Genetic Programming as an Explorative Tool in Early Software Development Phases

Robert Feldt
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Gˆ teborg, Sweden
Tel: +46 31 772 5217, Fax: +46 31 772 3663
E-mail: feldt@ce.chalmers.se

Abstract

Early in a software development project the developers lack knowledge about the problem to be solved by the software. Any knowledge that can be gained at an early stage can reduce the risk of making erroneous decisions and injecting defects that can be expensive to eliminate in later phases. This paper presents the idea of using genetic programming to explore the difficulty of different input data in the input space, determine the effects of different requirements and identify design trade-offs inherent in the problem. Data from a pilot experiment is analysed and the knowledge gained is used to question and prioritize the requirements on the target system. Coping with high-dimensional input spaces and establishing the relationship between GP- and human-developed programs are identified as the major outstanding problems. An extended experimental environment is proposed based on techniques for visual database exploration.

1. Introduction

Software engineers transform the, often fuzzy, demands on a new piece of software into an executable system through a process involving requirements engineering, design, implementation and testing [Sommerville92]. Despite years of efforts on developing better software engineering methods, programming languages and development tools some software projects still fail to meet scheduled milestones and resource limitations. In part, this can be attributed to the fact that software systems are often öone-offí products and experience from earlier projects can not be relied on in doing predictions and planning [Sommerville92]. In the early phases of a software project not much is known about the difficulty of the task, the inherent trade-offs in the problem or different possible strategies in devising a solution.

If problem-specific knowledge could be obtained early in a development project this could lead to fewer errors being made in the requirements and design phases that, in turn, could decrease development time and cost. Many authors have reported that the cost of finding defects rises sharply in later development phases: a relative time of up to 1000 to one for finding defects in field use compared to during requirements engineering and 80 to one for defects in testing compared to design reviews [Humphrey95]. Information about the problem to be solved that is obtained early in a development project can reduce the risk of making erroneous decisions and minimize the probability of injecting defects in the system that can be expensive to eliminate in later phases. Furthermore, this information could be used during project planning to predict the resources needed.

Soft computing techniques for machine learning can be used when very little is known about a problem and its possible solutions. The ultimate goal of the soft computing technique of genetic programming (GP) is to enable automatic programming, i.e. the computer programming itself [Banzhaf98]. When the requirements on the program have been coded in

an executable form, a fitness function, and the parameters of the genetic programming system have been initialized, a large number of programs can be obtained from a GP system. In this paper we propose that problem-specific information be obtained early in a software development project by using genetic programming to explore the input data space of a program and the effect of different requirements on the difficulty of the problem. We propose the term *software problem exploration using genetic programming* (SPE-GP) for this idea and try to identify its implications.

A pilot experiment to test the basic idea of SPE-GP is described in section 2 and the results from the experiment are given in section 3. In section 4, we discuss what knowledge was gained in the experiment and what limits the conclusions that can be drawn from it. We also identify important questions for further research. A general framework for software problem exploration using GP is outlined in section 5. Finally, section 6 concludes the paper.

2. Pilot experiment

In a previous research project we have used a genetic programming system to develop 400 variants of an aircraft arrestment system software [Feldt98] [USAF86]. The programs were developed using the GPSys genetic programming system running on a SUN Enterprise 10000 with the Sun Solaris OS 2.1 and Java Development Kit 1.2 [Quereshi98]. The GP system was run 25 times for sixteen different settings of parameters and the best-of-run individuals were retained for further analysis. These 400 programs were subjected to the same 10000 test cases and their behavior compared to the requirements. Below we describe the target system, the GP system and the testing procedure. A more thorough description is given in [Feldt98].

2.1. Target system

The target system is designed to arrest aircraft on a runway. Incoming aircraft attach to a cable and the system applies pressure on two drums of tape attached to the cable. A computer that determines the break pressure to be applied controls the system. By dynamically adapting the pressure to the energy of the incoming airplane the program should make the aircraft come to a smooth stop. The requirements on a system like this can be found in [USAF86]. The system has been used in other research at our department and a simulator simulating aircraft with different mass and velocity is available [Christmansson98].

The main function of the system is to brake aircraft smoothly without exceeding the limits of the braking system, the structural integrity of the aircraft or the pilot in the aircraft. The system should cope with aircraft having maximum energy of $8.81*10^7$ J and mass and velocity in the range 4000 to 25000 kg and 30 to 100 m/s, respectively. More formally the program should[1] (name of corresponding failure class in parentheses)

- stop aircraft at or as close as possible to a target distance
- stop the aircraft before the critical length of the tape (335 m) in the system (OVERRUN)
- not impose a force in the cable or tape of more than 360 kN (CABLE)
- not impose a retarding force on the pilot corresponding to more than 2.8g (RETARDATION)
- not impose a retarding force exceeding the structural limit of the aircraft, given for a number of different masses and velocities in [USAF86] (HOOKFORCE)

[1] Our system adopts the requirements of [USAF86] with the addition of the allowed ranges for mass and velocity and a critical length of 335 m (950 feet in [USAF86]).

The programs are allowed to use floating point numbers in their calculations. They are invoked every 10 meters of cable and calculate the break pressure, for the following 10 meters, given the current amount of rolled out cable and angular velocity of the tape drum. An existing simulator of the system has been ported from C to Java. It implements a simple mechanical model of the airplane and braking system and calculates the position, retardation, forces and velocities in the system. It does not model the inertia in the hydraulic system or oscillatory movement of the aircraft due to elasticity in the tape. The simulator has been set to simulate braking with a time step of 62.5 milliseconds.

2.2. Genetic programming system

Our development system is built on top of the GPSys genetic programming system written in Java by Adhil Quereshi at the University College in London [Quereshi98]. During evolution GPsys invokes the simulator to evaluate the fitness of programs. Values from the simulation are used to assign penalty values on the four fitness criteria. The penalties are assigned in a non-linear fashion with high values when the program fails on the criteria. For the OVERRUN criteria:

- If the stop position of the aircraft is larger than the critical length of the system a basic penalty is assigned. The basic penalty was chosen as 80% of the maximum penalty for the criteria.
- A guiding penalty is assigned if the velocity of the aircraft is larger than zero on the critical length. The penalty is assigned proportional to the velocity of the aircraft on the critical length. This is to distinguish programs that almost succeeded in braking the aircraft from programs that havenít even tried and ì guidesî the programs in the direction of good performance. The guiding penalty was chosen as 20% of the maximum penalty for the criteria.
- If the aircraft comes to a halt a linear penalty is assigned. It diminishes from its maximum value at position 0 up to the target distance and then increases up to its maximum again at the critical length. This is to ensure that a halt position close to the target distance will give the program a low penalty. The maximum amount of linear penalty is a parameter to the system but should be much smaller than 80%.

The penalties for the other criteria are assigned in a similar manner. For more details consult [Feldt98]. The penalty values on the four criteria are summed to give the total fitness for the test case. The total fitness of the program is the sum of the finesses on all the test cases. A perfect program would get a fitness value of zero.

The values for the parameters to the GP system that was used in the experiments are shown in tables 1 and 2.

Parameter	Value
Generations	200
Population size	1000
Tournament size	5
Max depth of program trees at creation	7
Max depth of program trees	19
Max depth of mutation trees	3
Functions that were always used	Add, Sub, Mul, Div
Create Method	Ramped-half-and-half

Table 1. Values of parameters that were not varied during the experiments

Factor	Level	Description
A	-	No effect.
	+	The statement IF, and operators LE, AND and NOT can be used in the programs.
B	-	No effect.
	+	The functions SIN and EXP can be used in the programs.
C	-	The average velocity, average retardation, and the index to the current checkpoint can be used in the programs.
	+	The angular velocity, current time since start of the braking, the previous angular velocity and the time of the previous checkpoint can be used in the programs.
D	-	Programs cannot use any subroutines.
	+	Two subroutines (automatically defined functions) can be used in the program. They are evolved in the same manner as the rest of the program.
E	-	Maximum penalty on the RETARDATION failure criteria is 1000.0.
	+	Maximum penalty on the RETARDATION failure criteria is 2000.0.
F	-	Linear penalties are not used.
	+	Linear penalties are used and a maximum penalty of 30.0 is assigned on each failure criteria.
G	-	25 test cases uniformly spread on the range of possible values for mass and velocity are used to evaluate fitness during evolution.
	+	25 test cases chosen randomly for each run of the GP system are used to evaluate fitness during evolution.
H	-	Probability of mutation is 0.05.
	+	Probability of mutation is 0.6.

Table 2. Parameters that were varied between different runs of the GP system

2.3. Testing procedure

After each run of the GP system the best-of-run individual is evaluated on 10000 test cases evenly spread on the range of valid masses and velocities. Dividing the range of allowed mass into 100 locations 212.12 kg apart generates these test cases. For each mass a maximum velocity is calculated so that the resulting energy does not exceed the $8.81*10^7$ J specified in [USAF86]. The range [30, max velocity for this mass] is divided into 100 velocities and a total of 100*100=10000 test cases result.

3. Experimental results

For each test case executed, a trace of the braking of the airplane is returned from the simulator. Four values are extracted from this trace to classify the behavior of the program: halt distance of the aircraft, maximum force in the cable, maximum force on the hook of the aircraft and maximum retardation during the braking. These values correspond to the four fitness criteria above. We record a failure for a particular version on a particular test case if *any* value exceeds its limits. Failure is indicated by one (1) and success by zero (0) and these

binary values are collected into a failure behavior vector giving the failure behavior for each test case.

The *difficulty* of a test case is defined as the proportion of GP-developed controllers that failed on the test case. Since two real-valued numbers characterize each test case the input data difficulty can be visualized in a 2D contour diagram. The diagram is shown in figure 1.

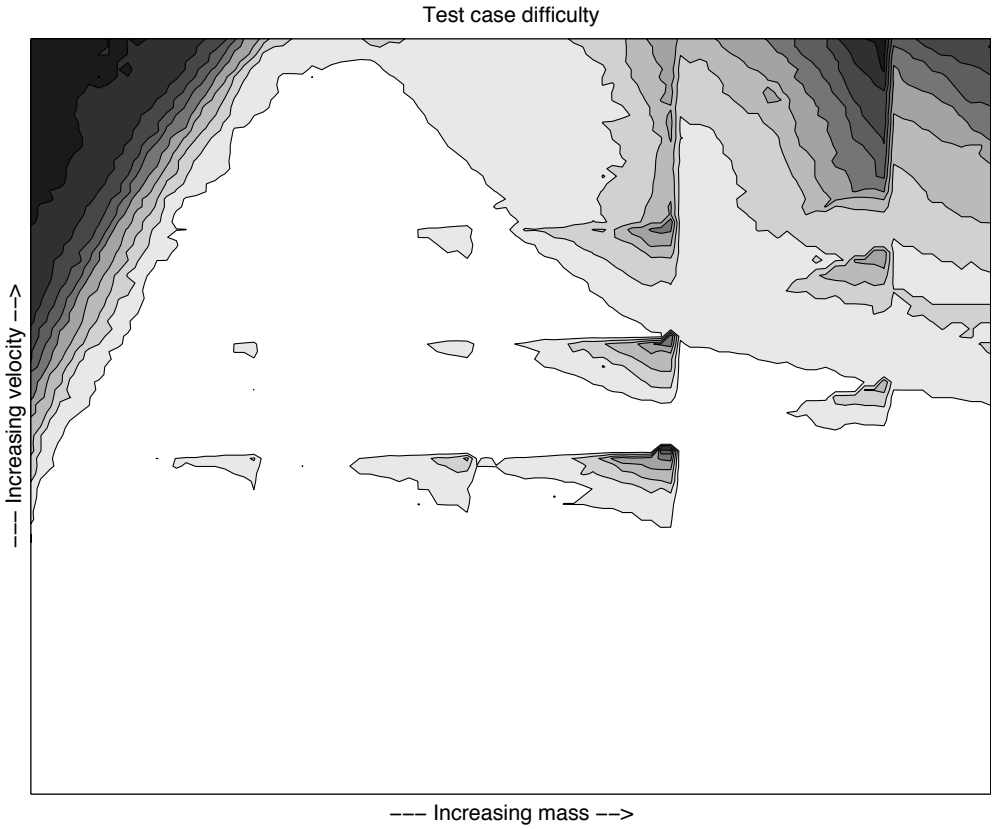

Figure 1. Contour diagram showing the difficulty of test cases with different mass and velocity

Darker areas indicate higher difficulty, i.e. indicating test cases where a larger proportion of the programs fail.

Detailed analysis of the diagram reveals that there are three main areas of difficulty. Visually these areas are located in the upper left corner, in equidistant clusters in the center and in the upper right corner, respectively.

For the upper left corner, where aircraft have high velocity and low mass, the programs generally fail because they violate the requirement on the maximum retardation on the pilot. It seems plausible that these failures arise because the programs do not properly measure and/or use a notion of the mass of the incoming aircraft in their control algorithm.

The failures in the center area are mainly due to excessive forces applied to the hook on the aircraft attaching to the cable of the braking system. A requirement in [USAF86] states the maximum allowed hook force for certain ìpointsî with specified mass and velocity. The clusters of failing programs seen in the center of figure 1 are located below (lower velocity)

and to the left (lower mass) of these points. These are the areas where the energy of the aircraft is at a maximum for the requirement of maximum hook force.

The failures in the upper right corner are made up of a combination of failing to restrict the force on the aircraft hook and failing to brake the aircraft before the end of the runway. The former can be explained by the same reasoning as above and the latter arises because the energies of the aircraft take on their largest values this area. If the programs do not exert a high enough brake pressure in the start of the braking they will not have time to brake the aircraft before the critical length.

4. Discussion

The experimental results imply that the central challenge in this problem is to cope with the high energies of heavy airplanes having large velocities. Failure to meet the OVERRUN criterion is fatal in that the aircraft has not come to a halt. Even though a large majority of the programs also have problems with the RETARDATION criterion for light airplanes with high velocities, detailed analysis of the braking behavior shows that the programs often only slightly exceed the threshold of 2.8g maximum retardation. The center clusters of difficulty show failures on the HOOKFORCE criterion but it is a smaller proportion of programs that fail here than on the previously mentioned criteria. The discontinuous property of these clusters can be attributed to the fact that the maximum hook force was specified at certain points in the requirements. The CABLE criterion was not violated by any of the programs. Detailed analysis of the experimental data also reveals that some programs make a trade-off between meeting different requirements. They trade better performance on the RETARDATION criterion for worse performance on the OVERRUN criterion. Other programs do the opposite. This indicates that there is a fundamental trade-off to be made between these criteria.

The knowledge gained by analyzing the behavior of the GP-developed programs could be used in a discussion with the client stating the requirements. It can be used to resolve questions, prioritize the requirements and show requirements that may be too easy to fulfill, indicating cost savings. An example of the former is to question the HOOKFORCE criterion that is only specified in certain points. Is this really what the client want or is it wiser to specify the requirement as a function of the incoming mass and velocity? The fact that the cable criterion is never violated might indicate that too strong a cable is used in the system. Based on the information gained from the experiment a weaker, but possibly less expensive, cable can be proposed. A suggestion to prioritize the requirements can be based on the trade-off between not retarding light airplanes too much and being able to brake high-energy airplanes at all. If the latter is preferred the developers can propose that a slight increase in the threshold for the RETARDATION criterion might change the difficulty, and therefore the cost, of designing a solution. Or they can question whether light airplanes with high velocities can be expected to appear.

It should be noted that the reported experiment is limited in several ways. Firstly, the target application is small, having few requirements and a low-dimensional input space. This latter feature makes it particularly well suited for visualization; with a high-dimensional input space visualization will be more difficult. Furthermore, an existing simulator could be used to evaluate the fitness and failure behavior of the programs. In general a simulator will not be available in early software development phases and it is unclear if the cost for developing one is motivated by the knowledge that can be gained. Further research is needed to clarify this.

Secondly, the presented experiment was not primarily designed to evaluate the SPE-GP idea put forward in this paper. For example, it would have been interesting to alter some of the requirements to assess their effect on the difficulty levels and their distribution.

In addition to these practical considerations there is a more fundamental question that needs to be addressed. If we are to gain any useful knowledge about the software problem at hand the behavior of the GP-developed programs must be representative for programs developed by human software developers. There is a risk that the behavioral data will be more GP-specific than problem-specific. An example of this is seen in our pilot experiment that is really showing the difficulty for *memory-less*[2] *controller programs with certain functions and terminals*. If the controllers were allowed to use, for example, indexed memory the difficulty landscape showed in figure 1 might change. A similar concern can be raised because of the limited applicability of present-day automatic programming systems, such as GP.

Provided that the SPE-GP technique can be used on more complex problems, the knowledge gained from using it can be compared to the outcome of the actual development project. For example, the average difficulty level obtained from using an SPE-GP scheme can be compared to the total development time. If, over several projects, correlations are found the average difficulty level can be used in project and resource planning.

Motivated by the deficiencies in the pilot experiment and the outstanding issues that need to be resolved in order to evaluate the power of SPE-GP we outline an extended environment for SPE-GP in the following section.

5. An extended environment for software problem exploration using genetic programming

In an extended environment for software problem exploration using genetic programming we need to address the problem of a high-dimensional input space. The two main problems with having many input parameters are the problem of selecting test cases for visualization and the problem of visualizing high-dimensional data. The former arises because there is a combinatorial explosion in the number of possible input cases; for most systems exhaustive testing of all possible combinations is not feasible. The latter problem arises because the high-dimensional data needs to be presented on our low-dimensional output devices.

One possible solution to the testcase-selection problem is to let the developer do the selection. Even though the developer might not be expected to have much knowledge about the problem they might have a hunch about what areas are more or less interesting. This scheme requires interaction between the developer and the SPE-GP environment. Another possible solution to the testcase-selection problem would be to use information from the GP development phase to guide the exploration and visualization system to the most difficult areas of the input space. One possible solution would be to let the testcases used during evolution, i.e. the fitness cases, be co-evolved with the programs.

There are a number of techniques for extracting knowledge from high-dimensional data. Often they involve some statistical technique for dimensional reduction, such as principal component analysis or multidimensional scaling, or use clever coding schemes to map the multiple dimensions to our 2D or 3D display devices [Keim97]. One promising approach is the Visor algorithm of Kˆnig et al [Kˆnig94]. Their system combines methods for feature extraction and class separation with a fast algorithm for visualizing high-dimensional data in 2D. These abilities can be useful in isolating clusters of similar test cases in the input space.

[2] The evolved programs can not access the memory used to calculate the terminals used in the programs.

Techniques for interactive exploration have also been studied in the area of visual database exploration [Keim97].

The environment we envisage is a developerís workbench for exploring the effects of different requirements and finding the difficult, and therefore crucial, areas of the input domain. With faster computers and more advanced display techniques one can imagine such a system performing the evolution of the programs in real-time in response to developer queries.

Figure 2 shows a conceptual diagram of an SPE-GP environment according to the discussion above. On the left side is the GP system developing a pool of programs that are tested on the right side to produce data that is displayed for the developer. The developer can interact with the visualization system to select areas of interest or display the data in different ways. The developer queries are sent to the test controller that uses existing data from a database or performs additional tests of the programs. The user can also alter the requirements and develop new programs that can give further information. Also note the evaluation controller supplying fitness cases to the evaluation of the programs in the development system. These fitness cases can be used as starting points for the tests in the exploration system.

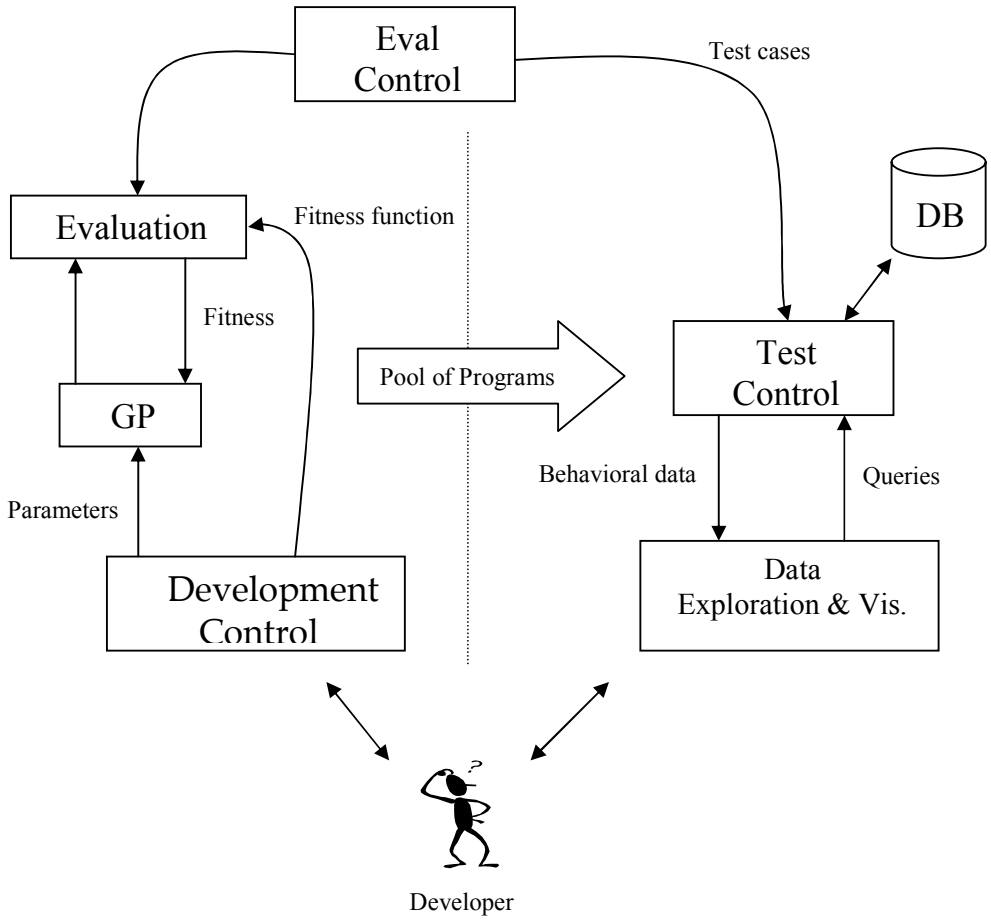

Figure 2. Conceptual diagram of a SPE-GP environment

6. Conclusions

Early in a software development project the software engineers lack knowledge about the problem to be solved by the software. Any knowledge that can be gained at an early stage can reduce the risk of making erroneous decisions and injecting defects that can be expensive to eliminate in later phases. Having such knowledge could thus lead to lower development costs and possibly a higher quality. Furthermore, it could be used in project planning to predict resource need. We propose that techniques for machine learning, especially genetic programming, be used to explore the software problem to be solved and to gain knowledge about difficult areas of the input space, the effects of different requirements and to identify design trade-offs that need to be addressed.

In an initial experiment, where genetic programming was used to develop 400 controller programs for an aircraft arrestment system, three main areas of difficulty in the input space were identified. These areas were related to the requirements, and the knowledge gained could be used to question and prioritize the requirements as well as indicating areas for cost savings. However, the target system was simple with few requirements and a low-dimensional input space well suited for visualization. For problems with high-dimensional input spaces it is unclear how to select the test cases for testing the programs and how to visualize the resulting data in a meaningful way for the human developers. We point out some possible solutions to these problems and outline a general environment for *software problem exploration using genetic programming* (SPE-GP).

The central outstanding research question is to establish what data on the behavior of programs developed using genetic programming tells us about the difficulties facing human developers. It is not likely that a general answer can be given to this question; the amount of knowledge gained will likely vary with the actual problem and the power of the genetic programming system used. We think that the importance of this software engineering problem motivates further study.

References

Banzhaf, Wolfgang et al. Genetic Programming – An Introduction. Morgan Kaufmann, San Fransisco, California, 1998.

Christmansson, Jörgen. An exploration of models for software faults and errors, PhD Dissertation, Department of Computer Engineering, Chalmers University of Technology, 1998.

Feldt, Robert. Generating Multiple Diverse Software Versions Using Genetic Programming - an Experimental Study. IEE Proceedings – Software, vol. 145, Issue 6, December 1998.

Humphrey, Watts. A discipline for software engineering, Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-54610-8.

Keim, D. A. Visual Techniques for Exploring Databases, Invited Tutorial, Int. Conference on Knowledge Discovery in Databases (KDD'97), Newport Beach, CA, 1997.

König, Andreas, Buhlman, Olaf, Glesner, Manfred. Systematic Methods for Multivariate Data Visualization and Numerical Assessment of Class Separability and Overlap in Automated Visual Industrial Quality Control, In *Proceedings of the 5th British Machine Vision Conference* (BMVC'94), pages 195-204, Sept. 1994.

Sommerville, Ian. Software Engineering, Addison-Wesley, Workingham, England, 1992. ISBN 0-201-56529-3.

Quereshi, Adil. GPSys 1.1 Homepage. Web Page. URL: http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html. 20 November 1998.

US Air Force. Military Specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction. 1986. MIL-A-38202C, Notice 1.