* Introduction
We have produced three prototype solutions to the NSWC challenge problem.

(1) A direct implementation of the requirements posed in Joe Caruso's original document. The functionality of each of the three processes -- sensor, classification and display -- are provided, but there is no modeling of the asynchronous operation of those processes.

(2) A refinement of (1) that models the asynchronous operation of the system. The three processes run in parallel and base their actions on a global notion of time.

(3) The classification algorithm in (1), while specified in such a way that it can be executed in parallel, is elaborated in this version to improve asymptotic efficiency for the case of a large number of radar contacts and doctrines.

The prototypes were all written by Lars Nyland alone, although at various stages in the effort, the problem and some of the solutions were discussed with others. Since the language and interpreter are at a different level of development, we made a decision that we would restrict ourselves to language features supported by the interpreter. We have included some notes on alternatives provided by more recent language changes.

The remainder of this document is organized as follows: we give a short problem overview, followed by our design and implementation descriptions. The design and implementation are then discussed in term of the actual development history of the prototype. There is a discussion of defects in the implementation. The last section is a discussion of metrics, which includes measurements of the executable codes and a discussion of code as a basis for further enhancements.

* Problem Overview
A high-level description of the problem can be stated as follows: Given a list of regions and a changing list of radar returns, compute the intersections of regions and radar returns. The radar data changes over time as the objects move, and some of the regions are based upon the location of a radar return (the region around an aircraft carrier, for instance). The radar data provides an identifier, a location, and optionally a velocity vector and altitude. The regions have an identifier, a location, and a shape (composed of a wide choice of shapes, such as arcs, circles, polygons).
An additional specification of the problem states that the solution should have some form of concurrency. This is inherently an asynchronous process, as the radar data always reflects data that is no more than \(t\) seconds old, and if another process misses some data, it simply misses.

Optional enhancements include:
- scaling up the number of tracks and regions from dozens to 1000s of tracks and 100s of regions
- asynchronous receipt of track reports in bundles of 1-50 tracks

The problem description is described in more detail in the original description and addendum from Joe Caruso at CSC.

* Design
Jan Prins, Peter Mills, and myself [Lars Nyland] got together to discuss possible solutions to the problem. Over the past year, the Proteus group has been concentrating on data-parallelism, so this problem caused a shift in our focus.

We discussed the possible implementations, and derived an asynchronous solution where the concurrent tasks were
- radar data gathering and posting
- intersection computation
- display of intersection results

The question on which we focussed was how to share the information among the concurrently operating processes. The choices are:
- atomically accessed variables
- linear variables
- once variables

The atomic variables have been part of Proteus for quite some time, and exist as part of the interpreter (declared as "shared" variables). It is our current feeling that these may not be the primitive that is needed for the sharing of information among concurrent threads. In our recent work on Proteus, we have developed the concept of "linear" variables, and operations that apply to them. The justification behind the development of linear variables is not only that it provides a more disciplined use of data sharing and synchronization, but they correspond more directly to message passing as well as being a better foundation on which to build many forms of data-transmission and thread synchronization.

Simply stated, a linear variable is akin to a mailbox. If it is empty, it may be filled, otherwise an attempt to store a value in it must wait until it is emptied. Reading a linear variable empties it, or alternatively, there is a non-destructive read. 'Once' variables are only given a value once, and read non-destructively from then on.

Linear variables, plus object-oriented capabilities, allow the creation of many kinds of data sharing constructs. The design of our solution relies on linear variables, but since they are not yet part
of our implementation, their use appears in comments, and shared variables are used.

Threads in Proteus are currently hierarchical, that is, one thread starts several sub-threads and must wait until all sub-threads finish before it can go on. Recently developed constructs allow the spawning of an independent thread, and in doing so, the parent thread can continue to execute. For the solution to this problem, either model of threads is usable, and since the hierarchical threads are implemented, they will be used.

Designs considered but discounted include any sort of synchronous parallel solution to the problem. It is an asynchronous situation, and is best solved in that realm.

One extension considered was the ability to handle larger numbers of tracks and regions. If the track and region data is subdivided over sub-areas of the geographic area, than fewer intersections will be examined. This can be understood as follows: If there are T tracks and R regions, than the straightforward calculation evaluates R*T intersections, keeping only those that actually intersect. If the geographic area is subdivided into N regions, and the tracks and regions are assigned to those geographic sub-areas, then there will be approximately N * (R/N * T/N) calculations, for a reduction in work to approximately 1/N of what it was originally. Of course, it isn't exact, each geographic region will not have exactly T/N tracks and R/N regions (especially since the regions will probably be in more than one geographic sub-area), but the number of comparisons is still substantially reduced.

* Implementation
The implementation proceeded based on the above design. The high-level asynchronous parallel structure is simple to describe within Proteus, the bulk of the time was spent designing the data structures for the radar and region data and implementing routines for the geometric computations.

In the files that constitute the code, there are two kinds of comments. There are ordinary comments, beginning with '---', that provide (hopefully) elucidating information about the associated code. The second form of commentary, beginning with '---+', are comments to other Proteus developers, or comments to explain what a further developed Proteus will support.

The implementation is now described. The top-level routine is simply:

```proteus
main := func() (
  initialize();
  clock() || radar_sensor() || intersect() || display();
);
```

This is a Proteus definition of the function named 'main'. It initializes data structures (including reading the radar data and
region data from a file), followed by the concurrent execution of the routines clock, radar_sensor, intersect, and display. The statement that specifies the concurrent execution finishes when all of the named routines finish, at which time the program is complete.

The 'clock' routine was introduced to allow the radar_sensor routine -know- when to make different data available. As part of the prototype, there is no actual radar to supply data, so that is simulated by having all the data, and making distinct pieces available at different times.

The intersect routine reads the track file data the region data from shared (or linear) variables, and then computes which tracks are in which regions, and sets a shared/linear variable to the result. It relies upon a set of geometric routines for computing the interaction.

The display routine loops reading the shared/linear variable set by the intersection routine, and for the purpose of the prototype, simply prints the data available in that variable.

The implementation of the code for larger numbers of track and regions has the following strategy. We use a data-parallel expression to build a nested data structure of the tracks and regions contained in each of many geographic sub-area. Then a nested operation is executed where for each geographic sub-area, the already developed intersection code is run. The top-level code for this enhancement was straightforward to develop (it is shown below in before/after form), but it required many more geometric routines, thus it took several hours to get it working.

Intersection evaluation (for small data sets):

```plaintext
-- Save all the data where a track is inside of a region.
intersection_data :=
    [ [ t(trkId), r(regId) ] : t in track_file, r in region | overlap(t, r) ];
```

Intersection evaluation (for large data sets):

```plaintext
-- Save all the data where a track is inside of a region.
intersection_data :=

%++[
    -- concatenate all the results
    [ [ t(trkId), r(regId) ] : t in subarea(srTrack), r in subarea(srRegion) | overlap(t, r) ] : subarea in space

where (  
    -- Assign tracks and regions to sub-areas of the whole
    -- area under consideration.
    space := sub_divide(my_track_file, my_region);
 )
```

This modification requires the introduction of an additional data type
for storing the tracks and regions for each geographic sub-area. It is a sequence of tuples, where each tuple contains the tracks and regions in that geographic sub-area.

The above code was purposely written in a nested sequence style, to make use of the parallel execution techniques we have developed for expressions in that style. If the execution of such an expression is performed on a relatively small asynchronous parallel computer (number of processors from 2 to 32), the number of threads used to perform the computation can be dynamically chosen, based on resource information provided. This may be important for reliability considerations.

* Development History and Experience

The first attempt was to build a concurrent thread-based program in Proteus relying on the interpreter for execution. The dual effort of building a concurrent program and writing geometric routines led to a certain amount of difficulty. The control of a concurrent application is inherently more difficult than the control of a single-threaded program, so it was difficult to control the program for purposes of debugging the geometric routines.

With some wise advice, I then wrote a sequential version of the program that was more functional. It has the main function:

```plaintext
serial_model := func() {
    var time;
    time := 0;
    initialize();
    while time < 50 do {
        track_file := extract_tracks(time);
        display_info := intersect(track_file, region);
        display(time, display_info);
        time := time + 10;
    }
};
```

This routine extracts the current track information, computes the intersection and then displays the intersection, and continues doing so until the track data has expired.

Developing the sequential version led to a more concise concurrent version, and a shared set of geometric routines.

The changes made to the routines to achieve concurrency had to do with supplying communication mechanisms other than parameters (i.e. shared variables), a way to get the routines to run more than once per call, and a graceful exit of each of the concurrent routines. The method used to get the routines to execute many times was to simply wrap the body of the routine in a while-loop that exited when a shared variable indicated the execution should end. That same shared variable is what provided a graceful termination of each of the threads.
A refinement was to have the threads terminate in order. The process is essentially a pipeline, as information passes from the sensor process to the intersection process to the display process, so the termination condition was set up similarly. The display process does not exit until the intersection process exits, and it doesn't exit until the sensor process exits. This was done by defining two shared variables, sensor_done and intersect_done, with the appropriate assignments inserted and the proper tests in while loops.

The last step was to build a version that supported better calculation of interactions (for the larger data sets suggested in the addendum). As stated in the Implementation section, the high-level additions were rather straightforward, however, several geometric routines were needed to aid in this computation. They were routines to determine if a region of any one of the allowable forms had any overlap with the geographic sub-area (which are all rectangular). Instead of writing all of the pairwise overlap routines (arc-arc, arc-circle, arc-rectangle, ...), I chose instead to implement a routine for each possible shape that would return its bounding box (a rectangle that encompasses the entire shape). This may lead to more interactions than necessary, but the overlap of shapes is not what is being prototyped here, so the generous choice of using bounding boxes simply gave me what I felt was a reasonable platform on which to build the code for the increased data.

While the geometric aspects of the prototype aren't what is being prototyped, I thought I'd add just a few words about how they were implemented. I defined some primitive shapes, which are rectangle, circle, polygon (convex only), and arc. I then defined some additional composite shapes, union, intersection, and diff. The constructive flavor of the geometric shapes allows the definition of all of the shapes necessary for the prototype. The tight zones are implemented as the union of enough convex polygons to create the region. The doughnut-shaped engagement region is implemented as the difference of two circles. No shape used the intersection-composition, even though the arc could have been implemented as such (the intersection or difference of a polygon and a circle). I felt it was easier to implement the arc as a primitive rather than calculate the necessary polygon that interacts with the circle.

While there is no current implementation for objects within Proteus, the availability of maps and functions-as-values allowed me to dispatch function calls based on the type of shape being used. This was used for both the 'in_region' functions and the 'bbox' functions. Each of these is a map whose domain is a string (such as "circle"), and whose range is the function to perform that operation for that shape. Thus, the definitions of the functions have the form:

```plaintext
in_region("circle") := func(c, point) (...);
in_region("polygon") := func(p, point) (...);
in_region("union") := func(u, point) (...);
...```

```
and the invocation of the correct function has the form

\[
\text{in\_region}(r.\text{rType})(\text{r.rData}, [x, y]);
\]

The expression 'in\_region(r.rType)' looks up (via a hash-table) the function definition corresponding to the type, and then calls that function with the parameter list '(r.rData, [x, y])'.

* Defects

Of course, during development, defects were introduced as I developed the code. And during development, I found and removed the defects until I was content with the program. The availability of a functioning environment allowed me to explore the interactions of different routines which run at different speeds (the intersect routine performs many more computations than the radar\_sensor routine for one iteration of each). I could see that if the clock routine allowed a time variable to increase too fast, then radar data would be written and then overwritten before the intersect routine was able to get a snapshot of the radar tracks.

One of the difficulties is getting everything started and stopped without any errors. The track\_file shared variable must be defined correctly for the intersect routine to work, there is a dependency on the track\_file to locate the slaved doctrines, so the radar data indicating the positions of the objects on which the slaved doctrines are based must be present.

Perhaps this is a current defect. The program will fail if the objects on which the slaved doctrines are based are not in the radar tracking data. One possible solution to this problem is to eliminate the slaved doctrines with no track information. This seems especially important in the case of slaved doctrines based on aircraft locations, as the aircraft may leave the region, so the doctrine slaved against that data should simply not be used in the intersection computation. There were no instructions about what to do in this case from the original specifications, but one could easily assume (and verify with the client) that slaved doctrines with no radar datum on which to base the doctrine can be silently ignored. To do so, a change in the code might be

[existing code]

```
locate := func(region, track\_file) {
    for r in domain(region) do {
        if region(r)(\text{Slaved\_To}) /= om then
            region(r)\{\text{Origin}\} := track\_file(region(r)\{\text{Slaved\_To}\})\{\text{Loc}\};
        
    
    return region;
    
};
```

to

[suggested code]

```
for r in domain(region) do {
    ```
if region(r)(Slaved_To) /= om then
    if track_file(region(r)(Slaved_To) /= om then
        region(r)(Origin) :=
    track_file(region(r)(Slaved_To))(Loc);
else
    -- remove the region r from the map.
    region(r) := om;

* Metrics
This section describes some of the metrics that we suggest for evaluating our solution.

** Lines of Code
There are 3 variants of the program for which the lines-of-code metric will be stated. The first (v1) is the single-threaded version, the second (v2) is the concurrent version of a handful of threads, and the third version (v3) is the extension of the second to handle much larger data sets. Only the needed geometric routines will be counted for each (v3 required more routines than v1 & v2).

<table>
<thead>
<tr>
<th>Lines</th>
<th>';'s</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1:</td>
<td>213</td>
<td>81</td>
</tr>
<tr>
<td>v2:</td>
<td>293</td>
<td>103</td>
</tr>
<tr>
<td>v3:</td>
<td>389</td>
<td>138</td>
</tr>
</tbody>
</table>

I did not count the file that contains the data for the radar or the regions, since this data is not part the program.

** Development Time
Three of us met for a two-hour meeting to design the solution. We had each read the problem description and the addendum supplied by Joe Caruso. In addition, I had attended both meetings (10/9, 11/4). I then spent 26 hours writing, debugging and commenting the code. This does not include the time spent writing this report, nor does it include the time required to boost the capabilities of our interpreter.

There was probably another hour or two of design discussion between other members of the group and myself, and there were some uncounted number of discussions among the other members of the group.

** Set/Sequence/Tuple Notation
The set/sequence/tuple notation may seem unfamiliar to many programmers, mainly due to the fact that it is not supported in many programming languages. It is derived from set notation, which is prevalent in math texts used as early as secondary or even primary school. Additional notation for the language must be specified, since there is, for instance, no capital sigma with which to specify the summation of the elements of a sequence.

For those who feel at sea with the notation, let me describe one
expression in the prototype being discussed with an approximate translation to C. The expression is

\[
\text{intersection_data} := \\
\{ \{ \text{t(trkId), r(regId)} \} : \\
\text{t in track file, } \text{r in region} \\
\text{| overlap(t, r) } \};
\]

The expression on the right-hand side specifies that for all values of \( t \) taken from the track_file (a sequence of radar data), and all values of \( r \) in the region variable (a sequence of regions), call the function overlap with the track \( t \) and region \( r \), and if it returns a value of true, include (a tuple consisting of) the id's of the track \( t \) and region \( r \) as one element of the resulting sequence.

The (roughly) equivalent C is (at the very least):

\[
\text{Noverlaps = 0;}
\text{if (intersection_data != NULL)}
\text{\quad free(intersection_data);}
\text{intersection_data = malloc(sizeof(struct overlap_s), ntracks*nregions);} \\
\text{for (t = 0; t < ntracks; t++) {}
\text{\quad for (r = 0; r < nregions; r++) {}
\text{\quad \quad if (overlap(track[t], region[r])) {}
\text{\quad \quad \quad intersection_data[Noverlaps].trk = track[t].trkId;}
\text{\quad \quad \quad intersection_data[Noverlaps].reg = track[t].regId;}
\text{\quad \quad \quad Noverlaps += 1;}
\text{\quad \quad }}
\text{\quad }}
\text{\quad }}
\text{\quad }}
\text{\}}
\text{\}}
\text{While the C code is not obviously detestable compared to the Proteus code, consider the following observations. The memory allocation cannot know ahead of time how large to actually make the intersection_data array, so the simple solution is to make it as big as will ever be needed. More exact methods are easily possible, but the mere fact that the programmer need worry about memory allocation is a hinderance.}

Next, if the array intersection_data is shared with a display procedure, then so must the counting variable Noverlaps. The order of the above C statements must be carefully chosen, or alternatively locks must be used for the updating and reading of the intersection_data array.

The types of all of the variables in C must be well defined (declared). This is not true in Proteus, the creation of the sequence of tuples representing the intersection_data can be changed as seen fit. Of course, the if the routine that the data is shared with depends on the structure of the data, then it will have to be modified, but in this case, it is simply printed, so it is easy to experiment with alternate forms (which was indeed done as I experimented with the rates of computation of the different threads,
the times at which the radar data was gathered, the intersection computed, and the display time were all a part of the data at different times in the development).

**Ease of Modification**
Given full Proteus, I'll make the quantitative claim that programs written using it are easier to modify, especially when we have a module construction environment. The biggest reason I have for saying this is that the amount of code that needs to be written is smaller than many other languages due to the support provided by the language. The programmer is freed from memory management, ensuring no memory leaks, allocation or initialization of data structures for concurrency, managing concurrent threads, declaration of many variables, such as loop index variables, and declaration of types (this is indeed an arguable point).

**Extensibility**
I have shown two extensions already, one for increased data, and one for more reliable computation when not all the expected data is present. The extension were short and direct. I would say that some of the other suggested extensions that would be as straightforward are those dealing with

- extending the slaved doctrines to aircraft
- bundling of radar track data
- maintaining a doctrine history
- taking the velocity and altitude of the radar returns into account
- adding a "tactical action aid"
- adding more shapes, and changing the underlying geometry to a spherical rather than flat surface

just to name the ones that seem readily included in our prototype.

**Enhancement, Refinement, and Analysis of Prototypes**
The prototypes developed to date represent several steps in an evolutionary process of software development -- an iterative process that proceeds along the dimensions of both enhancing the algorithmic solutions and of refining the specificity of the parallelism to take greater advantage of the underlying architecture (while preserving the functionality). Part of the development process is feedback through analysis and measurement of the prototype.

We thus envision the next steps in the prototyping experiment to address enhancements in the problem specification (e.g., track file size and tactical predictions for engageability), fundamental algorithmic enhancements for improved parallel performance, refinements in the form of the parallelism to more closely target the underlying parallel model, and application of techniques for predicting performance through simulation.
*** Algorithmic enhancements: problem decomposition

A fundamental parallelization technique applicable to the problem of detecting doctrine intersections is spatial decomposition -- subdividing the problem by decomposing the space into separate subregions and assigning to each a separate process to handle the task of intersecting targets with doctrines touching that subregion. The decomposition can be recursive, and the naive partitioning for the 2D case is into "quad-trees" (each region dividing orthogonally into 4 rectangles). In the hierarchical decomposition, a naive solution will have the intersection tasks located at the leaves (the finest partitioning). Since this is at heart a data decomposition it can be captured either in a data-parallel manner or explicitly through task-level parallelism.

There are some subtleties involved in this "image-based" partitioning approach. First, doctrines which overlap several subregions have to be broadcast to each of them. Secondly, a further parallel speedup can be achieved by replicating several tasks for a single subregion, and handing out a subset of the doctrines to each task. This might be termed "horizontal" decomposition. Thirdly, less naive intersection algorithms will do some intersection work at higher levels of the tree, akin to many graphics rendering algorithms (basically a computational geometry problem) and multi-grid methods such as Fast Multiple Algorithm.

*** Algorithmic enhancements: tactical prediction

Another elaboration is to provide some prediction of future target-doctrine intersection. This can be naturally handled in two ways. First, the introduction of time can be regarded as extending the 2D problem (of detecting target/doctrine intersections at one point in time) to a 3D intersection problem, where the third dimension is time. That is, the potential track of a target over some finite future duration forms a cylinder (or more correctly, an elliptically shaped object), which can be input into the intersection task to determine future contacts. Alternatively, one can time-step the predicted locations of objects, at each step performing 2D intersection task. The time-stepped processes can also be multiply spawned as independent processes in order to speed up predictive response.

*** Concurrency refinement

Several restrictions to the way that the concurrency is expressed greatly
impact performance and so bear exploration. For example, the track file to be used for geo-region intersection can be updated by the radar sensor task either in entire blocks or piecemeal. The latter would induce a more asynchronous algorithm, entailing finer-grained concurrency access. Furthermore, if the track file is extremely large, in-situ updates to the large concurrent aggregate may have to resort to explicit lock/release rather than linear-operators (which perform copying).

Furthermore, data-parallel solutions can be dynamically virtualized to a different number of processors, a promising technique for promoting fault-tolerant applications which must accommodate losing processors.

Performance Prediction and Real-Time Constraints
-----------------------------------------------------------------

Currently our sequential interpreter models real-time constructs (and the prediction of performance though simulation) by assuming maximal parallelism, uniform processor speeds, and uniform instruction cost. We would like to gather more detailed simulations of performance that relax some or all of these constraints, as well as be able to give some proof of temporal correctness -- i.e. that there is a feasible scheduling.

Rather than attempt to address these issues in isolation within our language and development system, we envision interfacing with the advanced real-time prototyping technology being developed by Jane Liu and colleagues at the University of Illinois at Urbana-Champaign. The PERTS system (Prototyping Environment for Real-Time Systems) supports the synthesis and validation of real-time systems, capitalizing on recent theoretical advances for rigorously predicting real-time behavior. In particular the PERTS system provides both a schedulability analyzer -- which determines the feasibility of scheduling parallelizable jobs with deadlines and which can suggest possible changes to task and resource parameters if needed to meet feasibility -- as well as simulation and measurement tools which extract the processing time and resource requirements from annotated source code. A promising avenue of investigation is to examine the possibility of using the PERTS system (and possibly other simulation tools such as Purdue's PAWS) as a back-end for analyzing performance.

** Ease of Understanding
The ease of understanding is yet to be judged. Certainly any solution written in a familiar-looking language or familiar model of programming will immediately seem more understandable. In the development of Proteus, we have strived to supply the user with the
ever-present imperative block-structured programming model (that of C, Ada, Pascal, etc.) while adding features that allow more expressibility using a familiar notation (e.g., set comprehension) while taking care of important but not intellectually challenging details underneath the surface.

While the Proteus programming language is established enough to develop and execute programs, it is still very much under development, so any suggestions about what could be better and what is misleading could quite easily have an effect on the next round of enhancements. Please feel free to make comments to me (nyland@cs.unc.edu) or any other member of the Proteus ProtoTech group.