TASK: PA18
CDRL: A023
26 February 1994

Verifying Launch Interceptor Routines With the Asymptotic Method

Informal Technical Data

STARS-AC-A023/006/00
26 February 1994

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	26 Feb 1994	Informal Technical Report

4. TITLE AND SUBTITLE
Verifying Launch Interceptor Routines With the Asymptotic Method

5. FUNDING NUMBERS
F19628-93-C-0130

6. AUTHOR(S)
ORA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER
STARS-AC-A023/006/00

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
Headquarters ESC
Hanscom, AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
A023

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes the results of an exercise in formal program verification conducted at ORA. In this exercise, we started with code written by students at Syracuse University as part of work conducted by Professor Amrit Goel on program development methods. The code was an implementation of the routines of the Launch Interceptor Program (LIP), a specification of a protocol for launching an interceptor missile. This specification was used by Knight and Leveson in a well-known experiment in N-version programming

14. SUBJECT TERMS

15. NUMBER OF PAGES
21

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TASK: PA18
CDRL: A023
26 February 1994

INFORMAL TECHNICAL REPORT

For

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Verifying Launch Interceptor Routines With the Asymptotic Method

STARS-AC-A023/006/00
26 February 1994

Data Type: Informal Technical Data

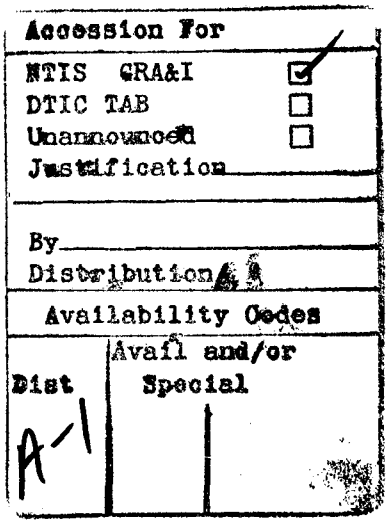
CONTRACT NO. F19628-93-C-0130

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

Odyssey Reserach Associates
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091



TASK: PA18
CDRL: A023
26 February 1994

Data Reference: STARS-AC-A023/006/00
INFORMAL TECHNICAL REPORT
Verifying Launch Interceptor
Routines With the
Asymptotic Method

Distribution Statement "A"
per DoD Directive 5230.24
Authorized for public release; Distribution is unlimited.

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Advanced Research Projects Agency (ARPA) under contract F19628-93-C-0130, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent. The information identified herein is subject to change. For further information, contact the authors at the following mailer address: delivery@stars.reston.paramax.com

TASK: PA18
CDRL: A023
26 February 1994

Data Reference: STARS-AC-A023/006/00
INFORMAL TECHNICAL REPORT
Verifying Launch Interceptor
Routines With the
Asymptotic Method

Principal Author(s):

Douglas N. Hoover *Date*

Daryl M. McCullough *Date*

Approvals:

Teri L. Payton 2/28/94

Program Manager *Teri F. Payton* *Date*

(Signatures on File)

Contents

1 Introduction

This report describes the results of an exercise in formal program verification conducted at ORA. In this exercise, we started with code written by students at Syracuse University as part of work conducted by Professor Amrit Goel on program development methods. The code was an implementation of the routines of the Launch Interceptor Program (LIP), a specification of a protocol for launching an interceptor missile. This specification was used by Knight and Leveson in a well-known experiment in N-version programming [3].

The exercise consisted of specifying the routines using the asymptotic method and verifying a number of them using Penelope [1]. The asymptotic method is a relatively simple approach to specification and verification of numerical programs that takes account of numerical error without quantifying it. It is important to take account of numerical error, even if it is small, because even a small error can change the result of a numerical program if it is a discontinuous function of the input. All non-constant discrete-valued (integer or Boolean) functions of real number arguments are discontinuous, and many real-valued functions appearing in geometry are discontinuous at certain degenerate configurations. When a specification, like the LIP specification, does not say anything quantitative about permissible error, the asymptotic method is the appropriate way to formalize that specification. The asymptotic method exposes problems of this kind, without requiring any more work than is necessary for that purpose. Classical error analysis will expose similar problems and give quantitative measures of how bad they are, but also requires more effort and expertise.

This verification exercise had the following results.

1. A number of weaknesses in the specification were found regarding the treatment of near-degenerate geometrical situations. We adopted what seemed to be a reasonable solution to these problems and adapted the programs accordingly.
2. Some problems were found with the programs in that they used discontinuous formulas to compute continuous quantities, creating problems dealing with exceptions and permitting unpredictable results near singularities.
3. A few other errors were found in the programs. In particular, in one place an incorrect formula was used and in another uninitialized variables were read.
4. Considering the effect of error cast doubt on the method that Knight and Leveson used to test programs for faults, and on their evaluation of some program variants as faulty.
5. We outlined a development method for numerical programs that would appear to address the problems exposed by this verification exercise.

2 The Launch Interceptor Program (LIP)

The Launch Interceptor Program (LIP) is part of the controller for a fictional antimissile system. The program is given input data taken from radar (representing points in the two-

dimensional radar field) and is given a large number of adjustable parameters and computes various functions of the data to determine whether the radar images represent a threat. If so, the program must generate a signal to launch an interceptor. Typical sorts of computations performed on the data include determining whether there are three consecutive points that are not collinear (that is, they make an angle that is sufficiently far from 0 or pi), and determining whether there are three consecutive points that make a triangle whose area falls within certain bounds (as determined by input parameters).

The program specification was used in a study of N-version programming by Knight and Leveson [3]. In this study, the specifications for the program were given to a number of students for the purpose of studying the types of errors made and the correlations in errors made by different programmers. N-version programming depends on a certain independence of the mistakes made by different programmers, so that in case of disagreement between outputs of programs, the correct answer can be obtained by majority vote (or other algorithm). The results of the study showed that the errors were not independent, but instead different students tended to make the same (or related) errors. An error in a program was considered to be any significant deviation of its output from the corresponding output in a standard implementation (called the "Gold" program). There is a problem with this method of testing because a Boolean result of a numerical program can be affected by roundoff error. Using different numerical methods can produce different roundoff errors, leading to different results in marginal cases, leading to disagreement even though neither program is really faulty.

2.1 Structure of the Launch Interceptor Program

There are three parts of the Launch Interceptor Program:

1. Geometrical utility functions

   These are programs for computing various geometrical characteristics of points in the plane, such as the area of a triangle formed by three points, the angle they form, etc.

2. Launch Interceptor Conditions (LICs)

   These are 15 short programs (written as tasks in the example program we looked at) that compute Boolean-valued functions of a sequence of two-dimensional points (representing radar data). Each Boolean function computed is a statement that there exist certain data points in some configuration that can be determined by one of the utility functions.

3. Combining the LICs

   The 15 values returned are combined in pairs according to a Logical Combinations Matrix (LCM) to form a 15 by 15 Boolean-valued matrix called the Preliminary Unlocking Matrix (PUM). Certain rows of the PUM are combined and placed in a Final Unlocking Vector (FUV). Launch is authorized if each entry in the FUV is `true`.

These three parts of the LIP involve different kinds of programming. The utility programs are almost purely numerical. The LICs use the utility programs, plus loops to search through the list of data points. The updating of the PUM is purely a discrete program, not involving floating point at all (and using integers only as matrix indices).

Our experience with the verification exercise was that the problems exposed by the specification using the asymptotic method had mainly to do with the geometric utilities, since those routines were completely numerical. The other two levels contained a smaller proportion of numerical computation, and accordingly the asymptotic method, though still necessary for proper specification and interpretation of the overall program, was less important. In particular, the asymptotic method was not relevant to writing and verifying the routines that performed the combinations of LICs, since those were purely discrete computations.

3 The Asymptotic Method

ORA's Penelope system [6, 1, 5] provides a fully detailed semantics for numerical programming that supports any method of analysis of numerical programs, such as error analysis; but the centerpiece of the system is the *asymptotic method* [7, 2, 6]. We believe that the asymptotic method will be the most useful approach to specifying and verifying numerical programs because it reveals the qualitative effects of numerical error without being too much harder than reasoning as if there were no numerical error. The asymptotic method is easy enough to use that it can be used by programmers who are not experts in error analysis and in situations where detailed error analysis is not warranted.

Informally, the asymptotic method models machine operations on floating point numbers as approximate, rather than exact. The notion of being approximately correct is made rigorous by considering running the same program on a sequence of machines with better and better accuracy. A program is asymptotically correct if any desired accuracy in the output can in principle be obtained by running the program on a sufficiently accurate machine.

In Penelope we indicate that quantities x and y are close by the notation

$$x \approx y \quad (\texttt{x ~~ y in typescript}).$$

The $\approx$ relation is an equivalence relation and is a congruence for continuous functions:

$$x \approx x', y \approx y' \Rightarrow x + y \approx x' + y'.$$

But $\approx$ is not a congruence for discontinuous operations like comparisons:

$$x \approx x',\ y \approx y',\ x \leq y \not\Rightarrow x' \leq y'.$$

In general, when numerical error is possible, discontinuous operations in a specification or program should raise a red flag. They should be avoided when possible, and when they cannot, special care must be taken to make sure that they are handled correctly.

The relation $x \approx y$ does not mean that the difference between x and y is too small to represent on the machine. Rather, in order for it to make sense to use the asymptotic method, the total machine roundoff error expected to occur in a program must be negligible, that is, ≈ 0.

4 The Value of the Asymptotic Method

Using the asymptotic method essentially helps us identify discontinuities in our specifications and programs and either get rid of them or accommodate them. Some of these benefits accrue during the process of specification and some during verification (whether formal or informal).

During specification, discontinuities in the specification will be detected. If the discontinuities are unavoidable, the specification usually must be modified.

As a general rule, if the naive specification is a relation $P(\vec{x}, \vec{y})$, where $\vec{x}$ are the input variables and $\vec{y}$ are the output variables, then the nearest implementable specification is $P'(\vec{x}, \vec{y})$ defined by

$$P'(\vec{x}, \vec{y}) \iff \exists \vec{x'}, \vec{y'} \left(\vec{x} \approx \vec{x'} \, and \, \vec{y} \approx \vec{y'} \, and \, P(\vec{x'}, \vec{y'}) \right).$$

Sometimes a stricter specification is implementable, but to prove such a thing requires reference to details of floating point arithmetic.

For example, a specification:

> "return `true` iff $x \leq y$"

must usually be replaced by

> "return `true` if $x \lesssim y$, `false` if $y \lesssim x$, and either `true` or false if $x \approx y$."

In the verification stage, asymptotic verification will find any instances in which discontinuous formulas are used to compute continuous functions. Such formulas should be replaced by continuous ones.

4.1 Examples: Discontinuous and Unimplementable Requirements

Naive specifications of numerical programs often are unimplementable because they ignore the combined effects of discontinuities and machine error. When modified in the simplest possible way to allow for error, such specifications often do not say enough about how marginal and degenerate cases should be handled.

4.1.1 Marginal Considerations

The simplest kind of discontinuous specification occurs in the specification of the first LIC.

> 1) There exists at least one set of two #consecutive# data points that are a distance greater than the length `LENGTH1` apart, where
>
> $$0 \leq \texttt{LENGTH1}.$$

The problem here is that the distance between two data points (x_1, y_1) and (x_2, y_2) can be computed only approximately, so it is not possible to test precisely the property

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \leq \texttt{LENGTH1}.$$

One could say "computed distance" instead of "distance," but then some novel way of computing the distance that produced a slightly different computed answer would give a different result. The specification has to make a choice between saying exactly how the distance should be computed or imposing a weaker requirement. The latter seems more appropriate for a requirements specification. In terms of the asymptotic method, the natural specification is:

> return `true` if
> $$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \lesssim \texttt{LENGTH1},$$
> `false` if
> $$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \gtrsim \texttt{LENGTH1},$$
> and either `true` or `false` otherwise.

Another version of the same kind of problem is also lurking in this example: the function *square root* is defined only for nonnegative numbers. Even if the mathematical value of an expression `E` is always nonnegative, the computed value may sometimes be slightly negative due to machine error, so that there may be problems in computing

`sqrt(E)`.

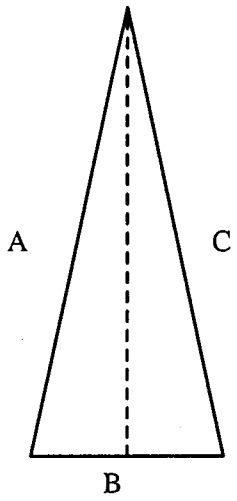
The best way to handle this problem seems to be as follows.

1. The function `sqrt` should raise an exception `ARGUMENT_ERROR` on a negative input.
2. When computing `sqrt(E)` where the mathematical value of `E` is known to be nonnegative (as when computing a length or an area), catch the exception `ARGUMENT_ERROR` and return 0. The computed value of `E` can be negative only if the mathematical value is near 0, so 0 will be a good approximation to the square root of the mathematical value.

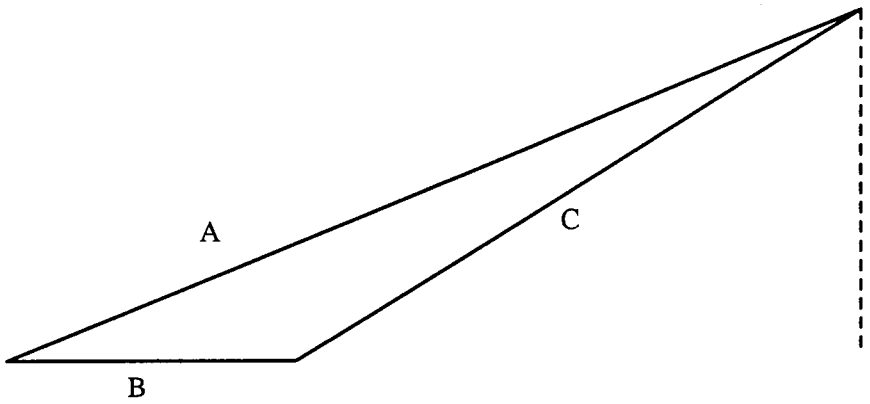
4.1.2 Discontinuities at Degenerate Points

A more serious kind of discontinuity occurs in the specification of another LIC:

> 7) There exists at least one set of "N_PTS" consecutive data points such that at least one of the points lies a distance greater than "DIST" from the line joining the first and last of the "N_PTS" points. If the first and last points of the "N_PTS" are identical, then the distance to compare with "DIST" will be the distance from the coincident point to all other points of the "N_PTS" consecutive points ...



Altitude is approximately C when B is small.



Altitude is approximately C/sqrt(2) when B is small.

Figure 1: Altitude is a discontinuous function of triangle side lengths

This requirement asks for the altitude of the second vertex of a triangle above the line made by the other two vertices. In the case of a degenerate triangle, in which the first and third vertices coincide, this requirement asks that the length of the line between the first and second vertices be given.

Such a requirement is *discontinuous* in the sense that a small change of input values does not always lead to a slight change of output values. As is shown in Figure 1, two triangles whose side lengths differ by only small amounts may have altitudes that differ by a large amount. As the length of the second side (labeled "B" in the diagram) goes to zero, the altitude of the triangle can approach any value between 0 and the length of the third side (labeled "C").

The requirements as stated call for the output of the length of "C" in the case where the length of "B" is exactly zero, but calls for the output of something close to $C/\sqrt{2}$ in the case where "B" is close to zero as in the bottom of the two pictures in Figure 1. From a computational point of view, the altitude is undefined when the length of "B" is small, so it does not make sense to specify it without further assumptions about the nature of the data (which must be mentioned in the specification). There are three reasonable approaches:

1. Do not specify the result at all when the length of "B" is small.
2. Raise an exception when the length of "B" is below a certain threshold ϵ such that $\epsilon \not\approx 0$.
3. Assume that the precision of the data is much less than the accuracy of the data, so that any two data points P and Q will either be equal or else

$$distance(P, Q) \not\approx 0$$

4.2 Program Correctness

Even if a requirements specification is implementable, the implementation can err by using discontinuous functions to implement the specification.

An example occurs in another one of the LICs.

> 4) There exists at least one set of three #consecutive# data points that are vertices of a triangle with area greater than "AREA1."

Of course, in trying to compute whether

$$area(p, q, r) > \texttt{AREA1},$$

we can only count on getting

`true`	if $area(p, q, r) \gtrsim \texttt{AREA1}$,
`false`	if $area(p, q, r) \lesssim \texttt{AREA1}$,
either	if $area(p, q, r) \approx \texttt{AREA1}$.

This modified specification is implementable because the area of a triangle is a continuous function of its vertices. But if we use high-school trigonometry to compute the area,

$$area(p, q, r) = distance(p, q) * distance(q, r) * \sin(angle(p, q, r)),$$

we will have a problem when either p or r is close to q, because the angle is discontinuous and undefined when $p = q$ or $r = q$. Instead, we should use either the vector formula

$$area(p, q, r) = \|(p - q) \times (r - q)\|/2$$

or Archimedes' formula

$$area(p, q, r) = \sqrt{s(s - a)(s - b)(s - c)}$$

where

$$\begin{aligned} a &= distance(p, q), \\ b &= distance(q, r), \\ c &= distance(r, p), \\ s &= (a + b + c)/2. \end{aligned}$$

As suggested above, the possibility of roundoff error leading to a slightly negative argument to the square root function should be handled by trapping the resulting exception and returning a value of 0.

5 Validity of the Knight-Leveson Study

As discussed in Section 2, the study of Knight and Leveson [3] cast doubts on the benefits of N-Version programming by showing that program faults are not statistically independent for different versions. Considering the study from the point of view of the asymptotic method casts doubt on the validity of their results.

We have already observed in Section 4.1.2 that their specification did not adequately deal with discontinuities in functions such as angle and altitude.

A second problem relates to how they determined program faults. A program was judged "faulty" on any test data for which it gave a different answer from a "gold program". But any Boolean function of real number inputs is discontinuous; therefore its results can be affected by differences in numerical error due to minor variations in the formulas used. If a condition P can be reasonably computed using methods A and B, and the gold program uses method A, then we would expect detected "faults" in programs that used method B to be strongly correlated.

We would also expect many faults to occur with test data containing degenerate configurations, as then values near discontinuities occur in the computation, possibly leading to widely differing numerical results.

It seems likely to us that real faults are often correlated because many programmers would be likely to use a formula for, say, area, as described in Section 4.2. It seems to us that most faults of this kind would be avoided if the specification gave adequate instructions on what to do about degenerate configurations and programmers were given adequate instructions to use continuous formulas to compute continuous mathematical functions.

In general, we do not see how mere statistics can distinguish between correlated errors due to common faults and correlated errors due to marginal data. One would have to investigate the code to find whether there really were any faults.

Knight and Leveson found that some programmers compared angles by comparing their cosines. They considered this a fault due to ignorance of the fact that comparing cosines of angles in $[0, \pi]$, although mathematically equivalent, is not numerically equivalent to comparing the angles themselves. But to find the angle $angle(p, q, r)$ formed by three points p, q, r, we must first compute the cosine of the angle using some such formula as

$$\cos(angle(p, q, r)) = \frac{(p - q) \cdot (r - q)}{\|p - q\| \|r - q\|}.$$

It is not clear to us why it is numerically better to compute the inverse cosine of this quantity than to compute the cosine of the angle to which it is to be compared.

We conclude that

- it may be necessary to investigate the code to determine whether a numerical program is really faulty, and
- statistical studies of the value of N-version programming should use a non-numerical program.

6 Methodology for Developing Numerical Programs

Our experiences with analyzing real number programs suggest the following enhancement of a standard model of program specification and development.

1. Naive requirements specification.
2. Asymptotic requirements specification.
3. Detailed specification of formulas.
4. Implementation.

By a naive requirements specification, we mean a specification like that used by Knight and Leveson. That specification is a perfectly good starting point. The specification would be written by experts in the application domain, but would require no special expertise in numerical programming or in the asymptotic method. Use of a formal specification language may make this specification more precise, concise and understandable. For instance, Lu et al. [4] translated the specification used by Knight and Leveson into Z, replacing confusing locutions about consecutive data points by clearer specifications like

$$\exists i \in 1..(p - 1) \ (distance((x[i], y[i]), (x[i + 1], y[i + 1]) > \texttt{LENGTH1})$$

An asymptotic requirements specification would just be a translation of the naive specification into an implementable specification expressed in terms of the asymptotic method, together with mathematical definitions of the concepts used. This specification would be written by experts in the asymptotic method, who would consult with the authors of the naive specification about how any discontinuities in the naive specification should be treated.

In the asymptotic specification, one should also consider any effects of *input error*, that is, of the likely possibility that the input data to the program will come from sensors or prior computation and will not be exact.

The formula specification is essentially a design specification phase. The part of it that relates to the asymptotic method is that the chosen formulas to compute mathematical functions should have only the same singularities as those mathematical functions.

Implementation should be divided between code that is intensive in numerical computation and code that is not. Numerical code should be carefully inspected or formally verified using the asymptotic method.

The asymptotic requirements specification and formula specification should be written by persons well-versed in the asymptotic method. Essentially this means that they must have a good grasp of the concept of continuity and how it relates to the effect that numerical error can have on the results of a program. Software engineers who do not understand the notion of continuity well should not write numerical code.

6.1 The Place of Formal Verification

Formal verification will be useful for both the asymptotic requirements specification and the formula specification, as well as for code verification.

Formal verification would be used to show that the asymptotic requirements specification satisfies necessary conditions for it to be implementable.

Proving that formulas chosen to meet the asymptotic requirements specification introduce no additional discontinuities is essentially a familiar kind of design verification.

6.2 Aids to Development and Verification

Various libraries of code and mathematical theories would assist the development and verification of numerical code.

6.2.1 Libraries of Numerical Subprograms

Many libraries of numerical subprograms exist. They should be annotated in a way that makes it clear how they treat discontinuities in the mathematical functions they implement.

Such annotations would make it easy to see whether a subprogram meets a given asymptotic specification.

6.2.2 Libraries of Mathematical Theories

The process of specification and verification of a numerical program must draw on the mathematical theory of the application domain. As discussed in Section 4.1, it is important for unambiguous and implementable specifications that special care be taken for degenerate cases of real number parameters, where functions may be undefined or discontinuous. For this purpose, standard mathematical treatments of (for example) triangles may be inadequate, since they often restrict their concern to nondegenerate cases.

Another issue is that some standard mathematical treatments may be correct, yet not especially useful for program verification if care was not taken to make definitions that can readily be translated into computable specifications. For the most efficient use of verification, it is good to have (whenever possible) the structure of the mathematical definition of a real number function parallel the structure of a program that could compute the function. In the ideal case, there should be two definitions of a function: one that clearly and intuitively describes the function, and a second that uses computable functions in the definition as far as possible. Then there should also be proofs that the intuitive specification and the computable specification agree.

An example that came up in the Launch Interceptor Program was the concept of the radius of the smallest circle enclosing a set of three points. The most straight-forward way to state mathematically that a radius r can enclose a set of three points $\{p_1, p_2, p_3\}$ is

$$\exists p : \mathit{Vector}\ \forall p' : \mathit{Vector}\ (\ p' \in \{p_1, p_2, p_3\} \rightarrow \|p - p'\| \leq r)\,,$$

where $\|v\|$ is the norm of vector v.

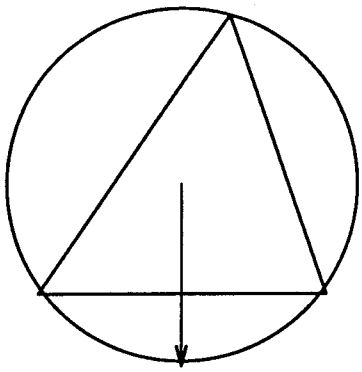
This definition is not computable as it stands, since it involves an existential quantification over an uncountable set.

We can see from Figures 2 and 3 that the smallest radius r of a disk containing the three points p_1, p_2, p_3 is

1. half the greatest distance between two of the points p_i, p_j, if the triangle $p_1p_2p_3$ is obtuse;
2. the radius of the circumscribing circle of the triangle $p_1p_2p_3$ if that triangle $p_1p_2p_3$ is acute. The circumradius is given by the formula

   $$r = a * b * c/4.0\mathrm{area}(p_1, p_2, p_3)$$

   where $a = \mathit{distance}(p_1, p_2)$, $b = \mathit{distance}(p_2, p_3)$, $c = \mathit{distance}(p_3, p_1)$. (The two formulas give the same result when $p_1p_2p_3$ is a right triangle.)



All angles acute: Use the circumscribing circle.

Figure 2: Circumscribing Circle

A proper verification would include the proof that this method of computing the radius is correct.

One problem we found in the code from Syracuse University was that the radius was always computed as the circumradius of the triangle, even when it was obtuse. It is clear from Figures 2 and 3 that for obtuse triangles the circumradius is larger than the radius of the smallest disk containing the triangle.

7 Summary of Verification Activity

We divided the Launch Interceptor Program implementation into three groups:

1. geometrical utilities that compute characteristics of groups of points in the plane, such as angles, areas of triangles, distances, etc.;
2. routines that compute the launch interceptor conditions (LICs);
3. routines that combine the LICs in order to obtain the final decision on whether to launch the interceptor missile.

The routines in the third group are not numerical at all. The numerical aspect of the LIC routines consists mainly of using the utilities to determine whether a given geometrical configuration exists in an input set of data points. The geometrical routines are purely numerical.

It quickly became clear not only that were the utilities were of primary interest from the point of view of numerical verification, but that they contained a number of problems and showed a number of problems with the specification. The problems we found in the utilities were as follows:

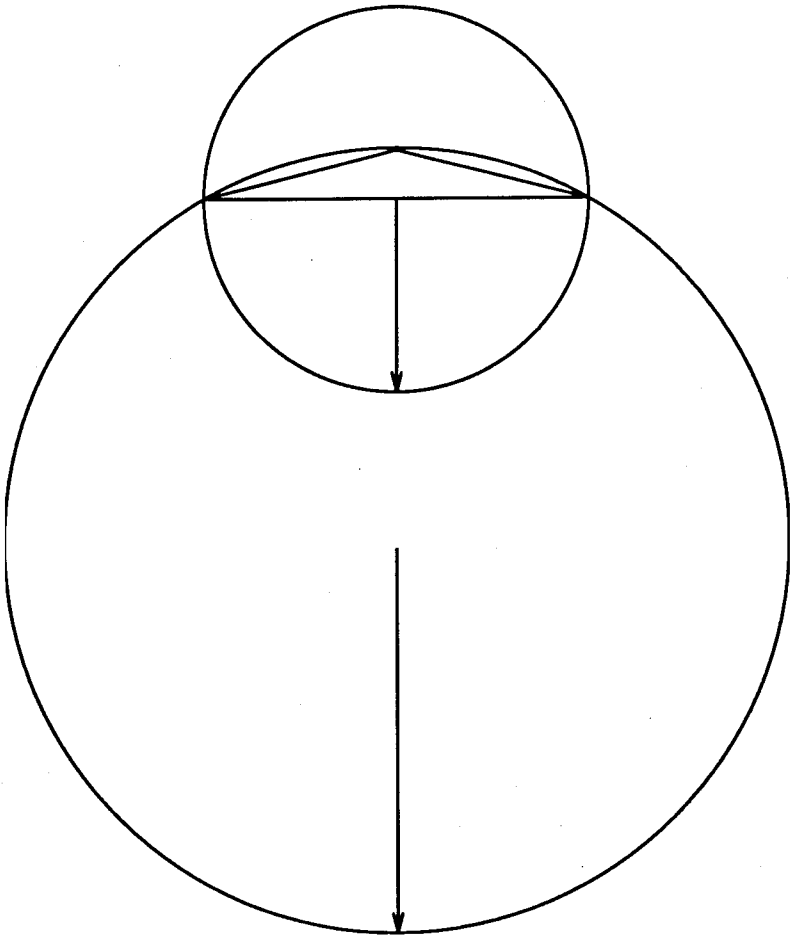


Figure 3: Radius of the smallest disk containing a triangle

1. unnecessary exceptions (the LIP experiment counted uncaught exceptions as faults) near degenerate configurations;
2. other unpredictable behavior near degenerate configurations;
3. use of discontinuous formulas to compute continuous functions;
4. use of an incorrect formula to compute the radius of the smallest disk containing three points.

In order to avoid these problems or make their effects less serious, we rewrote most of the utilities. Not all problems could be completely removed because the LIP specification does not give adequate instructions about what to do near degenerate configurations. (It does say what to do *at* degenerate configurations, but exceptions arise or the effects of roundoff error can be serious near degenerate configurations as well as at them.) In general, we tried to minimize problems with degenerate configurations and avoided exceptions by avoiding divisions by numbers that could be near zero. In spite of this, results of Boolean-value tests on near degenerate configurations remained unpredictable in terms of the mathematically correct results.

Typically, discontinuities arose from specifying something about or computing an angle. The basic problem is that the angle formed by three points PQR is undefined when, say, $P = Q$, and can have arbitrary values for P near Q. Hence when P is near Q, roundoff error can completely change the computed value of the angle.

Besides the problems with the utilities, formal verification revealed an error in one of the LIC routines, the use of uninitialized variables.

Next, we describe the specific problems we found in the utilities and how we rewrote them.

7.0.3 Comparing Real Numbers

The LIP specification requires that real numbers be compared using a function `realcompare (x,y)` that returns `lt` if the value of `x` is less than that of `y`, `eq` if the values of `x` and `y` are equal, and `gt` if the value of `x` is greater than that of `y`. The Ada real number model does not actually support the possibility of implementing a function with this specification (Ada allows the value of a variable to vary during execution according to whether it is kept in an extended precision register or stored in memory), but we used this specification anyway for the sake of the experiment. In any case, this specification can be realized on systems that use only a single, fixed precision for computations of a given floating-point type.

In Penelope we used -1, 0, and 1 instead of `lt`, `eq`, and `gt` because Penelope does not yet support enumerated types.

One of the problems of the Knight-Leveson study was that they counted any disagreement with their "gold" program as a programming fault, although perfectly valid programs could

produce different results on marginal data because the programs to their producing different numerical errors. Since such differences are often associated with comparisons, the testing process could be improved by conducting tests in the following way:

1. Run the gold program several times with a special version of the function `realcompare` that perturbs its arguments slightly and randomly before comparing them;
2. Run the program being tested for faults with the normal version of `realcompare`. The program's result is valid if the resule is the same as any of the results produced by the gold program.

7.0.4 Function `det_area`

The function `det_area(a,b,c)` computes the area of a triangle with sides of lengths approximately a, b, and c. The original routine computed the angle α between sides a and b and gave the area as

$$ab|\sin\alpha|.$$

This formula has an apparent problem because, a or b small, the computation of the angle is numerically ill-defined. Computing the angle could give an unpredictable result or an exception. In fact, this is not really a problem, because no matter what the computed value of the angle α may be, $|\sin\alpha| \leq 1$, so if a or b is small, the computed product $ab\sin\alpha$ will be small, as it should be. An exception can be raised only if either a or b is small, in which case the area will be small, so we could just trap the exception and return 0. All of our changes to the utilities tend toward eliminating explicit computation of angles, however, so to be consistent we did the same in this case.

Hence, we substituted Archimedes' formula for the area of a triangle,

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = (a+b+c)/2.$$

It is conceivable that, due to prior roundoff error in the computation of a, b, and c or s, the computed value of $s(s-a)(s-b)(s-c)$ might be slightly negative. Hence we trapped any possible argument error raised by the call to the square root function and returned 0 in that case.

```
function det_area(a, b, c : in float) return float

  --| where * * *
  --|   return area such that
        (forall p, q, r:Point::
             ((((a ~~ distance(p, q))
```

```
                and (b ~~ distance(q, r)))
                and (c ~~ distance(r, p)))
            ->
                (area ~~ area(p, q, r))));
    --| end where;

  is
    s : float := (((a+b)+c)/2.0);

  begin
    return  sqrt((((s*(s-a))*(s-b))*(s-c)));
  exception
    when argument_error =>
      return  0.0;
  end det_area;
```

7.0.5 Function `det_angle/compare_angle`

LIC's #3 and #10 call for comparing the angle formed by three points with values near π. The original utilities computed the angle formed by the three points and did the actual comparison. Here, we avoid the possible exception on computing the angle. Instead of computing the angle and then comparing, we have a function `compare_angle(a,b,c,beta)` that performs the comparison without actually computing the angle. The inputs `a,b,c` are the lengths of the sides of the triangle in question, and the angle α of interest is the angle between sides `a` and `b`. By the cosine law,

$$c^2 - a^2 - b^2 = ab\cos\alpha$$

We just compare $c^2 - a^2 - b^2$ with $ab\cos\beta$, avoiding a division by ab that might result in an exception.

This procedure flies in the face of the judgement by Knight and Leveson that comparing cosines of angles instead of the angles themselves is an error, but it is not obvious that from the point of view of numerical error it is better to take the inverse cosine of one number than the cosine of another.

We remark that our approach still does not guarantee even the approximate validity of the comparison between α and β when either a or b is small—no approach can, since in that case α is numerically ill-defined.

The notation

```
--| assert ...
```

indicates a cutpoint assertion, that is, a property that holds at that point in the code and

which is sufficient to prove that the subprogram, if it reaches that point and eventually completes, will have a result satisfying the return specification.

The notation

```
--| lemma ...
```

indicates a property that holds at that point in the code and will hold for the remainder of the subprogram.

```
function compare_angle(a, b, c, beta : in float)
      return boolean

  --| where * * *
  --|   in ((0.0 <~~ beta) and (beta <~~ pi));
  --|   return v such that ((((((c**2)-(a**2))-(b**2))
              <!
               (((2.0*a)*b)*cos(beta)))->v)
          and
           (((((c**2)-(a**2))-(b**2))
              >!
               (((2.0*a)*b)*cos(beta)))->(not v)));
  --|   return v such that
          (((a >! 0.0) and (b >! 0.0))
         ->
          (forall p, q, r:Point::
                (((a ~~ distance(q, p))
               and (b ~~ distance(r, q)))
               and (c ~~ distance(p, r)))
              ->
               ((v->(angle(p, q, r) <~~ beta)) and
               ((not v)->(angle(p, q, r) >~~ beta))))));
  --| end where;

is
  w : integer :=
        realcompare(
              (((c**2)-(a**2))-(b**2)),
              (((2.0*a)*b)*cosine(beta)));

begin
  --| lemma beta_bound: ((0.0 <~~ beta) and (beta <~~ pi()));
  if ((w=(-1)) or (w=0)) then
```

```
      --| assert ((c**2-a**2)-(b**2) <~~ 2.0*a*b*cos(beta));
      return  true;
    else

      --| assert ((c**2-a**2)-(b**2) >~~ 2.0*a*b*cos(beta));
      return  false;
    end if;
  end compare_angle;
```

7.0.6 Function `compare_radius`

LIC's #2 and #14 require a test whether the triangle formed by three points is contained in a disk of a given radius, which we will call *rad*.

The original code implements these LICs using a utility that computed the radius of the smallest disk containing a triangle with sides of length a, b, c using the formula for the radius of the circumscribing circle of a triangle,

$$\frac{abc}{4area(a,b,c)} = \frac{abc}{4\sqrt{s(s-a)(s-b)(s-c)}},$$

where $s = (a+b+c)/2$. The computed radius was then compared with the given maximum allowable value *rad*.

There are two problems with this approach.

1. When the triangle is obtuse, this formula is incorrect, as observed above in Section 3.
2. When a, b, c are small, the value computed using this formula may be seriously affected by roundoff error.

We solved the first problem by using the correct formula for radius (half the longest side) when the triangle is obtuse (it is obtuse if $a^2 > b^2 + c^2$, $b^2 > c^2 + a^2$, and $c^2 > a^2 + b^2$).

We solved the second problem in the following way.

1. Instead of computing the radius and then comparing, we wrote a subprogram `compare_radius (a,b,c,rad)` that does the comparison without computing the radius of the triangle of sides a, b, c when it is possible to avoid computing that radius.
2. `compare_radius` firsts tests whether $a + b + c \leq rad$. If it is, then certainly the radius of the triangle is smaller than *rad*.
3. `compare_radius` then tests for obtuseness, and in the obtuse case, returns half the longest side.

4. We assume that *rad* is large compared to floating point errors and the underflow threshold. If the triangle is acute and $a + b + c \gtrsim rad$, then we must have $a, b, c \not\approx 0$. In this case, there is no numerical problem with the formula for the circumradius of a triangle, so we use it and compare the result to *rad*.

```
function compare_radius(a, b, c, rad : in float)
      return boolean

  --| where * * *
  --|   in (rad >! 0.0);
  --|   in (exists p, q, r:Point::
                  (((distance(p, q) ~~ a)
              and (distance(q, r) ~~ b))
              and (distance(r, p) ~~ c)));
  --|   return v such that
         (forall p, q, r:Point::
              ((((distance(p, q) ~~ a)
             and (distance(q, r) ~~ b))
             and (distance(r, p) ~~ c))
          ->
                 ((v -> (radius(p, q, r) <~~ rad))
             and ((not v) -> (radius(p, q, r) >~~ rad)))));
  --| end where;

is

begin
  --| lemma rad: (rad >! 0.0);
  --| lemma triangle: (exists p, q, r:Point::
                   (((distance(p, q) ~~ a)
               and (distance(q, r) ~~ b))
               and (distance(r, p) ~~ c)));
  if (((a+b)+c)<=rad) then
    --| assert (((a+b)+c) <~~ rad);
    return  true;
  elsif (((a**2)+(b**2))<=(c**2)) then
    --| assert (((a**2)+(b**2)) <~~ (c**2));
    return  ((c/2.0)<=rad);
  elsif (((b**2)+(c**2))<=(a**2)) then
    --| assert (((b**2)+(c**2)) <~~ (a**2));
    return  ((a/2.0)<=rad);
  elsif (((c**2)+(a**2))<=(b**2)) then
    --| assert (((c**2)+(a**2)) <~~ (b**2));
    return  ((b/2.0)<=rad);
  else
```

```
      --| assert ((((((a**2)+(b**2)) >~~ (c**2)) and
                 (((b**2)+(c**2)) >~~ (a**2))) and
                 (((c**2)+(a**2)) >~~ (b**2)))
       and
        (((a+b)+c) >~~ rad));
      return  ((((a*b)*c)/(4.0*det_area(a, b, c)))<=rad);
    end if;
  end compare_radius;
```

Bibliography

[1] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16:1058–1075, September 1990.

[2] D. N. Hoover. Denotational semantics of numerical programs. Technical report, Odyssey Research Associates, Inc., Ithaca, NY 14850-1313, August 1992.

[3] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[4] Juin-Yeu Lu, Giuliana Dettori, and Shiu-Kai Chin. Applying formal methods to software engineering: Using Z to specify the launch interceptor program. Technical Report TR 15–7, Odyssey Research Associates, April 1989.

[5] Odyssey Research Associates, Inc., Ithaca, NY 14850-1313. *Larch/Ada Reference Manual*, July 1993.

[6] Sanjiva Prasad. Verification of numerical programs using Penelope/Ariel. In *COMPASS '92*. National Institute of Standards and Technology, June 1992.

[7] David Sutherland. Formal verification of mathematical software. Technical Report CR 172407, NASA Langley Research Center, Hampton VA, 23665, May 1984.