;; note that in using 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 gettrack (cpa vector)
  (read intrack)
;;  (format t "~%TRACK   TRACK   TRACK   TRACK   TRACK   TRACK  ~% ")
;;  (format t "~a~%" '(gettrack ',cpa ',vector))
;;  (format t "~%TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT~%")
  (format outtrack "~a~%" '(gettrack ',cpa ',vector))
;;   (format outtrack "~a~%" '(gettrack))
;;   (format outtrack "~a~%" cpa)
```

```
;;   (format outtrack "~a~%" vector)
;;  (prog1 (list (read intrack)(read intrack)) (read intrack))
  (read intrack)
  )
;;
;;


;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the main function which assigns the incoming hits to
;;  existing tracks.  It assumes that the number of tracks and hits
;;  match directly and that there is a one-to-one correspondence.
;;  This is the precondition material to freeze the context and
;;  and hold the information for the output.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(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
                                   :type 'segment
                                   :coord (list var)
                                   :time (list (get-hits-time)))
                   segments))
        (fifodequeue hits)
        (poptart ksarq assign-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))
               (lyst1 (get-segments-coord-with-time-for-nodes-y 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 lyst1) (length lyst2))
                  `(',lyst1 ',lyst2) `(',lyst2 ',lyst1)))
               )
;;      (format t "Inside pre-assignment function - arglyst is ~a~%" temp)
             (send ksarptr :set-arglyst temp)
             (send ksarptr :set-prelyst
                   (list snodelyst lyst1 lyst2 lyst3 time))
               ))))))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is post condition assign-hitswhich will be fired from the
;;  normal boot with arglyst
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun post-assign-hits (ksarptr)
  (let*
   (
        (temp (send ksarptr :anslyst))
        (lyst (send ksarptr :prelyst))
        (snodelyst (nth 0 lyst))
        (lyst1 (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 lyst1 is ~a " lyst1)
;;      (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 'lyst1 lyst1 'temp temp
                             'lyst2 lyst2 'snodelyst snodelyst))

        (update-segment-coord-and-time temp snodelyst lyst1 lyst3 time)

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

         (if (< (length lyst1) (length lyst2))
        (create-goal-for-unmatched-hit-data temp lyst2)
        nil)

;;       (update-segment-time)
;;       (update-segment-coord temp lyst3)

       (fifodequeue hits)
       ))
;;
;; note that in using 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 getspline (snodeptr)
  (let*
   (
   (lystt (send snodeptr :time))  ;; this gets time list
   (lystl (send snodeptr :coord)) ;; this gets coord list
   (lystt (first-n-elements 4 lystt)) ;; get last 4 time pts
   (lystl (first-n-elements 4 lystl)) ;; get last 4 pos pts
   )
   (format t "~a~%~a~%" lystt lystl)
  (format outspline "~a~%" lystt) ;; send ks time instances
  (format t "~a~%" (read inspline))
  (format outspline "~a~%" lystl) ;; send ks pos. instances
  (do (
       (work (read inspline) (read inspline)) ;; read coefficients
```

```
        (lyst nil (cons work lyst))  ;; continue reading coef x y z
       )
       ((null work) (return
                    (cddddr (reverse lyst))))  ;; reverse to get in order
  )))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this function compares two spline representations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun compare-spline-models (sptrone sptrtwo )
  (format t " in compare-spline-models ~a~a~% " sptrone sptrtwo)
  (let*
   (
    (lystone (getspline sptrone )) ; this gets the spline representation
    (lysttwo (getspline sptrtwo )) ; for both nodes
;;
;;    (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)
;;    (format t " END SPLINE END SPLINE END SPLINE END SPLINE ~%")
    (dist (apply 'add   ; this just sums the abs diff of coef for alltimes
          (mapcar 'vector-absolute-difference
                  lystone lysttwo))))
   (format t "~% SPLINE  SPLINE  SPLINE  SPLINE  SPLINE  SPLINE ~%")
   (format t " ERROR between pairs is ~a~% " dist)
    dist
  ))

;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this creates a segment for 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 (send bnodeptr :coord))  ; copy the coords
   (time (send bnodeptr :time))   ; remove time part of the coordinate
   (number (length coordinates))
   )
    (sendpushlevel (make-instance 'snode
                                  :type 'segment
                                  :coord coordinates
                                  :number number
                                  :time (list time)   )
                   segments)
   (remove-data-x-from-level-y bnodeptr hits)
   ))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; this program grabs a sample from
;;; the from the program and places it on the
;;; the hits level
;;;
;;;   assume that outbeam and inbeam are the out
;;;   and in ports to the beam KS
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun getbeam (ksarptr)
;;  (format outbeam "(fire)~%")
  (let* (
          (temp (send ksarptr :anslyst))  ; 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
          :type 'hit :number count :time timestamp)))
;;
;;  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)
     (send hits :set-right
           (cons xnode (send hits :right))))
    (t nil))))



;; (defun getbeam (ksarptr)
;;   (format outbeam "fire~%")
;;    (setq count (read inbeam))
;;    (setq timestamp (read inbeam))
;;    (cond
;;     (count
;;    (format t " count is ~a~% " count)
;;   (setq temp
;;   (make-instance 'bnode :coord (read inbeam)
;;              :type 'hit
;;              :number count
;;              :time timestamp)
;;   )
;;   (format t " beam node is ~a ~% " temp)
;;    (send hits :set-right
;;        (cons temp (send hits :right))))
;;     (t  nil)
;;     )
;;   )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  File is ggmacro.l (goal macros) part of partition of gksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The macros push and pop items from the queue instance variable
;;  of the flavors acting like queues - eg eventq and ksarq
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;

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

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  here is a macro push-value-onto-node-at-attribute which
;;  pushes value on the variable attribute of flavor node
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro push-value-onto-node-at-attribute (value node attribute)
  `(send ,node ,(keywordize (concat :set- attribute))
         (push ,value
               (send ,node ,(keywordize (concat : attribute))))))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  sendpush places a flavor node on the stack
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendpush (flav stack)
  `(send ,stack :set-queue
         (cons ,flav (send ,stack :queue))))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; this macro gets the queue length of queues like the eventq
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro qlength (stack)
  `(length (send ,stack :queue)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  sendpop pops top off stack using the send command
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendpop (stack)
  `(let ((temp (send ,stack :queue)))
        (send ,stack :set-queue (cdr temp))
        (car temp)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  sendhead finds the head of the queue but does not pop it
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendhead (stack)
  `(car
    (send ,stack :queue)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  get the nth entry in the stack
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendnth (n stack)
  `(nth n
        (send ,stack :queue)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  pushes a flavor instance onto the ksarq
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendksarpush (flav pqueue)
  `(send ,pqueue :set-queue ,flav))
;;        (merge
;;         (list ,flav)
;;         (send ,pqueue :atomic-queue)
;;         'porder)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  this pushes a flavor instance at level to the data BB
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendpushlevel (object level)
  `(send ,level :set-right
         (push ,object (send ,level :right))))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  this pushes a flavor instance at level to the goal BB
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro sendpushgoal (object level)
  `(send ,level :set-left
         (push ,object (send ,level :left))))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  this removes the last member of a queue
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro fifodequeue (level)
  `(send ,level :set-right
         (reverse (cdr (reverse (send ,level :right))))))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  order function for the sort function
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;

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

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; get-nodes returns a list of all the nodes on
;; the level named - levels are hits, segments,tracks
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro get-nodes (level)
  '(send ,level :right))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  this is an display function which shows the member of 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 "~% ========= QUEUES ================= CLOCK is ~a =======~%" clock)
  (do*
   ((wlyst '(:atomic-queue :beam-queue :assign-queue)(cdr wlyst))
    (plyst (car wlyst) (car wlyst))
;;          (keywordize (concat : (car queue-lyst))))
    (var ksarq)
    )
   ((null wlyst)
(format t " ========== END OF QUEUES ============== CLOCK is ~a =====~%" clock)
)
(format t " ************  ~a  ************************~%" plyst)
     (format t " * ~a~%" (send var plyst))
))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  goal selector pulling type x goals from level y
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro get-type-x-goals-from-level-y   (x  y)
  '(do (
        (temp (reverse (send ,y :left)) (cdr temp))
        (y-temp nil)
        (not-y-temp nil)
        )
       ((null temp) (return y-temp))
       (if (equal (send (car temp) :type) ',x)
           (setq y-temp (cons (car temp) y-temp))
           (setq not-y-temp (cons (car temp) not-y-temp)))
       (send ,y :set-left not-y-temp)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This macro gets the goal lyst to be processed by the forward
;;  based chaining system
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defmacro get-goals (level)
  '(let
    ((temp (send ,level :left)))
    ;; this must be modified later to filter only those new
    ;; goals that you want to process
    (send ,level :set-left nil)
    temp
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This macro copies the goal lyst on the given level
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defmacro copy-goals (level)
    '(send ,level :left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this macro removes a GOAL specified by x from
;;  level y
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro remove-goal-x-from-level-y (node level)
  '(let*
    ( (temp (send ,level :left))  ;copy goal list
    (tlyst (delete ,node temp)) ) ; delete node from temp list
    (send ,level :set-left 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 (send ,level :right))  ;copy goal list
    (tlyst (delete ,node temp)) ) ; delete node from temp list
    (send ,level :set-right tlyst)
    (format t "~% DATA NODE ~a has been removed from level ~a" ,node ,level)
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  construct some accessor functions to get the hits
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function gets the coord associated with the segments
;;

(defun get-segments-coord ()
  (mapcar 'car
```

```
         (mapcar
          '(lambda (x) (send x :coord))
          (send segments :right)
          )
         ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function gets the coord associated with the segments
;; 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 ()
         (mapcar
          '(lambda (x)
                   (cons
                   (car (send x :time))
                   (car (send x :coord))))
          (get-nodes segments)
         ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function gets the coord associated with the segments
;; specified by lyst i.e. 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
          '(lambda (x)
                   (cons
                   (car (send x :time))
                   (car (send x :coord))))
         y))

 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;
 ;;  this function gets the coords from the oldest hits flavor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;

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

 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;  this function gets the time from the oldest hits flavor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;

 (defun get-hits-time ()
       (send (car (last (send hits :right))) :time))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  this function gets the coord and time for last flavor on hits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun get-hits-coord-with-time ()
  (mapcar '(lambda(x) (cons (get-hits-time) x))
                          (get-hits-coord)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; this function gets the number of items enqueued on the
;; blackboard level
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; the following are track node ACCESSOR functions get-track-
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro get-track (operation tracklyst)
  `(mapcar #'(lambda (x)
                    (send x ,(keywordize ( concat : operation))))
          ,tracklyst))

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function removes a list of nodes from a 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 (send level :right))  ; this all the nodes on the level
    )
   ((null twork) (send level :set-right 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 (subl 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
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;


(defun get-recent-segments (time)
  (do*
   (
    (slyst (reverse (send segments :right)) (cdr slyst))
    (clyst (reverse (get-segments-coord-with-time)) (cdr clyst))
    (newlyst nil)
    )
   ((null slyst) (return newlyst))
   (if (>= (diff time (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 (send hits :right))
    (hmlyst (mapcar #'(lambda (x) (send x :time)) hlyst))
    (slyst (send segments :right))
    (smlyst (mapcar #'(lambda (x) (car (send x :time))) slyst))
    (testlyst (append hmlyst smlyst)))
   (if testlyst (apply 'max testlyst) 0)))


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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This subroutine 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 (send segments :right)(cdr slyst))
     (node nil)
     (smlyst (mapcar #'(lambda (x)
                              (car (send x :time))) 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)
      )))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  goal selector pulling from level y goals that satisfy f
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-nodes-at-level-y-satisfying-predicate-f (y f)
  (do (
        (temp (reverse (send y :left)) (cdr temp))
        (y-temp nil)
        (not-y-temp nil)
        )
       ((null temp) (return y-temp))
       (if (funcall f (car temp))
           (setq y-temp (cons (car temp) y-temp))
           (setq y-temp (cons (car temp) not-y-temp)))
       (send y :set-left not-y-temp)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  goal selector copying from level y goals that satisfy f
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun copy-goal-nodes-at-level-y-satisfying-predicate-f (y f)
  (do (
        (temp (reverse (send y :left)) (cdr temp))
        (y-temp nil)
        (not-y-temp nil)
        )
       ((null temp) (return y-temp))
       (if (funcall f (car temp))
           (setq y-temp (cons (car temp) y-temp))
           )))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  goal selector copying from level y goals that satisfy f
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun copy-data-nodes-at-level-y-satisfying-predicate-f (y f)
  (do (
        (temp (reverse (send y :right)) (cdr temp))
        (y-temp nil)
        (not-y-temp nil)
```

```
      )
      ((null temp) (return y-temp))
      (if (funcall f (car temp))
          (setq y-temp (cons (car temp) y-temp))
          )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun choosep (node)
  (or
   (equal (send node :type) 'clock)
   (equal (send node :type) 'purge-segments)))

(defun not-unmatched (node)
  (not (equal (send node :type) 'unmatched)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This macro gets the goal lyst to be processed by the forward
;;  based chaining system
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro get-goals-from-level-x-of-duration-y (level lifespan)
  `(let* (
        (temp (send ,level :left))
        (keep (my-remove-if
               '(lambda (x) (equal (send x :duration) ',lifespan))
               temp))
        )
    (send ,level :set-left keep)
    temp
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  here is a macro push-value-onto-node-at-attribute which
;;  pushes value on the variable attribute of flavor node
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defmacro queue-flavor-onto-node-at-attribute (flav node attribute)
  `(send ,node ,(keywordize (concat :set- attribute))
        (merge (list ,flav)
               (send ,node ,(keywordize (concat : attribute)))
               'forder
               )))

;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  forder is the function which orders the flavors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defun forder (x y)
  (< (send x :priority) (send y :priority)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  pops flavor off the queue at node, returns popped flavor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;

(defmacro pop-flavor-at-node-at-queue (node qname)
  `(let (
         (temp (send ,node (keywordize (concat : ,qname))))
         (send ,node (keywordize (concat :set- ,qname)) (cdr temp))
         (car temp) ; return the popped flavor
         ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the same as the above macro but the qname is
;;  entered directly since it is called directly with like beam-queue
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro poptart(node qname)
  `(let (
         (temp (send ,node ,(keywordize (concat : qname))))
         (send ,node ,(keywordize (concat :set- qname)) (cdr temp))
         (car temp) ; return the popped flavor
         ))

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

(defun shorten_to_n (lyst n)
  (reverse
  (nthcdr (diff (length lyst) n )
  (reverse lyst)
  ))
  )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is file ggmess.l, the file which establishes the
;;  communication flavors.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the flavor messenger
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defflavor messenger   ;; these should be named after ks's
  (
   write-port    ; the output port to the process
   write-fd      ; the output port file descriptor
   read-port     ; the input port to the process
   read-fd       ; the input port file descriptor
   pid           ; the process identity
   )
   ()
:gettable-instance-variables
:settable-instance-variables
:inittable-instance-variables
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is a mixing flavor called ks-protocol
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defflavor ks-protocol
  (
   command      ; the input function
   arglyst      ; the argument list
   anslyst      ; answer list
   preboot       ; command to start up function after read
   prelyst       ; argument list for reboot after read
   )

:gettable-instance-variables
:settable-instance-variables
:inittable-instance-variables
 )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This  method writes to the output port
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (messenger :after :write-port) ()
  (send self :set-write-fd (getfd write-port)))

(defmethod (messenger :after :read-port) ()
  (send self :set-read-fd (getfd read-port)))

;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This  method writes to the output port
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmethod (ks-protocol :write-ks) ()
  (format t "COMMAND sent to ks ~a~%"  (cons command arglyst))
 ;; (*break (equal command 'getassignment) 'before-write-getassignment)
  (format (send  ; get output port name from messenger flavor
           (send self :messenger)  ; get messenger name from variable
           :write-port) "~a~%"
          (cons command arglyst)) ; form function call
  (send self :set-stage -1)  ; change state of ksar to read
  )


;;(defmethod (ks-protocol :test ) ()
;  (format  "~a~%"  (send self :command))
;  (send self :print-self)
;;  (format  "~a~%"  command))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The macros push and pop items from the queue instance variable
;;  of the flavors acting like queues - eg eventq and ksarq
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;;
;; (defmacro push (object stack)
;;   '(cons ,object ,stack))
;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;  here is a macro push-value-onto-node-at-attribute which
;;  pushes value on the variable attribute of flavor node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (defmacro push-value-onto-node-at-attribute (value node attribute)
;;   '(send ,node ,(keywordize (concat :set- attribute))
;;       (push ,value
;;             (send ,node ,(keywordize (concat : attribute))))))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  keywordize inserts sym into the keyword package
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;;
;; (defmacro keywordize (sym) '(intern (symbol-name ,sym) *keyword-package*))
;;
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  create instance of messenger
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq beammsg (make-instance 'messenger ))
;;(send beammsg :set-command 'fire)
;;(send beammsg :set-arglyst nil )
```

```
(setq assignmsg (make-instance 'messenger ))
;; (send assignmsg :set-command 'getassignment)
;; (send assignmsg :set-arglyst `( ',r4  ',r3) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This  method reads from the output port and places it in the
;;  answer lyst in the flavor
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (ks-protocol :read-ks) ()
;;        (format t "~% READKS READKS READKS READKS READKS READKS ~%")
;;        (format t "INSIDE DEFMETHOD READ-KS clock is ~a ~% " clock)
;;        (describe self)
      ;; (*break (> clock 8) 'stop-to-test-read)
  (let* (
        (mtemp (send self :messenger))
        (ptemp (send mtemp :read-port))
;;      (xxx (format t "messenger is ~a and port is ~a %" mtemp ptemp))
        (temp (read ptemp))
        ;;      (send
        ;;       (send self :messenger)  ; address of messenger flavor
        ;;       :read-port)))  ; get the readport
         )
;;       (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)
       (send self :set-anslyst temp)
       (send self :set-stage 0)
       ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function a read predicate which tests if the read port is
;;  ready to receive data.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun readp (port)
  (cond
    ((zerop (pread port)) 0)
    (t
     (cond
      ((and
       (equal (tyipeek port) 10)
       (progn (readc port) (if (equal (pread port) 0) 1 nil))
       ) 0)
      (t 1))
    )
  ))


;; (defun readp (port)
;;   (cond
;;     ((zerop (pread port)) nil) ; test if empty channel
;;     (t                         ; else
;;      (cond
;;       ((and
;;        (equal (tyipeek port) 10)  ; see if next char is return ascii 10
;;        (progn (readc port) (if (equal (pread port) 0) 1 nil))
;;        ) nil)  ; if return and channel empty return nil
;;       (t 1))   ; else return 0
```

```
;;       )
;;    ))
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function poll-reads those KS's which have non-empty buffers
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(defun poll-reads (kslyst)
  (mapcar 'readp   ; check to see which ports are ready to read
          (mapcar '(lambda(x) (send x :read-port)) kslyst)))

(defun poll-writes (kslyst)
  (mapcar 'readp  ; check to see which ports are ready to write
          (mapcar '(lambda(x) (send x :write-port)) kslyst)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function opens ports to a process which will test output
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (defun openports ()
;;   (setq ports (*process 'path t t))
;;   (setq inbeam (car ports) outbeam (cadr ports)))
;;
;;
;;  now open the port
;;
;;
;; (openports)
(send beammsg :set-write-port outbeam)
(send beammsg :set-read-port  inbeam)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function opens ports to a process which will test output
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (defun open_assign_ports ()
;;   (setq ports (*process 'test t t))
;;   (setq inassign (car ports) outassign (cadr ports)))
;;
;;
;;  now open the port
;;

;; (open_assign_ports)
(send assignmsg :set-write-port outassign)
(send assignmsg :set-read-port  inassign)
(defvar KSSOURCES (list beammsg assignmsg
                        ))
;;
;;
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This function creates a list of ports
```

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

(defun forminports (kslyst)
  (mapcar '(lambda (x) (getfd (send x :read-port))) kslyst))

(defun formoutports (kslyst)
  (mapcar '(lambda (x) (getfd (send x :write-port))) kslyst))

;;
;; create inport and outport lists
;;

(defvar inports (forminports KSSOURCES))
(defvar outports (formoutports KSSOURCES))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  File is ggnode.l (goal ggnode) part of partition of gksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function puts a true or false in the those nodes which
;; are threat and have more than one segment associated with tnode
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun assign-threat (ksarptr)
  (let*
   ((nodelyst (cadar (send ksarptr :context))) ; 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 " ------------------------------------------- ~%")
   (send (send ksarptr :nodeptr) :set-check
         (if test t 'failed)
         ))) ; set check

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function puts the right coordinate with the matched
;; segment  assigned coordinates
;;  NOT USED CURRENTLY
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun update-segment-coord (order clyst)
  (do
   (
   (nlyst order (cdr nlyst)) ; this contains the order information
   (flavorlyst (send segments :right) (cdr flavorlyst)) ; snodes on segments
   )
   ((null nlyst) (format t "segments are updated ~%"))
   (format t " ~% the nlyst is ~a " nlyst)
   (format t " ~% the clyst is ~a " clyst)
   (send (car flavorlyst) :set-coord
         (push (nth (car nlyst) clyst)  ; choose proper coord to assoc with seg
               (send (car flavorlyst) :coord)))
   )
  )
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; update-segment-coord-and-time is the goal bb version of
;; update-segment-coord and it handles both cases when
;; 1. # segments >= # hits
;; 2. # segments < # hits
;; assigning the coordinates to the proper segments after
;; the assignment problem has been solved and the permutation
;; stored the permutation vector order
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
```

```
(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
       (push-value-onto-node-at-attribute time snode time)
       (send snode :set-coord
             (push value (send snode :coord)))
       )))
  ;;; ;; --- 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
      (push-value-onto-node-at-attribute time snode time)
      (send snode :set-coord
            (push value (send snode :coord)))
      )))
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this function generates a hit goal to account for those
;;  data points that are not matched to the current segments
;;  lyst1 is coordinates with time in the original data hit
;;  intset is the assignment of segments to data and this
;;  function should only be applied when there are more hit
;;  data then current segments.
;;  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
                  :type      'unmatched
                  :coord     lyst
                  :number    number
                  :time      time
                  ) hits)
  ))

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

;;
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function updates the time list in the segment nodes
;;    NOT CURRENTLY USED
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun update-segment-time ()
  (let
   (
   (time (get-hits-time)) ; this contains the time
   (flavorlyst (send segments :right) ) ; snodes on segments
   )
  ; (format t "entered update-segment-time ~%")
  ; (format t "time is ~a and flavorlyst is ~a ~%" time flavorlyst)
   (mapcar '(lambda (x) (send x :set-time
                               (cons time (send x :time))))
           flavorlyst)
  ))
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  create 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)
  (let*
   (
    (temp (gettrack (send segnode :cpa) (car (send segnode :linear))))
    (intervals (car temp))
    (threat (cadr temp))
    (trknode
     (make-instance 'tnode
                    :type 'track
                    :time (list (car (send segnode :time)))
                    :snode (list segnode)
                    :cpa-bracket intervals
                    :threat threat
                    :last-coord (car (send segnode :linear))
                    :last-velocity (cadr (send segnode :linear))
                    )
    )
    )
   (send segnode :set-tnode trknode)  ; establish forward ptr
   (send segnode :set-threat threat)  ; establish threat contribution
   (sendpushlevel trknode tracks)
   ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; assign-tracks is the boot 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 #'create-new-track-node (send segments :right))
      (merge-tracks) )
     (t
      (format t "ERROR no segment to construct tracks ~%"))))

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


   ((send (send ksarptr :nodeptr) :tnode)
    (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 '(lambda (x) (null (send x :tnode))))
    (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 (send x :tnode)))))
     (merge-tracks))
   (t (format t "~% ERROR - Fallen through inside assign-tracks"))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; the following is a function to detect two track nodes next to each other
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun find-formation (tnode1 tnode2)
  (let*
   (
   (pos1 (send tnode1 :last-coord)) ;pos of the first track
   (pos2 (send tnode2 :last-coord)) ;pos of the second track
   (vel1 (send tnode1 :last-velocity)) ;vel of the first track
```

```
    (vel2 (send tnode2 :last-velocity)) ;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
   )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; the following function returns a list of nearby tracks
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(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 equivalence classes for the tracks using
;; find-nearby-tracks to construct the relation.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-track-subsets ()
  ;(format t "~% ENTERED find-track-subsets ~%")
  (do*
   (
    (tlyst (send tracks :right)  ; 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 following function "make-merged-node" merges nodes and ignores
;; the ones constructed to find the merged nodes.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-merged-nodes (tsets)
  (format t "ENTERING MAKE-MERGED-NODES -----------~%")
  (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 nodes being merged is ~a ~%" twork)
   (let
    (
     (temp
      (make-instance 'tnode
                     :type 'track
                     :time (list
                            (average
                             (mapcar 'car (get-track 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 (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) (send
                          (car (send x :snode)) :set-tnode temp)) twork)
    (remove-nodes-from-level tracks twork) ; remove tracks grouped
    (sendpushlevel temp tracks)  ; replace with new group
    )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function is the main merging program
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun merge-tracks ()
  (let*
   ( (tsets (find-track-subsets)) );creates the groups
   (make-merged-nodes tsets)    ;creates an equivalence node
  (format t "TRACKS MERGED ~%")
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function updates the track from the segment nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun update-track (ksarptr)
  (let*
   (
    (snodeptr (send ksarptr :nodeptr)) ; gets the snode flavor for ksar
    (tnodeptr (send snodeptr :tnode))  ; get the tnode flavor from snode
    (lyst (send snodeptr :linear)) ; 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 (send tnodeptr :last-coord)) ; get track position from track node
    (trkvel (send tnodeptr :last-velocity)) ;obtain track velocity fm trk node
    (dt (diff (car (send snodeptr :time))
```

```
            (car (send tnodeptr :time))
            )) ;determine the time difference
      (n (length (send tnodeptr :snode)))  ; number of snodes in a track
      (s1 (diff 1.e0 (quotient 1.e0 n)))   ; convex weight 1/number in tracks
      (newtime (car (send snodeptr :time))) ; want the latest time on track
      (temp (gettrack (send snodeptr :cpa) ; want the threat and intervals
                 (car (send snodeptr :linear))))
      (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) (send tnodeptr :set-last-coord  ; just average positions
                    (convex-vector-average s1 trkpos segp)))
     (t
      (send tnodeptr :set-last-coord  ; just average updated positions
            (convex-vector-average s1 segp
                        (vector-sum trkpos (scale-vector dt trkvel))))
      (send tnodeptr :set-last-velocity segv)))

     (send tnodeptr :set-time (list newtime))   ; update the time

    (send tnodeptr :set-cpa-bracket   ; update the cpa intervals
          (list (union-intervals txint)
                (union-intervals tyint)))

    (send tnodeptr :set-threat      ; update the threat accessment
          (apply 'or
                 (list newthreat
                       (car
                        (mapcar '(lambda (x) (send x :threat))
                                (our-set-difference
                                 (send tnodeptr :snode)
                                 (list snodeptr))
                                )))))
  ))

(defun get-velocity (pl) (cadr pl))
(defun get-position (pl) (car pl))
;;
;;;;
;;;; this function finds the absolute difference between two numbers
;;;;
;;
;;(defun absolute-difference (x y)
;;  (abs (diff 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 'add
;;        (mapcar 'absolute-difference
;;                u v)))
;;;;
;;;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this function  gives those 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 ~a~a~% - - " snode lyst)
  (do*
   (                                          ;compare to others in group
    (worklyst (our-set-difference lyst (list snode))
              (cdr worklyst))  ; do it one at time
    (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 lyst ~a ~%" snode lyst)
    (format t " the dlyst ~a~%" dlyst)
;;    (format t "HOW CLOSE IS SPLINE MODEL - distance is ~a~%" thresh)
   (if (<  (compare-spline-models snode (car worklyst)) group-threshold)
       (setq dlyst (cons (car worklyst) dlyst))
      nil )))

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

(defun find-closest-pairs-new (snode lyst) ;assumes snode inside lyst
  (format t " in NEW find-closest-pairs-new -- ~a ~a ~%" snode lyst)
  (setq sumlyst nil)
  (dolist
   (var (our-set-difference lyst (list snode)) sumlyst)
   (format t " snode is ~a var is ~a ~% " snode var)
   (let (
         (dist (compare-spline-models snode var))
         )
        (format t " the spline distance is ~a ~%" dist)
        (if (< dist group-threshold)
            (setq sumlyst (cons var sumlyst))
            nil))))



;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this function gives the group check of the tracks
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-closest-set (lyst)  ; returns teh lyst of grouped tracks
  (format t " --- find-closest-set ~a~% ----" lyst)
  (let ((ulyst nil))  ; initialize union lyst as nil
       (dolist (ele lyst ulyst)
  (format t " ele lyst and ulyst are ~a~% ~a~% ~a~% " ele lyst ulyst)
             (setq ulyst (our-union
```

```
                    (find-closest-pairs ele lyst)
                    ulyst)))))
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  File is ggplan.l (goal ggplan) part of partition of gksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this program maps all the events into ksars and puts
;;  them on the ksar queue.  It is done in two stages :
;;  1. using the rule base to chain forward and store
;;      results in the workq
;;  2. use the termporary output to initiate a function
;;      call to construct and insert the ksar into the ksarq
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; (defun map-events (lyst)
;;   (try-all-events lyst)
;;   ;(*break t " just finished try-all-events ")
;;   ;(format t " just finished try-all-events ~%")
;;   ;(showq) (showl) (expandq workq)(expandq ksarq)
;;   ;; workqueue now contains all the events process by the
;;   ;; forward chaining system.
;;   ;;
;;   (do* ((worklyst (send workq :queue) (cdr worklyst))
;;         (var (car worklyst)(car worklyst))
;;         (accumulate nil))
;;         ((null worklyst)(send workq :set-queue accumulate))
;;         (format t " worklyst is prior to funcall ~a~%" worklyst)
;;        (format t " call to funcall with var ~%~a~%" var)
;;        (format t " call is to function  ~a ~% " (car var))
;;         (cond
;;     ((funcall (car var) (cdr var))
;;   (format t " ~% just finished funcall ~%")
;;   ;(showq) (showl) (expandq workq)(expandq ksarq)
;;      ; (*break t "inside map-events just finished funcall")
;;       )
;;      (t (cons var accumulate)
;;       (*break t " inside the mapevent loop"))
;;   )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; try this function separates the if and then parts
;; 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)
  (let*
   (
    (ifs   (subst node 'gnode (cadaddr rule)))
    (thens (subst node 'gnode (cadadddr rule)))
    )
 ;  (format t "~% ifs is now ~a " ifs)
 ;  (format t "~% eval of ifs is ~a " (eval ifs))
   (if (eval ifs)
       (progn (eval thens) t)
       nil)
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this is the start of monitor or a planner,
;;  functions it just determines which nodes will be placed
;;  back on the blackboard
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  try-all-rules  this procedure 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)
  ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this function in the start of the planner
;;  plan-goals gets the goals and maps them into KSAR's
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  ;(format t "You are about to map the events, the ksarq is follows ~%")

  (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))))

;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  File is ggports.l (goal ggports) part of partition of gksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; this funciton opens an in port called inbeam for the a.out
;;; process which represents the beam generation program. The out
;;; port is outbeam.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun openports ()
(setq ports (*process 'path t t ))
(setq inbeam (car ports) outbeam (cadr ports))
)

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

(openports)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this part opens ports to inassign and outassign to obtain
;;  an assignment given two lists of coordinates
;;  also intrack and outtrack functions
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; function which opens ports
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun open-assignment-ports ()
  (setq ports (*process 'test t t))
  (setq inassign (car ports) outassign (cadr ports)))

;;
(open-assignment-ports)
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; function which opens track ports
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun open-track-ports ()
  (setq ports (*process 'track t t))
  (setq intrack (car ports) outtrack (cadr ports)))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; function which opens ports for spline fucntion
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun opensplineports ()
  (setq ports (*process 'spline t t))
  (setq inspline (car ports) outspline (cadr ports))
  )

;;
(opensplineports)
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; function which opens ports for spline fucntion
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq outdata (outfile 'datafile))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;;  this file is gr  for goal rules
;;
;;   and contains the rules some of the code to work through
;;   these rules
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the initialization for a recurrent goal.  A recurrent is a
;;  goal which when satisfied, generates a ksar and then is disabled from
;;  generating anymore ksar's until the KS is activated.  Even if the
;;  KS fails at that point it only reenables the goal. This goal creation
;;  is for rule5
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(sendpushgoal
  (make-instance 'bbevent
                 :type 'merge-segments
                 :ksarptr nil
                 :duration 'recurrent
                )
  segments)


;;
;; purpose of this program is to determine if one can take
;; the rule format and feed it a node and have it implement
;; the rule if the precedent is satisfied.  This is a rule
;; which uses bbevent flavors as its facts
;;
;;
(setq rule0
  '(rule clocked-data-arrival
         (if
          (equal (send gnode :type) '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 (send gnode :type) 'hit) ; is goal a hit node
          ;; ;(null (send segments :right))  ; no segments
           (equal (send gnode :action) '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 (send gnode :coord)
                   (send gnode :number) (send gnode :time) gnode))
            (format t " ~%$$$$$$ rule assign hits that arrived fired $$$$$$")
            ))))
```

```
;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  new rule
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq rule1a
  '(rule process-unmatched-hits
         (if
          (and
          (equal (send gnode :type) 'unmatched) ; is type unmatched
          (setq rvar1 (send gnode :source)) ; what is hit node
          ))
          ;; -------- for now just remove these two nodes ----
          (then
           (progn
          (create-unmatched-hit-ksar rvar1) ; 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 (send gnode :type) 'track)
           (equal (send gnode :threat) t)
           (null  (send (send gnode :source) :check))
           (and (> (length (send gnode :snode)) 1)
                (>= (apply 'min
                           (mapcar
                            #'(lambda (y) (send y :number))
                             (send gnode :snode)
                             )) 4))))
         ;;--------------------------------------------------------
          (then
           (progn
            (create-ksar
             (list 'change 'track (send gnode :time)
                         (send gnode :threat) (send gnode :snode)
                         (send gnode :source)))
            (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 (send gnode :type) 'track)
           (equal (send gnode :threat) t)
           (equal (send (send gnode :source) :check) 'failed)))
         ;; ---------- generate subgoals ---------------------
          (then
```

```
          (progn
           (create-subgoals-to-break-track (send gnode :source))
           (format t "~% 2A 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 (send gnode :type) 'segment)
          (> (send gnode :number) 1)  ; are the number pts in a segment > 1
          (equal (send gnode :action) 'change)
          ))
        ;;--- construct the tracks from the segments ----
        (then
         (progn
         (create-ksar
          (list 'change 'segment (send gnode :coord)
                     (send gnode :number) (send gnode :time)
                     (send gnode :source)))
          (format t "~%$$$$$ just executed rule 3, create tracks $$$$$")
         ))))

;;
;;  rule 4 purges old rules from the goal BB
;;

(setq rule4
  '(rule purge-old-segment-nodes
       (if
       (and
        (equal (send gnode :type) 'purge-segments) ; is it a purge node
        (setq rvarl (find-oldest-segment))
        (> (abs (diff (car (send rvarl :time)) )
                      (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 (send rvarl :tnode)
                (<= (length (send (send rvarl :tnode) :snode)) 1) )
               ; then delete both track and segment nodes
          (progn (setq rvar2 (send rvarl :tnode))
                 (remove-data-x-from-level-y rvar2 tracks)
                 (remove-data-x-from-level-y rvarl segments))    nil)
         (format t "~%444444 rule purge-old-segment-nodes fired 44444")
         ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Rule 5 is to merge-segments when the appropriate conditions exits
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

 (setq rule5
  '(rule merge-segments
         (if
          (and
         (equal (send gnode :type) 'merge-segments) ; is it a purge node
         (null (send gnode :ksarptr)) ; no merge segment ksar active
         (setq rvarl (find-oldest-segment))
         (setq rvar3 (find-most-recently-started-segment-with-length-gt-y 1))
         (setq rvar2 (abs (diff (car (send rvarl :time))
                                (car (last (send rvar3 :time))))))
         (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
          (send gnode :set-ksarptr (car (last
                                 (create-segment-extension-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 (send gnode :type) 'track)
          (equal (send gnode :action) 'verify-track)
          ))
;; --------------- rule verify track composition ---------
        (then
         (progn
          (create-verify-track-ksar (send gnode :source)
                                    (send gnode :snode)
                                    (send gnode :coord))
          (format t "~% $$$$ rule verify-track-composition fired $$$$ ~%")
          ))
        ))

;;
;;
(setq rules (list rule0 rulel rulela rule2 rule2a rule3
                  rule4
                 rule5 rule6))
;; ;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   FILE NAME IS grloop
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   This function is used to update the global variable called clock
;;   and to push a clock event onto the eventq
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun clock-update ()
  (cond
   ((zerop (mod clock 4))
   (sendpushgoal
    (make-instance 'bbevent
                   :type 'clock
                   :duration 'one-shot)
             hits)
   (sendpushgoal
    (make-instance 'bbevent
                   :type 'purge-segments
                   :duration 'one-shot)
             hits)
   )
   (t))
  (setq clock (add1 clock))
  )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  the function goon is go-on function to step through the loop
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(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 is the new bootstrap function
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun bootstrap ()
  (cond
   ((equal (send ksarq :number) 0)
    (format t "number is ~a at first test of ksarq contents " number)
    nil)
   (t (format t "ksarq not empty - test if atomic queue not empty ~%")
    (cond ((> (length (send ksarq :atomic-queue)) 0)  ; if atomic not empty
    (let* (                  ; pull ksar out of queue
        (xxx (format t "Try executing KSAR fm ATOMIC QUEUE ~%"))
        (temp  (poptart ksarq atomic-queue))
        (ctemp (send temp :boot))  ; pull command out of boot
        (ctemp (append ctemp (list temp))) ; insert arg of ksarq ptr
        )
        (format t "~% BOOTSTRAPING COMMAND ~a BOOTSTRAPING COMMAND ~%"
               ctemp)
        (format t " bootstrap is to fire ~a  command     ~%" ctemp)
;;      (format t " bootstrap is to fire ~a  command     ~%" ctemp)
        (eval ctemp) ; this fires the command held in boot
        (format t " bootstrap is COMPLETE -- continuing on~% ")
        ))
          (t (format t "Number of Entries in Garbage Queue < 0 ~%"))
          )
  ;;


  ;;
  ;; now finish off ksars which need only put data on bb
  ;;

;;     (format t "Test if any ksar's needed to be finished off~%")
    (cond   ; take care of the reads, are any mask elements = 0
     ((and (nequal (send ksarq :number) 0)
     (apply 'or (mapcar '(lambda (x) (if (equal x 0) t nil))
                          (cdr (send ksarq :mask)))))
      (format t "Mask elements = 0,got down to the let statement~%")
      (let*(
;;;           (xxx (*break (equal clock 9) 'break-at-finish-work))
       (mtemp (cdr (send ksarq :mask)))
           ;; 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 "mtemp is ~a and ntemp is ~a ~%" mtemp ntemp))
       (ftemp (mapcar '(lambda (x)
                          (pop-flavor-at-node-at-queue ksarq x))
                   ntemp)) ; this is a list of instances
;;      (xxx (format t "ftemp is ~a ~%" ftemp ))
        (btemp (mapcar '(lambda (x) (car (send x :boot))) ftemp))
;;      (xxx (format t "btemp is ~a  ftemp is ~a ~%" btemp ftemp))
      (rtemp (mapcar 'list btemp ftemp)) ; list of functions to fire
      )
;;    (format t "mtemp is ~a, ntemp is ~a, ftemp is ~a, ~%
;;           btemp is ~a, and rtemp is ~a" mtemp ntemp ftemp btemp rtemp)
           (mapcar 'eval rtemp) ))  ; this fires all the functions
    (t
    (format t "None of subqueues ready to boot ~%"))
          )


  ;;
  ;; now take of the reads -  stage is -1
  ;;

;;    (format t "About to test if there are any READS to do ~%")
    (cond   ; take care of the reads, are any mask elements = -1?
     ((apply 'or (mapcar '(lambda (x) (if (equal x -1) t nil))
                          (cdr (send ksarq :mask))))
      (let*(
        (ktemp (poll-reads KSSOURCES))  ;ktemp is read ready ports
       (mtemp
      (mapcar '(lambda (x y) (times x (if (null y) 0 y)))
             ktemp (cdr (send ksarq :mask))))
           ;; 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
```

```
        )  ;; now read each of these KS's and place result in
;;    (format t " ktemp is ~a mtemp is ~a and rtemp is ~a ~%"
;;          ktemp mtemp rtemp)
         ;; (*break t 'boot-strap)
      (mapcar '(lambda (x) (send
                            (car (send ksarq  (keywordize (concat : x))))
                            :read-ks)) rtemp)
         ;; (*break t 'boot-strap)
          ));; message flavor
     (t
     (format t "None of KS's need to or are ready to read ~%")))

    ;;
    ;; now take of the writes - put this code inlater
    ;;

;;    (format t "About to test if there are any WRITES to do ~%")
    (cond   ; take care of the writes, are any mask elements = 1?
     ((apply 'or (mapcar '(lambda (x) (if (equal x 1) t nil))
                       (cdr (send ksarq :mask))))
;;    (format t "About to enter WRITE  seqence ~%")
        ;; (*break t 'boot-strap-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 (send ksarq :mask))))
         ;; 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
        )  ;; now read each of these KS's and place result in
;;    (format t " mtemp is ~a and rtemp is ~a ~%"
;;          mtemp rtemp)
      (mapcar '(lambda (x) (send
                            (car (send ksarq  (keywordize (concat : x))))
                            :write-ks)) rtemp)
        )) ;; message flavor
     (t
     (format t "None of KS's ready to or need to read or write ~%")))


    ;;
    ;; now use the preboot to establish the precondtions and
    ;; freeze the local context
    ;;

;;    (format t "About to test for precondtions ~%")
    (cond   ; take care of the reads, are any mask elements = 0
     ((and (nequal (send ksarq :number) 0)
     (apply 'or (mapcar '(lambda (x) (if (equal x 2) t nil))
                       (cdr (send ksarq :mask)))))
;;      (format t "Mask elements = 2,got down to the let statement~%")
      (let*(
       (mtemp (cdr (send ksarq :mask)))
         ;; 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 "mtemp is ~a and ntemp is ~a ~%" mtemp ntemp))
       (ftemp (mapcar '(lambda (x) (car
                                   (send ksarq
                                        (keywordize
                                         (concat : x)))))
                    ntemp)) ; this is a list of flavor instances
;;      (xxx (format t "ftemp is ~a ~%" ftemp ))
        (btemp (mapcar '(lambda (x) (car (send x :preboot))) ftemp))
;;      (xxx (format t "btemp is ~a  ftemp is ~a ~%" btemp ftemp))
      (rtemp (mapcar 'list btemp ftemp))) ; list of functions to fire
;;    (format t "~%mtemp is ~a, ntemp is ~a ~%" mtemp ntemp)
;;    (format t "~%ftemp is ~a, btemp is ~a ~%" ftemp btemp)
;;    (format t "~%rtemp is ~a~%" rtemp )
     ;;    (*break  t 'bootstrap-fire-pre-assign-hits-before-mapcar)
        (mapcar 'eval rtemp)  ; this fires all the functions
     ;;    (*break  t 'bootstrap-fire-pre-assign-hits)
        (mapcar '(lambda(x) (send x :set-stage 1)) ftemp)
        ))
   (t (format t " None of the queues had preconditions to execute ~%")))


   )))

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; this function fires the boot found in the ksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (defun bootstrap ()
;;   (if (> (qlength ksarq) 0)
;;   (let* (
;;      (temp (sendpop ksarq))  ; pull ksar out of queue
;;      (ctemp (send temp :boot))  ; pull command out of boot
;;      (ctemp (append ctemp (list temp))) ; insert arg of ksarq ptr
;;      )
;;      (format t "~% BOOTSTRAPING COMMAND ~a BOOTSTRAPING COMMAND ~%"
;;              ctemp)
;; ;;   (format t " bootstrap is to fire ~a  command     ~%" ctemp)
;;      (eval ctemp) ; this fires the command held in boot
;;      )
;;       nil)
;;       )
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this is the main loop for driving the BB
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun cloop ()
  (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)
  ;; (expandq ksarq)
  (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
  ;(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

  (if cloop-display
      (progn
      (format t "BEFORE BOOTSTRAP KSARQUEUE EXPAND ~%")
      (expandq ksarq))
     nil)


  (bootstrap)

;; (format t "AFTER BOOTSTRAP KSARQUEUE EXPAND ~%")
;;(expandq ksarq)

;  (showq) (showl) ; display queues and levels

  ;(goon)

  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this a display routine to exhibit the level 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 bbtwo.l
        (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~%" (send var :right))
   (format t " ------------------  ~a  goal------------------ ~%" (car varl))
       (format t "  ~a~%" (send var :left))
       ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this a display routine to expand the nodes on a level
;;  expandl expands the DATA side and expandg expands GOAL side
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this a display routine to expand the nodes in a queue
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expandq (queue)
;;  (format t "***********~a*queue expansion*************~%" queue)
  (dolist
   (var (cddr (si:flavor-allowed-init-keywords 'event)))
  (format t "***********~a*queue expansion*************~%" var)
  (dolist (ele (send ksarq var))
   (describe ele))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  function go-for-it allows you set the number of loops you want
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun go-for-it ()
;; (format t "~% cloop-count is at top  ~a " cloop-count)
  (cond
   ((plusp cloop-count) (setq cloop-count (subl 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 (subl answer))
             (format t "~% Do you want queues and levels displayed????")
             (setq cloop-display (y-or-n-p)))
          (t (reset)))))
;;  (format t "~% cloop-count it is at bottom  ~a " cloop-count)
  ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;;
;;  (defvar KSQUEUES (list 'beam-queue ;; 'assign-queue
;;                      ;; 'track-queue 'spline-queue 'merge-queue
;;                      ))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
/* Note that the lisp function (setq port (infile ...))
 * of (setq port (car (*process ...))) makes that lisp
 * variable port be bound to a pointer to a file pointer.
 * that is, what comes into this function is actually the
 * address of the pointer to the FILE thing. thus, we use
 * **port and set up fd to be fd=*port at the beginning
 * of the routine. In this way, we can use fd as expected.
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

pauls_new_read(port)
FILE *(*port);

{

FILE *FILE_ptr;
int fd, c, mask;
struct timeval timeout;

        FILE_ptr = *port;

        fd = fileno(FILE_ptr);
        mask = 1<<fd;

        timeout.tv_sec = 0; timeout.tv_usec = 0;
        /* causes polling see page PS1:8-9 */
        /* note mask is really a fd_set but that is integer */

        c = select(fd+1,&mask,(fd_set *)0,(fd_set *)0,&timeout);
        /*
        printf("\nmask %d c %d fd %d\n",mask,c,fd);
        */
/*  this returns 1 if something to read, zero otherwise */
        return(c);
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This file is smerge
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the extension of segments needed
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun merge-segments (ksarnode)
  (let*
   (
   (wlyst (find-segment-pair))  ; finds best pair of segments to merge
   (s1 (car wlyst))             ; first segment
   (s2 (cadr wlyst))            ; second segment
   (value (car (last wlyst)))   ; value of the cost
   )
   (merge-segment-x-to-segment-y s1 s2)  ; change the nodes
;;   (format t "~% 5555555555555555555555555")
;;   (*break  t 'beforeset)
   (send (send ksarnode :context) :set-ksarptr nil) ; reset goal node
;;   (format t "~% 5555555555555555555555555")
;;  (*break  t 'afterset)
   ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This function merges two segments which have been determined to be
;; the same.  It needs to remove the track that is associated with
;; s1 if it is only supported by s1.  If there are more than one
;; tracks 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)
  (send s2 :set-number
        (add1 (diff (car (send s2 :time))
                    (car (last (send s1 :time))))))
  (let*
   (
   (t1  (send s1 :tnode))
   (snodelyst (if t1 (send t1 :snode) nil))
   )
   (cond
    ((null t1))  ; if there is no track established just remove s1
    ((and t1 (equal (length snodelyst) 1)) ; if only one supporting
     (remove-data-x-from-level-y t1 tracks)) ;snode, remove track node
    ((and t1 (> (length snodelyst) 1)) ; if more than 1 snode supports
     (send t1 :set-snode (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 extension ksar
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-segment-extension-ksar (gnode)
  (sendksarpush
  (make-instance 'ksar
                 :priority 1
                 :ksar-id 'extension
                 :ks nil
                 :boot '(merge-segments)
                 :cycle clock
                 :context gnode
                 )
   ksarq)
  )


;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function loops through the
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 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 extension
```

```
    )
   ((null worklyst) (return dlyst)) ; return [a]-a
   (if (rab a (car worklyst))  ;relation of Rab, possible link
       (setq dlyst (cons (car worklyst) dlyst))  ; record if true
      nil )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the relation that we will test the above function upon
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun time-ordering (snode1 snode2)
  (let
   ((t1 (send snode1 :time))  ; time sequence of snode1
    (t2 (send snode2 :time))  ; time sequence of snode2
    )
   (or (< (car t1) (car (last t2)))  ; proper forward order
        (> (car (last t1)) (car t2)) ; proper backward order
        )))


;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is the relation that we will test the above function upon
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun forward-time-ordering (snode1 snode2)
  (let
   ((t1 (send snode1 :time))  ; time sequence of node1
    (t2 (send snode2 :time))  ; time sequence of node2
    )
    (< (car t1) (car (last t2))))) ; proper forward time ordering


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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Computes the cost of merging a path forward over the time gap
;;  to the possible candidate extensions.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;

(defun cost (s1 s2)   ; from s1 to s2 where these are segment nodes
  (let*(
   (deltat (diff (car (last (send s2 :time))) (car (send s1 :time))))
   (model  (send s1 :linear)) (estimate (vector-sum (car model)
                         (scale-vector deltat (cadr model))))
   (dx     (vector-difference (car (last (send s2 :coord))) estimate))
   (d      (vector-magnitude dx))
   (mean   (quotient (vector-magnitude (cadr (send s1 :linear))) 2.e0))
   (prob   (exp-cdf d mean))
   (cprob  (diff 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 (send s2 :coord))))
(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 " (diff 1.e0 prob))
(cond               ; temp criterion, if 1/cprob > 3 then eliminate
  ((< cprob 0.33333333e0) 101.e0)
  (t (sum deltat (quotient (diff 1.e0 prob)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This function finds the particular the best pair of eligiable
;;  segments for extension
;;  Note the problem may not be commutative.
;;  That is, it matters which pair is merged first.  So unless you
;;  are able to look for a global optimum via a linear program.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-segment-pair ()
  (do*(
   (slyst (send segments :right)(cdr slyst))  ; original segment lyst
   (olyst (sort slyst '(lambda (x y)  (< (car (send x :time))
                              (car (send y :time))))))
   (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 (send segments :right))
    (tlyst (mapcar
           '(lambda (x)
```

```
                   (car (last (send x :time)))) slyst))
   )
  (apply 'max tlyst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This functtion finds the flavor representing the most recently started
;;   segment
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-most-recently-started-segment ()
  (let*
   ((slyst (send segments :right))
    (tlyst (if slyst (mapcar
            '(lambda (x)
                   (car (last (send x :time)))) slyst) nil))
    (tmax (apply 'max tlyst))
    (smax (if slyst
     (my-remove-if '(lambda (x) (< (car (last (send x :time))) tmax))
                   slyst) nil))
    )
   (if smax (car smax) nil)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This finds the most recently started segment with length greater
;;  than y
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-most-recently-started-segment-with-length-gt-y (y)
  (let*
   ((slyst (send segments :right))
    (tlyst (if slyst (mapcar
            '(lambda (x)
                   (car (last (send x :time)))) slyst) nil))
    (tmax (apply 'max tlyst))
    (smax (if slyst
     (my-remove-if '(lambda (x) (< (car (last (send x :time))) tmax))
                   slyst) nil))
    (ymax (if smax
             (my-remove-if-not
              '(lambda (x) (> (length (send x :time)) y)) smax)))
    )
   (if ymax (car ymax) nil)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  THIS FILE IS  setoperations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  our-union are the Winston set functions page 34 of winstons 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 are the Winston set functions 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 are the Winston set functions 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 fills the list with n-1 ... 1 0
;;  this is needed to form the complement of a set which
;;  then is used to choose the children
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun numset (n)
  (numset1 (- n 1) nil)) ; call with n-1 to initialize recursion

(defun numset1 (n lyst)
  (cond
   ((lessp n 0) lyst) ; n less than 0, set to nil
   (t
    (cons n (numset1 (- n 1) lyst)))
  )
 )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;       THIS FILES IS testutilities
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   This function returns the vector magnitude
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

(defun vector-magnitude1 (vector)
  (cond
   ((null vector) 0)
   (t
    (add (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) (quotient (float x) mag))
                 vector)))
       )
  )


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

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

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

(defun dot-product (u v)
  (dot-product1 u v))

(defun dot-product (u v)
  (cond
   ((null u) 0)
   (t
    (sum
     (times (car u)  (car v))
     (dot-product (cdr u)(cdr v))
     )
   ))
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   This function returns the vector difference and sum of two vectors
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

(defun vector-sum (u v)
  (cond
   ((null u) nil)
   (t
    (cons (add (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) (quotient (float x) deltat))
          vector))


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

(defun find-cpa (coord time)
  (let* (
         (r (car coord))
         (u (vector-difference (car coord)
                               (nth (sub1 prediction-threshold) coord)))
          (v (find-unit-vector u))
          (scale (minus (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)
  (let ((r (car coord))
```

```
        (u (vector-difference
            (car coord) (nth (sub1 prediction-threshold) coord)))
        (dt (diff (car time)(nth (sub1 prediction-threshold) time)))
        )
       (list
        r
        (find-velocity u dt)
        ))
  )



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

(defun average (lyst)
  (let* (
         (asum (apply 'sum lyst))
         (len  (length lyst))
         )
        (quotient asum len)))

(defun convex-average (alpha x y)
  (sum (times alpha x)
       (times (diff 1.e0 alpha) y)
       ))

(defun vector-average (lyst)
  (do*
   (
    (scale (quotient 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 (sub1 (length worklyst))) (return result))
   ))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   This function returns the union of intervals
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun union-intervals (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 (diff 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 'add
         (mapcar 'absolute-difference
                 u v)))
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Gaussian density function
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun gauss (x)
  (let ((const (quotient 1.e0 (sqrt 3.141592653589793))))
       (times const (exp (minus (times x x))))))


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

(defun exp-cdf (x mean)
  (diff 1.e0 (exp (minus (quotient (float x) mean)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; this is only a limited version of replace if 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)
```

```
  (mapcan '(lambda (x y) (if x nil (list y)))
          (mapcar func lyst) lyst))

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

(defun my-remove-if-not (func lyst)
  (mapcan '(lambda (x y) (if x (list y) nil))
          (mapcar func lyst) lyst))

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