An Experiment Using Haskell to Prototype "Geometric Region Servers" for Navy Command And Control[1]

William E. Carlson
Intermetrics, Inc.

Paul Hudak
Yale University

Mark P. Jones
Yale University

November 19, 1993

Intermetrics Report IR-MA-1390

Research Report YALEU/DCS/RR-1031

[1]This work was supported by the Advanced Research Project Agency and the Office of Naval Research under ARPA Order 8888, Contract N00014-92-C-0153.

Executive Summary

We describe the results of using the functional programming language *Haskell* for prototyping a Naval Surface Warfare Center (NSWC) requirement for a *Geometric Region Server*. Our findings support the belief that Haskell can be used effectively to prototype large software systems: the prototypes that were developed took significantly less time and resulted in more concise and easier to understand code than the corresponding prototypes written in Ada or Common Lisp. This allowed us to more quickly search the design space before committing to an implementation. Furthermore, we found that Haskell complements Ada 9X. Haskell is an excellent tool for exploring algorithms in the problem space, while Ada 9X is a powerful tool for engineering a readable, maintainable production quality implementation. We conclude that using Haskell in a software development process that incorporates disciplined prototyping will yield higher quality software at lower cost.

1 Introduction

The Intermetrics-Yale team is pleased to provide the Naval Surface Warfare Center (NSWC) in Dahlgren, VA, with this report on an initial experiment comparing prototyping processes and programming languages. This experiment relates to several larger and more comprehensive experiments being conducted jointly by the Advanced Research Project Agency (ARPA) Prototyping Technology (ProtoTech) program, ARPA's HiPer-D program, the Office of Naval Research, and NSWC.

The Intermetrics-Yale team has used Intermetrics' prototyping methodology to compare use of the Haskell language to the current state-of-the-practice based on Ada and Lisp. Other teams participating in this experiment have used related methodologies but different languages. The purpose of this report is to document our experimental procedures and results, and to provide a basis for comparing our experience with that of the other teams.

The ARPA and ONR goal for this experiment was to test the hypothesis that Haskell is an excellent prototyping language and an effective tool in the context of the Intermetrics prototyping methodology. Our experimental results support the truth of this hypothesis. The report that follows contains detailed information and metrics on the experiment. Our approach took advantage of Haskell's support for literate programming; the LaTeX manuscript for each Haskell prototype also serves as a directly executable version of the prototype program.

The NSWC goal for the experiment was to quickly develop a prototype *Geometric Region Server*, or *GEO Server* to test certain architectural hypotheses related to the AEGIS Weapons System (AWS) redesign. Thus, our approach to meeting the ARPA and ONR goals was: 1) To do the best possible job of meeting the NSWC goal, 2) To keep careful metrics on the prototyping process used to develop the prototype for NSWC, and 3) To analyze those metrics from the point of view of research in prototyping processes, languages, and tools.

Mark Jones of Yale developed three Haskell prototypes of increasing capability and generality for the "Prototyping Demonstration Problem for the Prototech HiPer-D Joint Prototyping Demonstration Project," dated 6 August 1993 [Reference 1], as modified by the addendum dated

9 November 1993 [Reference 2]. This report provides the design rationale and assumptions, our approach to exploring the problem space, a mapping between the prototyping language Haskell and the problem domain, the prototype code and scenario output, the evaluation metrics, and a list of deliverables. The Haskell prototypes are characterized as follows:

Prototype 1: The purpose of Prototype 1 was to see how quickly a good solution to the stated prototyping requirements could be written. A first draft of Prototype 1 was written, documented, and tested in three hours after the author spent one hour studying Reference 1. This draft was based entirely on Reference 1; the author did not interact with anyone in developing the solution. In a "walkthrough" of the code produced and its accompanying documentation, no logical errors were found; but a decision was made to make certain cosmetic improvements to the documentation in order to simplify future review and maintenance. The updates were completed in less than an hour, resulting in the source code, documentation, and test results presented in Appendix B.

Prototype 2: We believe that our Prototype 2 will be most comparable to the prototypes prepared by the other teams. This prototype satisfies all mandatory requirements of the problem as defined in References 1 and 2, and also incorporates generalizations to support anticipated future requirements. To accomplish this task, the original author of Prototype 1 made a number of modifications and enhancements requiring about five hours (including all coding, documentation, and testing). The documentation, annotated source code, and test results are presented in Section 7 of this document.

Prototype 3: Because Haskell is such a powerful tool for exploring this problem domain, we were able to extend Prototype 2 to include about half of the possible 14 enhancements currently being considered for future implementation. A total of 25 hours were spent extending Prototype 2 to create what we refer to as Prototype 3, again including all coding, documentation, and testing. A large part of this time was spent simply determining the mathematics required to capture notions of "engageability." This prototype is presented in Appendix A, and an analysis of it is included in Section 6.

Our team developed the three prototypes in an ascending order of generality and functionality in order to show the ease and speed with which original and additional requirements can be implemented and communicated, thereby demonstrating extensibility, understandability and appropriateness. Productivity metrics for each prototype are contained in Section 6 of this document, which includes a comprehensive view of all metrics collected during the experiment.

We also addressed two additional issues associated with the use of Haskell as a prototyping language. First, we conducted a concurrent experiment on the ability of "new hires" to learn Haskell and to become productive. Second, we conducted an experiment on the usability of a Haskell program as a design for an Ada program. The methodology for these experiments is discussed in Sectio 5 of this document, and the results are analyzed in Section 6.

1.1 Report Overview

This report is organized into eight sections and seven appendices. A brief description of each is given below.

- Section 1 – Introduction: This Introduction briefly describes the background and context of the experiment, as well as our team's approach. It also includes this description of the structure and content of the remainder of the report.
- Section 2 – Applicable Documents: This section lists the documents referenced in developing the report.
- Section 3 – A Brief Introduction to Haskell: This section presents an overview of the Haskell prototyping language.
- Section 4 – Intermetrics Prototyping Methodology: This section summarizes the prototyping software development process that Intermetrics recommends be used for large system development. Haskell is a powerful tool which supports this approach. The experiment described in this report was a prototyping experiment in its own right, so it was performed in accordance with the Intermetrics Prototyping approach, and was also a (small) experiment in the overall design process for the next generation Aegis Weapons System.
- Section 5 – Experiment Design: This section defines the objectives of the experiment, the evaluation metrics, and a list of deliverables. It also contains experience profiles on the key participants of the experiment.
- Section 6 – Metrics and Analysis: This section provides an analysis of the metrics data gathered during the experiment.
- Section 7 – Baseline Solution (Prototype 2): This section provides the design rationale and assumptions for solving the specified experimental problem, the approach to exploring the problem space, a mapping between the prototype language and problem domain, and complete source code and documentation for one solution to the problem.
- Section 8 – Conclusion: This section summarizes the overall conclusions of the experiment.

Seven appendices are included for completeness, and as a convenience to the reader. They include three Haskell solutions to the problem, an Ada 9X solution, a Relational LISP solution, and copies of References 1 and 2.

1.2 Executable Versions of Programs

The Yale Haskell system, documentation and solutions to the experimental problem are available on the Internet. Anonymous ftp can be used to retrieve the following files from the host nebula.cs.yale.edu:

1. The Yale Haskell system can be downloaded from pub/haskell/yale/haskell-206*.

2. The Haskell report is available from pub/haskell/report/report-1.2.{dvi,ps}.Z.
3. The Haskell tutorial is available from pub/haskell/tutorial/tutorial.ps.Z.
4. The Haskell solution presented in Section 7 is available from pub/proto/s2.lhs.
5. The extended Haskell solution which includes engageability, corresponding to Appendix A of this report, is available from pub/proto/s3.lhs.
6. The rapid prototype written as the first step in the prototyping process, a copy of which is provided in Appendix B, is available from pub/proto/s1.lhs.

2 Applicable Documents

1. J. Caruso, "Prototyping Demonstration Problem for the Prototech HiPer-D Joint Prototyping Demonstration Project," CCB Report, Version 0.2, August 6, 1993 and last modified on October 27, 1993.
2. J. Caruso, Addendum to "Prototyping Demonstration Problem for the Prototech HiPer-D Joint Prototyping Demonstration Project," November 9, 1993.
3. P. Hudak and J. Fasel, "A Gentle Introduction to Haskell," *ACM SIGPLAN Notices*, Vol. 27(5), May 1992.
4. P. Hudak, S.L. Peyton Jones and P. Wadler (eds.), "Report on the Programming Language Haskell, version 1.2," *ACM SIGPLAN Notices*, Vol. 27(5), May 1992.
5. M.P. Jones and P. Hudak, "Implicit and Explicit Parallel Programming in Haskell," Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-982, July 1993.
6. *Programming Language Ada: The Language and Standard Libraries*, ISO/IEC CD 8652, September 15, 1993. Also available from the Ada9X Mapping/Revision Team, Intermetrics, Inc., IR-MA-1363-3.
7. *Rationale for the Programming Language Ada*, ISO/IEC CD 8652, September 15, 1993.
8. G.L. Steele, Jr. et. al., COMMON LISP, Digital Press, 1980.

3 A Brief Overview of Haskell

Haskell is a general purpose, purely functional programming language, named after the logician Haskell B. Curry, and designed by a 15-member international committee. The committee's goal was to design a "common" functional language that satisfied the following constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.

2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available; anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should serve to reduce unnecessary diversity in functional programming languages.
6. It should serve as a basis for future research in language design; it is expected and desirable that extensions or variants of the language may appear.

The culmination of this effort was the release in April, 1990, of the *Haskell Report* [Reference 4], the official document that defines the language (the current revision, Haskell 1.2, is dated March 1992).

For those not familiar with functional languages, it is helpful to keep a few basic concepts in mind when reading the prototypes presented later. First, Haskell is *purely functional*, which means that it has no constructs for inducing side effects to an implicit store (as found in imperative languages as well as in "almost-functional" languages such as ML and Scheme). This also means that there is no sequential flow of control in a Haskell program (since there are no side effects that require sequencing) – there are only *data dependencies* – and thus Haskell programs typically contain many implicit opportunities for parallelism.

Programming in Haskell requires thinking *declaratively*: i.e. describing *what* is to be computed rather than *how*. In general, very little committment is made to operational detail. Yet Haskell programs can be easily refined later for efficiency (thus supporting the notion of "get-it-right-first"). For these reasons alone, Haskell is an excellent prototyping language.

In addition, the declarative nature of Haskell gives programs a mathematical, specification-like quality. This enhances application of formal methods in the form of either program synthesis or program verification. We encourage the reader to note the many examples in the prototype where, once the mathematics of the solution is determined, the Haskell code to implement the algorithm is essentially a direct translation of the mathematics. Similarly, we are able to prove many interesting properties of the prototype using equational reasoning that is no more difficult than high-school algebra. In short, Haskell programs are very abstract, yet executable.

Haskell also represents the state-of-the-art in high-level language design, containing advanced language features such as higher-order functions, non-strict semantics (i.e. lazy evaluation), a rich polymorphic type system, and pattern-matching. In addition, it has several novel features that give it additional expressiveness, including an elegant, object-oriented form of overloading using a notion of *type classes*, a flexible purely functional I/O system, and a purely functional array datatype.

Haskell is now being used world-wide in a variety of applications. Several excellent implementation exist, including two being distributed at Yale: the Yale Haskell System (compiler-based), and Gofer (interpreter-based). All of the prototypes described in this document have been compiled and executed.

4 Intermetrics Prototyping Methodology

The ProtoTech program defines prototyping as *disciplined experimentation*. A prototype is an executable model, specification, or implementation for a component, subsystem, or complete system. Some prototyping is purely experimental in nature, producing information which is incorporated into requirements and designs. In other cases, however, prototype code and documentation can be reused to form part of a desired product. A key difference between a prototype and a product is the extent to which the quality of the product must have been assured.

In a disciplined prototyping process, each experiment which is performed must have clearly defined objectives, schedule and budget. If there is any possibility that prototype source code will be reused in a product, then the code must be written with the attitude that it is production quality code. It has been shown time and time again that testing should be used to confirm the absense of defects, rather than to find and fix defects.

Intermetrics is focusing on prototyping processes for developing large systems which push the state of the art in one or more dimensions. The major steps and milestones in software and systems development using prototyping are the same as the milestones in traditional processes:

- Requirements definition stage, during which requirements are clarified and a requirements document is produced. This stage is completed when the requirements document and development plan are approved.
- Top level, or architecture design stage, during which design alternatives are explored, a design for the system is written, and its feasibility is demonstrated. This stage is completed when the preliminary system design is approved.
- Detailed design and implementation stage, during which the code is produced, documented, integrated and tested. This stage is completed when analysis, testing and reviews confirm that a high quality implementation of the design and requirements has been completed.
- Initial deployment, post deployment support and enhancement, during which the system is used for a period of time as initially installed, and then upgraded to adapt to changing circumstances and new requirements.

In the Intermetrics process, experiments using prototypes have a role in each stage of development and post deployment support. During requirements definition, numerous experiments are often performed to clarify requirements and to determine technical feasibility. Often, these experiments are quite independent of each other, integrated only by the vision defined in an operational concept document and a list of the risks which must be mitigated.

During top level design, there will be a few experiments aimed at converging on a good top level architectural design for the desired system and demonstrating that performance goals can be achieved. After the preliminary design is approved, a relatively conventional approach is followed during the implementation period. The order in which modules are coded is chosen so that the system can be integrated incrementally, rather then deferring integration until all modules have been coded and unit tested. An effort is made to implement the most difficult thread through

the system first, so that any problems can be identified early and corrected in order to minimize rework.

After deployment, experiments are used to determine the most cost effective and least disruptive way to implement system enhancements. Each system enhancement is a development project in its own right.

The disciplined prototyping process requires peer reviews and code walkthroughs of every document and every module which is produced. Even very early in the requirements exploration phase, people writing documents and code should have the attitude that what they write may survive and become part of the delivered product. By striving for zero defects during every process activity, the probability of defects being introduced and being passed from one process step to the next is reduced significantly.

Each fine grain process step is similar to writing a paper for publication:

1. write a high quality draft as quickly as possible,
2. have the draft reviewed by peers in a formal walkthrough,
3. update the draft based on corrective actions identified during the walkthrough.

This same process is followed whether the artifact is a document providing the overall vision for the system or the production source code for a module. If the artifact is executable, then a fourth step can be added:

4. test the artifact to confirm that no symptoms of defects can be found.

The amount of effort expended on testing depends on the objectives for that process step. The Intermetrics philosophy is to test enough to ensure that the specific experimental objectives are met, and to reserve a greater part of the testing budget for the more complex system integration period. If everyone involved in the development has been committed to developing high quality artifacts, then relatively few defects should be found during system integration, and the testing should confirm the system's overall quality.

Project plans for development using prototyping take into account that some experiments will produce unexpected results and that some code which is written will be thrown away. The reason for doing experiments is to manage risk and to eliminate uncertainties early in the development process. Hence, it is important to treat the entire development as a continuous improvement process in which everyone on the team is working together to produce the best possible solution to user needs within budget and on schedule. We work hard to avoid associating a negative value judgement with a decision to throw a module away, and strive for an "egoless" approach to choosing among alternate approaches. Decisions are always forward looking: "based on the information which has been learned, should a particular artifact be reused in the next development stage?."

We are convinced that using prototyping experiments to reduce risk as early in a project as possible actually reduces cost while improving quality. The paradoxical conclusion that you can lower costs by planning to throw some code away is consistent with numerous studies of software

engineering productivity. Defects cost much more to remove late in a process than early, and design problems are much more costly to correct than coding errors. Furthermore, productivity declines exponentially as project size increases, so, for example, five projects with five peaple each will produce much more useful code than one twenty-five person project. By using prototyping to finalize subsystem interfaces early and to determine experimentally that the interfaces are well designed, one can substantially reduce the overhead of human communication on a large project. In effect, the architecture and interfaces partition the large project into several smaller ones, thereby increasing productivity substantially.

5 Experiment Design

The experiment described in this report was very small compared to the overall objectives of both Aegis and the ARPA ProtoTech program. The Aegis objectives are described in Reference 1 (see Appendix F) as follows:

> The AEGIS Weapons System (AWS) is an extensive array of sensors and weapons designed to defend a battle group against air, surface, and subsurface threats.
>
> . . .
>
> "The AWS is undergoing a major redesign. As part of this redesign several functionally similar algorithms (currently dispersed throughout the system) have been proposed as targets for consolidation. It is speculated that consolidation of these algorithms can reduce redundant computation and offer a better opportunity to utilize the parallel computation capability of modern architectures. It is also felt that large blocks of consolidated processing can better foster reusability and the benefits associated with the client/server architecture."
>
> . . .
>
> ". . . the role of AEGIS is changing from one stressing defense against massive raids in open ocean to conflicts fought in the complex and contact rich environments of confined bays and straits. This change requires . . . a refined ability to differentiate and separate friend and foe. This complexity makes the ability to quickly test and fine tune algorithms through prototyping very compelling." [p. 2]
>
> . . .
>
> The current AEGIS consists of millions of lines of CMS-2 code. This experiment, on the other hand, was to take place during a two week period, and the initial guidance was that about five person days of effort should be devoted to each prototype.

Everyone involved viewed this experiment as just a first step in applying prototyping technology to help NSWC develop future generation AEGIS and similar systems. Reference 1 carefully defines the objectives for the experiment:

- prototype "algorithms which compute presence of tracks within geometric regions in space." [p. 3] These are referred to as "doctrine-like capabilities." [p. 2]
- "minimize work to achieve a first delivery" [p. 8]
- "allow room for incremental enhancement." [p.8]

Reference 1 also defines the criteria for evaluating the prototypes which are produced:

- Extensibility – ease with which the prototype can be extended to fulfill additional requirements.
- Understandability – rapidity with which the prototype can be absorbed and altered.
- Appropriateness – the expressiveness of the language relative to the task at hand.
- Accuracy – correct functioning of the prototype.
- Compactness – capability per line of code.

Given these objectives and criteria, the Intermetrics-Yale team adopted the following plan:

1. Have an experienced Haskell programmer write a solution to the problem as quickly as possible (Prototype 1). Have this person work independently, using only the information available in the written problem statement. This person was Mark Jones of Yale. He received his PhD in Computer Science in 1992, and has been at Yale as an Associate Research Scientist for about 15 months. He has programmed in Haskell for about 3 years, and also wrote one of the popular Haskell implementations (Gofer).
2. Conduct a formal walkthrough of Prototype 1. Participants in the walkthrough included:
   - Bill Carlson, Intermetrics VP and Chief Technology Officer, with 25 years industry experience including successful management of over $100M in software development projects.
   - Bernie Domanski, manager of the Intermetrics Naval Warfare Systems Department and a retired Naval Officer (Qualified Surface Warfare Officer). Mr. Domanski was the team representative at the kickoff meeting for the experiment.
   - Virgil Banowetz, a senior systems engineer with 24 years total experience and 3 years experience implementing Navy Command and Control Systems in Ada.
   - Paul Hudak, Professor of Computer Science at Yale, with 5 years industrial experience and 11 years of programming language research and teaching experience. Prof. Hudak was a key leader in the Haskell design effort.
3. Have the participants in the walkthrough identify (a) defects and improvements to Prototype 1, and (b) objectives for evolutionary improvements to Prototype 1.

The intent was that the time for Dr. Jones to code Prototype 1 provide a quantitative metric of success relative to the NSWC objective that the time to produce an initial solution be minimized. Mr. Carlson, Mr. Domanski and Mr. Banowetz would all evaluate the prototype for understandability, appropriateness and accuracy. Extensibility would be evaluated by creating the two enhanced versions of the prototype.

It happened that some modifications and extensions to Reference 1 were agreed to at the experiment kickoff meeting at NSWC. Dr. Jones and Dr. Hudak were not informed of those extensions until after the completion and review of the initial draft of Prototype 1. After the review, we distributed the written statement of extensions to the problem statement (reference 2, Appendix G). We then discussed each of the requested extensions. We went beyond the extensions, however, to use Mr. Domanski's problem domain expertise to discuss likely future directions of evolution and generalization of the prototype. The objectives for Prototype 2 and Prototype 3 were defined based on these discussions.

To get a sense of how easy Haskell is to learn, we had a recent college graduate spend eight days learning Haskell from publicly available documentation. This new hire received no formal training, but was allowed to ask an experienced Haskell programmer questions as issues came up during the self study. After the eight days training, we gave him a day and a half to write a solution to the GEO Server problem. The trainee's prototype is included in Appendix C, and statistics on his approach are discussed in Section 6.

To address the issue of whether a Haskell prototype is a useful design document from which to develop an Ada program, we gave References 1 and 2 and the Haskell Prototype 1 to a member of the Ada 9X design team who has many years of Ada programming experience. He was asked to code a solution to the problem defined in the references. His solution is provided in Appendix D.

Lisp has often been used to write prototypes. We gave References 1 and 2 to Dr. Robert Balzer of the University of Southern California Information Sciences Institute. He is a very experienced Lisp programmer, and has developed a special Lisp system enhanced with a capability called Relational Abstraction for Prototyping. There was no technical interaction between Dr. Balzer and the people writing in Haskell. Dr. Balzer will be submitting his prototype as an independent submission, but some preliminary data on his prototype is discussed in Section 6, and an initial version of the prototype is provided in Appendix E.

6 Metrics and Analysis

As discussed above, this small experiment supports two research objectives. First, and most importantly, it is an initial exploration of NSWC's hypothesis that consolidation of algorithms to compute the presence of tracks within geometric regions in space and packaging those algorithms into a GEO Server can significantly improve the next generation of ship defense systems. Second, the experiment is an evaluation of Haskell as a language for exploring such questions.

This section first presents the metric data collected on the prototyping process and on the prototypes which were produced. It then provides a preliminary analysis of this data.

6.1 Product Metrics

The Haskell prototype was constructed by iterative refinement. Briefly, the three versions are as follows:

- P1: Original problem as specified in Reference 1.
- P2: Modified problem based on a later email message (Reference 2). This solution goes beyond P1 as follows:
  1. Allows an arbitrary number of slave ships and tracked aircraft (the specification asks for up to 400).
  2. Includes notion of an explicit "track file" to be used for logging, display update, etc.
  3. Includes notions of concurency via explicit annotations for parallelism.
- P3: Extends P2 as follows:
  1. Includes notion of "engageability" (including the computation of velocity vectors given a sequence of positions).
  2. Has notion of "operator interaction" to update system parameters.

The engageability problem is not solved in P2, since it is optional. Solution P2 is included as Section 7 of this report, and is the recommended starting point for someone who wants to study the Haskell prototype. After reading P2, we suggest studying P3 (Appendix A) to see how P2 has been easily extended, and then reading P1 (Appendix B) to see how the two more sophisticated solutions have evolved as natural extensions of the first prototype which was written.

Each Haskell prototype is structured as three layers. First, there are primitive definitions and equations which form a domain specific language for geometric region processing. Then there are a set of definitions for problem specific concepts such as reachability, engageability, the sensor/track file, and dispositions. Finally, there is the "main" function which solves the problem as specified by calling the primitives which have been assembled. In addition, parameters and data are needed for each test of the prototype. This test data is interspersed through the code in P1, but is moved into a separate section in P2 and P3. It is important to distinguish the metric for

source code at each layer, because the primitives and problem specific concepts could be reused as building blocks to solve a variety of problems.

An important attribute of the Haskell prototypes is the extensive documentation which accompanies them. The desired style for a Haskell program is a publication quality technical paper with equations interspersed to formalize the notions which are discussed. Those equations are typically written first in standard mathemtical notation, then translated directly into Haskell. Each line of Haskell code is marked with a ">" sign in the left-most margin, and together they comprise the executable source code for the prototype. The Yale Haskell compiler reads and executes the same file that is processed with LaTeX to produce the documentation. This is called the *literate style* of programming (first proposed by Donald Knuth).

In counting source code lines, we use a simple definition: a line of program text is one with a ">" in the left-most margin. The following are counts of the number of source lines of Haskell code in each version of the Haskell prototype, broken down in terms of functionality:

	P1	P2	P3
Primitives	32	42	89
Problem Specific Concepts	16	21	60
Main function	4	2	3
Total program size (not counting separable input data and parameters)	52	65	152
Parameters/Data	18	20	29

Below is a different breakdown of the program source code, based on whether not the code is "essential" or "for clarity", along with measures of lines of documentation:

	P1	P2	P3
Program text:			
essential	28	36	102
for clarity	24	29	55
Parameters/Data	18	20	24
Documentation	234	465	700
Total Lines Delivered	304	550	881

The source program lines are partitioned into lines which are essential, and lines which could be deleted without affecting the execution of the program. Haskell supports very rapid prototyping, in which the programmer writes a terse program and leaves it to the compiler to figure out type signatures. Haskell also supports a more disciplined style, in which the programmer declares types signatures and synonyms which the compiler uses to check the prototype for internal consistency. The Yale Haskell compiler could have synthesized type signatures equivalent to the lines which were added for clarity. This experiment was performed using the disciplined style, because that style is required by the Intermetrics prototyping process, and because it makes the Haskell programs more reliable, easier to read, and more useful as a design document for later stages in the development process.

As discussed in Section 5, a concurrent experiment was conducted to determine whether a relatively inexperienced programmer could use Haskell to prototype a GEO Server. This programmer's prototype is contained in Appendix C. Statistics on his prototype follow:

Program text	111
Parameters/Data	45
Documentation	112
Total Lines Delivered	268

This program is comparable in functionality to Haskell Prototype P1, and thus the above figures should be compared to those for P1.

As discussed in Section 5, prototypes were also written in Ada 9X and in Relational Lisp. The following are counts of the non-blank non-comment lines of source program text and of comments and documentation that comprise the prototype:

	Ada 9X	Relational Lisp
Program text	765	218
Parameters/Data	35	56
Comments	200	12
Total Lines Delivered	1000	286

These programs are comparable in functionality to Haskell Prototype P2, and thus the above figures should be compared to those for P2.

6.2 Process Metrics

Time spent developing the prototypes was categorized as follows:

- "thinking": Trying to understand the problem and coming up with an *approach* to solving it.
- "solving": Given an approach, working out the details of the mathematics / algorithm / whatever (if required). This is the kind of information that a domain expert would have at his/her fingertips, but the prototypers had to create from scratch.
- "coding": Given the solution spec, writing the code, *including* debug and test.
- "doc": Writing up the documentation to go along with the code – primarily to explain the *solution* rather than the code.
- "review": Independent review.
- "mods": Incorporating changes agreed to during a review.

Total time spent (in hours) on the various prototypes:

	P1	P2	P3	Trainee	Ada 9X	Relational Lisp
thinking	1	1	3	2	4	
solving	0	0	15[1]	1.5	6	
coding	1	1	3	4	12[2]	4[3]
doc[4]	2	2	4	0.5	4	
review	0.75	0.50	–	–	1	
mods	0.50	0.25	–	–	1	
Total	5.25	4.75	25	8	28	4

The superscripts in the table correspond to the following notes:

1. P3 has extended functionality beyond those in any of the other prototypes. In particular, its notion of "engageability" is quite sophisticated (to get a feel for this, the reader is encouraged to look at the printout of the running program in Appendix A). The 15 hours "thinking" in the table reflect the time spent developing the mathematics to solve the physical/geometric constraints. This was all done with no thought given to its expression in Haskell.

2. The Ada 9X solution has not been compiled and tested. The other five prototypes were tested, and execute correctly.

3. In addition to this 4 hours of coding, Bob Balzer spent 4 hours in the kickoff meeting at NSWC. After that meeting, his thinking, solving and coding were done as an inseparable activity. Mark Jones, on the other hand, did not attend the kickoff meeting; his time thinking about and solving all parts of the problem are included in the table above.

6.3 Analysis of Prototyping Process

The metrics presented above allow a variety of ratios to be computed. The most obvious is the relative size of program text and documentation. Since the Haskell programs were the shortest in lines of program text, we use them as the normalization factor, resulting in the following table:

	Haskell	Ada9X	R-Lisp
Program text (normalized)	1.0	9.4	3.2
Documentation (normalized)	1.0	.43	.026

In other words, the Ada prototype was 9.4 times longer than the Haskell prototype in terms of lines of program text, and had 43% of the number of lines of documentation. For the Relational Lisp prototype these figures were 3.2 times and 2.6%, respectively.

Below is a table of three more derived ratios: *delivered lines per hour* (number of delivered lines divided by coding plus documentation time), *lines of program text per hour* (number of lines of program code divided by coding plus documentation time), and *program text per total lines* (ratio of program text to total delivered text):

	P1	P2	P3	Trainee	Ada9X	R-Lisp
Delivered Lines per Hour	101.3	184.2	125.9	59.6	62.5	71.5
Lines of Program Text per Hour	23.3	28.3	25.9	34.7	50.0	68.5
Program Text per Total Lines	23.0	15.5	20.5	58.2	80.0	95.8

Another interesting analysis is to compare the time needed to produce the Haskell prototypes with the time that would normally be needed to produce a production implementation of a GEO Server. While the following is crude, it provides a good indication that the prototyping process is meeting expectations:

- Use the Ada 9X program to estimate that the final product will have approximately 1000 source lines of code.
- Use standard productivity models to estimate that 1000 source lines of code would normally take between 300 person hours and 1600 person hours for top level design, implementation and integration. (For example, typical parameters to the Cocomo model will produce estimates in this range.)
- Then, take the ratio of time to create each of the prototypes to these ranges:

	P1	P1+P2	P1+P2+P3
Percentage of total time to build prototype	0.3%-1.8%	0.6%-3.3%	2.2%-11.7%

The same ratios for the Ada 9X prototype are 1.8%-9.3% (achieving the functionality of P2). Thus, both the Haskell prototyping productivity and the Ada 9X prototyping productivity are consistent with an overall productivity to a high quality product of greater than 500 source lines of code per person month. The main risk with the Haskell prototype is that a problem may arise in converting the problem specific algorithm into an efficient implementation. The main risk for the Ada 9X prototype is that it may make a design assumption that proves to be invalid.

In summary, the metrics indicate that the goals of prototyping are being achieved, but since the experiment is very small and there are no controls for individual productivity factors, we can only say that the results are encouraging rather than definitive.

6.4 Subjective Analysis of GEO Server Prototypes

A variety of design issues are identified in the problem statement which are addressed to varying degrees by the various prototypes. Each solution is a prototype of a GEO Server. Further experimentation will be required to determine the extent to which these prototypes represent models of the GEO Server required by NSWC. This section makes some subjective comments about the relative strengths and weakness of the various prototypes.

Despite the fact that the Ada 9X program has not been compiled and tested, it is at a similar quality level to Haskell Prototype P2. Both prototypes were written by experts who are probably in the highest productivity category.

It is worth noting the high percentage of comments relative to program code in the Haskell prototypes, even in the trainee's solution. The largest amount of time in the creation of Haskell prototypes was spent on documentation, not coding. We view this as an advantage of the literate style of programming, which becomes more like writing a paper than writing a program. The analogy to writing a paper carries over to the issue of reuse – the process of loading existing text into an editor and modifying it to make it as perfectly suited to the new prototype as if it had been written from scratch is used extensively in prototyping.

The Haskell prototypes are very good at providing an understandable and testable specification for mathematical algorithms in the problem domain. The Ada 9X prototype, on the other hand, is strong in providing a candidate set of data representations and partitioning of the problem into concurrent tasks (although these might represent overcommittments to concrete representations).

The Haskell prototypes are very good at providing specifications for correct behavior. The Ada prototype is very good for exploring where to detect and recover from abnormal situations (we did not test Haskell's capabilities in this regard; it may be that if such abnormalities were part of the requirements specification that the Haskell prototype would have handled them perfectly well).

The Haskell prototype is very good at high level exploration of alternative algorithmic partitions. For example, the problem assumes that track data includes the identification of the object whose position is being reported, but the problem statement says that a key issue is identification of friends and foes. We suspect that algorithms for identifying tracks will be much easier to prototype as modifications to the Haskell prototype than as modifications to the Ada prototype; that is an experiment which would be worthwhile to perform.

The Haskell prototype is very good at exposing opportunities for parallelism. The Ada prototype implements a particular partitioning for concurrency.

We assume that either prototype can be interfaced to existing software to test the GEO Server hypothesis, but such interface experiments should be performed since they are important to both AEGIS and ProtoTech. An issue which should be explored is whether to have multiple copies of the GEO Server, and what criteria to use in distributing the copies.

In summary, the prototype GEO Servers which have been created can be used for a broad range of experiments. We need guidance from NSWC as to the relative importance of the experiments that can be performed.

6.5 Extra Credit Options

In Reference 2 a list of 14 "possible enhancements" was given (which we like to think of as "extra credit"). Aside from the specific options implemented in Prototype P3, Mark Jones did not consider these options while developing his prototypes. It is nevertheless interesting to evaluate how well the prototypes might already meet these optional requirements. The list below summarizes this analysis.

a. Scale the problem to 1000s of tracks and 100s of regions of any mix.

   Done at the specification level.. The Haskell prototypes can handle arbitrary numbers of tracks and regions, but we have not specifically addressed processor and memory utilization.

b. Allow for the asynchronous receipt of tracks in bundles of 1 to 50 track packets.

   Not done.

c. Scale the problem to 3000 track reports per second update rate.

   N/A. (We did no performance measurements on any of the prototypes.)

d. Add the ability to add, modify, and delete regions during tactical operations.

   Done. Prototype P3 implements this capability.

e. Adopt a curved earth model.

   Partially done. The geometry module was purposely made "abstract" and can be easily replaced with a different one. In fact Mark Jones wrote such a "curved earth geometry module," but it is not included in this report.

f. Take into account altitude readings.

   Given (e) above, *done.*

g. Allow for regions to be oddly shaped vertically (i.e., at the edges) and have ceilings.

   Given (e) above, *done.*

h. Allow for regions to be oddly shaped horizontally.

   Done. Arbitrary polygonal (including concave) regions, curved (circular) regions, non-contiguous regions, and regions with internal "holes" are easily created. Possible extensions (using the "region authoring system" mentioned below) might include, for example, splines for more complex curves.

i. Allow the orientation of weapons doctrine to be fixed relative to the orientation of the ship or relative to a threat access emanating from a single point on the surface of the earth.

   Not done. However, this would be an easy enhancement, since the weapons doctrine is sufficiently abstract.

j. Extend slaved doctrine to aircraft.

   Done. We believe the abstractions used for slaved ships can also be applied to slaved aircraft.

k. Maintain a doctrine history (i.e., parameters such as when and where a track first entered a doctrine region, length of time in the region, etc.).

Done. Prototype P3 implements this capability.

l. The doctrine problem was purposefully divided into the "region comparison engine" and a heretofore unmentioned "tactical action aid" which determines what should be done with the information based on the type of doctrine, the ID and other information about the contact. One possible enhancement might be to build features of the second part. The two problems are actually very different in character. The region comparison engine's goal is the rapid determination of the presence of tracks in regions. The tactical action aid determines what, if any, significance that event holds. Therefore, the tactical action aid's job is more rule based or AI'ish in nature.

Not done.

m. There is actually a third part to the process and that is the "doctrine region authoring system". This component would work in conjunction with tactical action aid to produce the desired tactical goals.

Partially Done. The domain-specific language developed for constructing regions in Haskell is an excellent "doctrine region authoring system," even for the non-Haskell user. We estimate that a few hours of explanation would be all that is required for proper use. On the other hand, users may demand a graphical user interface.

n. A possible additional goal for the prototype experiment might be to determine how these three pieces fit together optimally.

Not done.

6.6 Analysis of Haskell as Tool for Prototyping

The experience in the formal code walkthroughs was that Haskell prototypes are highly **understandable**. A major benefit was that the discussions focused on whether the Haskell program correctly modeled the problem. No time was spent on irrelevant implementation details. The Haskell style of programming is ideally suited to understanding the algorithms in problem specific rather than computer specific terms.

The **extensibility** of the Haskell programs was demonstrated by the three-step refinement process. The fact that the reuse from one prototype to the next was at a source code level, and involved an examination of every line in the program, is analogous to the process of writing a paper for publication. Some of the most valuable changes are global changes or reorganizations that allow the same ideas to be communicated much more effectively and succinctly. The Haskell compiler is an invaluable assistant in checking that such reorganizations and global changes result in a consistent set of equations and constraints.

Haskell was **appropriate** to the problem in at least three ways. First, mathematical formulas are a good notation for the GEO Server problem, and Haskell has excellent support for equations. Second, Haskell is extensible, so a problem specific notation, which many people would call a

domain specific language, could be defined and the remainder of the problem expressed in terms of that notation. Finally, Haskell did not force premature resolution of design decisions.

The Haskell prototypes were **accurate** – they produced the correct answers on the defined test scenarios. Haskell's type checking was a big help in avoiding coding errors.

The Haskell prototypes were also **compact**. As the analysis above has shown, the Haskell prototypes had between 10% and 30% as many lines of code as the prototypes written in Ada and Lisp, and readability was not sacrificed to gain the conciseness.

Finally, the Haskell prototypes enhanced the use of **formal methods**. Several interesting properties were proved about the Haskell prototypes using simple equational reasoning. A similar proof using the other prototypes would be much more difficult.

Another concern in evaluating a language is the **maturity** of the language definition and of the documentation which describes the language. Haskell is in production use worldwide. It was designed by an international committee, and it has been thoroughly reviewed and polished. There are multiple implementations. The language specification and a tutorial have been published by the Association of Computing Machinery (see references 3, 4 and 5). In summary, Haskell is a production quality language will a substantial and growing following worldwide.

Haskell and Ada address very different kinds of concerns. With Haskell, one focuses on declaring a formal model of the objects and operations in the problem domain. The GEO Server algorithms are algebraic, and can be clearly, easily and succinctly expressed in Haskell. It is significant that Mr. Domanski, Mr. Banowetz and Dr. Brosgol were all surprised and suspicious when we told them that Haskell prototype P1 (see appendix B) is a complete tested executable program. We provided them with a copy of P1 without explaining that it was a program, and based on preconceptions from their past experience, they had studied P1 under the assumption that it was a mixture of requirements specification and top level design. They were convinced it was incomplete because it did not address issues such as data structure design and execution order.

Haskell is a much better tool than Ada for comparing different application specific algorithms, because the algorithms can be expressed mathematically without concern for choice of data representation, memory management, or synchronization of access to data. Specifying and testing an algorithm to predict when tracks will intercept regions is an ideal use of Haskell. Once an algorithm is selected, it can then be implemented in Ada. It is also possible that when implementing a chosen algorithm in Ada, an issue will arise as to modifications to the algorithm which could yield a large factor in execution efficiency. In such a case, it will make sense to try the revised algorithm in Haskell before investing in the engineering to do the production quality implementation in Ada.

The issue of concurrency in the Haskell prototype is worthy of special discussion. Algorithms expressed in Haskell provide many potentials for parallel execution, and that potential can be specified explicitly with an annotation. On the other hand, the design of efficient data representations and control flow to take advantage of the potential parallelism involves substantial engineering. The Proteus project involving Univ. of North Carolina, Duke and Kestrel has demonstrated a variety of transformation techniques for this purpose, and indeed the Haskell annotations for parallelism have a strong resemblance to those used in Proteus (see Reference 5). It will also

be desirable to prototype the mapping of Ada tasks and access to protect objects on the target hardware, in order to optimize the implementation and to obtain performance metrics to guide the transformation of the architecture independent algorithms onto the actual hardware.

6.7 Analysis of Yale Haskell Implementation

The Yale Haskell implementation is a production quality tool available by anonymous ftp from Yale. No bugs were encountered in programming any of the prototypes. It is an excellent interactive tool for prototyping, so in comparing productivity data with standard models the tool support should be rated as excellent.

We did not measure the performance of the models which were created. Haskell is most useful for defining the functionality of algorithms. A research area being addressed in ProtoTech is how to relate the performance of a model written in Haskell to the performance of a production implementation written in a language such as Ada.

6.8 Comments on This Experiment

The discipline used in performing the experiment sets an example for future larger experiments. The experiment problem statement was excellent. It provided a concise definition of a problem with wide ramifications which could be absorbed quickly. It was also a problem that could be explored at many different levels of sophistication.

This experiment was constrained to be completed in two weeks, and with limited resources. Therefore, there was no way to compensate for differences in the productivity of individual programmers. It is safe to assume that the programmers were all at the highest productivity levels for their profession, and that none of them had any prior experience with the problem domain. Also, since the time spent on each task was so small, time measurements are subject to error; all numbers should be taken as indicative rather than precise.

7 Baseline Haskell Solution (Prototype 2)

This section presents our "Prototype 2," a Haskell solution to the problem outlined in Reference 1 and modified by Reference 2. This prototype meets all required specifications.

Taking advantage of Haskell's support for "literate programming," the LaTeX source code for this section also serves as a directly executable version of the prototype. Each line of Haskell code is preceded by a right angle-bracket (>) in the left-most column – all other lines are treated as comments.

To aid understanding by those not familiar with Haskell, we have also included an explanation of Haskell language features one-by-one as they are arise in the program. These comments are indented and printed in smaller font, and would not normally be included if Haskell were familier to the majority of readers.

The remainder of this section is organized into four subsections:

1. Primitive definitions and equations which establish basic mathematical relationships for geometrical analysis. These primitives had to be written for this problem, but in most situations they would already have existed in a library. We refer to these primitives as the *domain specific language for geometric regions*. Examples of concepts defined include point, origin, regions, and various operations on regions. By reading this section, you can see how the mapping between Haskell and the problem domain is created. This can also be viewed as the foundation for a "doctrine region authoring system" as mentioned in Reference 2.

2. Definition of issues related to object tracking, such as sensor/track file representations, dispositions, and situation assessment. These definitions and equations build upon the primitives defined in the first section, and create a richer mapping between Haskell and the problem domain.

3. A function which solves the problem defined in references 1 and 2. This "program" is only three lines long, because it is implemented in terms of the functions which have been defined previously.

4. Sample data and execution results. This section provides the test data specified in the problem statement, and demonstrates that the correct results are printed.

This presentation of the program is "bottom-up," in that it starts with primitive definitions, and gradually builds up to the problem solution. However, order is not important in a Haskell program, and we could have chosen to present the program function first in the source text. We have chosen this presentation order to facilitate readability and understandability, and to highlight the generality and reusability of the primitives from which the program is built.

7.1 Defining a Domain Specific Language for Geometric Regions

In this section a set of datatypes and functions are defined for abstractly specifying geometric regions.

7.1.1 Points

A point is used to identify a location in space and will be represented by a vector of floating point coordinates.

```
> data Vector a = Pt a a   deriving (Eq, Text)
> type Point    = Vector Float
```

> `data` and `type` are keywords. The first line above defines a new polymorphic datatype called `Vector`, and can be read as: "For any type `a`, an element of the datatype `Vector a` consists of values of the form `Pt x y`, where `x` and `y` are of type `a`." (The `deriving` addendum ensures that equality (through the class `Eq`) and printing (through the class `Text`) are defined on `Vector`.) The second line above is a *type synonym*, which simply gives a name to an existing type; in this case `Vector Float` is being given the name `Point`.

In particular, the origin is described by the point with zero coordinates:

```
> origin :: Point
> origin  = Pt 0 0
```

> The first line above is a *type signature*, which simply declares the type of an identifier. The second line binds `origin` to a specific value of type `Point`.

These definitions are easily modified to extend the program to three-dimensional space. For example, some form of spherical coordinates (e.g. longitude, latitude and altitude) would obviously be more appropriate to model regions around the (curved) surface of the earth.

To simplify our work, we define the usual operations of vector addition and subtraction:

```
> instance Num a => Num (Vector a) where
>    Pt x y + Pt u v  =  Pt (x+u) (y+v)
>    negate (Pt x y)  =  Pt (-x)  (-y)
>    Pt x y - Pt u v  =  Pt (x-u) (y-v)
```

> `Num` is a *type class*. Types that are instances of `Num` can be given method definitions for the operations in the class. In this case `Vector a` is being given definitions for `(+)`, `(-)`, and `negate`, so that we can use these operations on `Vectors` just as we do with numbers. Also note the use of *pattern matching* on the left hand sides of the equations.

It will also be useful to calculate the length of a vector (i.e. distance from the origin):

```
> sqrDist             :: Num a => Vector a -> a
> sqrDist (Pt x y)     = x*x + y*y
```

The second line above defines a new function called `sqrDist`. From this definition its type can be inferred by the compiler, but we provide the type signature on the first line to aid readability. It should be read as: "`sqrDist` is a function that, for any type `a` in the class `Num`, maps elements of `Vector a` into elements of type `a`."

For convenience, we have chosen to use a `sqrDist` function that returns the square of the distance of its argument point from the origin. This avoids unnecessary calculations and loss of accuracy that would be caused by taking square roots.

In fact, the definitions given above are more general than necessary for the current problem. However, the extra flexibility that this gives may be useful in other situations or in further extensions to the program at a later date.

7.1.2 Regions

A region is a set of points in space. However, the only thing we need to know about a region is whether or not it contains a given point. This leads to a simple and elegant implementation, modelling a region as a function that takes a point as its argument and returns a boolean value to indicate if the given point is inside the region that the function represents. In Haskell notation, this can be described as follows:

```
> type Region  =  Point -> Bool
```

`T1 -> T2` is the type of functions that map elements of `T1` into elements of `T2`.

We will also define a simple function to allow us to determine whether a point is in a region.

```
> inRegion       :: Point -> Region -> Bool
> p `inRegion` r = r p
```

The left-hand-side could have been written `inRegion p r` but we have chosen (stylistically) to use `inRegion` as an infix operator to aid readability, which is accomplished in Haskell by enclosing the identifier in backquotes.

In fact, for the current representation of regions, we can obtain the same result just by applying a region to a point. However, the use of an operator like `inRegion` gives us more of the feel of an abstract data type and will allow us to modify the representation of regions at a later point, for example to deal with engageability, without requiring changes to other parts of the program using regions.

7.1.3 Some Primitive Regions

We start by defining a collection of primitive regions from which the full regions that we are interested in can be constructed. In fact, for the purposes of this project, only two different kinds of primitive regions are required; circles and halfplanes. However, we can easily extend this framework to build a larger library to accomodate other kinds of region as required. Thus this has the feel of a domain specific language, or "authoring tool", for region construction.

Circlular Regions The point `p` lies inside the circle of radius `r` if and only if its distance from the origin is less than `r`. This translates directly into Haskell notation (using the square of the distance and the square of the radius instead for convenience):

```
> type Radius = Float

> circle  :: Radius -> Region
> circle r = \p -> sqrDist p < r*r
```

> An expression of the form `\x -> exp` is a *lambda expression*, i.e. a (nameless) *function*. The second line is equivalent to: `circle r p = sqrDist p < r*r`, but we prefer the above version because it emphasizes the way in which `circle` will typically be used: it will be applied to one argument to yield a *region* (which just happens to be a function).

Half Plane Regions Half planes are another useful form of region, representing the set of points that are on one particular side of a given line. We will define the region `halfPlane a b` to be the half plane of points to the left of the line joining the point `a` to the point `b`.

With some simple geometry using the vector cross product, the sign of the component of $(\mathbf{a} - \mathbf{p}) \times (\mathbf{b} - \mathbf{a})$ in the perpendicular plane can be used to determine whether a particular point $\mathbf{p}$ lies in the specified half plane. Once again, this translates directly into Haskell notation:

```
> halfPlane     :: Point -> Point -> Region
> halfPlane a b = \p -> zcross (a - p) (b - a) > 0
>  where zcross (Pt x y) (Pt u v) = x*v - y*u
```

7.1.4 General Operations on Regions

This section defines some useful general purpose operators that can be used to obtain new regions of various kinds. There are many other examples of useful general operations in addition to those described in the sections below that can also be described elegantly and concisely in this framework, for example to modify a region by arbitrary linear transformations such as rotations and reflections on the coordinate space. However, none of these is required for the problems that we deal with in the current prototype.

Complements It is sometimes useful to be able to specify a region as a complement; i.e. as the set of points that are `outside` some other given region. This is captured by the following definition:

```
> outside  :: Region -> Region
> outside r = \p -> not (r p)
```

Intersections and Unions It is often useful to be able to describe regions as the intersection or union of some simpler regions. These operations can be described directly using the `(/\)` and `(\/)` functions, respectively, as defined below:

```
> (/\) , (\/) :: Region -> Region -> Region
> r1 /\ r2     = \p -> (r1 p && r2 p)
> r1 \/ r2     = \p -> (r1 p || r2 p)
```

> `(&&)` and `(||)` are Haskell's boolean "and" and "or" operations, respectively.

Using the standard Haskell function `foldr1`, we can extend the `(/\)` and `(\/)` operators to calculate the intersection of arbitrary (non-empty) lists of regions:

```
> intersect, union :: [Region] -> Region
> intersect         = foldr1 (/\)
> union             = foldr1 (\/)
```

> `[T]` is the type of *lists* whose elements are all of type `T`. The use of `foldr1` is a typical way to extend an infix operation to work on a list.

For example, `intersect [r1,r2,r2] = r1 /\ r2 /\ r3`.

> If `x`, `y`, and `z` have type `T`, then the list `[x,y,z]` has type `[T]`.

Translations The regions that we have defined so far are all fixed at particular locations in space. However, it is easy to describe the translation of regions by specifying the position of objects relative to some new origin. We represent this in Haskell using the operator `at` defined by:

```
> at         :: Region -> Point -> Region
> r `at` p0   = \p -> r (p - p0)
```

For example, all of the regions produced by the `circle` primitive described in Section 7.1.3 are centered on the origin, but we can use the `at` operator to obtain a circle centered at some other point. For example, `circle r `at` c` is a circular region with radius `r` centered at `c`.

The `at` operator satisfies a number of useful and intuitive laws, including, for example:

```
                        r `at` origin =  r
(r `at` p1) `at` p2  =  r `at` (p1+p2) =  (r `at` p2) `at` p1
```

The first of these shows that setting a new origin that is the same as the current origin leaves the region unchanged. The second describes the way that repeated translations are combined. The ability to state and prove results like this is useful because it helps to ensure that our definitions are reasonable and that they behave in the ways we expect. In addition, laws like these can also be very useful in program development and optimization.

7.1.5 Derived Regions

The tools introduced in the previous sections can be used to define a wide range of useful regions. We illustrate this with some examples that will be useful in the following work.

Annulus Regions An annulus, containing all those points whose distance from the origin is between radius values `r1` and `r2` can be described as the intersection of the outside of a circle of radius `r1` with a circle of radius `r2`:

```
> annulus       :: Radius -> Radius -> Region
> annulus r1 r2 = outside (circle r1) /\ circle r2
```

> Function application always has higher precedence than infix application. Thus the rhs of the definition above parses as: `(outside (circle r1)) /\ (circle r2)`.

Convex Polygons Many useful regions can be described using convex polygons. The following definition allows us to construct arbitrary convex polygons by listing the vertices in counter-clockwise order:

```
> convexPoly        :: [Point] -> Region
> convexPoly (v:vs) = intersect (zipWith halfPlane ([v]++vs) (vs++[v]))
```

> `(++)` is the list append (i.e. concatenation) function.

The key idea here is that, if the vertices are given by points `[v0, v1, ..., vn]`, then the interior of the polygon is just the intersection of the halfplanes joining adjacent points (including a line from `vn` to `v0`). In the definition above, `zipWith halfPlane` is used to produce this list of half-planes, and then `intersect` is used to produce their intersection. The `zipWith` function is a standard function included as part of the Haskell standard prelude.

Segment and Pie Regions It will also be useful to be able to deal with segment-shaped regions of the plane; i.e. regions obtained by sweeping a line emanating from the origin through a given range of angles (like a beam from a lighthouse). The easiest way to deal with this is to think of the segment as the intersection of two half planes:

```
> type Angle = Float

> segment     :: Angle -> Angle -> Region
> segment l u
>   = halfPlane (radial u) origin /\ halfPlane origin (radial l)
>     where radial a = Pt (cos theta) (sin theta)
>                      where theta = a * (pi / 180)
```

A `where` clause is simply a way to add local definitions.

The auxiliary function `radial` defined here is used to calculate a point on the unit circle corresponding to a given angle `a`. Note that the angles `l` and `u`, in degrees, are multiplied by `pi/180` to convert them to radian measure as required by the `sin` and `cos` functions.

Using `segment` regions, we can define another function for constructing pie-shaped portions of a circle:

```
> pie       :: Radius -> Angle -> Angle -> Region
> pie r l u = segment l u /\ circle r
```

This will be used, for example, to define weapon doctrine regions.

7.2 Object Tracking

In this section the object representations and tracking algorithms are fully defined.

7.2.1 Sensor/Track File Data

We assume that sensor information is supplied as a stream of "packets" (presumably from a sensory subsystem). Individual packets will be represented by a triple containing:

- The position of ownship,
- The positions of an arbitrary collection of slaved objects, and
- The positions of an arbitrary collection of objects to be tracked.

In Haskell notation, each packet can be represented as value of type `trackingData` where:

```
> type TrackingData = (Point, [Point], [Point])
```

7.2.2 Dispositions

A *disposition* provides a snapshot of the current positions of the regions and objects of interest at a given point in time. It is represented by a pair containing a list of named regions, and a list of the objects that are being tracked:

```
> type Disposition = ([NamedRegion], [NamedObject])
> type NamedObject = (String,Point)
> type NamedRegion = (String,Region)
```

The representation for a named object includes a string (used to identify the object in diagnostic messages), and a vector giving its current position.

A disposition can be obtained from the data in an input packet specifying the position of each object:

```
> makeDisposition :: TrackingData -> Disposition
> makeDisposition (pown, pslaves, pobjs)
>   = ([("weapon doctrine",     weaponDoctrine `at` pown),
>       ("engageability zone", engageability `at` pown)]
>      ++ zipWith (\(name,r) p -> (name, r `at` p)) slaves pslaves
>      ++ tightZones,
>      zip objectIds pobjs)
```

The list of named regions is made up from two regions centered on ownship, together with the slave doctrine regions and the tight zones. (For simplicity, we have assumed a single ownship, although it would be easy to extend the program to use an arbitrary list of ownships.) Note that there are several free variables in this definition that can be set within a specific application as appropriate. This includes `weaponDoctrine`, `engageability`, `slaves`, `tightZones` and `objectsIDs`. Suitable values for these variables corresponding to the sample data provided in the original problem description document are presented in Section 7.4.

7.2.3 Situation Assessment

Having constructed a suitable disposition, the next task is to assess the situation that it describes, producing reports to indicate when an object is in a given region. This is captured by the `assess` function which produces a single string containing all the reports for the current disposition. We use the `par` function to indicate that these individual reports can be generated concurrently in a parallel implementation (Reference 5 contains details on these issues of parallelism in Haskell):

```
> assess :: Disposition -> String
> assess (regs, objs)
>  = unlines (par [ r | obj <- objs, r <- objectReport obj regs ])
```

> An expression of the form `[exp | x <- xs, y <- ys]` is called a *list comprehension*, and is a way to construct a list that resembles set construction in mathematics. It can be read: "The list of all `exp`'s such that `x` is taken from the list `xs` and `y` from the list `ys`."

7.2.4 Report Generation

The `objectReport` function is used to produce the list of reports for a specific object relative to the list of regions of interest. The report begins with a display of the current position of the object, followed by reports for individual regions. As before, the `par` function is used to indicate that these reports can be produced in parallel.

```
> objectReport :: NamedObject -> [NamedRegion] -> [String]
> objectReport (what,p) rs
>  = [ what ++ ": " ++ showP p ] ++
>    par [ rep | reg <- rs, rep <- regionReport p reg ]
>    where showP (Pt x y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Finally, we have the `regionReport` function that produces the reports for a single object at point `p` and a specific named region.

```
> regionReport                :: Point -> NamedRegion -> [String]
> regionReport p (region,r) = [ " -- In " ++ region | p `inRegion` r ]
```

Note that, in the first case, if the object is currently outside the region concerned, then no reports are generated (or more precisely, the list of reports is empty).

7.3 The Complete Program

The complete program is described by a function that takes an input stream of tracking data `td` and turns each one into a disposition which can then be assessed to produce the output reports that are required:

```
> program :: [TrackingData] -> [String]
> program  = map (assess . makeDisposition)
```

> `.` is Haskell's infix function composition operator. The `map` function is used to apply a function to a list of values, producing the list of results. e.g. `map f [x1, ..., xn] = [f x1, ..., f xn]`.

7.4 Sample Data and Execution

This section provides the sample data corresponding to the information given in the original problem description, and subsequent extensions.

We start with the descriptions of the weapon doctrine, engageability, slave doctrine and tight zone regions, and the list of objects that are being tracked:

```
> weaponDoctrine = pie 59 (-120) 120
> engageability  = annulus 22 44

> slaves     = [("carrier slave doctrine", circle 40)]

> tightZones = [("tight zone",
>                convexPoly [Pt 0 5, Pt 118 32, Pt 118 62, Pt 0 25] \/
>                convexPoly [Pt 118 32, Pt 259 5, Pt 259 25, Pt 118 62])]

> objectIds  = ["commercial aircraft", "hostile craft"]
```

The positions of the various objects at different times are given by the following table of packets, with coordinates taken directly from the original problem description:

```
> trackingData :: [TrackingData]
> trackingData  = [(Pt 113 64, [Pt 180 140], [Pt 38 25,  Pt 258 183]),
>                  (Pt 123 64, [Pt 180 130], [Pt 58 30,  Pt 239 164]),
>                  (Pt 133 64, [Pt 180 119], [Pt 100 43, Pt 210 136]),
>                  (Pt 163 64, [Pt 180 81],  [Pt 159 36, Pt 148 73]),
>                  (Pt 192 64, [Pt 180 60],  [Pt 198 27, Pt 110 37])]
```

For convenience, we assume a time interval of 20 between each of these packets:

```
> timeInterval  :: Float
> timeInterval   = 20
```

To demonstrate how this data can be used with our prototype implementation, we define the following function to print the list of reports, including an indication of the elapsed time at the beginning of each report:

```
> demo :: Dialogue
> demo  = display (zipWith (\t r -> "Time " ++ show t ++ ":\n" ++ r ++"\n")
>                          (iterate (timeInterval+) 0)
>                          (program trackingData))
>  where display = foldr (\rep -> appendChan stdout rep  exit .
>                                 appendFile "trackFile" rep exit) done
```

`(timeInterval+)` is an example of the partial application of an infix operator; it is equivalent to `\x-> timeInterval+x`.

The `display` function defined here is used to interleave writing of the reports to the standard output channel and the `trackFile` file.

Running this program produces the following output:

```
Time 0.0:
commercial aircraft: (38.0,25.0)
 -- In tight zone
hostile craft: (258.0,183.0)

Time 20.0:
commercial aircraft: (58.0,30.0)
 -- In tight zone
hostile craft: (239.0,164.0)

Time 40.0:
commercial aircraft: (100.0,43.0)
 -- In engageability zone
 -- In tight zone
hostile craft: (210.0,136.0)
 -- In carrier slave doctrine

Time 60.0:
commercial aircraft: (159.0,36.0)
 -- In engageability zone
 -- In tight zone
hostile craft: (148.0,73.0)
 -- In carrier slave doctrine

Time 80.0:
commercial aircraft: (198.0,27.0)
 -- In engageability zone
 -- In carrier slave doctrine
 -- In tight zone
hostile craft: (110.0,37.0)
 -- In tight zone
```

8 Conclusions

A disciplined prototyping experiment has been conducted to explore algorithms for an AEGIS GEO Server. Product and process metrics have been evaluated relative to both NSWC and ProtoTech goals. Subjective information collected during the experiment has also been documented.

We have shown that the mapping between the Haskell language and this problem is extremely good. The mathematics of the problem translates into functional expressions, parallelism falls out naturally, and communication between subsystems maps directly into streams (i.e. lists) of data. There was absolutely nothing in the problem specification that was difficult to model because of a mismatch between language and domain.

We have also shown that Haskell can be used to address very different and complementary issues to those addressed by prototypes written in Ada 9X. With Ada, one can focus on software engineering issues associated with the efficient mapping of algorithms onto physical hardware. The object oriented capabilities of Ada 9X were particularily valuable in developing a solution for this problem.

The results suggest many additional experiments which should be performed. We look forward to working with NSWC to define additional experiments which build upon the foundation which has been created. This foundation is in two areas. The first consists of the prototypes themselves. The prototypes which have been written are flexible tools which can be enhanced and used to address a variety of issues. The second area is the methodology for performing experiments in a short period time with rigorous record keeping and careful documentation of experimental results. An investment has been made in learning to work together that can pay large dividends in the future.

A Enhanced Functionality: Prototype 3

This Appendix presents our "Prototype 3," an enhanced version of Prototype 2 that, in addition to satisfying all required specifications, includes notions of "engageability" and "operator interaction" (which were labeled "optional" in the email correspondence, Reference 2).

This appendix A, which is a complete executable Haskell program, was written by Mark Jones of Yale University, and is covered by the copyright notice on this report.

A.1 Defining a Domain Specific Language for Geometric Regions

In this section a set of datatypes and functions are defined for abstractly specifying geometric regions.

A.1.1 Points

A point is used to identify a location in space and will be represented by a vector of floating point coordinates. The same representation will be used for directions/velocities.

```
> data Vector a = Pt a a   deriving (Eq, Text)
> type Point    = Vector Float
> type Velocity = Vector Float
```

In particular, the origin is described by the point with zero coordinates:

```
> origin :: Point
> origin  = Pt 0 0
```

These definitions are easily modified to extend the program to three-dimensional space. For example, some form of spherical coordinates (e.g. longitude, latitude and altitude) would obviously be more appropriate to model regions around the (curved) surface of the earth.

To simplify our work, we define the usual operations of vector addition and subtraction:

```
> instance Num a => Num (Vector a) where
>    Pt x y + Pt u v  =  Pt (x+u) (y+v)
>    negate (Pt x y)  =  Pt (-x)  (-y)
>    Pt x y - Pt u v  =  Pt (x-u) (y-v)
```

Other useful operations on vectors include multiplying a vector by a given scalar, taking the dot product of a pair of vectors, and calculating the length of a vector (i.e. distance from the origin):

```
> scale              :: Num a => a -> Vector a -> Vector a
> scale k (Pt x y)    = Pt (k*x) (k*y)

> dot                :: Num a => Vector a -> Vector a -> a
> Pt x y `dot` Pt u v = x*u + y*v

> sqrDist            :: Num a => Vector a -> a
> sqrDist (Pt x y)    = x*x + y*y
```

For convenience, we have chosen to use a `sqrDist` function that returns the square of the distance of its argument point from the origin. This avoids unnecessary calculations and loss of accuracy that would be caused by taking square roots.

In fact, the definitions given above are more general than necessary for the current problem. However, the extra flexibility that this gives may be useful in other situations or in further extensions to the program at a later date.

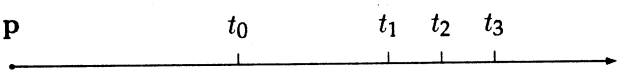
A.1.2 Regions

A region is a set of points in space. In the original problem description, the only use for a region was to determine when it contained specific points. This leads to a simple and elegant implementation, modeling a region as a function that takes a point as its argument and returns a boolean value to indicate if the given point is inside the region that the function represents. This approach was used in our prototype implementation of the problem presented in the original problem description document.

After the original problem description was circulated, it was decided to extend the prototype to allow prediction of the times when a given object might be expected to enter or leave a region, based on its current position and velocity.

This can be dealt with by generalizing the simple notion of regions used in our original prototype. To understand how this works, suppose that the position of an object is given by the line $\mathbf{p} + t\mathbf{v}$, where $\mathbf{p}$ is the position at time $t = 0$ and $\mathbf{v}$ is the velocity vector. We can describe the progress of the object through a region by a *trail*, containing a boolean to indicate whether the object is initially inside the region and a list of times at which it subsequently enters or leaves the region.

As a simple illustration, an object that starts outside the region at point $\mathbf{p}$, enters, leaves, re-enters and finally leaves again at times t_0, t_1, t_2 and t_3, respectively might be pictured as:



and represented by the trail `(False,[t0,t1,t2,t3])`. Note that we are only interested in predicting where an object will go, not where it has come from; i.e. we restrict our attention to trails in which all time values are positive.

In Haskell notation, this representation for regions is described as follows:

```
> type Region  =  Point -> Velocity -> Trail
> type Trail   =  (Bool, [Time])
> type Time    =  Float
```

We will also define a simple function to allow us to determine whether a point is in a region, discarding the rest of the trail (lazy evaluation ensures that we don't waste any time trying to calculate these extra values):

```
> inRegion       :: Point -> Region -> Bool
> p `inRegion` r = b where (b,ts) = r p origin
```

A.1.3 Some Primitive Regions

We start by defining a collection of primitive regions from which the full regions that we are interested in can be constructed. In fact, for the purposes of this project, only two different kinds

of primitive regions are required; circles and halfplanes. However, we can easily extend this framework to build a larger library of regions that is easily extended to accommodate other kinds of region as required. Thus this has the feel of a domain specific language, or "authoring tool", for region construction.

Circular Regions In vector notation, the set of points inside a circle centered on the origin with radius r is given by the equation $|\mathbf{x}| < r$. To find the points of intersection of a line of the form $\mathbf{p} + t\mathbf{v}$ with this circle, we need to find values for t such that:

$$r^2 > |\mathbf{p} + t\mathbf{v}|^2 = |\mathbf{p}|^2 + 2t\mathbf{p} \cdot \mathbf{v} + t^2 |\mathbf{v}|^2$$

In other words, the only possible values of t are those between the roots of the quadratic:

$$t^2 |\mathbf{v}|^2 + 2t\mathbf{p} \cdot \mathbf{v} + (|\mathbf{p}|^2 - r^2) = 0$$

By standard results, the roots of the quadratic $ax^2 + bx + c = 0$ are given by the formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

It follows that, for $a > 0$, the list of (real) roots of this quadratic can be obtained directly (in ascending order) using the Haskell definition:

```
> quadRoots                        :: Float -> Float -> Float -> [Float]
> quadRoots a b c | discr >  0  = [(-b - s) / twoa, (-b + s) / twoa]
>                 | discr == 0  = [-b / twoa]
>                 | discr <  0  = []
>                  where discr = b*b - 4*a*c
>                        s     = sqrt discr
>                        twoa  = 2*a
```

To produce the trail for a given point we use `quadRoots` to solve the quadratic above for t. If there are no real roots, then the line $\mathbf{p} + t\mathbf{v}$ does not meet the circle. Similarly, if there is only one root, then the line is a tangent to the circle. In the remaining case where the quadratic has two distinct roots t_1 and t_2, then the line must pass through the circle, and we can determine whether point $\mathbf{p}$ lies inside the circle by testing whether $t_1 < 0 < t_2$. Since we are only interested in positive values of t, we obtain the following Haskell definition:

```
> type Radius = Float

> circle                    :: Radius -> Region
> circle r p v | a == 0     = (c<0, [])
>              | otherwise = between True (quadRoots a b c)
>               where a = sqrDist v
>                     b = 2 * v `dot` p
>                     c = sqrDist p - r*r
```

The definition of `circle` uses the auxiliary function `between` defined below to produce an appropriate trail from the list of roots of a quadratic. The argument `s` specifies whether the portion of the line between the two roots is to be considered as being inside the region.

```
> between                          :: Bool -> [Time] -> Trail
> between s [t1,t2] | t1 > 0       = (not s, [t1, t2])
>                   | t2 > 0       = (s,     [t2])
>                   | otherwise    = (not s, [])
> between s other                  = (not s, [])
```

Half Plane Regions Half planes are another useful form of region, representing the set of points that are on one particular side of a given line. We will define the region `halfPlane a b` to be the half plane of points to the left of the line joining the point `a` to the point `b`.

With some simple geometry using the vector cross product, the sign of the component of $(\mathbf{a} - \mathbf{p}) \times (\mathbf{b} - \mathbf{a})$ in the perpendicular plane can be used to determine whether a particular point $\mathbf{p}$ lies in the specified half plane.

For the intersection of a line $\mathbf{p} + t\mathbf{v}$ with the line joining two points $\mathbf{a}$ and $\mathbf{b}$, we need to find values of μ and t such that:

$$\mathbf{p} + t\mathbf{v} = \mathbf{a} + \mu(\mathbf{b} - \mathbf{a}).$$

Taking the vector product with $\mathbf{b} - \mathbf{a}$, this gives:

$$t\mathbf{v} \times (\mathbf{b} - \mathbf{a}) = (\mathbf{a} - \mathbf{p}) \times (\mathbf{b} - \mathbf{a})$$

If $\mathbf{v}$ and $\mathbf{b} - \mathbf{a}$ are linearly dependent, then the two lines are parallel, and the cross product on the left is zero. Otherwise, taking components in the plane perpendicular to both $\mathbf{v}$ and $\mathbf{b} - \mathbf{a}$, we can calculate the required solution (ignoring values $t < 0$):

```
> halfPlane                        :: Point -> Point -> Region
> halfPlane a b p v | par == 0     = (pos>0, [])
>                   | otherwise    = (pos>0, filter (>0) [ pos/par ])
>                    where par = zcross v (b - a)
>                          pos = zcross (a - p) (b - a)
```

The `zcross` function uses the standard definition to calculate the component of the vector product in the direction perpendicular to the two dimensional plane that we are working in:

```
> zcross                    :: Num a => Vector a -> Vector a -> a
> zcross (Pt x y) (Pt u v) = x*v - y*u
```

A.1.4 General Operations on Regions

This section defines some useful general purpose operators that can be used to obtain new regions of various kinds. There are many other examples of useful general operations in addition to

those described in the sections below that can also be described elegantly and concisely in this framework, for example to to modify a region by arbitrary linear transformations such as rotations and reflections on the coordinate space. However, none of these is required for the problems that we deal with in the current prototype.

Complements It is sometimes useful to be able to specify a region as a complement; i.e. as the set of points that are `outside` some other given region. In this case, the list of times in any given trail is the same for both the original region and its outside; only the boolean flag indicating when the starting point is inside the region needs to be changed.

```
> outside :: Region -> Region
> outside r = \p v -> let (x,ts) = r p v  in  (not x, ts)
```

Intersections and Unions It is often useful to be able to describe regions as the intersection or union of some simpler regions. These operations can be described directly using the `(/\)` and `(\/)` functions, respectively, as defined below. (These symbols were chosen because of their similarity to the ∩ and ∪ symbols used in typeset mathematics to denote intersection and union, respectively.) Both intersection and union will be defined in terms of a `trailMerge` function that combines a pair of trails, merging the events in chronological order. A function parameter is used to determine where the object enters the appropriate combined region; the boolean 'and' operator, `(&&)` is used for the intersection, while boolean 'or' `(||)` is used for the union:

```
> (/\) , (\/) :: Region -> Region -> Region
> r1 /\ r2     = \p v -> trailMerge (&&) (r1 p v) (r2 p v)
> r1 \/ r2     = \p v -> trailMerge (||) (r1 p v) (r2 p v)
```

The definition of `trailMerge` is as follows, using three auxiliary functions described below:

```
> trailMerge :: (Bool -> Bool -> Bool) -> Trail -> Trail -> Trail
> trailMerge f tr1 tr2 = (x0, changes x0 (tmerge f x1 x2 ts1 ts2))
>  where x0       = f x1 x2
>        (x1,ts1) = trailExpand tr1
>        (x2,ts2) = trailExpand tr2
```

The `trailExpand` function is used to convert a trail into a form with a boolean associated with each time to indicate whether the object is entering or leaving the region at that time:

```
> trailExpand       :: Trail -> (Bool, [(Time, Bool)])
> trailExpand (x,ts) = (b, zip ts bs)
>  where (b:bs) = iterate not x
```

Next, `tmerge` is used to combine these modified trails, taking account of the times for each event, and using earlier values `x0` and `y0` when and object enters or leaves one region without a transition in the other:

```
> tmerge f x0 y0 [] rs = [ (f x0 y1, r) | (r,y1) <- rs ]
> tmerge f x0 y0 ls [] = [ (f x1 y0, l) | (l,x1) <- ls ]
> tmerge f x0 y0 ls@((l,x1):lxs) rs@((r,y1):rys)
>    | l <  r  = (f x1 y0, l) : tmerge f x1 y0 lxs rs
>    | l == r  = (f x1 y1, l) : tmerge f x1 y1 lxs rys
>    | l >  r  = (f x0 y1, r) : tmerge f x0 y1 ls  rys
```

Finally, the `changes` function is used to find the list of times at which the output for the resulting merged trail actually changes:

```
> changes            :: Bool -> [(Bool,Time)] -> [Time]
> changes s0 []       = []
> changes s0 ((s1,t):sts)
>        | s0==s1     = changes s0 sts
>        | otherwise = t : changes s1 sts
```

Using the standard Haskell function `foldr1`, we can extend the `(/\)` and `(\/)` operators to calculate the intersection of arbitrary (non-empty) lists of regions:

```
> intersect, union :: [Region] -> Region
> intersect         = foldr1 (/\)
> union             = foldr1 (\/)
```

For example, `intersect [r1,r2,r2] = r1 /\ r2 /\ r3`.

Translations The regions that we have defined so far are all fixed at particular locations in space. However, it is easy to describe the translation of regions by specifying the position of objects relative to some new origin. We represent this in Haskell using the operator `at` defined by:

```
> at        :: Region -> Point -> Region
> r `at` p0  = \p v -> r (p - p0) v
```

For example, all of the regions produced by the `circle` primitive described in Section A.1.3 are centered on the origin, but we can use the `at` operator to obtain a circle centered at some other point. For example, `circle r `at` c` is a circular region with radius `r` centered at `c`.

The `at` operator satisfies a number of useful and intuitive laws, including, for example:

```
r `at` origin =  r

(r `at` p1) `at` p2  =  r `at` (p1+p2) =  (r `at` p2) `at` p1
```

The first of these shows that setting a new origin that is the same as the current origin leaves the region unchanged. The second describes the way that repeated translations are combined. The ability to state and prove results like this is useful because it helps to ensure that our definitions

are reasonable and that they behave in the ways we expect. In addition, laws like these can also be very useful in program development and optimization.

The `at` function allows us to define translations in space, but we can also define translations in time, for example, to describe a region that is moving through space at some constant speed. This is accomplished using the `movingAt` operator defined below, replacing the velocity `v` of the specified object with its velocity relative to the moving region:

```
> movingAt          :: Region -> Velocity -> Region
> r `movingAt` v0   = \p v -> r p (v - v0)
```

As with `at`, there are a number of useful laws that the `movingAt` operator satisfies:

```
                        r `movingAt` origin = r

(r `movingAt` v1) `movingAt` v2
                            =  r `movingAt` (v1+v2)
                            =  (r `movingAt` v2) `movingAt` v1
```

In addition, we have the following law demonstrating the independence of translations in space using `at` with those in time using `movingAt`:

```
(r `movingAt` v) `at` p  =  (r `at` p) `movingAt` v
```

A.1.5 Derived Regions

The tools introduced in the previous sections can be used to define a wide range of useful regions. We illustrate this with some examples that will be useful in the following work.

Annulus Regions An annulus, containing all those points whose distance from the origin is between radius values `r1` and `r2` can be described as the intersection of the outside of a circle of radius `r1` with a circle of radius `r2`:

```
> annulus       :: Radius -> Radius -> Region
> annulus r1 r2 = outside (circle r1) /\ circle r2
```

Convex Polygons Many useful regions can be described using convex polygons. The following definition allows us to construct arbitrary convex polygons by listing the vertices in counter-clockwise order:

```
> convexPoly        :: [Point] -> Region
> convexPoly (v:vs) = intersect (zipWith halfPlane ([v]++vs) (vs++[v]))
```

The key idea here is that, if the vertices are given by points `[v0, v1, ..., vn]`, then the interior of the polygon is just the intersection of the halfplanes joining adjacent points (including a line from `vn` to `v0`). In the definition above, `zipWith halfPlane` is used to produce this list of half-planes, and then `intersect` is used to produce their intersection. The `zipWith` function is a standard function included as part of the Haskell standard prelude.

Segment and Pie Regions It will also be useful to be able to deal with segment-shaped regions of the plane; i.e. regions obtained by sweeping a line emanating from the origin through a given range of angles (like a beam from a lighthouse). The easiest way to deal with this is to think of the segment as the intersection of two half planes:

```
> type Angle = Float

> segment     :: Angle -> Angle -> Region
> segment l u
>   = halfPlane (radial u) origin /\ halfPlane origin (radial l)
>     where radial a = Pt (cos theta) (sin theta)
>                      where theta = a * (pi / 180)
```

The auxiliary function `radial` defined here is used to calculate a point on the unit circle corresponding to a given angle `a`. Note that the angles `l` and `u`, in degrees, are multiplied by `pi/180` to convert them to radian measure as required by the `sin` and `cos` functions.

Using `segment` regions, we can define another function for constructing pie-shaped portions of a circle:

```
> pie       :: Radius -> Angle -> Angle -> Region
> pie r l u = segment l u /\ circle r
```

This will be used, for example, to define weapon doctrine regions.

A.2 Reacheability and Engageability

One remaining problem that we need to deal with is to describe the set of points that can be reached by some object traveling from the origin at some given speed. For example, this might be used to describe when an object is within range of a missile or radar signal.

This can be described using a circular region around the origin whose radius increases at the given speed. For simplicity, we have assumed that the speed is constant and that trajectories can be treated as straight lines. Although the mathematics would be a little more complex, the ideas used here can be extended to more realistic assumptions..

If s is the speed of expansion, then we can reach a point $\mathbf{p} + t\mathbf{v}$ at time t if:

$$|\mathbf{p} + t\mathbf{v}| < t\,s.$$

Squaring both sides, gives a quadratic:

$$(s^2 - |\mathbf{v}|^2)t^2 - 2t\mathbf{p} \cdot \mathbf{v} - |\mathbf{p}|^2 > 0$$

There are three cases to consider, depending on the comparative difference in the speeds s and $|\mathbf{v}|$ which determine the sign of the leading coefficient of the quadratic:

- If $s > |\mathbf{v}|$, then the region is expanding faster than the object is moving and the quadratic above takes positive values outside the range between its roots.
- If $s < |\mathbf{v}|$, then the object is moving faster than the region is expanding. The leading term of the quadratic is negative so it only takes positive values between the roots (if there are any).
- If $s = |\mathbf{v}|$, then the leading coefficient is zero and the quadratic reduces to:

$$-2t\mathbf{p} \cdot \mathbf{v} > |\mathbf{p}|^2 .$$

  There are two further subcases: If $\mathbf{p} \cdot \mathbf{v} \geq 0$ then there are no solutions for $t \geq 0$. But if $\mathbf{p} \cdot \mathbf{v} < 0$, then we require $t > -|\mathbf{p}|^2 / 2\mathbf{p} \cdot \mathbf{v}$.

Combining these various cases, gives the following definition for a circular region that is `expanding` with speed `s`:

```
> type Speed = Float

> expanding                    :: Speed -> Region
> expanding s p v | a >  0     = between False (quadRoots a (-b) (-c))
>                 | a <  0     = between True (quadRoots (-a) b c)
>                 | b <  0     = (False, [ -c / b ])
>                 | otherwise = (False, [])
>                   where a = s*s - sqrDist v
>                         b = 2 * p `dot` v
>                         c = sqrDist p
```

A.3 Object Tracking

In this section the object representations and tracking algorithms are fully defined.

A.3.1 Sensor/Track File Data

For the purpose of this program, we assume that sensor information is supplied as a stream of packets. Individual packets will be represented by a triple containing:

- The position of ownship,
- The positions of an arbitrary collection of slaved objects, and
- The positions of an arbitrary collection of objects to be tracked.

In fact, it will be useful to take a slightly more general approach and allow other kinds of data – for example, velocities – to be held in a packet. We will therefore use the following representation for tracking data, together with a more general `Packet` datatype:

```
> type TrackingData = Packet Point
> type Packet a = (a, [a], [a])
```

To estimate the velocity of an object, we just take the difference in the position of the object over a period of time and divide by the length of the interval:

```
> estimateVelocity          :: Point -> Point -> Velocity
> estimateVelocity new old = scale (1/timeInterval) (new - old)
```

For simplicity, we have assumed that packets are supplied at regular intervals, specified by the constant `timeInterval` defined in Section A.5, and we have used a very simple formula to estimate velocities. More sophisticated approaches can be accommodated by providing a timestamp for different input packets, or using calculations that make more use of earlier position data to predict velocities, even when some intermediate readings are missing.

A.3.2 Estimating Velocities

The next thing we want to do is extend the approach to take a stream of packets of positions and produce a stream of estimated velocity packets as our output. Since there is no previous data, the velocities produced for the first input packet are all zero. Subsequent velocities can be estimated using the `estimateVelocity` function defined above:

```
> estimateVelocities   :: [TrackingData] -> [Packet Velocity]
> estimateVelocities [] = []
> estimateVelocities (pk:pks)
>   = mapPacket (\p -> origin) pk :
>       zipWith (zipPacket estimateVelocity) pks (pk:pks)
```

The `estimateVelocities` function above was defined in terms of two general purpose functions for working with packets of information of various kinds. The first of these applies a specified function to each value held in a packet, the second combines each of the corresponding components in two packets to produce a packet of results. These functions are analogues of the standard Haskell functions `map` and `zipWith` for lists, motivating the choice of names for these functions here:

```
> mapPacket :: (a -> b) -> Packet a -> Packet b
> mapPacket f (own, slaves, objs)
>   = (f own, map f slaves, map f objs)

> zipPacket :: (a -> b -> c) -> Packet a -> Packet b -> Packet c
> zipPacket f (own, slaves, objs) (own', slaves', objs')
>   = (f own own', zipWith f slaves slaves', zipWith f objs objs')
```

It would certainly have been possible to present the definition of `estimateVelocities` without introducing these additional operators. However, our experience with programming in Haskell suggests that the additional generality that this provides is often very useful because they can be reused when additional features are added to the program at a later stage. In addition, to an experienced functional programmer, the use of auxiliary functions like this makes the program much easier to read and understand.

A.3.3 Dispositions

A disposition provides a snapshot of the current positions of the regions and objects of interest at a given point in time. It is represented by a pair containing a list of named regions, and a list of the objects that are being tracked:

```
> type Disposition = ([NamedRegion], [NamedObject])
> type NamedObject = (ObjectId,Point,Velocity)
> type NamedRegion = (String,Region)
> type ObjectId    = String
```

The representation for a named object includes a string (used to identify the object in diagnostic messages), and vectors giving the current position and (estimated) velocity.

Dispositions are parameterized by the list of regions centered on the ownship, the current slave doctrines, the (fixed) positions of tight zones and the names being used to identify the regions that are currently being tracked. We represent these parameters by a tuple:

```
> type Params = ([NamedRegion], -- regions centered on ownship
>                [NamedRegion], -- slave doctrine regions
>                [NamedRegion], -- tight zones
>                [ObjectId])    -- objects being tracked
```

(For simplicity, we have assumed a single ownship, although it would be easy to extend the program to use an arbitrary list of ownships.)

A disposition can be obtained by combining the data from these parameters, an input packet specifying the position of each object and a corresponding packet of estimated velocities:

```
> makeDisposition :: Params -> Packet Point -> Packet Velocity -> Disposition
> makeDisposition (ownRegions, slaveRegions, tightZones, objectIds)
>                   (pown, pslaves, pobjs) (vown, vslaves, vobjs)
>      = (map (position pown vown) ownRegions
>         ++ zipWith3 position pslaves vslaves slaveRegions
>         ++ tightZones,
>         zip3 objectIds pobjs vobjs)
>        where position p v (name,r) = (name, r `at` p `movingAt` v)
```

A.3.4 Situation Assessment

Having constructed a suitable disposition, the next task is to assess the situation that it describes, producing reports to indicate when an object is in a given region, or when it can be expected to enter or leave a given region. This is captured by the `assess` function which produces a single string containing all the reports for the current disposition. We use the `par` function to indicate that these individual reports can be generated concurrently in a parallel implementation:

```
> assess :: Disposition -> String
> assess (regs, objs)
>  = unlines (par [ r | obj <- objs, r <- objectReport obj regs ])
```

A.3.5 Report Generation

The `objectReport` function is used to produce the list of reports for a specific object relative to the list of regions of interest. The report begins with a display of the current position and velocity of the object, followed by reports for individual regions. As before, the `par` function is used to indicate that these reports can be produced in parallel.

```
> objectReport :: NamedObject -> [NamedRegion] -> [String]
> objectReport (what,p,v) rs
>  = [ what ++ ": " ++ showP p ++ " --> " ++ showP v ] ++
>    par [ rep | reg <- rs, rep <- regionReport p v reg ]
>    where showP (Pt x y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Finally, we have the `regionReport` function that produces the reports for a single object at point `p` moving with velocity `v` through a particular named region. These reports are generated by using the information obtained from the trail for the current object and region.

```
> regionReport :: Point -> Velocity -> NamedRegion -> [String]
> regionReport p v (region,r)
>   = case r p v of
>       (False, [])    -> []
>       (False, (t:_)) -> [" -- Expected in " ++ region ++
>                          " at t+" ++ showApprox t]
>       (True, [])     -> [" -- Currently in " ++ region]
>       (True, (t:_))  -> [" -- In " ++ region ++
>                          ", expected to leave at t+" ++ showApprox t]
```

Note that, in the first case, if the object is currently outside the region concerned with no expected entry time (indicated by the trail `(False,[])`, then no reports are actually generated in this case (or more precisely, the list of reports is empty).

Finally, we have included a simple routine to print the time values produced by the trail calculations to a single digit of accuracy (of course, the accuracy of the output figures is determined by the accuracy of the input data).

```
> showApprox  :: Time -> String
> showApprox t = front ++ take 2 rest
>                where (front,rest) = span ('.'/=) (show (t+0.05))
```

A.4 Complete Program

The complete program is described by a function that takes a set of disposition parameters `params`, and an input stream of tracking data `td` and produces a list of `String` reports as its output. The input stream `td` is combined with the estimated velocities to produce a stream of dispositions, each of which is passed to the `assess` function to obtain the reports required:

```
> program :: [Params] -> [TrackingData] -> [String]
> program params td
>  = map assess (zipWith3 makeDisposition params td (estimateVelocities td))
```

The parameter stream used here, typically generated from operator inputs, allows the situation being monitored to change dynamically over a period of time.

A.5 Sample Data and Execution

This section provides the sample data corresponding to the information given in the original problem description, and subsequent extensions.

We start with the descriptions of the weapon doctrine, engageability, missile range, slave doctrine and tight zone regions, and the list of objects that are being tracked.

```
> params :: Params
> params  = (ownRegions, slaveRegions, tightZones, objectIds)
>  where
>   ownRegions   = [("weapon doctrine",    pie 59 (-120) 120),
>                   ("engageability zone", ezone),
>                   ("missile range",      missile)]
>                  where ezone   = annulus 22 44
>                        missile = ezone /\ expanding missileSpeed
>   slaveRegions = [("carrier slave doctrine", circle 40)]
>   tightZones   = [("tight zone",
>                    convexPoly [Pt 0 5, Pt 118 32, Pt 118 62, Pt 0 25] \/
>                    convexPoly [Pt 118 32, Pt 259 5, Pt 259 25, Pt 118 62])]
>   objectIds    = ["commercial aircraft", "hostile craft"]
```

This data will be used as parameters to `makeDisposition`. For the data given in the problem description, the same parameters are used throughout, so the corresponding stream of input parameters required is just `repeat params`.

Two additional parameters, supplied in the list of extensions to the original prototyping document are the interval between packets and the speed of missiles:

```
> timeInterval  :: Time
> timeInterval   = 20

> missileSpeed  :: Speed
> missileSpeed   = 20 / timeInterval
```

Finally, the positions of the various objects at different times are given by the following table of packets, with coordinates taken directly from the Figures in the original problem description:

```
> trackingData :: [TrackingData]
> trackingData  = [(Pt 113 64, [Pt 180 140], [Pt 38 25,  Pt 258 183]),
>                  (Pt 123 64, [Pt 180 130], [Pt 58 30,  Pt 239 164]),
>                  (Pt 133 64, [Pt 180 119], [Pt 100 43, Pt 210 136]),
>                  (Pt 163 64, [Pt 180 81],  [Pt 159 36, Pt 148 73]),
>                  (Pt 192 64, [Pt 180 60],  [Pt 198 27, Pt 110 37])]
```

To demonstrate how this data can be used with our prototype implementation, we define the following function to print the list of reports, including an indication of the elapsed time at the beginning of each report:

```
> demo :: Dialogue
> demo  = display (zipWith (\t r -> "Time " ++ show t ++ ":\n" ++ r ++ "\n")
>                          (iterate (timeInterval+) 0)
>                          (program (repeat params) trackingData))
>  where display = foldr (\rep -> appendChan stdout rep  exit .
>                                 appendFile "trackFile" rep exit) done
```

The `display` function defined here is used to output the reports on the standard output channel with a newline character inserted after each (as a result of the call to `unlines`).

Running this program produces the following output:

```
Time 0.0:
commercial aircraft: (38.0,25.0) --> (0.0,0.0)
 -- Currently in tight zone
hostile craft: (258.0,183.0) --> (0.0,0.0)

Time 20.0:
commercial aircraft: (58.0,30.0) --> (1.0,0.25)
 -- Expected in weapon doctrine at t+131.3
 -- Expected in engageability zone at t+52.5
 -- Expected in missile range at t+52.5
 -- Expected in carrier slave doctrine at t+94.5
 -- In tight zone, expected to leave at t+93.2
hostile craft: (239.0,164.0) --> (-0.95,-0.95)
```

```
 -- Expected in weapon doctrine at t+55.6
 -- Expected in engageability zone at t+65.0
 -- Expected in missile range at t+65.0
 -- Expected in carrier slave doctrine at t+26.9
 -- Expected in tight zone at t+111.5

Time 40.0:
commercial aircraft: (100.0,43.0) --> (2.1,0.65)
 -- Expected in weapon doctrine at t+22.8
 -- In engageability zone, expected to leave at t+10.2
 -- Expected in missile range at t+34.3
 -- Expected in carrier slave doctrine at t+31.8
 -- In tight zone, expected to leave at t+19.8
hostile craft: (210.0,136.0) --> (-1.45,-1.4)
 -- Expected in weapon doctrine at t+19.6
 -- Expected in engageability zone at t+26.1
 -- Expected in missile range at t+31.6
 -- In carrier slave doctrine, expected to leave at t+44.3
 -- Expected in tight zone at t+55.1

Time 60.0:
commercial aircraft: (159.0,36.0) --> (2.95,-0.35)
 -- Expected in weapon doctrine at t+16.2
 -- In engageability zone, expected to leave at t+20.9
 -- Expected in carrier slave doctrine at t+3.9
 -- In tight zone, expected to leave at t+33.9
hostile craft: (148.0,73.0) --> (-3.1,-3.15)
 -- Expected in engageability zone at t+1.4
 -- In carrier slave doctrine, expected to leave at t+2.1
 -- Expected in tight zone at t+4.8

Time 80.0:
commercial aircraft: (198.0,27.0) --> (1.95,-0.45)
 -- In engageability zone, expected to leave at t+11.9
 -- In carrier slave doctrine, expected to leave at t+4.0
 -- In tight zone, expected to leave at t+31.3
hostile craft: (110.0,37.0) --> (-1.9,-1.8)
 -- In tight zone, expected to leave at t+5.0
```

B Prototype 1

This Appendix presents our "Prototype 1," a Haskell solution to the original problem specification given in Reference 1. It is being included to fully document the evolution of our prototype, and to demonstrate the graceful extensibility of programs written in Haskell.

This appendix B, which is a complete executable Haskell program, was written by Mark Jones of Yale University, and is covered by the copyright notice on this report.

B.1 Point

A point is used to identify a location in space. Following the simplifications required by the problem statement, we will represent a point by a pair of coordinates:

```
> type Point = (Float, Float)
```

These definitions are easily modified to extend the program to work in three dimensional space with other kinds of values, for example, floating point numbers, as coordinates.

To simplify our work with points, we define operations of addition and subtraction, treating points as vectors:

```
> instance (Num a, Num b) => Num (a,b) where
>     (x,y) + (u,v) = (x+u, y+v)
>     negate (x,y)  = (-x, -y)
>     (x,y) - (u,v) = (x-u, y-v)
```

It will also be convenient to be able to determine the distance of a point from the origin; this can be calculated using Pythagoras' theorem:

```
> type Distance = Float

> dist       :: Point -> Distance
> dist (x,y) = sqrt (x*x + y*y)
```

B.2 Regions

A region is a set of points in space. Since our main object is to construct regions and test to see when an object is inside a region, we will represent regions as predicates, returning `True` if a given point lies within a given region:

```
> type Region = Point -> Bool
```

B.2.1 Some Primitive Regions

We start by defining a collection of primitive regions from which the full regions that we are interested in can be constructed. This provides a library of regions that can easily be extended as necessary to accommodate new kinds of region as necessary.

Circular Regions A simple example of the kind of regions that will be useful to us in following work, is a circle with a specified centered and radius. The points inside this region are those whose distance from the center is less than or equal to the radius:

```
> type Radius = Float

> circle        :: Point -> Radius -> Region
> circle c r     = \p -> dist (p - c) <= r
```

Half Plane Regions Half planes are another useful form of region, representing all those points on one particular side of a given line. With a simple bit of geometry using the vector cross product, the half plane of points to the left of the line from a point `a` to a second point `b` can be described by the region `halfPlane a b`, where:

```
> halfPlane     :: Point -> Point -> Region
> halfPlane a b = \p -> let (x,y) = p - a
>                           (u,v) = p - b
>                       in  (x*v - y*u) >= 0
```

Segment Regions It will be also useful to have a way of representing regions of the plane that are obtained by sweeping a line emanating from a give point `p0` through the angles `l` to `u` (like the beam from a lighthouse). This can be defined by:

```
> type Angle = Float

> segment         :: Point -> Angle -> Angle -> Region
> segment p0 l u = \p -> let (x,y) = p - p0
>                            theta = degrees (atan2 y x)
>                        in  l <= theta && theta <= u
```

Note that the angles specified by $l \leq u$ should be in the range $-180°$ to $180°$ (i.e. $-\pi$ to π radians).

We have also used an auxiliary function `degrees` to convert the angle (in radians) returned by the `atan2` function to the corresponding number of degrees. We could have simply included this as a constant in the definition of `segment`. However, it seemed sensible to make `degrees` available as a general utility, since this may be useful in other parts of the program.

```
> degrees       :: Angle -> Angle
> degrees angle = (angle / pi) * 180
```

B.2.2 General Operations on Regions

There are also a number of more general operations on regions. For example, we can represent the intersection and the union of two regions using the operators `(/\)` and `(\/)`, respectively, defined by:

```
> (/\) , (\/) :: Region -> Region -> Region
> r1 /\ r2     = \p -> (r1 p && r2 p)
> r1 \/ r2     = \p -> (r1 p || r2 p)
```

It is sometimes useful to specify a region as the set of points that are `outside` some other given region:

```
> outside      :: Region -> Region
> outside r     = \p -> not (r p)
```

It is easy to extend this framework with other operations for working with regions, for example to calculate the intersection or union of a list of regions, or to modify a region by an arbitrary transformation on the coordinate space (for example, linear transformations such as reflections and rotations). However, none of these is required for the problems that we deal with in this prototype.

B.2.3 Derived Regions

The tools introduced in the previous sections can be used to define a wide range of useful regions. For example, an annulus containing all those points whose distance from a center point `c` is between `r1` and `r2` can be described by:

```
> annulus        :: Point -> Radius -> Radius -> Region
> annulus c r1 r2 = outside (circle c r1) /\ circle c r2
```

It will also be useful to have ways of specifying regions contained within arbitrary convex polygons, specified by listing the vertices in counter-clockwise order:

```
> convexPoly       :: [Point] -> Region
> convexPoly (v:vs) = foldr1 (/\) (zipWith halfPlane ([v]++vs) (vs++[v]))
```

The key idea here is that, if the vertices are given by points `[v0, v1, ..., vn]`, then the interior of the polygon is just the intersection of the halfplanes joining adjacent points (including a line from `vn` to `v0`). In the definition above, `zipWith halfPlane` is used to produce this list of half-planes, and `foldr1 (/\)` is used to construct their intersection. Both `zipWith` and `foldr1` are standard functions in Haskell.

B.3 Regions for Prototyping Problem

According to the data provided in Figure 6 of the problem description, there are four regions of interest in the sample problem, each of which will be described below. In a larger program these might be placed in a separate module as constants that can be changed without needing to modify the rest of the program.

- The Weapon doctrine is described by a portion of a circle, radius 59, centered around ownship with all points in the range −120° to 120°:

```
> weaponDoctrine          :: Point -> Region
> weaponDoctrine ownship = circle ownship 59 /\
>                          segment ownship (-120.0) 120.0
```

- The engageability zone is an annulus with inner radius 22 and outer radius 44, centered around ownship:

```
> engageabilityZone          :: Point -> Region
> engageabilityZone ownship = annulus ownship 22 44
```

- The slave doctrine for the carrier, given by a circle of radius 40 around the carrier:

```
> slaveDoctrine          :: Point -> Region
> slaveDoctrine carrier = circle carrier 40
```

- The tight zone for commercial traffic. This can be described as the union of two convex polygons:

```
> tightZone  :: Region
> tightZone   = left \/ right
>  where left  = convexPoly [(0,5),     (118,32), (118,62), (0,25)]
>        right = convexPoly [(118,32), (259,5),  (259,25), (118,62)]
```

A complete disposition, given the position of ownship and the carrier, and including names for each of the zones, is given by the following:

```
> type Disposition = [(String, Region)]

> makeDisposition :: Point -> Point -> Disposition
> makeDisposition ownship carrier
>  = [ ("weapons doctrine",   weaponDoctrine ownship),
>      ("engageability zone", engageabilityZone ownship),
>      ("slave doctrine",     slaveDoctrine carrier),
>      ("tight zone",         tightZone) ]
```

B.4 Putting Everything Together

The aim of the prototype is to assess configurations of regions and objects, indicating when particular objects are inside specified regions. We will describe this by a function:

```
> type Object = (String, Point)

> assess           :: Point -> Point -> [Object] -> String
> assess ownship carrier situation
>   | null reports = "all quiet!\n"
>   | otherwise    = unlines reports
>     where disp    = makeDisposition ownship carrier
>           reports = [ what++" in "++region | (what,p) <- situation,
>                                              (region,r) <- disp,
>                                              r p ]
```

The first two arguments give the current positions of ownship and the carrier, respectively. The third argument is a list of pairs containing the name (represented by a `String`) and a point (indicating the position) of each object that is being tracked. The output is a string providing the required status report.

To specify the set of problems that we want to assess, we use the following data, giving the names of the objects being tracked and their positions:

```
> objectNames  = [ "commercial aircraft", "hostile present" ]

> trackingData :: [(Point, Point, [Point])]
> trackingData  = [((113,64), (180,140), [(38,25),   (258,183)]),
>                  ((123,64), (180,130), [(58,30),   (239,164)]),
>                  ((133,64), (180,119), [(100,43),  (210,136)]),
>                  ((163,64), (180,81),  [(159,36),  (148,73)]),
>                  ((192,64), (180,60),  [(198,27),  (110,37)])]
```

To display the reports for each of these situations using the definitions given in this paper, we just run the following program:

```
> demo :: Dialogue
> demo  = appendChan stdout (unlines reports) exit done
>  where reports = [ assess ownship carrier (zip objectNames positions)
>                  | (ownship, carrier, positions) <- trackingData ]
```

Using the data above the `demo` program produces the following output:

```
commercial aircraft in tight zone

commercial aircraft in tight zone

commercial aircraft in engageability zone
commercial aircraft in tight zone
hostile present in slave doctrine
```

commercial aircraft in weapons doctrine
commercial aircraft in engageability zone
commercial aircraft in tight zone
hostile present in slave doctrine

commercial aircraft in weapons doctrine
commercial aircraft in engageability zone
commercial aircraft in slave doctrine
commercial aircraft in tight zone
hostile present in tight zone

C Trainee Version of Prototype 1

This Appendix presents a Haskell solution to the original problem specification written by a recent college graduate who was given only eight days to learn Haskell just prior to the experiment.

This appendix C, which is a complete executable Haskell program, was written by Kevin Coram of Intermetrics, and is covered by the copyright notice on this report.

```
> module AEGIS where
```

There are three basic types of things we will need to keep track of, each with their own geometric regions. These are: the Aegis ship with its weapon doctrine, and engageability zone; objects being tracked with their slave doctrine; and fixed Tight Zones, which can be specified by a two functions, which make up the lower and upper boundaries of the zone. Since the tight zone boundaries can be piecewise continuous functions, we will store the lower and upper boundaries as lists of functions. For all object types, there is a position. Since none of the information in a report will change from time instance to time instance other than the position, the position information will be stored external to the object information.

```
> data Object_type = Aegis Name Doctrine E_Zone
>                  | Track Name Doctrine
>                  | T_Zone [Function] [Function]
```

Give every object some type of identifying name. The names must be unique so we can filter out things like an object being in its own slave doctrine.

```
> type Name      = [Char]
```

The position of any object is specified by its Cartesian coordinates.

```
> type Position = (Float, Float)
```

There are two types of doctrines. The first is the weapon doctrine of the Aegis ship. It is the area of a circle of given radius, possibly with a wedge removed. Thus, it can be represented buy the radius of the circle, and the information for the wedge. We do not have a wedge associated with a slave doctrine, and therefore only need know the radius.

```
> data Doctrine = Weapon Radius Wedge
>               | Slave Radius
```

The engageability zone is modeled as the volume that a rectangle rotated around the Aegis ship sweeps out. Since for this model we are ignoring the altitude component, this becomes the area between two concentric circles centered at the ship.

```
> data E_Zone    = Zone (Radius, Radius)
```

For this model, represent the radius of a circle by a real number.

```
> type Radius    = Float
```

Represent the wedge removed from the weapon doctrine as the degrees for the start of the removed wedge, paired with the degrees for the end of the wedge. Use the X-axis in the Cartesian coordinate grid as the zero for the angles. The lack of a wedge can be represented by (0.0,0.0).

```
> type Wedge     = (Float, Float)
```

For this model, the functions used to bound a tight zone are straight lines. We can specify them with two points on the line, and the lower and upper bounds of the function along the X-axis.

```
> type Function = (Position, Position, Float, Float)
```

An object can be specified by the object's type, and it's current position. Note that the position field for a Tight Zone is redundant.

```
> type Object = (Object_type, Position)
```

Define the data for each of the objects in the model:

```
> aegis_ship = Aegis "Aegis Ship"
>              (Weapon 59.0 (120, 240))
>              (Zone (22.0, 44.0))

> carrier    = Track "Carrier" (Slave 40.0)

> hostile    = Track "Hostile" (Slave 0.0)

> commercial = Track "Commercial Airline" (Slave 0.0)

> tight_zone = T_Zone [((  0.0, 25.0), (118.0, 62.0),   0.0, 118.0),
>                      ((118.0, 62.0), (259.0, 25.0), 118.0, 259.0)]
>                     [((  0.0,  5.0), (118.0, 32.0),   0.0, 118.0),
>                      ((118.0, 32.0), (259.0,  5.0), 118.0, 295.0)]
```

For each positional report, make a list of type Object, with one entry per object, specifying the position for each object. By convention, the Aegis ship will always be the first record in the list.

```
> initial :: [Object]
> initial  = [ (aegis_ship, (118.0,  64.0)),
>              (carrier   , (180.0, 140.0)),
>              (hostile   , (258.0, 183.0)),
>              (commercial, ( 38.0,  25.0)),
>              (tight_zone, (  0.0,   0.0))
>            ]

> report2 :: [Object]
> report2  = [ (aegis_ship, (133.0,  64.0)),
>              (carrier   , (180.0, 119.0)),
>              (hostile   , (210.0, 136.0)),
>              (commercial, (100.0,  43.0)),
```

```
>               (tight_zone, (  0.0,   0.0))
>             ]

> report3 :: [Object]
> report3  = [ (aegis_ship, (123.0,  64.0)),
>               (carrier   , (180.0, 130.0)),
>               (hostile   , (239.0, 164.0)),
>               (commercial, ( 58.0,  30.0)),
>               (tight_zone, (  0.0,   0.0))
>             ]

> report4 :: [Object]
> report4  = [ (aegis_ship, (163.0,  64.0)),
>               (carrier   , (180.0,  81.0)),
>               (hostile   , (148.0,  73.0)),
>               (commercial, (159.0,  30.0)),
>               (tight_zone, (  0.0,   0.0))
>             ]

> report5 :: [Object]
> report5  = [ (aegis_ship, (192.0,  64.0)),
>               (carrier   , (180.0,  60.0)),
>               (hostile   , (110.0,  37.0)),
>               (commercial, (198.0,  27.0)),
>               (tight_zone, (  0.0,   0.0))
>             ]
```

The distance between two points on the Cartesian plane can be found by the following function.

```
> distance                  :: Position -> Position -> Float
> distance (x,y) (x',y')    = sqrt ((x-x')^2 + (y-y')^2)
```

Given a point, and a circle (center point paired with the radius), determine whether or not the point is inside the circle.

```
> inCircle              :: (Position, Radius) -> Position -> Bool
> inCircle (p1,r) p2    = (distance p1 p2) <= r
```

Given an engagement zone, paired with the center of the zone, and a point, determine if the point is within the engagement zone. It is in the zone if the point is inside the outer radius and not inside the inner radius.

```
> inZone                              :: (Position, E_Zone) -> Position -> Bool
> inZone (p1, Zone (r1, r2)) p2  = (inCircle (p1, r2) p2) &&
>                                     not (inCircle (p1, r1) p2)
```

Given a weapon doctrine, paired with the center of the doctrine, and a point, determine if the point is within the doctrine.

```
> inWeapon                          :: (Position, Doctrine) -> Position -> Bool
> inWeapon (p1, (Weapon r w)) p2 = (inCircle (p1, r) p2) &&
>                                     not (inWedge (p1, w) p2)
```

Given a wedge paired with the coordinates of the 'tip' of it, and a point, determine if the point is in that wedge.

```
> inWedge                    :: (Position, Wedge) -> Position -> Bool
> inWedge (p1, (d1,d2)) p2  = ((angle p1 p2) > d1) && ((angle p1 p2) < d2)
```

Given two points in the Cartesian plane, determine the angle a ray from the first to the second makes with the positive X-axis.

```
> angle                                          :: Position -> Position -> Float
> angle (x,y) (x',y') | x /= x' && y /= y' = toDegree (phase ((x'-x):+(y'-y)))
>                     | otherwise           = 0.0
```

The phase (and indeed all the Haskell trig functions) operate on radians. However, we are storing our angles as degrees. Function toDegree will change radian measures into degrees. The constant 57.29577951 is the number of degrees per radian. Also, the range for the Haskell trig functions is (-pi,pi) and what we want is (0,360), thus we add 180 degrees to the result of the radians-to-degrees calculation when the radian measure is negative.

```
> toDegree             :: Float -> Float
> toDegree x | x <  0  = 180 + (x * 57.29577951)
>            | x >= 0  = x * 57.29577951
```

Given an Aegis ship and an object being tracked, return a string telling us whether the tracked object is in the Weapon doctrine of the ship. If the first object is not an Aegis ship, return the null string.

```
> weapon_doctrine                              :: Object -> Object -> String
> weapon_doctrine ((Aegis n1 doct _), p1)
>                 ((Track n2 _)     , p2) = if (inWeapon (p1,doct) p2)
>                                           then
>                                             (n2 ++
>                                              " within weapon doctrine of "
>                                              ++ n1 ++ ".")
>                                           else ""
> weapon_doctrine _ _                      = ""
```

Given an Aegis ship and an object being tracked, return a string telling us whether the tracked object is in the Engageability Zone of the ship. Again, if the first object is not an Aegis ship, return the null string.

```
> engageable                              :: Object -> Object -> String
> engageable ((Aegis n1 _ zone), p1)
>            ((Track n2 _)     , p2) = if (inZone (p1,zone) p2)
>                                      then
>                                        (n2 ++
>                                         " within engageability zone of "
>                                         ++ n1 ++ ".")
>                                      else ""
> engageable _ _                     = ""
```

Given two objects, return a string telling us if the second object is in the Slave Doctrine of the first. The Aegis ship does not have a slave doctrine, so if it is the first object, return a null string. Likewise, tight zones do not have any doctrines, so return a null string for them as well.

```
> inSlave                                      :: Object -> Object -> String
> inSlave ((Aegis _ _ _), _) _                 = ""
> inSlave ((Track n1 (Slave r)), p1)
>         ((Aegis n2 _ _), p2) | n1 /= n2 = if (inCircle (p1,r) p2)
>                                           then
>                                             (n2 ++
>                                              " within slave doctrine of "
>                                              ++ n1 ++ ".")
>                                           else ""
>                              | n1 == n2 = ""
> inSlave ((Track n1 (Slave r)), p1)
>         ((Track n2 _) , p2) | n1 /= n2  = if (inCircle (p1,r) p2)
>                                           then
>                                             (n2 ++
>                                              " within slave doctrine of "
>                                              ++ n1 ++ ".")
>                                           else ""
>                             | n1 == n2  = ""
>                                           = ""
> inSlave _ _
```

Given a function (as defined above), we may want to evaluate the Y value of that function for a given X value. Function evaluate does this for us. It takes in the list of the piecewise function and finds the range that our X value is in. It then calls the appropriate function evaluator given the function definition for the range on the X value, returning the Y value for the function. If the X value is not within one of the ranges where the function is defined, the error routine is called, givin us a run-time error.

```
> evaluate :: [Function] -> Float -> Float
> evaluate [] _ = error "Point out of range in function evaluate"
> evaluate (x:xs) val = if inBounds x val then
>                          straight_line x val
>                       else
>                          evaluate xs val
```

Function inBounds takes a single function and an X value and returns True is the X value is within the bounds for the function definition.

```
> inBounds                  :: Function -> Float -> Bool
> inBounds (_, _, l, u) x  = (l <= x) && (x <= u)
```

Our functions are straight lines, specified by two points on the line. The function below uses those two points to form a function that will map X values into Y values. Notice that m is our slope, and the (y-m*x) term is our Y intercept.

```
> straight_line                                  :: Function -> Float -> Float
> straight_line ((x,y), (x',y'), _, _) val  = m * val + y - m * x
>                                               where m = (y-y')/(x-x')
```

Given a point and a function, determine if that point is above or below the function curve.

```
> below            :: Position -> [Function] -> Bool
> (x,y) `below` fs  = (evaluate fs x) <= y

> above            :: Position -> [Function] -> Bool
> (x,y) `above` fs  = (evaluate fs x) >= y
```

Given a tight zone and a point, determine if the point is within the tight zone. The point is inside the tight zone if it is above the lower bounding function and below the upper boundng function.

```
> tight                   :: Object_type -> Position -> Bool
> tight (T_Zone ls us) p  = (p `below` us) && (p `above` ls)
```

Using the above function, output a string if the object is in the tight zone. If the first object is not a tight zone, return the null string. Likewise, if both objects are tight zones, return null since having one tight zone inside another is meaningless.

```
> inTightZone                        :: Object -> Object -> String
> inTightZone ((T_Zone ls us), _)
>             ((Aegis n1 _ _), p1) = if (tight (T_Zone ls us) p1)
>                                    then
>                                      n1 ++ " is in a tight zone."
>                                    else ""
```

```
> inTightZone ((T_Zone ls us), _)
>             ((Track n1 _), p1)   = if (tight (T_Zone ls us) p1)
>                                    then
>                                      n1 ++ " is in a tight zone."
>                                    else ""
> inTightZone _ _                  = ""
```

To generate a report, we will splice together the output strings from applying the `weapon_doctrine`, `engageable`, `inSlave`, and `inTightZone` functions together. For `weapon_doctrine` and `engageable`, we map the functions, with the Aegis ship as the argument, onto all the rest of the objects in the list of objects for the position report. To check the slave doctrines and tight zones, we must look at all pairs of objects. To do this, we use list comprehension rather than mapping. All this generates a list of strings. Using the filter function to remove any strings that are null, we then use the unlines function to splice the list of strings into a single, longer string.

```
> genReport :: [Object] -> String
> genReport (x:xs)  = unlines                              $
>                     filter ( /= "")                      $
>                     ["Weapon Doctrine Report:"]         ++
>                     (map (weapon_doctrine x) xs)        ++
>                     ["\nEngageability Zone Report:"]    ++
>                     (map (engageable x) xs)             ++
>                     ["\nSlave Doctrine Report:"]        ++
>                     [ inSlave y z | y <- x:xs, z <- x:xs ] ++
>                     ["\nTight Zone Report:"]                 ++
>                     [ inTightZone y z | y <- x:xs,
>                                         z <- x:xs]
```

The heart of the program interface is the printReport dialogue. It will print out onto the standard output the report for the given positional report list. To run it, select C-c r to run a dialogue and then type printReport <report list> where <report list> is the name of the positional report list for which we want a report.

```
> printReport :: [Object] -> Dialogue
> printReport xs = appendChan stdout (genReport xs) exit done
```

Appendix D

An Ada 9X Solution to the AEGIS Weapons System (AWS) Prototyping Demonstration Project*

* Prepared by Benjamin M. Brosgol, consultant to Intermetrics, Inc.

This appendix provides a solution to the AWS prototyping problem, written in Ada 9X. A solution to this problem has two aspects. First, it is necessary to *model the architecture* of the system: the *objects* (ships, tracks), *attributes* (position, weapon doctrine, etc) and *relationships* (such as when a track is in a given region). Second, it is necessary to *implement the geometry* to prototype the algorithms. Both of these are essential. Focusing solely on the 1st may yield a design that, though sound in theory, overlooks issues (such as performance) that are important in practice. On the other hand, focusing solely on the second may yield a simple working demo quickly but does not necessarily "scale up"; major redesign may be needed to meet the requirements of the full problem.

Ada provides facilities that allow us to address both aspects. In this paper we will present a solution using Ada 9X. An Ada 83 version is also feasible, but for a more complete exploitation of Object-Oriented Programming (OOP) methods, the emerging Ada 9X standard supplies the needed capabilities.

D.1 System Architecture

Table 1 of the Prototype Description reveals the kinds (classes) of objects that the program will deal with. In this section we define these object classes, the attributes of each class, and the relationships among the classes. (Note that the term "attribute" is used in this paper in the OOP sense of a fetchable/storable component of an object, rather than in the Ada syntactic sense.) We also summarize the required processing, indicating the potential parallelism.

D.1.1 Class/Attribute/Relation Notation

The notation defined in this section allows the succinct expression of an OOP-based system design. It is an abstraction of OOP syntax as found in traditional OO languages, and the description employs conventional OOP terminology ("class", "subclass", "attribute"). It can be easily (in fact, mechanically) mapped to Ada 9X packages, with a "class" corresponding to both a package (possibly a child) and a tagged type.

The form:

```
Class_Name [ Attr_1 : Attr_Type_1;
             Attr_2 : Attr_Type_2;
             ...]
           { Func_1(Parameter_Types_1) : Func_1_Return_Type;
             ...}
```

defines the class Class_Name; any object of this class has attributes Attr_1 of type Attr_Type_1, Attr_2 of type Attr_Type_2, etc. Moreover, Func_1 applied to the object and also to parameters of the types specified in the list returns a value of type Func_1_Return_Type, etc.

Implicitly associated with the class Class_Name are two operations, one to construct a new object and the other to delete an existing object.

```
function New_Class_Name(Attr_1 : Attr_Type_1;
                        Attr_2 : Attr_Type_2;
                        ...) return Class_Name_Ref;

procedure Delete ( Object : in out Class_Name_Ref);
```

where Class_Name_Ref is an access type whose designated type is Class_Name'Class (see below for definition of 'Class).

Each attribute [Attr : Attr_Type] is implemented by two operations:

- a procedure that sets a new value for the attribute:

```
procedure Set_Attr( Object    : in out Class_Name;
                    New_Value : Attr_Type);
```

- a function that retrieves the current value of the attribute:

```
function Attr (Object : Class_Name) return Attr_Type;
```

Each function {Func(Parameter_List) : Return_Type} is implemented by a corresponding function:

```
function Func (Object : Class_Name; Parm_1 : Parm_Type_1; ...)
return Return_Type;
```

where Parm_Type_1 etc correspond to the Parameter_List.

Indentation defines the class-subclass relationship; a subclass may introduce new attributes and functions, override existing attributes and functions, or both. For example,

```
Class_1 [Attr_1 : Attr_Type_1]
      Class_2 [Attr_2 : Attr_Type_2]
      Class_3 [Attr_1 : Attr_Type_1;
               Attr_3 : Attr_Type_3]
```

defines the class Class_1 and two "subclasses", Class_2 and Class_3. Let Class_1'Class denote the discriminated union of Class_1 and its subclasses. Then the following properties hold:

- An object of class Class_1 has (only) the attribute Attr_1. An object of class Class_2 has the attributes Attr_1 (inherited from the parent class) and Attr_2. Similarly, an object of class Class_3 has the attributes Attr_1 and Attr_3. Note that the value of Attr_1 for a Class_3 object is given by the "inner" attribute definition.

- An object from Class_1'Class is an object from one of the classes Class_1, Class_2, and Class_3, together with a "tag" identifiying which class.

Those familiar with OOP will see that the above notation is a succinct way of identifying *inheritance* (the subclass/class hierarchy), *polymorphism* (an object from C'Class can be in class

C, or in any subclass of C), and *dynamic binding* (the interpretation of an attribute of an object from C'Class depends on which class it is in).

The notation

```
Relation_Name (Class_1, ..., Class_n)
```

indicates that there is a relation Relation_Name such that for any objects Obj_1, ..., Obj_n in classes Class_1, ..., Class_n, respectiveley, the condition Relation_Name(Obj_1, ..., Obj_n) is either true or False.

D.1.2 Definition of AWS Prototype Object Classes

With the above notational conventions in place, we can succinctly portray the object classes relevant to the AWS example. The attribute types shown will be defined later.

```
Ship [Position : Ship_Position_Type ]
      Ownship [Weapon_Doctrine     : W_Doctrine_Type;
               Engageability_Zone  : E_Zone_Type]
      Carrier [Slaved_Doctrine     : S_Doctrine_Type]

Track [Position : Track_Position_Type] {Kind() : String};
      Friendly[]
      Hostile[]
      Commercial[]

Tight_Zone [Boundary : Boundary_Type]
```

Note that since weapon doctrine, engageability zone, and slaved doctrine exist only in terms of a host ownship or carrier, they are modeled as attributes as opposed to object classes. On the other hand, since a tight zone is an object independent of ships and tracks, it is modeled by an object class.

Some attribute values are updated periodically as part of the behavior of the program. (In the above example, these are the Position attributes of Ship and Track.) Other attribute values change either infrequently or not at all.

The Track hierarchy lets us model the situation where an aircraft is either known to be friendly, hostile, or commmercial, or is unknown. In particular, the function Kind returns "Unknown" when the parameter is in class Track, "Hostile" when in class Hostile, and "Commercial" when in class Commercial.

D.1.3 Relationships among the object classes

The following relations are of relevance to the program; each may be regarded as a function that delivers a Boolean result based on whether the condition holds:

```
In_Weapon_Doctrine(Track_Ref, Ownship)
```

```
In_Engageability_Zone(Track_Ref, Ownship)

In_Slaved_Doctrine(Track_Ref, Carrier)

In_Tight_Zone(Track_Ref, Tight_Zone)
```

Note that there are many different ways in which these functions could be implemented. For example, each track object could initiate a check of which regions it is in, every time its position is updated. Alternatively, a separate part of the program could do this checking asynchronously. The above relations are thus an abstract statement of what is required, independent of the implementation.

D.1.4 Processing Model

The addendum to the problem description states:

> "Prototypers are encouraged to think in terms of three concurrent processes, sensor/track file building, geo-region computation management, and operator interface; and two data structures with protection for concurrent access, track file and geo-region descriptions."

This is readily modeled in Ada; in fact, the above object class summary is independent of whether the underlying processing model is sequential or parallel. As we will see in the next section, each object class can be implemented by one or more protected objects, thus ensuring that in the parallel case an update operation to an attribute value is mutually exclusive with all other accesses to that value.

The problem statement leaves open a number of alternatives for modeling the potential concurrency, but strongly suggests that the track updates and engagement calculations be independent. To satisfy this requirement, our design has the following tasking structure:

- One periodic task updates the position of each ship and track.
- For each track, there is a periodic task that uses the current positions to update the relationships (therein performing the engagement calculations) and to produce an appropriate output message identifying when a track is in one of the regions.
- A server task manages the output messages, ensuring that a shared device such as the operator console is not the simultaneous target of several output operations.

D.2 Ada Specification of the System Architecture

Each of the package specifications below can be derived mechanically from the Class/Attribute/Relation notation of section 2. To enforce the "concurrent read, exclusive write" protocol on attribute accesses, each attribute is implemented via an Ada 9X protected type.

The attributes at the various levels of the type hierarchy have independent protection; thus it is possible, for example, that the Weapon_Doctrine of an Ownship could be accessed while the

Position of the Ownship is being updated. This increases the opportunity for parallelism. If the attributes were not independent, a richer protection strategy would be required. Note that the enforcement of this strategy would be part of the definition of the ship/ownship abstraction and thus transparent to clients. (That is, the traditional but error-prone technique of having client code explicitly lock and release semaphores can be avoided in Ada; more reliable mechanisms are available.)

D.2.1 Ship Package Hierarchy

```
--Ship [Position : Ship_Position_Type ]
--    Ownship [Weapon_Doctrine     : W_Doctrine_Type;
--             Engageability_Zone : E_Zone_Type]
--    Carrier [Slaved_Doctrine     : S_Doctrine_Type]
```

D.2.1.1 Package Ship_Pckg

```
with Geometry_Pckg; use Geometry_Pckg;
package Ship_Pckg is
   type Ship is tagged limited private
   type Ship_Ref is access all Ship'Class;

   function New_Ship (Position : Ship_Position_Type)
      return Ship_Ref;

   procedure Delete (Object : in out Ship_Ref);

   function Position (Object : Ship) return Ship_Position_Type;

   procedure Set_Position (Object    : in out Ship;
                           New_Value : Ship_Position_Type);

private
   protected type Position_Protector is
      function Position return Ship_Position_Type;
      procedure Set_Position (New_Value : Ship_Position_Type);
   private
      Position_ : Ship_Position_Type;
   end Position_Protector;

   type Ship is tagged limited
      record
         Position : Position_Protector;
      end record;
end Ship_Pckg;
```

D.2.1.2 Package Ship_Pckg.Ownship_Pckg

```
package Ship_Pckg.Ownship_Pckg is
   type Ownship is new Ship with private;
```

```
   function New_Ownship
     (Position           : Ship_Position_Type;
      Weapon_Doctrine    : W_Doctrine_Type;
      Engageability_Zone : E_Zone_Type)
   return Ship_Ref;

   procedure Delete (Object : in out Ownship_Ref);

   function Weapon_Doctrine (Object : Ownship) return W_Doctrine_Type;
   function Engageability_Zone (Object : Ownship) return E_Zone_Type;

   procedure Set_Weapon_Doctrine (Object    : in out Ownship;
                                  New_Value : W_Doctrine_Type);

   procedure Set_Engageability_Zone (Object    : in out Ownship;
                                     New_Value : E_Zone_Type);

private
   protected type Weapon_Doctrine_Protector is
      function Weapon_Doctrine return W_Doctrine_Type;
      procedure Set_Weapon_Doctrine(New_Value : W_Doctrine_Type);
   private
      Weapon_Doctrine_ : W_Doctrine_Type;
   end Weapon_Doctrine_Protector;

   protected type Engagagement_Zone_Protector is
      function Engageability_Zone return E_Zone_Type;
      procedure Set_Engageability_Zone(New_Value : E_Zone_Type);
   private
      Engageability_Zone_ : E_Zone_Type;
   end Engagament_Zone_Protector;

   type Ownship is new Ship with
      record
         Weapon_Doctrine : Weapon_Doctrine_Protector;
         Engagement_Zone : Engagement_Zone_Protector;
      end record;
end Ownship_Pckg;
```

D.2.1.3 Package Ship_Pckg.Carrier_Pckg

```
package Ship_Pckg.Carrier_Pckg is
   type Carrier is new Ship with private;

   function New_Carrier
     (Position           : Ship_Position_Type;
      Slaved_Doctrine    : S_Doctrine_Type;)
   return Ship_Ref;

   procedure Delete ( Object : in out Carrier_Ref );

   function Slaved_Doctrine (Object : Ownship) return S_Doctrine_Type;
```

```
   procedure Set_Slaved_Doctrine (Object     : in out Ownship;
                                  New_Value : S_Doctrine_Type);

private
   protected type Slaved_Doctrine_Protector is
      function Slaved_Doctrine return S_Doctrine_Type;
      procedure Set_Slaved_Doctrine (New_Value : S_Doctrine_Type);
   private
      Slaved_Doctrine_ : S_Doctrine_Type;
   end Slaved_Doctrine_Protector;

  type Carrier is new Ship with
      record
         Slaved_Doctrine : Slaved_Doctrine_Protector;
      end record;
end Ship_Pckg.Carrier_Pckg;
```

D.2.2 Track Package Hierarchy

```
--Track [Position : Track_Position_Type] {Kind() : String};
--    Friendly[] {Kind() : String}
--    Hostile[] {Kind() : String}
--    Commercial[] {Kind() : String}
```

D.2.2.1 Package Track_Pckg

```
with Geometry_Pckg; use Geometry_Pckg;
package Track_Pckg is
   type Track is tagged limited private;
   type Track_Ref is access Track'Class;

   function New_Track(Position : Track_Position_Type)
      return Track_Ref;

   procedure Delete ( Object : in out Track_Ref );

   function Position (Object : Track) return Track_Position_Type;

   procedure Set_Position (Object     : in out Track;
                           New_Value : Track_Position_Type);

   function Kind (Object : Track) return String;
private
   protected type Position_Protector is
      function Position return Track_Position_Type;
      procedure Set_Position (New_Value : Track_Position_Type);
   private
      Position_ : Track_Position_Type;
   end Track_Protector;
```

```
   type Track is tagged limited
      record
         Position : Position_Protector;
      end record;
end Track_Pckg;
```

D.2.2.2 Package Track_Pckg.Friendly_Pckg

```
package Track_Pckg.Friendly_Pckg is
   type Friendly is new Track with private;

   function New_Friendly(Position : Track_Position_Type)
   return Track_Ref;

   function New_Friendly(Position : Track_Position_Type)
   return Friendly_Ref;

   function Kind (Object : Friendly) return String;

private
   type Friendly is new Track with null record;
end Track_Pckg.Friendly_Pckg;
```

D.2.2.3 Package Track_Pckg.Hostile_Pckg

```
package Track_Pckg.Hostile_Pckg is
   type Hostile is new Track with private;
   type Hostile_Ref is access Hostile'Class;

   function New_Hostile(Position : Track_Position_Type)
   return Track_Ref;

   function New_Hostile(Position : Track_Position_Type)
   return Hostile_Ref;

   function Kind (Object : Hostile) return String;

private
   type Hostile is new Track with null record;
end Track_Pckg.Hostile_Pckg;
```

D.2.2.4 Package Track_Pckg.Commercial_Pckg

```
package Track_Pckg.Commercial_Pckg is
   type Commercial is new Track with private;
   type Commercial_Ref is access Commercial'Class;

   function New_Commercial(Position : Track_Position_Type)
   return Track_Ref;

   function New_Commercial(Position : Track_Position_Type)
   return Commercial_Ref;
```

```
   function Kind (Object : Commercial) return String;

private
   type Commercial is new Track with null record;
end Track_Pckg.Commercial_Pckg;
```

D.2.3 Tight Zone

```
--Tight_Zone [Boundary : Boundary_Type]

with Geometry_Pckg; use Geometry_Pckg;
package Tight_Zone_Pckg is
   type Tight_Zone is tagged limited private;
   type Tight_Zone_Ref is access Tight_Zone'Class;

   function New_Tight_Zone (Boundary : Boundary_Type)
      return Tight_Zone_Ref;

   procedure Delete ( Object : in out Tight_Zone_Ref );

   function Boundary (Object : Tight_Zone) return Boundary_Type;

   procedure Set_Boundary (Object    : in out Tight_Zone;
                           New_Value : Boundary_Type);

private
   protected type Boundary_Protector is
      function Boundary return Boundary_Type;
      procedure Set_Boundary (New_Value : Boundary_Type);
   private
      Boundary_ : Boundary_Type;
   end Boundary_Protector;

   type Tight_Zone is tagged limited
      record
         Boundary : Boundary_Protector;
      end record;
end Tight_Zone_Pckg;
```

D.2.4 Package Relations

```
package Relations is

   function In_Weapon_Doctrine (Target : Track_Ref;
                                Vessel : Ownship) return Boolean;

   function In_Engageability_Zone (Target : Track_Ref;
                                   Vessel : Ownship) return Boolean;

   function In_Slaved_Doctrine (Target : Track_Ref;
                                Vessel : Carrier) return Boolean;
```

```
   function In_Tight_Zone (Target : Track_Ref;
                           Region : Tight_Zone) return Boolean;

end Relations;
```

D.2.5 Package Geometry

```
package Geometry is

   subtype Real is Float;
   subtype Degrees is Float range -180.0 .. 180.0;

   type Point is
      record
         X, Y : Real;
      end record;

   type Ship_Position_Type  is new Point;
   type Track_Position_Type is new Point;

   type W_Doctrine_Type is
      record
         Radius : Real;
         Angle  : Degrees;
      end record;
      -- Assumes that the direction of motion is parallel to X-axis

   type S_Doctrine_Type is
      record
         Radius : Real;
      end record;

   type E_Zone_Type is
      record
         Inner_Radius : Real;
         Outer_Radius : Real;
      end record;

   type T_Zone_Endpoint is new Point;

   type T_Zone_Array is array (Positive range <>) of
          T_Zone_Endpoint;

   type Boundary_Type is access T_Zone_Array;
   -- A tight zone is modeled by a pointer to an array of points
   -- The points are considered to be connected to form a closed polygon

end Geometry;
```

D.3 Ada Implementation of System Architecture

D.3.1 Ship Package Hierarchy

D.3.1.1 Package Ship_Pckg

```
with Unchecked_Deallocation;
package body Ship_Pckg is

   procedure Free_Ship is
      new Unchecked_Deallocation(Ship'Class
                                 Ship_Ref);

   function New_Ship(Position : Ship_Position_Type)
   return Ship_Ref is
      Ref : Ship_Ref;
   begin
      Ref := new Ship;
      Ref.all.Position.Set_Position(Position);
      return Ref;
   end New_Ship;

   procedure Delete ( Object : in out Ship_Ref ) is
   begin
      Free_Ship(Object);
   end Delete'

   protected body Position_Protector is
      function Position return Ship_Position_Type is
      begin
         return Position_;
      end Position;

      procedure Set_Position (New_Value : Ship_Position_Type) is
      begin
         Position_ := New_Value;
      end Set_Position;

   end Position_Protector;

   function Position (Object : Ship) return Ship_Position_Type is
   begin
      return Object.Position.Position;  -- Calls function
   end Position;

   procedure Set_Position (Object    : Ship;
                           New_Value : Ship_Position_Type) is
   begin
      Object.Position.Set_Position(New_Value);
   end Set_Position;

end Ship_Pckg;
```

D.3.1.2 Package Ship_Pckg.Ownship_Pckg

```
package body Ship_Pckg.Ownship_Pckg is

   function New_Ownship
     (Position           : Ship_Position_Type;
      Weapon_Doctrine    : W_Doctrine_Type;
      Engageability_Zone : E_Zone_Type)
   return Ship_Ref is
      Ref : Ship_Ref;
   begin
      Ref := new Ownship;
      Ref.all.Position.Set_Position(Position);
      Ref.all.Weapon_Doctrine.Set_Weapon_Doctrine(Weapon_Doctrine);
      Ref.all.Engageability_Zone.
              Set_Engageability_Zone(Weapon_Doctrine);
      return Ref;
   end New_Ownship;

   function Weapon_Doctrine (Object : Ownship)
      return W_Doctrine_Type is
   begin
      return Object.Weapon_Doctrine.Weapon_Doctrine;
   end Weapon_Doctrine;

   function Engageability_Zone (Object : Ownship)
      return E_Zone_Type is
   begin
      return Object.Engageability_Zone.Engageability_Zone;
   end Engageability_Zone;


   procedure Set_Weapon_Doctrine (Object    : in out Ownship;
                                  New_Value : W_Doctrine_Type) is
   begin
      Object.Weapon_Doctrine.Set_Weapon_Doctrine(New_Value);
   end Set_Weapon_Doctrine;

   procedure Set_Engageability_Zone (Object    : in out Ownship;
                                     New_Value : E_Zone_Type) is
   begin
      Object.Engageability_Zone.Set_Engageability_Zone(New_Value);
   end Set_Engageability_Zone;


   protected body Weapon_Doctrine_Protector is
      function Weapon_Doctrine return W_Doctrine_Type is
      begin
         return Weapon_Doctrine_;
      end Position;
```

```
      procedure Set_Weapon_Doctrine (New_Value : W_Doctrine_Type) is
      begin
         Weapon_Doctrine_ := New_Value;
      end Set_Weapon_Doctrine;

   end Weapon_Doctrine_Protector;

   protected body Engageability_Zone_Protector is
      function Engageability_Zone return E_Zone_Type is
      begin
         return Engageability_Zone_;
      end Position;

      procedure Set_Engageability_Zone (New_Value : E_Zone_Type) is
      begin
         Engageability_Zone_ := New_Value;
      end Set_Engageability_Zone;

   end Engageability_Zone_Protector;

end Ship_Pckg.Ownship_Pckg;
```

D.3.1.3 Package Ship_Pckg.Carrier_Pckg

```
package body Ship_Pckg.Carrier_Pckg is

   function New_Carrier
     (Position           : Ship_Position_Type;
      Slaved_Doctrine    : S_Doctrine_Type)
   return Ship_Ref is
      Ref : Ship_Ref;
   begin
      Ref := new Carrier;
      Ref.all.Position.Set_Position(Position);
      Ref.all.Slaved_Doctrine.Set_Slaved_Doctrine(Slaved_Doctrine);
      return Ref;
   end New_Carrier;

   function Slaved_Doctrine (Object : Carrier)
      return S_Doctrine_Type is
   begin
      return Object.Slaved_Doctrine.Slaved_Doctrine;
   end Slaved_Doctrine;

   procedure Set_Slaved_Doctrine (Object    : in out Carrier;
                                  New_Value : S_Doctrine_Type) is
   begin
      Object.Slaved_Doctrine.Set_Slaved_Doctrine(New_Value);
   end Set_Slaved_Doctrine;
```

```
   protected body Slaved_Doctrine_Protector is
      function Slaved_Doctrine return S_Doctrine_Type is
      begin
         return Slaved_Doctrine_;
      end Slaved_Doctrine;

      procedure Set_Slaved_Doctrine (New_Value : S_Doctrine_Type) is
      begin
         Slaved_Doctrine_ := New_Value;
      end Set_Slaved_Doctrine;

   end Slaved_Doctrine_Protector;

end Ship_Pckg.Carrier_Pckg;
```

D.3.2 Track Package Hierarchy

D.3.2.1 Package Track_Pckg

```
with Unchecked_Deallocation;
package body Track_Pckg is
   function New_Track(Position : Track_Position_Type)
      return Track_Ref is
      Ref : Track_Ref;
   begin
      Ref := new Track;
      Ref.all.Position.Set_Position(Position);
      return Ref;
   end New_Track;

   procedure Free_Track is
     new Unchecked_Deallocation(Track'Class, Track_Ref);

   procedure Delete ( Object : in out Track_Ref ) is
   begin
      Free_Track(Object);
   end Delete;

   function Position (Object : Track) return Track_Position_Type is
   begin
      return Object.Position.Position;
   end Position;

   procedure Set_Position (Object    : in out Track;
                           New_Value : Track_Position_Type) is
   begin
      Object.Position.Set_Position(New_Value);;
   end Set_Position;
```

```
   function Kind (Object : Track) return String is
   begin
      return "Unknown";
   end Kind;

   protected body Position_Protector is
      function Position return Track_Position_Type is
      begin
         return Position_;
      end Position;

      procedure Set_Position (New_Value : Track_Position_Type) is
      begin
         Position_ := New_Value;
      end Set_Position;
   end Position_Protector;

end Track_Pckg;
```

D.3.2.2 Children of Track_Pckg

```
package body Track_Pckg.Friendly_Pckg is
   function New_Friendly (Position : Track_Position_Type)
   return Track_Ref is
      Ref : Track_Ref;
   begin
      Ref := new Friendly;
      Ref.all.Position.Set_Position(Position);
      return Ref;
   end New_Friendly;

   function Kind (Object : Friendly) return String is
   begin
      return "Friendly";
   end Kind;
end Track_Pckg.Friendly_Pckg;

package body Track_Pckg.Hostile_Pckg is
   function New_Hostile (Position : Track_Position_Type)
   return Track_Ref is
      Ref : Track_Ref;
   begin
      Ref := new Hostile;
      Ref.all.Position.Set_Position(Position);
      return Ref;
   end New_Hostile;

   function Kind (Object : Hostile) return String is
   begin
      return "Hostile";
   end Kind;
end Track_Pckg.Hostile_Pckg;
```

```
package body Track_Pckg.Commercial_Pckg is
   function New_Commercial (Position : Track_Position_Type)
   return Track_Ref is
      Ref : Track_Ref;
   begin
      Ref := new Commercial;
      Ref.all.Position.Set_Position(Position);
      return Ref;
   end New_Commercial;

   function Kind (Object : Commercial) return String is
   begin
      return "Commercial";
   end Kind;
end Track_Pckg.Commercial_Pckg;
```

D.3.3 Tight Zone

```
with Unchecked_Deallocation;
package body Tight_Zone_Pckg is

   procedure Free_Tight_Zone is
      new Unchecked_Deallocation(Tight_Zone'Class
                                 Tight_Zone_Ref);

   function New_Tight_Zone(Boundary : Boundary_Type)
   return Tight_Zone_Ref is
      Ref : Tight_Zone_Ref;
   begin
      Ref := new Tight_Zone;
      Ref.all.Boundary.Set_Boundary(Boundary);
      return Ref;
   end New_Tight_Zone;

   procedure Delete ( Object : in out Tight_Zone_Ref ) is
   begin
      Free_Tight_Zone(Object);
   end Delete'

   protected body Boundary_Protector is
      function Boundary return Boundary_Type is
      begin
         return Boundary_;
      end Boundary;

      procedure Set_Boundary (New_Value : Boundary_Type) is
      begin
         Boundary_ := New_Value;
      end Set_Boundary;

   end Boundary_Protector;
```

```
   function Boundary (Object : Tight_Zone) return Boundary_Type is
   begin
      return Object.Boundary.Boundary;  -- Calls function
   end Boundary;

   procedure Set_Boundary (Object    : Tight_Zone;
                           New_Value : Boundary_Type) is
   begin
      Object.Boundary.Set_Boundary(New_Value);
   end Set_Boundary;

end Tight_Zone_Pckg;
```

D.3.4 Implementation of Package Relations

All of the assumptions about the geometry are encapsulated in the processing described in this section.

D.3.4.1 Package Real_Elementary_Functions

```
with Numerics.Generic_Elementary_Functions;
with Geometry;
package Real_Elementary_Functions is
   new Numerics.Generic_Elementary_Functions(Geometry.Real);
```

D.3.4.2 Math Utilities

```
with Geometry; use Geometry;
package Math_Utilities is
   Degrees_Per_Radian : constant := 180.0 / Numerics.Pi;

   Math_Error : exception;

   function Distance (Point_1, Point_2 : Real) return Real;

   function Angle (Origin, Destination : Point) return Degrees;
   -- Measures the angle made by a ray from Origin as it is rotated
   --  until it reaches Destination
   -- Returns a result in the interval (-180.0, 180.0]
   -- If the rotation is counterclockwise, the sign is positive
   -- If the rotation is clockwise, the sign is negative

   procedure Get_X_Intercept(End_1, End_2 : Point;
                             X_Intercept  : out Real;
                             Parallel     : out Boolean);
   -- If the line defined by the endpoints End_1, End_2 are
   --  parallel to the X-axis, then set Parallel to True
   -- If the line defined by the endpoints is not parallel
   --  to the X-axis, set Parallel to False, and set X_Intercept
   --  to the value where the line intersects the X-axis
```

```
   -- Note: the intercept might not be on the line segment
   -- Precondition: End_1 /= End_2 (else Math_Error raised)

end Math_Utilities;

with Real_Elementary_Functions;
package body Math_Utilities is
   use Real_Elementary_Functions;

   function Distance (Point_1, Point_2 : Point) return Real is
   begin
      return Sqrt( (Point_2.X - Point_1.X)**2 +
                   (Point_2.Y - Point_1.Y)**2 );
   end Distance;

   function Angle (Origin, Destination : Point) return Degrees is
      Radians : Real;
   begin
      Radians := Arctan (Y => Destination.Y - Origin.Y,
                         X => Destination.X - Origin.X);
      return Radians * Degrees_Per_Radian;
   end Angle;

   procedure Get_X_Intercept(End_1, End_2 : Point;
                             X_Intercept  : out Real;
                             Parallel     : out Boolean) is
      Parallel : Boolean := False;
      m, b     : Real;  -- Slope, Y-Intercept
   begin
      if End_1 = End_2 then
         raise Math_Error;
      elsif End_1.Y = End_2.Y then
         Parallel := True;
         return;
      elsif End_1.X = End_2.X then
         X_Intercept := End.X;
         return;
      else
         -- Equation is y=mx+b
         -- b = y1 - m*x1
         -- x = (y-b)/m
         -- x-intercept is thus -b/m
         m := (End_2.Y - End_1.Y)/(End_2.X-End_1.X)
         b := End_1.Y - m * End_1.X;
         X_Intercept := -b/m;
       end if;
    end Get_X_Intercept;

end Math_Utilities;
```

D.3.4.3 Body of Package Relations

```
with Real_Elementary_Functions; use Real_Elementary_Functions;
with Geometry; use Geometry;
with Ship_Pckg.Ownship_Pckg, Ship_Pckg.Carrier_Pckg;
use Ship_Pckg.Ownship_Pckg, Ship_Pckg.Carrier_Pckg;
with Track_Pckg.Hostile_Pckg, Track_Pckg.Commercial_Pckg;
use Track_Pckg.Hostile_Pckg, Track_Pckg.Commercial_Pckg;
package body Relations is

   function In_Weapon_Doctrine (Target : Track_Ref;
                                Vessel : Ownship)
   return Boolean is
      Target_Position : Track_Position_Type :=
         Position(Target.all);  -- Dispatches
      Vessel_Position : Ship_Position_Type :=
         Position(Vessel);

      Radius : Real    := Weapon_Doctrine(Vessel).Radius;
      Angle  : Degrees := Weapon_Doctrine(Vessel).Angle;
      Result : Boolean;
   begin
      Result :=
        Distance(Target_Position, Vessel_Position) <= Radius and then
        Angle(Origin => Vessel_Position, Destination => Target_Position)
          not in 180.0 - Angle .. 180.0 + Angle;
      return Result;
   end In_Weapon_Doctrine;

   function In_Engageability_Zone (Target : Track;
                                   Vessel : Ownship)
   return Boolean is
      Target_Position : Track_Position_Type :=
         Position(Target.all);  -- Dispatches
      Vessel_Position : Ship_Position_Type :=
         Position(Vessel);
      Inner_Radius : Real := Engageability_Zone(Vessel).Inner_Radius;
      Outer_Radius : Real := Engageability_Zone(Vessel).Outer_Radius;
   begin
      return Distance(Target_Position, Vessel_Position) in
         Inner_Radius .. Outer_Radius;
   end In_Engageability_Zone;

   function In_Slaved_Doctrine (Target : Track;
                                Vessel : Carrier)
   return Boolean is
      Target_Position : Track_Position_Type :=
         Position(Target.all);  -- Dispatches
      Vessel_Position : Ship_Position_Type :=
         Position(Vessel);
   begin
      return Distance(Target_Position, Vessel_Position) <= Radius;
   end In_Slaved_Doctrine;
```

```
function In_Tight_Zone (Target : Track;
                        Region : Tight_Zone)
return Boolean is
-- Translate coordinates so that Target is (0.0, 0.0)
-- Draw a ray from the origin to +infinity, along the x-axis
-- Count the number of region segments with which the ray intersects
-- The target point is in the region if and only if that number
--   is odd
   Translated_Boundary : Boundary_Type;
   Trandlated_Boundary_Length : Positive;
   Translated_Region   : Tight_Zone;
   Intersections       : Natural;
   X_Intercept         : Real;
   Parallel            : Boolean;
begin
   -- First extend boundary so that last point is first point
   Translated_Boundary_Length := Boundary(Region).all'Length+1;
   Translated_Boundary :=
      new T_Zone_Array(Translated_Boundary_Length);
   Translated_Boundary.all(1..Translated_Boundary_Length-1) :=
      Boundary(Region).all;
   Translated_Boundary.all(Translated_Boundary_Length) :=
      Translated_Boundary.all(1);

   -- Now translate to put Target at (0.0, 0.0)
   for I in Translated_Boundary.all'Range loop
      Translated_Boundary.all(I).X :=
         Translated_Boundary.all(I).X - Position(Target).X;
      Translated_Boundary.all(I).Y :=
         Translated_Boundary.all(I).Y - Position(Target).Y;
   end loop;

   Intersections := 0;
   for I in Translated_Boundary.all'First+1 ..
            Translated_Boundary.all'Last loop

      X1 := Translated_Boundary.all(I-1).X;
      X2 := Translated_Boundary.all(I).X;

      Y1 := Translated_Boundary.all(I-1).Y;
      Y2 := Translated_Boundary.all(I).Y;

      Get_X_Intercept(End_1 => (X1, Y1),
                      End_2 => (X2, Y2),
                      X_Intercept => X_Intercept,
                      Parallel    => Parallel);
      if not Parallel then
         if X_Intercept > 0 and then
            X_Intercept in Real'Min(X1, X2) .. Real'Max(X1, X2)
         then
            Intersections := Intersections+1;
         end if;
      else
```

```
            -- If the segment is along the x-axis itself
            -- then check if it includes the origin
            if Y1 = 0.0 and then
               0.0 in Real'Min(X1, X2) .. Real'Max(X1, X2)
            then
               Intersections := Intersections+1;
            end if;
         end if;
      end loop;
      return (Intersections mod 2) = 1; -- True if odd, False if even
   end In_Tight_Zone;

end Relations;
```

D.4 Specific Prototyping Example

```
with Geometry; use Geometry;
package Simulation_Script is

   type Initial_Data_Record is
      record
         Aegis           : Ship_Position;
         Weapon_Doctrine : W_Doctrine_Type;
         Engagement_Zone : E_Zone_Type;
         Carrier         : Ship_Position;
         Slaved_Doctrine : S_Doctrine_Type;
         Commercial      : Track_Position;
         Hostile         : Track_Position;
         Tight_Zone      : Boundary_Type;
      end record;

   Initial_Data : Initial_Data_Record :=
      ( Aegis           => (X => 113.0, Y => 64.0),
        Weapon_Doctrine => (Radius => 59.0, Degrees => 60.0),
        Engagement_Zone => (Inner_Radius => 22.0,
                            Outer_Radius => 44.0),
        Carrier         => (X => 180.0, Y => 40.0),
        Slaved_Doctrine => (Radius => 40.0),
        Commercial      => (X => 38.0,  Y => 25.0),
        Hostile         => (X => 258.0, Y => 183.0),
        Tight_Zone      => new T_Zone_Array'(
                             (X =>   0.0, Y =>  5.0),
                             (X =>   0.0, Y => 25.0),
                             (X => 118.0, Y => 62.0),
                             (X => 259.0, Y => 25.0),
                             (X => 259.0, Y =>  5.0),
                             (X => 118.0, Y => 32.0) ) ;
```

```
   type Position_Data_Record is
      record
         Aegis      : Ship_Position;
         Carrier    : Ship_Position;
         Commercial : Track_Position;
         Hostile    : Track_Position;
      end record;

   Position_Data : array (Positive range <>)
                   of Position_Data_Record :=

      (1 => (Aegis      => (X => 133.0, Y =>  64.0),
             Carrier    => (X => 180.0, Y => 119.0),
             Commercial => (X => 100.0, Y =>  43.0),
             Hostile    => (X => 210.0, Y => 136.0) ),

      (2 => (Aegis      => (X => 123.0, Y =>  64.0),
             Carrier    => (X => 180.0, Y => 130.0),
             Commercial => (X =>  58.0, Y =>  30.0),
             Hostile    => (X => 239.0, Y => 164.0) ),

      (3 => (Aegis      => (X => 163.0, Y =>  64.0),
             Carrier    => (X => 180.0, Y =>  81.0),
             Commercial => (X => 159.0, Y =>  38.0),
             Hostile    => (X => 148.0, Y =>  73.0) ),

      (4 => (Aegis      => (X => 192.0, Y =>  64.0),
             Carrier    => (X => 180.0, Y =>  60.0),
             Commercial => (X => 198.0, Y =>  27.0),
             Hostile    => (X => 110.0, Y =>  37.0) ) );

   Time_Periods : constant Natural := Position_Data'Length;

end Simulation_Script;

with Calendar; use Calendar;
with Simulation_Script; use Simulation_Script;
with Strings.Unbounded; use Strings.Unbounded;
with Geometry; use Geometry;
with Ship_Pckg.Ownship_Pckg, Ship_Pckg.Carrier_Pckg;
use Ship_Pckg.Ownship_Pckg, Ship_Pckg.Carrier_Pckg;
with Track_Pckg.Hostile_Pckg, Track_Pckg.Commercial_Pckg;
use Track_Pckg.Hostile_Pckg, Track_Pckg.Commercial_Pckg;
procedure Simulation is

   The_Aegis_Ship, The_Carrier : Ship_Ref;
   The_Tight_Zone              : Tight_Zone_Ref;
   The_Commercial_Aircraft, The_Hostile_Aircraft : Track_Ref;

   task type Track_Monitor is
      entry Initialize (Identity : Track_Ref;
                        Period   : Integer);
   end Track_Monitor;
```

```
Commercial_Monitor, Hostile_Monitor : Track_Monitor;

task Position_Updater is
   entry Set_Period(Seconds : Integer);
end Position_Updater;

task Output_Manager is
   entry Put_Line(Item : Unbounded_String);
end Output_Manager;

task body Position_Updater is
   Next_Time : Time;
   Period    : Duration;
begin
   accept Set_Period(Seconds : Integer) do
      Period := Duration(Seconds);
   end Set_Period;
   Next_Time := Clock;
   for I in 1..Time_Periods loop
      Set_Position(The_Aegis_Ship, Position_Data(I).Aegis);
      Set_Position(The_Carrier, Position_Data(I).Carrier);
      Set_Position(The_Commercial_Aircraft,
                   Position_Data(I).Commercial);
      Set_Position(The_Hostile_Aircraft,
                   Position_Data(I).Hostile);
      Next_Time := Next_Time+Period;
      delay until Next_Time;
   end loop;
end Position_Updater;

task body Track_Monitor is -- per track
   Next_Time : Time;
   Identity  : Track_Ref;
   Period    : Duration;
   Output_Message : Unbounded_String;
begin
   accept Initialize(Identity : Track_Ref;
                     Period   : Integer) do
      Track_Monitor.Identity := Identity;
      Track_Monitor.Period   := Duration(Period);
   end Initialize;

   Next_Time := Clock;

   for I in 1..Time_Periods loop

      if In_Weapon_Doctrine(Identity, The_Aegis_Ship) then
         Output_Message :=
            To_Unbounded_String (Integer'Image(I) & ": " &
                                 Kind(Identity) &
                                 " in Weapon Doctrine" );
         Output_Manager.Put_Line(Output_Message);
      end if;
```

```
            if In_Engagement_Zone(Identity, The_Aegis_Ship) then
               Output_Message :=
                  To_Unbounded_String (Integer'Image(I) & ": " &
                                       Kind(Identity) &
                                       " in Engagement Zone" );
               Output_Manager.Put_Line(Output_Message);
            end if;

            if In_Slaved_Doctrine(Identity, The_Carrier) then
               Output_Message :=
                  To_Unbounded_String (Integer'Image(I) & ": " &
                                       Kind(Identity) &
                                       " in Slaved Doctrine" );
               Output_Manager.Put_Line(Output_Message);
            end if;

            if In_Tight_Zone(Identity, The_Tight_Zone) then
               Output_Message :=
                  To_Unbounded_String (Integer'Image(I) & ": " &
                                       Kind(Identity) &
                                       " in Tight Zone" );
               Output_Manager.Put_Line(Output_Message);
            end if;

            Next_Time := Next_Time+Period;
            delay until Next_Time;
         end loop;
      end Track_Monitor;

      task body Output_Manager is
         Local : Unbounded_String;
      begin
         loop
            select
               accept Put_Line (Item : Unbounded_String) do
                  Local := Item;
               end Put_Line;
               Text_IO.Put_Line(To_String(Local));
            or
               terminate;
            end select;
         end loop;
      end Output_Manager;

   begin
      The_Aegis_Ship :=
         New_Ownship(Position =>
                        Initial_Data.Aegis,
                     Weapon_Doctrine =>
                        Initial_Data.Weapon_Doctrine,
                     Engageability_Zone =>
                        Initial_Data.Engageability_Zone);
```

```
   The_Carrier :=
      New_Carrier(Position =>
                     Initial_Data.Carrier,
                  Slaved_Doctrine =>
                     Initial_Data.Slaved_Doctrine);

   The_Commercial_Aircraft :=
      New_Commercial(Position => Initial_Data.Commercial);

   The_Tight_Zone :=
      New_Tight_Zone(Boundary => Initial_Data.Tight_Zone);

   Position_Updater.Set_Period(Seconds => 2);

   Commercial_Monitor.Initialize(Commercial_Monitor, 2)
   Hostile_Monitor.Initialize(Hostile_Monitor, 2)
end Simulation;
```

D.5 Conclusions

Although Ada 9X compilers are not yet at the implementation level that would allow us to run the code displayed above, a manual execution step-through gives the following output:

```
1: Commercial in Tight Zone
2: Commercial in Tight Zone
3: Commercial in Engageability Zone
3: Commercial in Tight Zone
3: Hosile in Slave Doctrine
4: Commercial in Weapon Doctrine
4: Commercial in Engageability Zone
4: Commercial in Tight Zone
4: Hostile in Slave Doctrine
5: Commercial in Weapon Doctrine
5: Commercial in Engageability Zone
4: Commercial in Slave Doctrine
5: Commercial in Tight Zone
5: Hostile in Tight Zone
```

As an example of the adaptability of the design, it is straightforward to enhance the Track_Monitor implementation so that it maintains a history of positions (versus just using the most recent position) or so that it performs engageability analysis. These enhancements are localized; no other modules are affected. Moreover, an Ada program comprising periodic tasks is amenable to analysis through results from mathematical scheduling theory (in particular, Rate Monotonic Analysis), allowing the system designer to predict in advance whether it is feasible for deadlines of all the periodic tasks to be met.

In summary, the Ada 9X solution offers flexibility and has the software engineering benefits of packages for modularization, private and tagged types for data abstraction and object-oriented programming, and tasking and protected types for concurrency. As an upcoming standard, it also provides portability.

E A Relational LISP Solution to the AEGIS Wepons System (AWS) Prototyping Demonstration Project

Relational Abstraction Solution for ProtoTech HiPer-D Joint Demonstration Problem

USC Information Sciences Institute
(part of the SOYII ProtoTech Team)

Abstract

Relational Abstraction is a prototyping language extension for constructing executable functional prototypes with delayable implementation commitments, and for migrating these functional prototypes into production code via declarative choices for those delayed implementation commitments. These delayable commitments include data representation selection, what intermediate results are saved, when constraints are checked, and the order in which expressions are evaluated.

Relational Abstraction employs a database metaphor in which relations are used to model the logical structure of the data, the data is accessed associatively via queries, transactions encapsulate compound updates, constraints ensure the well-formedness of the data after each transaction, and computations can be activated (triggered) by those transactions.

Relational Abstraction has been implemented and used as a Lisp extension for many years and is currently being implemented as an extension to Ada and C++.

Enclosed below is our solution to the ProtoTech HiPer-D Joint Demonstration Problem in Relational Lisp. This solution was produced and debugged in three hours and successfully run on the example data sets. With a few declarative implementation choices, its performance was improved by a factor of three. Moreover, without modification, it handles possible enhancements B (asynchronous receipt of tracks in bundles), D (dynamically added, modified, and/or deleted regions), and J (slaved doctrine extended to aircraft).

It is particularly instructive to understand that coverage of the asynchronous track receipt and dynamic region definition enhancements results directly from the reactive nature of Relational Abstraction. A trigger (automation rule) was created for a tracked-object entering a geometric-region, and this region-entering condition was defined by a set of logic formulas — one for each type of region. This automation rule fires no matter how its trigger becomes true. The two enhancements just altered how this condition might become true (one by dynamically creating, changing, or deleting regions, and the other by altering the timing of location updates for tracked-objects).

E.1 Relational Abstraction Concepts

In Relational Abstraction relations are used to model the various attributes of an object. Some of these attributes, such as the location of tracked-objects, are primitive and are changed by direct update. Other attributes, such as the location of a doctrine, are defined logically in terms of other relational data (either primitive or itself logically defined). In the case of doctrine-location, it is defined as the

location of the tracked-object over which the doctrine is placed (which we name the covered-object). Once this logical relationship is established, the doctrine-location will always be derived by this formula whenever it is needed so that it will always be the same as the location of that covered-object as that object moves around (or the doctrine is placed over some other tracked-object).

This is a very powerful feature because higher and higher level attributes can be defined in terms of lower level ones. For example, the bearing of the center of the wedge in a weapon doctrine is defined as the sum of the weapon doctrine's threat axis and the offset of the wedge from this threat axis. The bearings of the two edges of the wedge are then defined as the sum and difference of this bearing and the angle which is the size of half the wedge. These bearings are then in turn part of the determination of whether a given bearing is inside or outside this wedge. Finally, in-weapon-doctrine is defined as those tracked-objects whose location is in the circle defined by the radius of the weapon-doctrine and is outside the weapon doctrine's wedge.

Thus, as objects move (or the primitive attributes of a weapon doctrine are changed) the in-weapon-doctrine attribute is **automatically rederived as needed.** In a similar way, in-slave-doctrine, in-engagability-region, and in-tight-zone are defined as derived logical formulas. These four attributes are combined to define the top level attribute in-region which defines which objects are in which geometric-regions.

Since in this problem we need to know when tracked-objects enter into a geometric-region, we define an automation-rule which triggers whenever this attribute (relation) becomes true. Its body will be executed each time any tracked-object enters any geometric-region. These automation rules can have the start, end, or change in the truth-value of any primitive, derived, or compound relation (i.e. one defined by a quantified well-formed formula) as its trigger. Thus, we could easily change the solution to react to tracked-objects leaving geometric-regions, or have it do both.

The imperative portion of this solution is very, very small. It consists only of the code to reposition the tracked-objects (in response to data theoretically coming from the sensors) and the automation rule body which reacts to tracked-objects entering geometric-regions (here just notifying the user). **Everything else is declarative — the type hierarchy, the primitive relations between the types, and derived relations such as the formulas by which in-region is defined.**

Most of this declarative information is expressed in quantified well-formed formulas that are fully understood by the Relational Compiler and can be used multi-directionally to find and test the objects satisfying the relation (that is, (R X Y) can be used to find Ys given an X, or find Xs given a Y, or find the Xs and Ys given neither, or to test the relationship given both an X and a Y). However, some of this declarative information is more naturally expressed as Lisp predicates or Lisp functions which can act as testers of a relation or as generators which find the objects occupying some set of columns of the relation given values for the objects occupying the other columns of the relation. One such computationally defined relation is tight-zone-tester which uses a library routine (actually transliterated from C) to test whether a point is within a polygon. Another computationally defined relation is +. This relation uses the built-in Lisp *plus* function to sum two numbers, and the built-in Lisp *minus* function to determine one of the addends given the other addend and the sum.

Thus, any functional (non-state changing) computation can be relationally encapsulated and used like any other derived relation. Similarly, functions and procedures (state-changing routines) can be supplied to perform the access and update operations for a relation. This is how different representations are chosen for relations. In this example, we added a declaration that the representation of the X and Y attributes of positions should be fields of a record rather than the default representa-

tion of relations (a linked list). These implementation declarations resulted in a three-fold increase in performance.

E.2 Demonstration Problem Solution

This is the executable code for the demonstration problem expressed in Relational Lisp. Timings for its execution and the performance improvement obtained by adding declarative optimization choices are given in the Collected Data section.

Commentary has been added to the code to describe the Relational Abstraction notation and its semantics.

E.2.1 Data Declarations

In this section we define the modeled objects and their attributes.

Define a new type named *position* and has required attributes x and y which are of type *number*. Similarly define *tracked-object* with a required attribute *location* of type *position*, and *Aegis-ship*, *carrier*, *commercial-airliner*, and *hostile-aircraft* as subtypes of *tracked-object* without any additional attributes. These subtypes inherit the *location* attribute defined on their parent type *tracked-object*.

```
(define-type position () x number y number)
(define-type tracked-object () location position)
(define-type Aegis-ship tracked-object)
(define-type carrier tracked-object)
(define-type commercial-airliner tracked-object)
(define-type hostile-aircraft tracked-object)
```

Define *geometric-region* as a new type (it will become the supertype of *doctrine*, *engagability-region*, and *tight-zone*. *Doctrine* in turn will become the supertype of *weapon-doctrine* and *slave-doctrine*).

```
(define-type geometric-region ())
(define-type doctrine geometric-region)
```

Define required attributes *covered-object* (the object at the center of the doctrine), *ownership* (the owner of the doctrine), *radius*, and *doctrine-location* for *doctrine*. *Doctrine-location* is derived as the location of the object which is the *covered-object* of the *doctrine*. These attributes are inherited by the subtypes of *doctrine* (i.e. *weapon-doctrine* and *slave-doctrine*).

```
(define-type doctrine geometric-region covered-object tracked-object
        ownership tracked-object radius number)
(defattribute doctrine-location doctrine position
        :definition ((doctrine position)
                s.t. (E (tracked-object)
                        (and (covered-object doctrine tracked-object)
                             (location tracked-object position)))))
```

We restrict the *ownership* of *weapon-doctrine* to be the *covered-object* and require this not be true of *slave-doctrine*.

```
(alwaysrequired ownership-equals-covered-object
        (A (weapon-doctrine#)
          (E (tracked-object)
```

```
                (and (ownership weapon-doctrine tracked-object)
                     (covered-object weapon-doctrine tracked-object)))))
    (neverpermitted ownership-equals-covered-object-in-slaved-doctrine
                    (E (slave-doctrine# tracked-object#)
                       (and (ownership slave-doctrine tracked-object)
                            (covered-object slave-doctrine tracked-object))))
```

Define *angle* and *compass-direction*[1] as types whose range is restricted to be between 0 and 360. These types are used to restrict the valid range values of the attributes that follow.

```
    (defrelation angle :definition ((x) s.t. (and (>= x 0) (< x 360))))
    (defrelation compass-direction :definition ((x) s.t. (and (>= x 0) (< x 360))))
```

Define *threat-axis*, *engagement-region*, *removed-wedge-half-size*, and *removed-wedge-offset* as primitive attributes of *weapon-doctrine*, and *removed-wedge-direction*, *wedge-min-bearing*, and *wedge-max-bearing* as derived attributes of *weapon-doctrine*. The removed wedge attributes (*removed-wedge-half-size* and *removed-wedge-offset* are optional, but they must both either be present or absent.

```
    (define-type weapon-doctrine doctrine threat-axis compass-direction
                 engagement-region engagability-region removed-wedge-half-size angle :optional
                 removed-wedge-offset angle :optional)
    (defattribute removed-wedge-direction weapon-doctrine compass-direction
      :definition ((weapon-doctrine wedge-direction)
                   s.t. (E (wedge-offset threat-direction)
                           (and (removed-wedge-offset weapon-doctrine wedge-offset)
                                (threat-axis weapon-doctrine threat-direction)
                                (+ threat-direction wedge-offset wedge-direction)))))
    (defattribute wedge-min-bearing weapon-doctrine compass-direction
      :definition ((weapon-doctrine wedge-min-bearing)
          s.t. (E (wedge-direction angle)
                  (and (removed-wedge-direction weapon-doctrine wedge-direction)
                       (removed-wedge-half-size weapon-doctrine angle)
                       (- wedge-direction angle wedge-min-bearing)))))
    (defattribute wedge-max-bearing weapon-doctrine compass-direction
      :definition ((weapon-doctrine wedge-max-bearing)
          s.t. (E (wedge-direction angle)
                  (and (removed-wedge-direction weapon-doctrine wedge-direction)
                       (removed-wedge-half-size weapon-doctrine angle)
                       (+ wedge-direction angle wedge-max-bearing)))))
    (alwaysrequired legitimate-removed-wedge-description
      (A (weapon-doctrine#)
        (equiv (removed-wedge-half-size weapon-doctrine $)
               (removed-wedge-offset weapon-doctrine $)))))
```

Define *min-radius* and *max-radius* as attributes of *engagability-region* and require that the former is always less than the latter.

```
    (define-type engagability-region geometric-region min-radius number max-radius number)
    (neverpermitted max-radius-less-than-min-radius
```

[1] *Compass-direction* has an implicit orientation as degrees from North.

```
(E (engagability-region# max-radius min-radius)
  (and (max-radius engagability-region max-radius)
       (min-radius engagability-region min-radius)
       (< max-radius min-radius))))
```

Define *range* and *bearing* as computed relations which given two positions respectively derives the distance and angle between them.

```
(defrelation range :types (position position number)
 :derivation (lisp-function
              (lambda (pos1 pos2)
                (let ((delta-x (- (x pos1) (x pos2)))
                      (delta-y (- (y pos1) (y pos2))))
                  (sqrt (+ (* delta-x delta-x) (* delta-y delta-y))))) 2 1))
(defconstant degrees-per-radian 57.2958)
(defrelation bearing :types (position position angle)
 :derivation (lisp-function
              (lambda (pos1 pos2)
                (let ((delta-x (- (x pos1) (x pos2)))
                      (delta-y (- (y pos1) (y pos2))))
                  (* (atan delta-x delta-y) degrees-per-radian))) 2 1))
```

Define the *directed-line* type with attributes *start-endpoint*, *end-endpoint*, and *line-slope* (a derived attribute which given a *directed-line*, *Line-slope* computes its slope) which will be used to form the polygon that defines a *tight-zone* and is held in the *tight-zone-lines* attribute of *tight-zone*. This attribute is set by the procedure that creates tight zones from a list of its vertices.

```
(define-type directed-line () start-endpoint position end-endpoint position)
(defattribute line-slope directed-line number
 :derivation (lisp-function
              (lambda (directed-line)
                (let ((p1 (start-endpoint directed-line))
                      (p2 (end-endpoint directed-line)))
                  (/ (- (y p2) (y p1)) (- (x p2) (x p1))))) 1 1))
(define-type tight-zone geometric-region tight-zone-lines list)
(defun define-tight-zone (point-list)
 (atomic
  (let ((points (cdr point-list))
        (prev (new-position (car (car point-list)) (cadr (car point-list))))
        (tight-zone (create tight-zone))
        line-sequence)
    (loop for next in points
          for next-pos = (new-position (car next) (cadr next))
          for line = (create directed-line start-endpoint = prev
                                           end-endpoint = next-pos)
          do (push line line-sequence)
             (setq prev next-pos))
    (++ tight-zone-lines tight-zone (reverse line-sequence))
    tight-zone)))
```

Now we define the relations that derive whether a *tracked-object* is in a *geometric-region.* A separate definition exists for each type of *geometric-region. In-circle* is used in the definition of several of these regions. It determines whether the first *position* is inside of a circle with its origin at the second *position* and radius *number.* It does so by determining whether the distance between the two positions is less than the circle's radius.

```
(defrelation in-circle :types (position position number)
 :definition ((object-position circle-origin circle-radius)
              s.t. (E (distance)
                      (and (range object-position circle-origin distance)
                           (< distance circle-radius)))))
```

In-slave-doctrine determines whether a *tracked-object* is in a *slave-doctrine* by seeing whether the *location* of the *tracked-object* is inside of the circle whose origin is the *location* of the *slave-doctrine* and whose radius is the *radius* of the *slave-doctrine.*

```
(defattribute in-slave-doctrine tracked-object slave-doctrine
 :definition ((tracked-object# slave-doctrine#)
              s.t. (E (doctrine-loc tracked-object-loc doctrine-radius)
                      (and (doctrine-location slave-doctrine doctrine-loc)
                           (location tracked-object tracked-object-loc)
                           (radius slave-doctrine doctrine-radius)
                           (in-circle tracked-object-loc doctrine-loc doctrine-radius)))))
```

Similarly, *in-engagability-region* determines whether a *tracked-object* is in a *engagability-region* by seeing whether the *location* of the *tracked-object* is inside of the circle whose origin is the *location* of the *weapon-doctrine* for which this is the *engagement-region* and whose radius is the *max-radius* of the *engagability-region*, and is outside of the circle whose radius is the *max-radius.*

```
(defattribute in-engagability-region tracked-object engagability-region
 :definition ((tracked-object# engagability-region#)
              s.t. (E (weapon-doctrine max-radius min-radius doctrine-loc tracked-object-loc)
                      (and (engagement-region weapon-doctrine engagability-region)
                           (doctrine-location weapon-doctrine doctrine-loc)
                           (location tracked-object tracked-object-loc)
                           (max-radius engagability-region max-radius)
                           (in-circle tracked-object-loc doctrine-loc max-radius)
                           (min-radius engagability-region min-radius)
                           (not (in-circle tracked-object-loc doctrine-loc min-radius))))))
```

In-weapon-doctrine tests whether the *tracked-object* is inside the circle defined by the *radius* of the *doctrine* and outside of the wedge removed from the *in-weapon-doctrine. Outside-doctrine-wedge* sees whether the bearing of the *tracked-object* from the *doctrine-location* is between (proceeding clockwise) the max and min bearings of the wedge (that is outside the wedge). *Clockwise-between* is defined computationally.

```
(defun test-clockwise-between (bearing start end)
 (let ((heading (mod bearing 360))
       (nstart  (mod start 360))
       (nend    (mod end 360)))
  (if (> nend nstart)
      (and (<= nstart heading) (<= heading nend))
```

```
    ;; 360 break occurs between start and end
    (or (>= heading nstart) (<= heading nend)))))
(defrelation clockwise-between :types (compass-direction compass-direction compass-direction)
 :computation (lisp-predicate
               (lambda (bearing wedge-max-bearing wedge-min-bearing)
                (test-clockwise-between bearing wedge-max-bearing wedge-min-bearing))))
(defattribute outside-doctrine-wedge tracked-object weapon-doctrine
 :definition ((tracked-object weapon-doctrine) s.t.
   (E (doctrine-loc tracked-object-loc)
     (and (doctrine-location weapon-doctrine doctrine-loc)
          (location tracked-object tracked-object-loc)
          (or (eql doctrine-loc tracked-object-loc)
              (E (direction wedge-max-bearing wedge-min-bearing)
                (and(bearing tracked-object-loc doctrine-loc direction)
                    (wedge-max-bearing weapon-doctrine wedge-max-bearing)
                    (wedge-min-bearing weapon-doctrine wedge-min-bearing)
                    (clockwise-between direction wedge-max-bearing wedge-min-bearing))))))))
(defattribute in-weapon-doctrine tracked-object weapon-doctrine
 :definition ((tracked-object# weapon-doctrine#)
   s.t. (E (doctrine-loc tracked-object-loc doctrine-radius)
          (and (doctrine-location weapon-doctrine doctrine-loc)
               (location tracked-object tracked-object-loc)
               (radius weapon-doctrine doctrine-radius)
               (in-circle tracked-object-loc doctrine-loc doctrine-radius)
               (implies (removed-wedge-half-size weapon-doctrine $)
                        (outside-doctrine-wedge tracked-object weapon-doctrine)))))))
```

In-tight-zone determines whether a *tracked-object* is inside a *tight-zone* by invoking a library routine (transliterated from C) which computes whether the *location* of the *tracked-object* is inside of the polygon which defines the *tight-zone.*

```
(defun point-in-polygon (pnt poly)
 "Returns T if point PNT is inside polygon POLY.  A polygon is represented
as a list of three or more line segments forming a closed loop."
 (let ((targetX (x pnt))
       (targetY (y pnt))
       (polytail poly)
       polyedge inside-p)
   (labels ((xcrossed (edge &aux (side (>= (x (start-endpoint edge)) targetX)))
              ;; returns two booland values.
              ;; first is true iff EDGE horizontally crosses targetX
              ;; (one end must be strictly to left, other at or to right)
              ;; second is true iff EDGE begins at or to right of targetX
              (values
               (not (eq side (>= (x (end-endpoint edge)) targetX)))
               side))
            (process-crossing-edge (edge)
```

```
            ;; edge is one which crosses the target y coord.
            (multiple-value-bind (xcrossing side)  (xcrossed edge)
              (cond
                (xcrossing
                 ;; edge cannot be vertical
                 (when
                   (< targetX
                     ;; xcoordinate of y crossing
                     (- (x (start-endpoint edge))
                       (/  (- (y (start-endpoint edge)) targetY)
                           (line-slope edge))))
                   ;; y coord crossing is strictly RIGHT of targetX
                   (setf inside-p (not inside-p))))
                (side;; y-crossing occurred to the RIGHT of target point
                 (setf inside-p (not inside-p)))))))
        (loop named outer while polytail do
          (cond
            ((>= (y (start-endpoint (first polytail))) targetY)
             ;; follow edges until we drop below targetY
             (loop  while (>= (y (end-endpoint (setf polyedge (pop polytail)))) targetY)
                do (unless polytail (return-from outer nil)))
             (process-crossing-edge polyedge))
            (t          ;; follow edges until we rise above targetY
             (loop while (< (y (end-endpoint (setf polyedge (pop polytail)))) targetY)
                do (unless polytail (return-from outer nil)))
             (process-crossing-edge polyedge)))))
        inside-p))
    (defrelation tight-zone-tester :types (position tight-zone)
      :computation (lisp-predicate
                    (lambda (position tight-zone)
                      (point-in-polygon position (tight-zone-lines tight-zone)))))
    (defrelation in-tight-zone :types (tracked-object tight-zone)
      :definition ((tracked-object# tight-zone#)
                   s.t. (E (position)
                           (and (location tracked-object position)
                                (tight-zone-tester position tight-zone))))))
```

Finally, *in-region* is merely the logical-or of each of the types of *geometric-region.*[2]

```
    (defattribute in-region tracked-object geometric-region
      :definition ((tracked-object geometric-region)
                   s.t. (or (in-slave-doctrine tracked-object geometric-region)
                            (in-weapon-doctrine tracked-object geometric-region)
                            (in-engagability-region tracked-object geometric-region)
```

[2]Polymorphism allows the proper in-region definition to be used for each type of region — that is, the type of region determines which in-region calculation is appropriate.

(in-tight-zone tracked-object geometric-region))))

E.2.2 Imperative Code

We define the automation rule that will trigger when a *tracked-object* enters a *geometric-region* (that is when *in-region* starts to be true). It's body just prints out a notification for the user.

```
(defautomation notify-operator-about-region-entry
  ((tracked-object geometric-region) s.t. (start (in-region tracked-object geometric-region)))
  (lambda (tracked-object geometric-region)
    (format T "s̃ entering s̃%" tracked-object geometric-region)))
```

We also define the "driver" code which establishes the initial conditions and updates the location of the *tracked-objects* as new sensor data arrives.

```
(defun new-position (x-coord y-coord) (create position x = x-coord y = y-coord))
(defun move-object (tracked-object x-pos y-pos)
  (atomic (update location of tracked-object to (new-position x-pos y-pos))))
(atomic ;; initial conditions
  (setq obj1 (create Aegis-ship location = (new-position 113 64)))
  (setq obj2 (create carrier location = (new-position 180 140)))
  (setq obj3 (create hostile-aircraft location = (new-position 258 183)))
  (setq obj4 (create commercial-airliner location = (new-position 38 25)))
  (setq region1 (create engagability-region max-radius = 44 min-radius = 22))
  (create weapon-doctrine covered-object = obj1 ownership = obj1
        radius = 59 threat-axis = 90 removed-wedge-half-size = 60
        removed-wedge-offset = 180 engagement-region = region1)
  (create slave-doctrine covered-object = obj2 ownership = obj1 radius = 40)
  (define-tight-zone '((0 5) (0 25) (118 62) (259 25) (259 5) (118 32) (0 5))))
(atomic ;; time increment 1
  (move-object obj1 123 64)
  (move-object obj2 180 130)
  (move-object obj3 239 164)
  (move-object obj4  58  30))
(atomic ;; time increment 2
  (move-object obj1 133 64)
  (move-object obj2 180 119)
  (move-object obj3 210 136)
  (move-object obj4 100  43))
(atomic ;; time increment 3
  (move-object obj1 163  64)
  (move-object obj2 180  81)
  (move-object obj3 148  73)
  (move-object obj4 159  36))
(atomic ;; time increment 4
  (move-object obj1 192  64)
  (move-object obj2 180  60)
  (move-object obj3 110  37)
```

```
(move-object obj4 198 27))
```

Separate Processes

As defined above, the Relational Lisp program will execute as required except that everything will occur in a single process. To allocate the computation to separate processes the automation rule is replaced by the following declarations which set up a sentinel that queues up the tuples that satisfies the triggering predicate (that a *tracked-object* enters a *geometric-region*) and spawn a process that services that queue. Thus the body of the original automation rule now operates in the newly established process.

```
(def-process-sentinel notify-operator-about-region-entry
  ((tracked-object geometric-region)
   s.t. (start (in-region tracked-object geometric-region))))
(fsd-process:spawn-process "operator interaction"
    #'(lambda ()
        (loop ;;infinite loop
          (process-db-wait
            (notify-operator-about-region-entry (tpls)
              ;; tpls is a list of tpls that triggered the sentinel rule
              ;; in a single atomic transaction.
              (loop for (tracked-object geometric-region) in tpls do
                 (format T "s entering s %"
                    tracked-object geometric-region)))))))
```

We use the same mechanism to spawn a process that determines which *tracked-objects* are in which *geometric-regions.* This process services the *track-file* data produced by the sensor process by moving the objects to the location indicated in the track file, which causes the *in-region* derivations to be done, which triggers the sentinel for *tracked-objects* entering *geometric-regions.*

```
(defrelation track-file :types (tracked-object number number)
  :count ((tracked-object output output) :countspec :optional))
(def-process-sentinel process-updated-track-file
  ((tracked-object x-coord y-coord)
   s.t. (start (track-file tracked-object x-coord y-coord))))
(fsd-process:spawn-process "In Region Calculations"
    #'(lambda ()
        (loop ;;infinite loop
          (process-db-wait
            (process-updated-track-file (tpls)
              ;; tpls is a list of tpls that triggered the sentinel rule
              ;; in a single atomic transaction.
              (atomic (loop for (tracked-object x-coord y-coord) in tpls
                          do (move-object tracked-object x-coord y-coord)))))))))
```

E.3 Collected Data

E.3.1 Development Time

Understanding: 20 minutes

Although the problem was well explained ahead of time, detailed questions still arose during modeling (this is inevitable). These questions included, which attributes were required and which were optional, what was the relationship between ownership and slave-doctrines (I assumed that the owner of the slave-doctrine around the carrier was the Aegis-ship), and what was the relationship between weapon-doctrine and engagability-regions (I assumed that they were paired, i.e. each weapon-doctrine had a single engagability-region).

Modeling(Specification): 105 minutes

This was the time to devise the Relational Abstraction model (in Relational Lisp) and enter it into the computer. The model developed was executable but was uncompiled (unanalyzed), undebugged, and unoptimized. It handled all parts of the problem except for the inclusion of the point-in-polygon function (which had not yet been transliterated from the C library) and the separation of the computation into different processes (which we view as part of the implementation (migration).

Debugging: 50 minutes

Compile-Time Errors There were a variety of compile-time errors that were found before execution. These included:

- changing the name of a relation (*doctrine-location*) to eliminate its definition from inadvertently being recursive (it was originally called location and was defined in terms of location (of tracked-objects)).
- Avoiding name conflicts with existing relations (*degree* was already defined, so we renamed our relation to *angle*).
- Misuse of an attribute name (*wedge-direction*) where its type (*compass-direction*) was required.
- Mis-clicking (the modern day equivalent of typos): picking up *threat-direction* instead of *wedge-direction* (this was detected as an unbound variable).
- Real typos: asked to create a *commercial-airline* instead of a *commercial-airliner.*
- Stubbing: Had to define a stub routine for the missing library function point-in-polygon.

Type Errors One type error occurred when I forgot to return the newly created position as the value of the routine new-position. In strongly typed languages such as Ada these types of errors would be found at compile time. In Lisp, such errors aren't detected (if at all) until run-time. In Relational Lisp, this showed up as a run-time type error (relations are type checked unless the user individually disables such checking).

A second type error occurred when I tried to create the initial data state. The transaction aborted with an error from the *ownership-must-not-equal-covered-object-in-slaved-doctrine* constraint that the

owner of the newly created *weapon-doctrine* was not allowed to be the same as the *covered-object* of that *weapon-doctrine.* This rule was being inappropriately being applied to *weapon-doctrines* instead of being limited only to *slave-doctrines.* The fix was to add a type restriction to this constraint.[3]

Run-Time Errors Two semantic errors occurred within the solution which were only found by testing and noticing inappropriate behavior (inaccurate reports positive or negative of objects in regions).

The first concerned a boundary condition — was the point at the center of a weapon-doctrine in that weapon-doctrine? I forgot to consider that case and the way the logic equations were defined made that point outside the weapon-doctrine (by including it in the removed wedge). I thought that was wrong and changed the logic equations to make that point inside the weapon-doctrine.

The second semantic error resulted from sloppiness in the non-relational Lisp function that defined the relation *clockwise-between.* It originally attempted to normalize the compass-directions by ensuring that they were non-negative, neglecting to ensure that they were also less than 360. This partial normalization caused erroneous derivation of the *clockwise-between* relation.

Delayed Optimization: 10 minutes

As we mentioned previously Relational Abstraction allows declarative optimization choices to be made for how relations should be represented and which derived relations (intermediate results) should be cached.

We selected representations (record fields) for the x and y attributes of *position.* This resulted in a three fold increase in performance (the default representation was a linked list which had to be linearly searched for the appropriate tuple). This speed-up would be much larger if realistic amounts of data were being processed. To perform this delayed optimization, we merely added ":representation structprop" to the declarations of the x and y represented)./footnoteIn Relational Abstraction when the representation of a relation changes, the existing data (i.e. tuples) in the relation are extracted from the old representation and put into the new one, and this is done invisibly so that no state change occurs. This is analogous to schema evolution in database systems.

We also decided to cache the values of the derived relations (i.e. intermediate results) *doctrine-location, removed-wedge-direction, wedge-min-bearing,* and *wedge-max-bearing* which resulted in an additional 15% performance improvement. These caching decisions are also declared by adding ":representation structprop" to the relation definitions (we are also selected how the cache should be relations.

E.3.2 Size & Experience

Excluding the library routine for calculating *points-in-polygon* and the various well-formedness constraints (which aren't needed to run and don't have any effect unless bad data is entered and are probably not part of any other submitted solution) the Relational Lisp solution contained 148 lines.

Developer: Robert Balzer **Experience with Formalism:** Expert

[3]In Relational Lisp when such errors occur which abort a transaction, the user can back out of the atomic (reestablishing the state before the transaction began), change the state or the rules in effect (here we corrected the overzealous constraint) and retry the transaction. Thus, we were able to fix the error on the fly and continue execution.

[3]This page intentionally left blank.