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

P. R. Kersten
Code 2211, Naval Undersea Warfare Center Division, Newport, Rhode Island 02841–5047

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

This exposition is a tutorial on how object-oriented programming (OOP) in LISP can be used for programming a blackboard. Since we have used Common LISP and the Common LISP Object System (CLOS), the exposition demonstrates how object classes and the primary, before, and after methods associated with the classes can be used for this purpose. The reader should note that the different approaches to object-oriented programming share considerable similarity and, therefore, the exposition should be helpful to even those who may not wish to use CLOS. We have used the radar tracking problem as a "medium" for explaining the concepts underlying blackboard programming. The blackboard database is constructed solely of classes which act as data structures as well as method-bearing objects. Class instances form the nodes and the levels of the blackboard. The methods associated with these classes constitute a distributed monitor and support the knowledge sources in modifying the blackboard data. A rule-based planner is used to construct knowledge source activation records from the goals residing in the blackboard. These activation records are enqueued in a cyclic queueing system. A scheduler cycles through the queues and selects knowledge sources to fire. © John Wiley & Sons, Inc.

I. 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

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

speech understanding,[1] image understanding,[2–5] planning,[6,7] high-level signal processing,[8–11] distributed problem solving,[12] and general problem solving.[13,14] Usually, a blackboard system consists of three parts, a *global database, knowledge sources* (KSs), 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 KSs. The KSs 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 KSs. In an event-driven BB, a scheduler uses the events as the primary information source to schedule the KSs 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 KSs. 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.[15] Note that if goals are isomorphic to the events, then a BB is essentially event driven.

As was eloquently pointed out by Nii,[10] there is a great difference between understanding the concept of a blackboard model and its implementation. Implementation is made all the more difficult by the lack in the current literature of a suitable exposition on how to actually go about writing a computer program for a blackboard. A blackboard is a complex computational structure, not amenable to a quick description as an algorithm. To program a blackboard, one must specify data structures for the items that are posted on the blackboard, explicitly state the nature of interaction between the data on the blackboard and the KSs, clearly define how the high-level goals get decomposed into lower-level goals during problem solving, and so forth. The purpose of this tutorial exposition is to rectify this deficiency in the literature, at least from the standpoint of helping someone to get started with the task of programming a blackboard.

For this tutorial, we have used the radar tracking problem (RTP) to illustrate how object-oriented programming (OOP) in LISP can be used to establish the flow of control required for blackboard-based problem solving. The RTP is defined as follows: Given the radar returns, find the best partition of these returns into disjoint time sequences that represent the trajectories of craft or any other moving body. For craft flying in tight formations, we will associate a single trajectory with each formation. Each trajectory, whether associated with a single craft or a formation, will be called a *track*. Since craft may break away from a formation, any single track can lead to multiple tracks. Shown in Figure 1 is a flight of three craft. Originally, their tight formation results in a single track. But as the flight progresses, one of the craft breaks away to the right, and then we have two tracks. The RTP problem then consists of assigning a radar return to one of the existing tracks or allowing it to initiate a new track. This problem is not new and has been solved with varying degrees of success

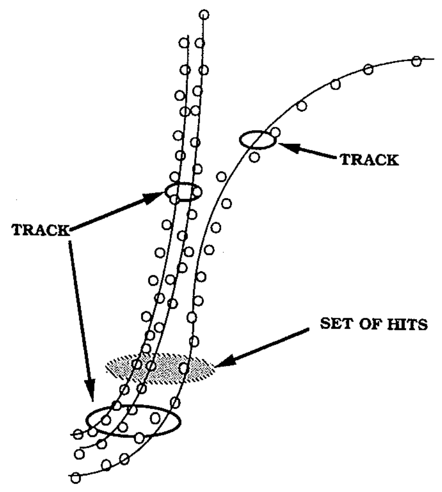

Figure 1. Shown here are radar returns from a flight of three craft. The circles represent the returned echos. Initially, the three craft are in a tight formation and form a single track. But subsequently, one of the craft breaks away to the right, the result being two tracks.

and implemented in numerous systems. In fact, a blackboard solution of the RTP may already exist, although it is probably proprietary. In fact, there is indication[8,11,16] that TRICERO has a radar tracker embedded in it.

On the basis of the criteria advanced by Nii,[11] it can be rationalizd that the RTP problem is well suited to the blackboard approach. We will now provide this rationalization in the following paragraph; the blackboard-suitability criteria, as advanced by Nii, will be expressed as italicized phrases.

The radar returns vary widely in quality. Returns may have high signal-to-noise ratio (SNR) in uncluttered backgrounds but may also be noisy, cluttered, and weak. Obviously you design for the worst case that includes *noisy and unreliable data*. While it is true that tracks can legitimately cross, merge, and split, noise and clutter can also induce these as anomalies in the actual tracks, in addition to, of course, causing the tracks to fade (Fig. 2). Track formation in a noisy environment requires not only significant signal processing but, in general, also requires forward and backward reasoning at a symbolic level. For example, backward reasoning can verify a track by a hypothesize-and-test scheme that may invoke procedures requiring higher spatial resolution and longer signal integration times for hypothesis verification. In other words, under noisy conditions we may use coarse resolution and forward reasoning to form track hypotheses, and then invoke backward reasoning to verify strongly held hypotheses. So, there is a need to use *multiple reasoning methods;* combined forward and backward reasoning steps can be easily embedded in a goal-driven blackboard. In addition to multiple reasoning methods, the system must also reason simultaneously along multiple lines. For example, when track splits

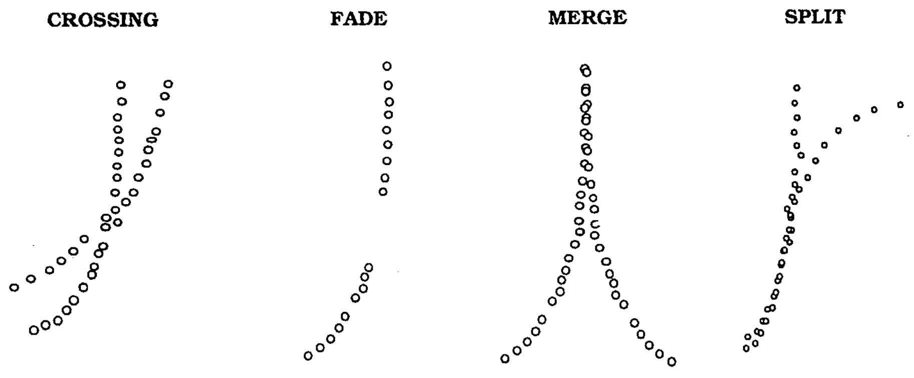

Figure 2. While it is true that tracks can legitimately cross, merge, and split, low signal-to-noise ratios can also induce these as anomalies in the actual tracks. Note noise can also cause the tracks to fade.

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 tracking systems of the future will be equipped with multisensor capability. Therefore future target tracking systems will have to allow fusion of information from diverse sensors, not to speak of the intelligence information that will also have to be integrated. With these additional inputs, the solution space quickly becomes large and complex, necessitating modularized computational structures, like blackboards, that are capable of handling *a variety of input data.*

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

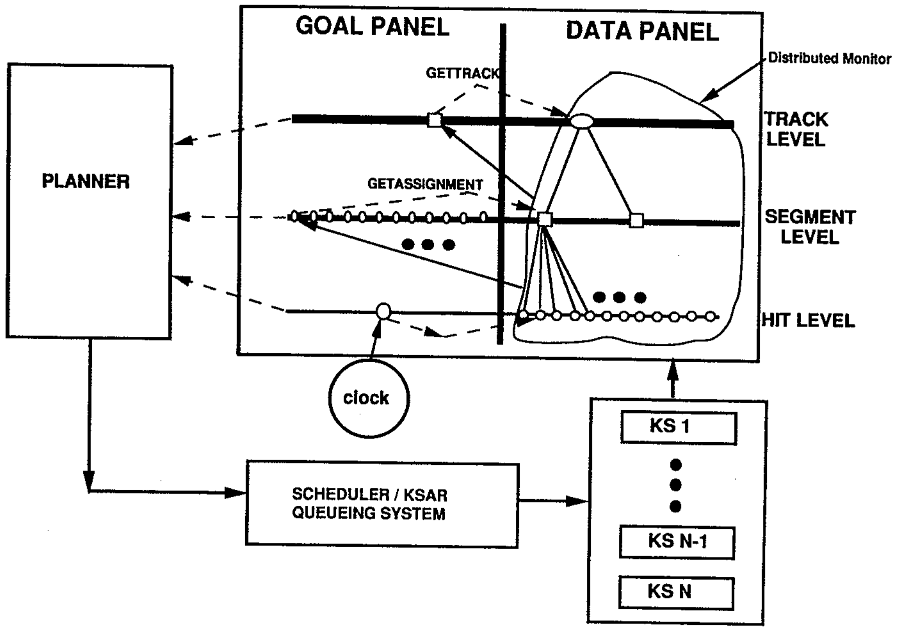

Figure 3. Architecture of radar tracking blackboard (RTBB).

Our radar tracking blackboard (RTBB), the subject of this tutorial exposition, is constructed in CLOS (Common LISP Object System)[17–19] with KSs written either in Common LISP[20–22] or in C. The overall organization of RTBB is shown in Figure 3. The database part of RTBB consists of two panels, the data panel and the goal panel, each containing three abstraction levels. Time-stamped radar returns reside in the form of beam nodes at the lowest level of abstraction in the data panel, the hit level. Spatially adjacent returns are grouped together into segments and reside as segment nodes at the next level of data abstraction. Finally, segments are grouped into track-level nodes at the highest level of abstraction in the data panel. A track-level data-panel node is capable of representing a formation of craft; multiple formations will require multiple track-level nodes. The data abstraction hierarchy is shown in greater detail in Figure 4. On the goal side in Figure 3, goal nodes at the hit level are simply requests to generate time-stamped radar returns. Goals at the segment level are more varied: there can be goal nodes that are requests to assign incoming radar returns to already existing segments, goals to deal with the problem of fading in radar returns, and so forth. 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 subgoals 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 subgoals is a special benefit of a goal-driven BB. A rule-based planner maps the goals

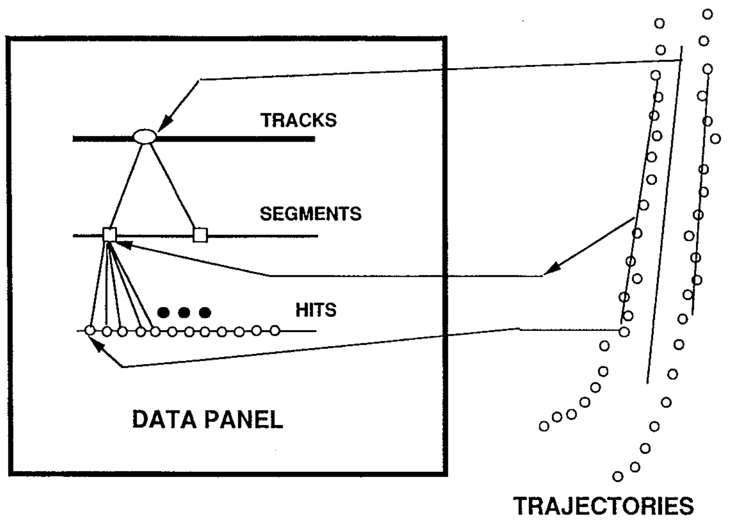

Figure 4. Shown here is the data abstraction hierarchy in greater detail. Hits that are spatially adjacent are grouped into segments. And segments that are approximately co-directional and spatially adjacent are grouped into tracks.

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

In the rest of this tutorial, we will start in Sec. II with a brief introduction to CLOS. As we mentioned in the Abstract, the different approaches to object-oriented programming share considerable similarities, and therefore even a reader who does not use CLOS should find this tutorial useful; such a reader may want to browse through Sec. II if only to become familiar with some of the main data structures used for RTBB. In Sec. III, we describe the different abstraction levels used in RTBB. Section IV briefly discusses the different KSs used. Control flow and scheduling are presented in Sec. V. Finally, Sec. VI 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 article, we believe the reader would find it very helpful to go through the examples in the Appendix for a fuller comprehension of the various aspects of the blackboard. For those wishing to see the source code, it is included in the technical report cited in Ref. 23.

II. THE REPRESENTATION PROBLEM—CLOS

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

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

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

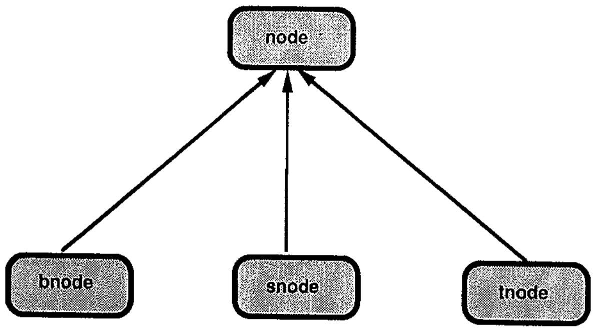

Figure 5. Organization of the node classes. The data abstractions used in RTBB are subclasses of the generic class node.

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

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

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

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

The argument *node* in the first line of this *defclass* macro asserts that the *tnode* class has *node* as its superclass, in accordance with Figure 5. Because the class *node* is a superclass of the class *tnode,* the latter inherits all the slots of the former, together with the read and write accessor functions, if any. We will explain in the next section the semantics of the seven local slots defined explicitly for the class *tnode*. What we wish to point out here is the new slot option *:initform* that appears in four of the local slots. This option permits us to give a default initial value to a slot, these being all nil for the four slots carrying this option. Without the *:initform* option and an associated default value, the value of a slot is left unbound at the time an instance of a class is created, unless the *:initarg* option allows a value to be assigned at that time. If the value of a slot for an object instance is unbounded, and if an attempt is made to read the values of such slots, CLOS will signal an error.

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

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

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

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

Since an after-method may be invoked automatically after initializing or altering critical slots in a node, it is possible to have such specialized methods report the changes to a queue or another portion of the BB. In an event-driven BB the changes are reported to an event queue and in a goal-driven BB the changes are reported to either a buffer in a centralized monitor or directly to the goal side of the BB. Since RTBB is a goal-driven blackboard, any changes in the data are reported directly to the goal panel of the BB by after-methods associated with classes defining the data objects. One can think of these after-methods as constituting a distributed monitor. The methods may also be visualized as being part of the KSs or as a shared utility of these KSs for reporting changes in the data. The reader should note, however, that there do exist alternatives for designing monitors. For example, polling techniques along with change bits or variables in the class instantiations could be used to create a centralized monitor. As another alternative, KSs themselves could report all the changes to a centralized monitor since KSs are the only entities allowed to alter the blackboard.

The following *defmethod* is an example of an after-method that places a node in the goal panel after the *event-time* slot has been given a value by a primary method.* In the definition of the after-method, the role of the method is declared by the qualifier keyword *:after*. The primary method to which the defined method is an after-method appears immediately before the keyword *:after*; in this case the primary method is (*setf event-time*), which is the generic writer function for altering the value of the slot *event-time*. Note in particular the lambda list† of this after-method: (*new-slot-value* (*ele tnode*)). When this after-method is invoked for execution, the first parameter, *new-slot-value,* is instantiated to the new value of the slot *event-time*. (Recall, it is the change in the value of this slot to whatever will be instantiated to *new-slot-value* that causes this after-method to be invoked in the first place.) The second parameter in the lambda list is the symbol *ele,* short for element, which has a specialization constraint placed on it. This specialization constraint, implied by the form (*ele tnode*), says that the parameter *ele* can only be bound to an object of class *tnode*. In other words, the after-method is only defined for track-level nodes on the blackboard. As is evident from its definition, this after-method first makes an instance of the class *bbgoal* and then deposits this goal instance at the track level (how precisely that is done will be explained later). With regard to the values given to the slots when an instance of *bbgoal* is created, the *event-time* slot takes on the value bound to the symbol *new-slot-value* that is the updated value of the slot *event-time* in the *tnode* object bound to the symbol *ele*. The slot *source* is set to the name of the *tnode* object that invoked the method. The slot *purpose* is set to 'change to reflect that the goal was caused by changing the *event-time* value, in contrast with, for example, a goal node that might be created by subgoaling. The slot *initiating-data-level* takes the value 'track since the formation of the goal node was caused by a change in a data node at the track level. The slot *threat* inherits its value from the tnode that caused the method to be invoked. The value of the slot *snode* is a list of pointers to the

*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 Sec. IV. 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 KSARs that could produce these subgoals.

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

snodes that support this track node. Finally, the slot *duration* is set to 'one-shot; this causes only one attempt to be made for this goal node to be satisfied. The reader should note that in the syntax of a defmethod, function calls such as (*snode ele*) are accessor functions, in this case a reader function that retrieves the value of the slot *snode* from the object bound to the symbol *ele*.

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

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

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

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

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

So the set of goals on the track level of the goal BB is just a stack of these class instances. As mentioned before, this after-method is invoked after a change has been made to the *event-time* slot of a track node on the data panel. This occurs whenever a track node is updated. The message that triggers this change will look something like (*setf* (*event-time tnode_object*) *new-event-time*)).

When only one or two methods are associated with each node type, it is a simple matter to write one method for each slot. However, as the number of slots associated with each node class on the blackboard increases, this becomes cumbersome. Seth Hutchinson suggested using a macro to generate these automatically and actually wrote a macro that did this using flavors.[24] The following version is a modified version of that macro designed for a goal driven BB using CLOS.

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

In this macro, a class instance is created of type *class* with slots whose names are supplied in the list *slot-list*. When this macro is invoked, the parameter *super-list* is bound to the list of superclasses of *class*. The first do loop, in lines 8 through 20, repeatedly executes the code in lines 17 through 19 and generates slots of form (*slot-name :initarg :slotname :initform nil :accessor slotname*) for each slot name in the list bound to the parameter *slot-list*. Subsequent execution of the *defclass* in line 5 then creates the appropriate class. The do loop in lines 22 through 43 creates an after-method of the type shown previously for each slot name in the list bound to the parameter *monitor-list*. Therefore, whenever the value of each slot named in *monitor-list* is updated, a goal node is automatically created and deposited at the *datalevel* of the blackboard. The reader might note in particular that the *newclass* macro uses the macro *keywordize,* a procedure used to intern the name bound to the symbol *op* into the keyword package. Here is an example of how *newclass* is called:

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

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

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

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

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

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

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

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

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

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

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

When slot values are allowed to be lists in the manner explained above, such lists may be used either as queues or stacks for the purpose of deciding which objects should be processed first. Here, we use the word *queue* in a generic sense and associate with it three components: its arrival process, its queueing discipline, and its service mechanism. The arrival process is characterized by an interarrival time distribution for items stored in the queue. The service mechanism is composed of servers and the service time distribution; note there can be multiple servers (e.g., processors) catering to a queue. The queueing discipline describes how an item is to be selected from those in the queue. Items arriving at a queue may be enqueued (stored) until serviced, or the items may be blocked (discarded) if no server is free at that time. This makes it possible for us to use the generic term *queue* to mean any queueing system, such as a LIFO queue (stack), a FIFO queue, or some prioritized queue. For an extensive discussion on queueing concepts, the reader is referred to Ref. 25.

III. REPRESENTATION OF THE ABSTRACTION LEVELS

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

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

*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.

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

where *coord* is a list of coordinates (*x*, *y*, *z*) of a time-stamped radar return, which consist of a set of echoes received during a single scan of the entire search space. The time stamp of such a set of echoes becomes the value of the *event-time* slot inherited from the superclass *node*. The slot *number* contains the actual number of distinct returns in the set of echoes. Another slot inherited from the superclass is *level*, whose value is set to *hit* for nodes of *bnode* class. Such nodes are generated every *n*th clock cycle, where at present *n* is set to 8.

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

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

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

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

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

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

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

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

When a segment object is created from this class, the slot *coord* contains a list of the coordinates of the hits that constitute that particular segment. Note in particular that whereas the similarly named slot for bnodes contains a list of hits for the same time stamp, the slot here has a time-sequenced list of hits constituting a geometrical segment in space. In other words, the coordinates in the slot *coord* are grouped on the basis of spatial continuity, as opposed to the temporal continuity used in bnodes. The value of the slot *event-time,* inherited from the superclass *node,* is the list of event-times corresponding to the hits in the slot *coord.* In order to clarify the access discipline used for processing the lists in the slots *coord* and *event-time,* both lists are treated as stacks. The value of the slot *cpa* is the closest point of approach if the segment were to be extended all the way to the radar site, assumed to be located at the origin of the (x, y, z) space. The value for the slot *cpa* is calculated by an after-method using the position and velocity information contained in the slot *linear,* the reference here being to the position and velocity of the target computed from the two most recent hits in the segment. More specifically, the value of the slot *cpa* is the perpendicular distance from the origin to a straight line that is an extension of the two most recent hits in the segment. The slot *threat* is set to true if the value of *cpa* falls within a small region around the origin; otherwise it is false. The extent of this region is ε times the *last-coord* (a slot for track level nodes to be discussed later), the comparison threshold being dependent on the distance since greater directional uncertainty goes with more distant craft (this point will be explained further in the discussion on the GETTRACK KS). While the computation of the value for *cpa* occurs when a segment node is first created, determination of whether *threat* is true or false does not occur until a track-level node is updated with the segment.

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

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

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

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

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

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

The inherited slot *duration* has a very important role to play in the control of the blackboard, a fact that will become more obvious in Sec. V. The slot *duration* refers to the length of time the goal is allowed to stay on the blackboard. For example, a one-shot duration means there is only one opportunity for the planner to test a node against the rules to see if it matches any of the antecedents; if the match fails, the goal node is discarded. Most goal nodes are of one-shot type; for example, the goal to update a tnode with new segments is of one-shot type. Only one KSAR for this goal node, which contains a pointer to the segment that should be used for updating, will ever be formed by the rule-based planner. The goal node is purged as soon as the KSAR is formed. Therefore, if this KSAR fails to satisfy the goal node, the goal node will not be there to reattempt updating of the tnode with the same segment.

In addition to the one-shot type, RTBB also contains a recurrent goal node. A recurrent goal node is disabled after it satisfies the antecedent of specific rules, and then is reenabled after a KS is fired from the subsequently generated KSAR. Recurrent goal nodes are never removed from the blackboard. The job of the recurrent goal node that is curently 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 hits

($h1_1, \ldots, h1_n$), and let us say the time stamp of $h1_1$ is 7, of $h1_2$ 8, and so on. Also, assume that there exists another snode made up of hits ($h2_1$, $h2_2$, $h2_3$) where the time stamp of $h2_3$ is 3. Then the job of the recurrent goal node will be to merge the two segments because the time stamp of $h2_3$ is so close to that of $h1_1$. The actual merging, carried out by the MERGE-SEGMENTS KS, will only take place if the extension of the $h2$ segment to the time instant corresponding to the beginning of the $h1$ segment is within an acceptable circle.

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

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

Given that the reader is now familiar with the organization of RTBB, the overall method for solution formation is restated, hopefully in a more precise manner. All the radar returns or hits generated on a scan of the search space are given the same time stamp. The list of hits occurring in one scan is contained in a class instance on the hit level of the data panel shown in Figure 3. Arrival of a new list of hits triggers the distributed monitor to place a goal node on the segment level of the goal panel. This goal node represents a desire or a request to use the new list of hits to update existing segments. If no existing segment can be found to match a particular hit, a new segment is started with the new hit.

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

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

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

IV. KNOWLEDGE SOURCES

There are six KSs 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 KSs and a short description of their purpose.

A. Hit Generation (GETBEAM)

This KS is written in C and simulates the trajectories for the various craft. Trajectories are generated by using Bezier curves in 3-space.[26] 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 a craft at normalized time u:

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

where it is assumed that time is normalized such that $0 \le u \le 1$ for the entire flight of the craft. Every nth clock cycle (currently $n = 8$) a goal node is placed on the hit level of the goal panel, the goal being to fire the GETBEAM KS. A KSAR is then 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 4×3 parameter matrix for each craft in the formation, and upon each call the GETBEAM KS returns the coordinates of all the craft in the formation.

B. 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 procedure (a special case of A* search[27] implemented via a best-first search algorithm; see Refs. 22 and 28 for implementation of best-first search. Further discussion on the complexities of the assignment problem, also called the data association problem, can be found in Ref. 29.

C. Track Formation (GETTRACK)

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

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

D. 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 immediately be 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 nonthreatening or even diverging away from the origin. The GETSPLINE KS does this verification by fitting a spline to each of the trajectories within a track and comparing the trajectories on the basis of the spline coefficients. This KS, written in C, is based on a spline routine in Refs. 26 and 30 and obtains a polynomial expression for the track between sample points based on the coordinates and time stamps held in the segment nodes.

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

E. Merge Segments (MERGE-SEGMENTS)

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

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

The MERGE-SEGMENTS KS is implemented within the BB process itself since it requires extensive access to the data nodes on the BB itself. (If shared memory were available on the BB, one could implement the KS as a separate process.) RTBB uses a recurrent goal node, at the segment level, to monitor and schedule the MERGE-SEGMENTS KS, implying that a goal to invoke this KS is placed permanently at the segments level of the goal panel. This goal node is an instance of the class bbgoal; in this instance, the slot *purpose* is set to 'merge-segments,' the *duration* to 'recurrent,' the *initiating-data-level* to 'segment, and the *ksarptr* to nil. When segments satisfy the rule to activate the MERGE-SEGMENTS KS, a flag pointing to the generated KSAR is established in the goal node. As was stated in Sec. III, 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.

F. The Segment Verify Knowledge Source (VERIFY)

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

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

This ends the introduction to the various KSs in the system. To put the KSs 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 KSs, GETSPLINE and VERIFY, constitute a backward type of reasoning. And lastly, the MERGE-SEGMENTS KS attempts to maintain track continuity across fades in trajectories.

V. BLACKBOARD CONTROL

Ideally, the control of the blackboard should be opportunistic in nature, that is, choose the KS that advances the solution the most.[31] However, whether or not control can be exercised in an "optimal" manner depends ultimately on the programmer, who presumably has an understanding of the application domain. In RTBB, data events are mapped into goal nodes, which in turn are mapped either into subgoals or KSARs. The KSARs are enqueued into a priority queueing system, the queueing discipline determining the order in which the

KSARs appear in their respective queues. The scheduler then cycles through the KSAR queues and selects KS to fire.

We will now describe various possible approaches to the representation and processing of KSAR queues. Then, at the end of this section, the current RTBB implementation of the KSAR queueing system will be discussed. Ideally, the KSAR priorities should be dynamically determined based on the threat a craft presents to the command and control center presumed to be located at the origin of the coordinate system. For dynamic prioritization, the planner must contain rules for assessing the relative severity and immediacy of a threat. Furthermore, the scheduling of the threatening tnodes must allow the other goal nodes in the system to be serviced often enough so that any future threats would not be ignored. Evidently, designing a planner and scheduler for such dynamic prioritization is a complex task and is not addressed in RTBB. We have chosen a simpler approach to KSAR prioritization that has the virtue of allowing for the main BB process to activate KS computations while the main process attends to other chores. As we will show below, our approach is an approximation to what may be thought of as a desirable approach to KSAR prioritization in which a separate KSAR queue is used for each KS and then, as shown in Figure 7, each KSAR queue is visited once in a cyclic fashion, perhaps using a FIFO access discipline.

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

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

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

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

IF–the purpose of the goal node is "merge-segments"
and there are no KSARs 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,

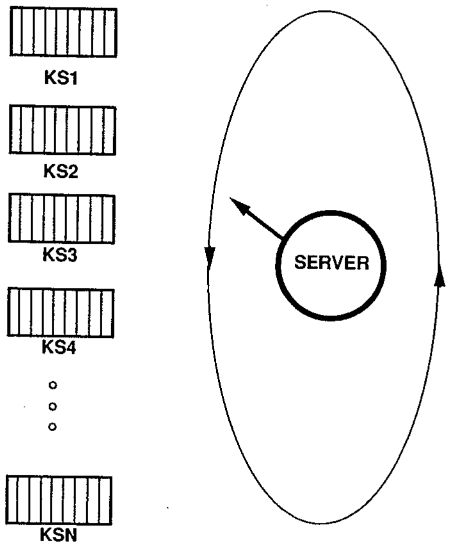

Figure 7. This figure depicts a desirable KSAR queueing system.

THEN–create a KSAR to merge the two segments.

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

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

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

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

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

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

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

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

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

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

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

When a KSAR of the type shown above (see #(ksar @ #x6269fe)) is selected and its corresponding KS executes, the control resides with the KS until such time when the execution of the KS is over. In other words, the main BB process simply waits for the KS to finish up before focusing on any other activity. KSARs that hand over control to their respective KSs are defined to be of type *atomic*. In RTBB, atomic KSARs have been used for most of the KSs. 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 cannot be altered during the execution of the KS, no intermediate incorrect information can be returned by the KS. Clearly, if the information on the BB was allowed to change during the execution of the KS, it is entirely possible that what is returned by the KS may not be relevant to the new state of the BB.

One major disadvantage of an atomic KSAR is that it does not permit exploitation of parallelism that is inherent to problem solving with blackboards. As we mentioned in the Introduction, one main attraction of using the BB paradigm is that the KSs, if they represent independent modules of domain knowledge, should lend themselves to parallel invocation. Although from the standpoint of enhancing performance, parallel execution of KSs is highly desirable, the reader beware, however. Parallel execution also demands that attention be paid to the elimination of interference between the KSs, in the sense that one KS should not destroy the conditions that must exist on the BB for the results returned by another KS to be relevant. Researchers have proposed methods for dealing with these difficulties; the methods consist of either locking regions of the BB database or tagging different nodes with the identities of the KSs that need them.[32] There is also the opinion that one should not bother with the overhead associated with region locking or data tagging, and should simply let the BB resolve on its own any inconsistencies that might arise due to interference between the KSs.

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

The KS corresponding to a distributed KSAR is executed in three stages. The first stage sends a command to the KS containing all the information needed to execute the KS. The format is just a list that represents a function call with all the information as arguments. The KS then just eval's the list. The second stage occurs when the system first polls the KS port to see if the KS is finished.

Note that this polling cannot be accomplished by pressing into service a regular read function, such as the Common LISP *read,* because such a function would simply wait for the data to appear or do something unpredictable, but that is not what we want. What we wished was that we be able to poll the KS every few clock cycles, check for whether or not the KS had returned the results, then read the results if available. In the absence of any results from the KS, we wanted the system to move on to other tasks and to return to the KS at a later time. Hence, the use of the Common LISP function *listen* for implementing a nonblocking read which takes the KS results and stores them in the KSAR. The third stage occurs when the BB takes the answer returned from the KS and modifies the BB accordingly. Between the stages, the BB is actively working on other parts of the problem. The result is a speedup due to the parallel processing carried out by the system.

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

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

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

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

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

As will be evident from the following definitions of the slots, the mixin class is only useful for a distributed KSAR. Since all KSAR instances use the mixin, the reader might wonder why we use the mixin class ks-protocol-mixin at all;

after all, the slots in the mixin could have been incorporated in the definition of the KSAR class. The reader should note that even when a mixin is always used for defining objects, its separate definition allows the definitions of objects to be expanded incrementally as the software is being developed. Also, one can take advantage of the fact that mixin associated methods will be invoked in a certain order depending upon the order of appearance of mixins, etc.

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

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

The GETASSIGNMENT KS then tries to assign each new hit to a segment based on the spatial and temporal closeness of the hit to the most recent entry in the segment.

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

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


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

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

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

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


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

*Note that this method is neither an after-method nor a before-method. The method that is shown here is a *primary* method that is invoked by calling the generic function "write-ks" with an object, which will be bound to the parameter *ele* and that must be of class *ks-protocol-mixin*.

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

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

The function call (openports) produces a bi-directional stream with

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

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

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

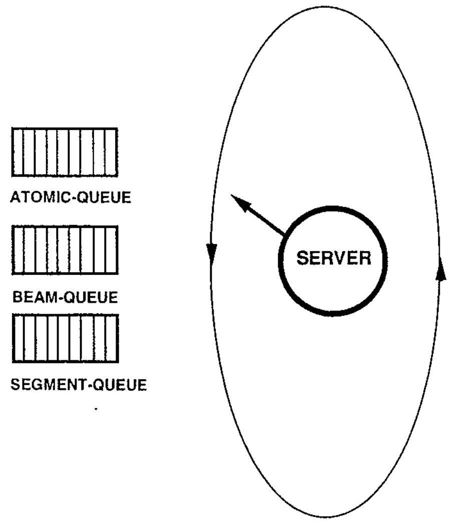

Figure 8. This is the actual KSAR queueing system currently employed in RTBB.

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

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

The slot *number* is set to the total number of KSARs held in all the queues. Note that *mask* represents the status of the KSARs that are currently at the head of the queues. The list that is the value of *mask* has a status entry for each of the distributed-KSAR queues, and the interpretation to be given to each entry in the list is the same as that given to the values for the slot *channel* in a distributed KSAR. In the *defclass,* the initial mask values have been set to 1 for the head KSARs in both the *beam-queue* and the *segment-queue,* meaning that if any KSARs are found at the heads of the respective queues, they are in stage 1. Recall that the channel value of 1 corresponds to the write stage in which commands are written out to the KSs.

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

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

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

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

The following is an example of bbksarq during execution.

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

Here the first item in the list that is the value of the slot *mask* represents the status of the atomic queue (in the case shown here, the value is set to 1, a meaningless number for atomic queues); the second item, pertaining to the beam queue, is set to -1 and means that the KS is ready to read; and, finally, the third item is set to nil since the segment queue is empty. The slot *number* is set to the total number of KSARs held in the queueing system and is automatically updated after every change by an after-method. As mentioned previously, the *track, spline,* and *merge* queues are not being used at this time.

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

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

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

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

        (go-for-it )    ;; limits cycles, throws control back for loop breakout
(clock-update) ;; update the clock variable and place a goal at the
               ;; hit level every eighth clock unit.

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

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

)))
```

VI. CONCLUSIONS

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

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

RTBB is an evolving program and many aspects of it could be further refined. For example, one of our future goals is to implement a separate queue for each KS; that would enhance a parallel or multiprocessor implementation of RTBB. We would also like all the KSARs to be of distributed type, which would make it necessary that we somehow "split" those KSs that are currently processed via atomic KSARs 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 preceding 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.

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

APPENDIX

We will now present four examples to illustrate the working of RTBB. These examples, at increasing levels of difficulty, start with the case of a single

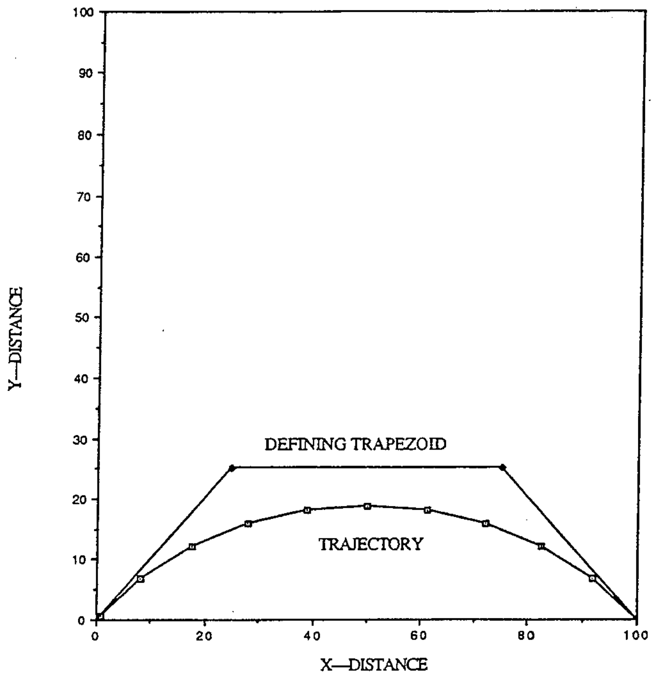

Figure 9. Single trajectory generated via Bezier's curve with every tenth point shown. Defining trapezoid shown with curve.

track formed by a single craft in the first example; progress to two stable tracks formed by three separate craft in the second example; further progress to the case where initially three craft form a single track, but eventually form only two tracks as one craft breaks away; and, finally, deal with the problem of fading tracks in the last example.

Example 1

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

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

The data nodes on the hit level (the bnodes) are initiated by periodically placing a goal node on the hit level of the goal panel. The goal node causes the generation of a KSAR, which causes the hit generation KS GETBEAM to fire.

Recall from the BB Control section that this KSAR is of distributed type although no *preboot* function is necessary since the function call is so simple. The KSAR is

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

After the segment information has extended to more than 14 points, the list is truncated by dropping the oldest hits. This is accomplished with an after-method so that after, say, 20 time units, the snode takes on the following appearance:

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

In the above snode, the maximum lengths of the *coord* and *event-time* slots are now only of length 14 as fixed by a global variable. The truncation length may be set to any fixed value but this threshold is not totally independent of the other parameters. For example, a track may only be generated when the segment length exceeds another fixed parameter. Certainly the truncation length must exceed this minimum length needed to initiate a track; otherwise data will truncated as soon as it is placed on the segment level.

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

Example 2

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

Figure 11 shows a plot of the three trajectories. Two of these trajectories are very close and logically form a track. The other trajectory forms a separate track by itself. The plots of Figure 11 are mirrored in the data structures on

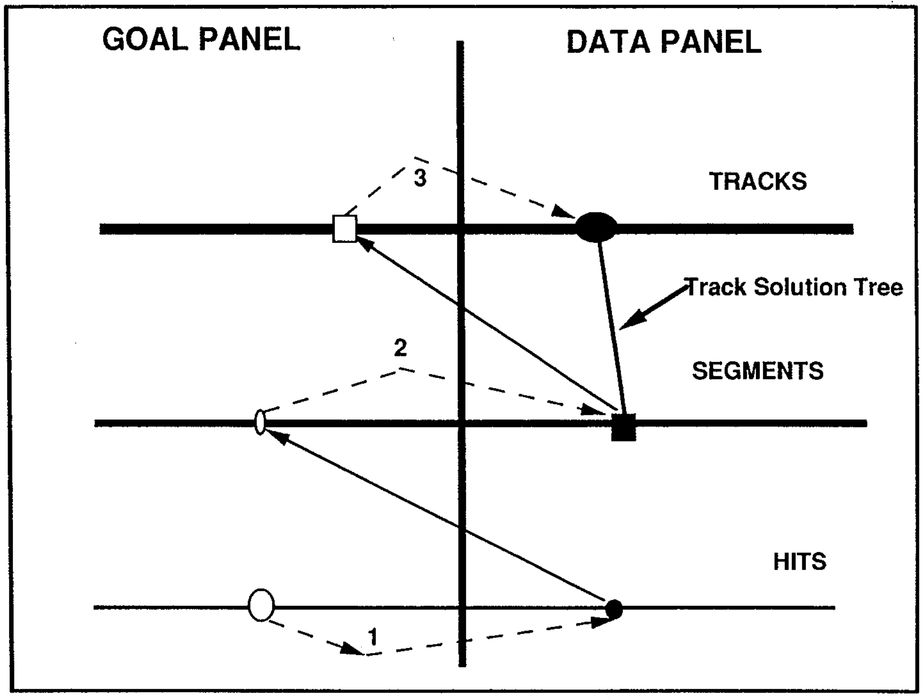

Figure 10. The track formation for a single trajectory is outlined here.

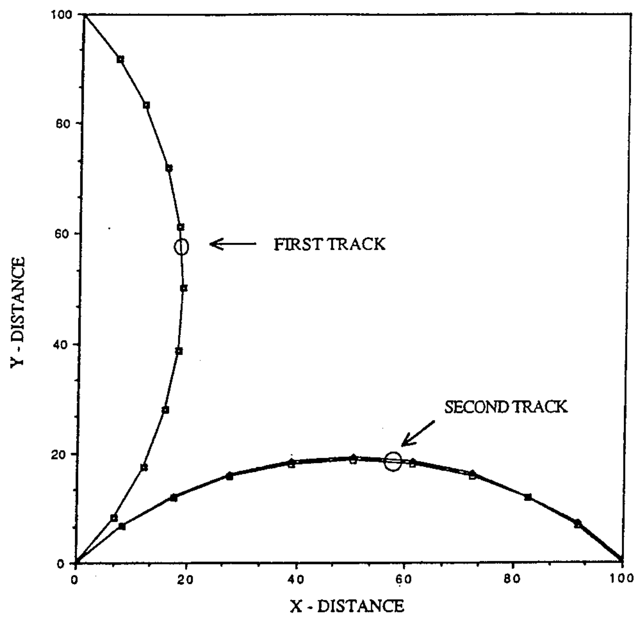

Figure 11. Trajectories for three separate craft with two tracks indicated.

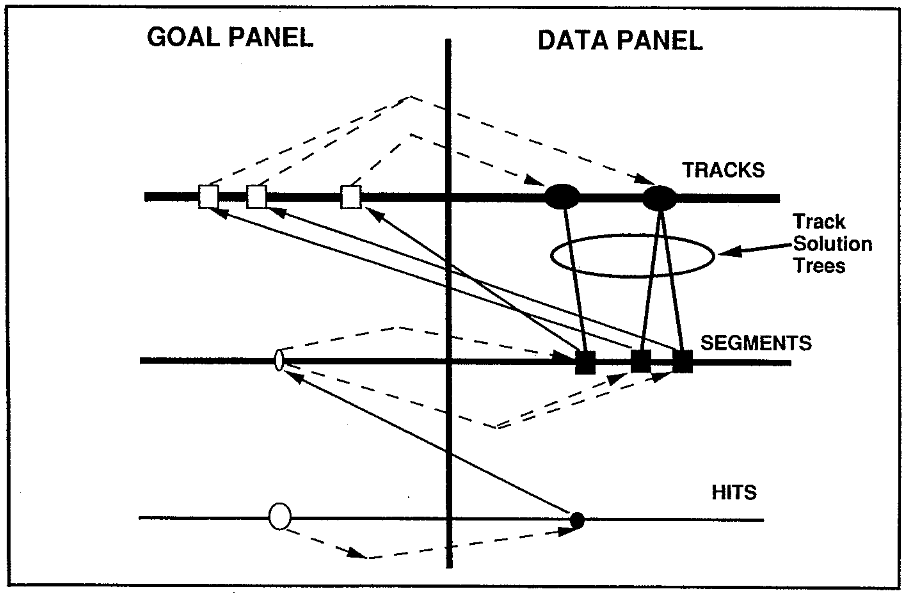

Figure 12. The nodes on the BB for the 3-trajectory, 2-track example. Note the two solution trees.

the blackboard panels. Since a tree represents a track, one tree will represent two trajectories and the other will represent a single trajectory.

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

Let us look at some of the data nodes on the BB after the tracks are established. The two tracks are represented by the following two tnodes:

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

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

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

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

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

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

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

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

```
#<tnode @ #x584f46>
is an instance of class #<clos:standard-class tnode @ #x56dc76>:
 event-time        (41)
 level             track
```

```
last-coord        (60.09016 18.42884 0.0)
last-velocity     (-1.1114502 0.14003944 0.0)
snode             (#<snode @ #x5838a6> #<snode @ #x583886>)
threat            t
cpa-bracket       ((-2.437229 54.252422) (25.053728 55.14532))
check             36
checklyst         nil

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

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

Example 3

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

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

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

The nodes or solution tree should reflect the history of the trajectories. Indeed, Figure 14 shows the correspondences between the physical trajectories and data structures that represent these trajectories. First a tree that has only one root, i.e., one trajectory with three branches representing the three distinct craft, will form on the blackboard. Once the track is established and determined to be a threat, the track grouping will be checked via the spline KS. When the track grouping is not verified by the spline KS, subgoals for each segment are

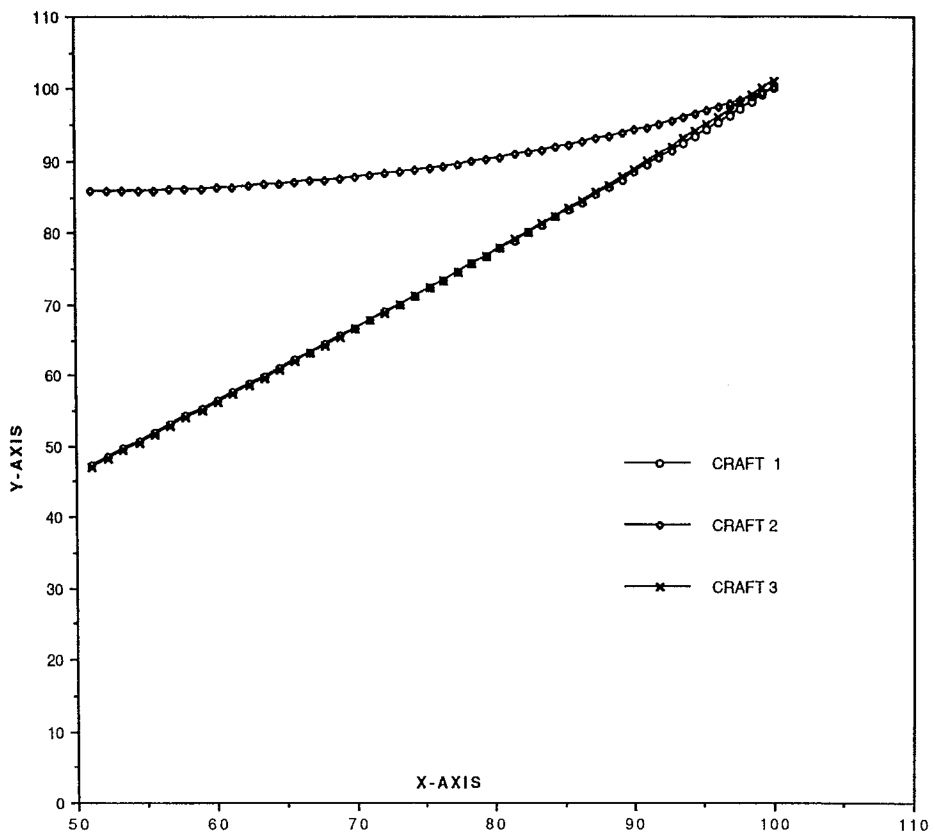

Figure 13. Shows breakaway craft 2 from the formation.

created and placed on the goal segment level. Each goal represents the desire to determine if that segment is in the same formation as the average track representing the root of the track. If the segment does not satisfy the grouping criterion against the track, it is spun off as a segment with no parent pointers. This means the BB will establish this segment as a separate track. The following paragraphs will show the state of the nodes in this sequence of events.

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

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

Observe that there are three snodes or branches supporting this track. The three segments supporting the trajectory are given below. Note that the track node

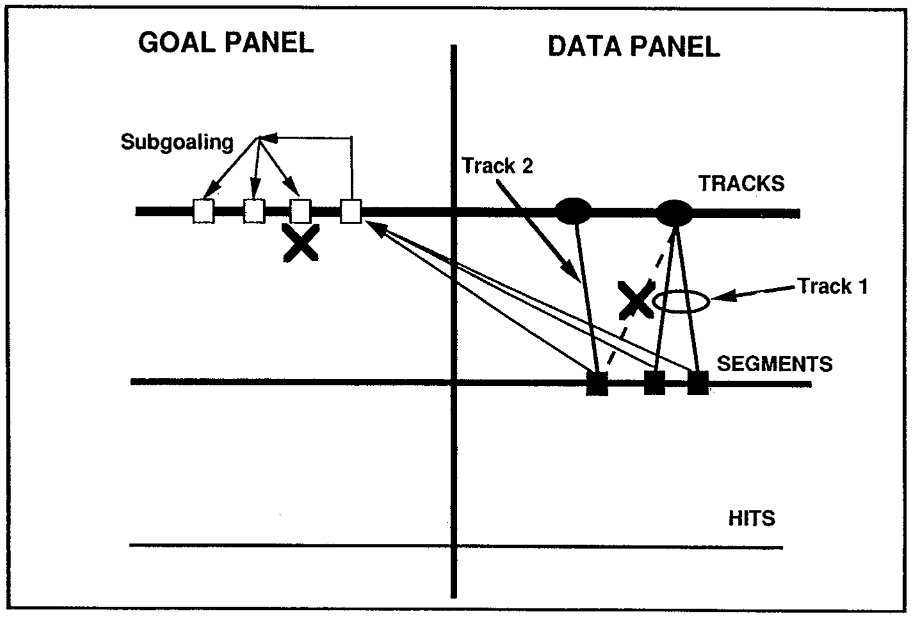

Figure 14. This example illustrates the cancellation of the segment node support of the track 1 hypothesis. Subgoaling triggered by the failure of the spline test is illustrated in the goal panel.

pointers in these nodes are really the parent pointers or the edges of the graph pointing to the root of the tree which represents the track.

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

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

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

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

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

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

After the spline test detects the breakaway 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-based planner will detect a failed track in the goal blackboard, and then generate a subgoal for each segment that supports the track. Each goal expresses the desire to reevaluate the track formation grouping criterion of each segment against the averaged track. The following are the subgoals generated by the rule base:

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

#<bbgoal @ #x7ccl8e>
is an instance of class #<clos:standard-class bbgoal @ #x56dab6>:
 purpose                 verify-track
 event-time              (4)
 initiating-data-level   track
 source                  #<tnode @ #x584e96>
 duration                one-shot
 ksarptr                 <unbound>
 snode                   #<snode @ #x5838a6>
 threat                  nil
 number                  <unbound>
 coord                   ((96.767204 97.039505 0.0) (-0.8006439 -0.99399567 0.0))

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

Each of these subgoals points to the parent track as the source and the supporting segment node as the snode. The KSAR generated from each of these subgoals will activate the VERIFY KS. This KS is part of the blackboard process—that is, it is not spun off as a separate process. If the segment is reverified 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—that is, 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. Lastly, it removes the pointer from the *checklyst* from the track node.

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

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

After the blackboard detects the unmatched segment node, it constructs a distinct track for this segment and the resulting solution consists of the two track nodes given below. The first track node is the newly created node from the unmatched segment 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 racks. The tnode corresponding to Track 1 of Figure 14 is

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

The tnode corresponding to Track 2 of Figure 14 is

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

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.

References

1. 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 Comput. Surv.* 213–253 (1980).
2. K.M. Andress and A.C. Kak, "Evidence accumulation and flow of control in a hierarchical spatial reasoning system," *AI Mag.* **9,** (2), 75–94 (1988).
3. K.M. Andress and A.C. Kak, *The PSEIKI Report—Version 3, Evidence Accumulation and Flow of Control in a Hierarchical Spatial Reasoning System,* Technical Report TR–EE 89–35, School of Electrical Engineering, Purdue University, November 1989.
4. A.R. Hanson and E.M. Riseman, "VISIONS: A computer system for interpreting scenes," in *Computer Vision Systems,* A.R. Hanson and E.M. Riseman, Eds., Academic, New York, 1978.
5. M. Nagao and T. Matsuyama, *A Structural Analysis of Complex Aerial Photographs,* Plenum, New York, 1980.
6. B. Hayes-Roth, "A blackboard architecture for control," *Artif. Intell.* **26,** 251–321 (1985).
7. E. Durfee and V. Lesser, "Incremental Planning in a Blackboard-Based Problem Solver," *Proceedings of the Fifth National Conference on Artificial Intelligence, AAAI '86,* pp. 58–64, Philadelphia, 1986.
8. R. Englemore and T. Morgen, *Blackboard Systems,* edited by R. Englemore and T. Morgen Addison-Wesley, Reading, MA 1988.
9. H.P. Nii *et al.,* Signal-to-symbol transformation: HASP/SIAP case study," *AI Mag.* 23–35 (1982).
10. H.P. Nii, "Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures," *AI Mag.* 38–53 (1986).
11. H.P. Nii, "Blackboard systems: Blackboard application systems, blackboard systems from a knowledge engineering perspective," *AI Mag.* 82–106 (1986).
12. V. Lesser and D. Corkill, "Functionally Accurate, Cooperative, Distributed Systems," *IEEE Trans. Sys. Man Cybern.* **SMC–11**(1), 81–96 (1981).
13. D.D. Corkill, *et al.,* "GBB: A Generic Blackboard Development Systems," *AAAI Conference,* Philadelphia, 1986.
14. I.D. Craig, "The Ariadne-1 blackboard system," *Comput. J.* **29**(3), 235–240 (1986).
15. D.D. Corkill, *A Framework for Organizational Self-Design in Distributed Problem Solving Networks,* Ph.D. thesis, University of Massachusetts, Feb. 1983.
16. 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.
17. R.P. Gabriel, J.L. White, and D.G. Bobrow, CLOS: Integrating object-oriented and functional programming, *Commun. ACM,* **34**(9), 29–38 (1991).
18. S.E. Keene, *Object-Oriented Programming in Common Lisp, A Programmer's Guide to CLOS* Addison-Wesley, Reading, MA, 1989.
19. J.A. Lawless and M.M. Miller, *Understanding CLOS The Common Lisp Object System* Digital Press, 1991.
20. *Allego CL User Guide, Version 4.1 beta,* Franz Inc., Aug. 1991.
21. K.H. Sinclair and D.A. Moon, The Philosophy of LISP, *Commun. ACM* **34**(9), 49–57 (1991).
22. P.H. Winston and B.K.P. Horn, *LISP,* 2nd ed., Addison-Wesley, Reading, MA, 1984.
23. P.R. Kersten and A.C. Kak, *A Tutorial on Using Lisp Object-Oriented Programming for Blackboard Computation (Solving the Radar Tracking Problem),* School of

Electrical Engineering Technical Report, (in preparation) Purdue University, W. Lafayette, IN.
24. S. Hutchinson (personal communication).
25. R.B. Cooper, *Introduction to Queueing Theory,* 2nd ed., North-Holland, Amsterdam, 1981.
26. I. Faux and M. Pratt, *Computational Geometry for Design and Manufacture,* Ellis Horwood, 1979.
27. N.J. Nilsson, *Principles of Artificial Intelligence,* Tioga, Palo Alto, CA, 1980.
28. F.S. Hillier and G.J. Lieberman, *Introduction to Operations Research,* Holden-Day, San Francisco, 1980, Chap. 18.
29. Y. Bar-Shalom and T.E. Fortmann, *Tracking and Data Association,* Academic, New York, 1987.
30. G. Forsythe, M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations,* Prentice-Hall, Englewood Cliffs, NJ, 1977.
31. R. Worden, ''Blackboard systems,'' in *Computer Assisted Decision Making,* G. Mitra Ed., North-Holland, Amsterdam 1986, pp. 95–106.
32. V. Lesser and R. Fennell, ''Parallelism in artificial intelligence problem solving: A case study of hearsay II,'' *IEEE Trans. Comput.* **C-26**(2) 98–143 (1977).