When Computers Fly, It Has to Be Right: Using SPARK for Flight Control of Small Unmanned Aerial Vehicles

Lt. Col. Ricky E. Sward, Ph.D., Lt. Col. Mark Gerken, Ph.D., and 2nd Lt. Dan Casey
U.S. Air Force Academy

One approach to software assurance is to use an annotated language such as SPARK. For safety critical software programs such as Unmanned Aerial Vehicle flight control software, the risk of software failure demands high assurance that the software will perform its intended function. Using an example from work being done at the U.S. Air Force Academy, this article describes SPARK and the formal process of proving correctness of software implementations.

In recent years, we have seen small Unmanned Aerial Vehicles (UAVs) emerge onto the battlespace. These small UAVs, which weigh less than 50 pounds, carry five to 10 pounds of payload, have mission endurances of about one hour, and field missions to support tactical surveillance and reconnaissance operations. As the number of missions for these small UAVs increases, more reliance will be placed on the computer systems that will control these aircraft. It is already possible for a computer program to create a flight plan for a UAV and place the craft in an orbit to observe an item of interest. Flight control computer programs are *safety critical* because errors in these programs could result in the loss of the UAV or cause damage to buildings and/or people on the ground [1].

Because of the critical nature of this software, development processes that result in *high-integrity* software (software systems that have a very low probability of failure) [1] must be used. Verification and validation (V&V) is often an important part of these processes. However, V&V of a safety-critical computer program can consume up to 50 percent of the development time [2]. One approach to reuse dedicated V&V schedule time while still assuring software quality is to use an annotated programming language. For example, the SPARK programming language and its associated tools increase the quality of software developed while reducing overall development time [3, 4]. This approach to software development can easily become a valuable tool when dealing with safety-critical systems. The SPARK language, which is a commercial product available from Praxis High Integrity Systems[1], is ideal for systems such as UAV flight planning software. The SPARK language has been used on many successful software development projects such as the C-130J [4] and is the language we have used on our UAV project. SPARK is an annotated language similar to the annotated Ada language [5] and the Larch annotated language for C [6]. The annotations in SPARK create extra work for the development team, but it has been shown the return on time investment can be as high as an 80 percent reduction in testing costs [4].

This article briefly discusses the SPARK programming language and development process. We examine a safety critical software example developed for small UAVs developed in a senior-level computer science course at the Air Force Academy. We elucidate the SPARK process by examining a small section of the UAV control software dedicated to orbital control.

Background

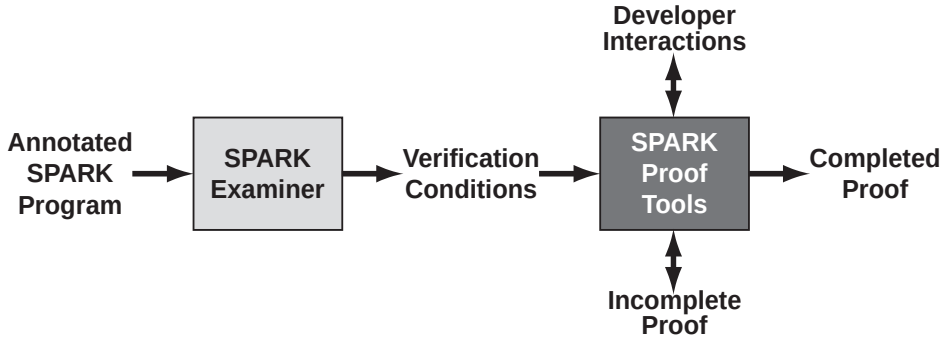
SPARK is an annotated subset of the Ada programming language, and every SPARK program can be compiled by an Ada compiler. However, SPARK includes several restrictions and rules governing the use of various programming constructs. These rules not only serve to simplify the V&V process, but also have a secondary benefit of helping assure good coding practice. For example, SPARK does not support dynamic allocation of memory so things such as pointers and heap variables are not supported. The use of GOTO statements is also not supported and there are restrictions on EXIT statements in loops. These rules and restrictions in SPARK allow the developer to predict the exact outcome when the code is executed. This prediction ability is a key feature of SPARK.

In order to show that the code developed has a very low error rate, SPARK uses formal, mathematical techniques to prove that the code is correct. The proof of correctness relies on the mathematical description of what is true before the code is executed and what is true after the code is executed. These are referred to as preconditions and postconditions, respectively. SPARK includes annotations that allow the programmer to embed the *precondition* and *postcondition* into a segment of code in the form of comments.

The proof of correctness uses a technique that begins with the postcondition and works backward through a segment of code, *hoisting* the postcondition up through the SPARK code [1]. Since the code is predictable, it is possible to determine a general effect of each statement and to reverse that effect, working backward from the postcondition toward the precondition. The result of this hoisting is called a *verification condition* in SPARK. If it can be shown that the precondition for the segment of code implies the verification condition developed from the hoisting process, the proof of correctness is complete.

SPARK includes several tools that help with the proof of correctness. The Examiner tool performs the hoisting operation and produces a collection of

Figure 1: *SPARK Tool Set*



verification conditions. The development team uses the Simplifier tool to reduce the verification conditions as much as possible before attempting to prove that they are implied by the precondition. The Proof Checker tool is used to prove that the verification conditions imply the precondition for a segment of code.

Figure 1 shows the process used by a software developer. The developer is responsible for annotating the code with preconditions and postconditions. The developer then executes the Examiner tool on the annotated SPARK code, and the Examiner returns the verification conditions. The developer then executes the Simplifier tool on the verification conditions, and the Simplifier returns the simplified verification conditions. The developer then executes the Proof Checker on the verification conditions to see if the tool can build a proof of program correctness automatically. If such a proof cannot be built, then the developer attempts to build a proof manually. In this situation, either the code is incorrect or the approach to constructing such a proof is insufficient. An incomplete proof does not necessarily mean that the code is incorrect, but a complete proof does mean the code is correctly implemented.

The burden placed on the developer is to build correct preconditions and postconditions, as well as correct code. If a proof cannot be built automatically for the code, then the developer will also need to examine the code or the verification conditions to see if a proof can be built manually. This may seem like an undue burden on the developer, but the return on this investment can be as much as an 80 percent reduction in costs during the testing phase [4] due to the fact that the code being developed is provably correct while being built.

It should be noted that since the preconditions and postconditions are written by the code developer, they are subject to human error. Therefore, these preconditions and postconditions should be reviewed and verified by a separate software development team. This moves the review process to a higher level of abstraction since the development team is now reviewing general mathematical preconditions and postconditions instead of pages of code written in a programming language.

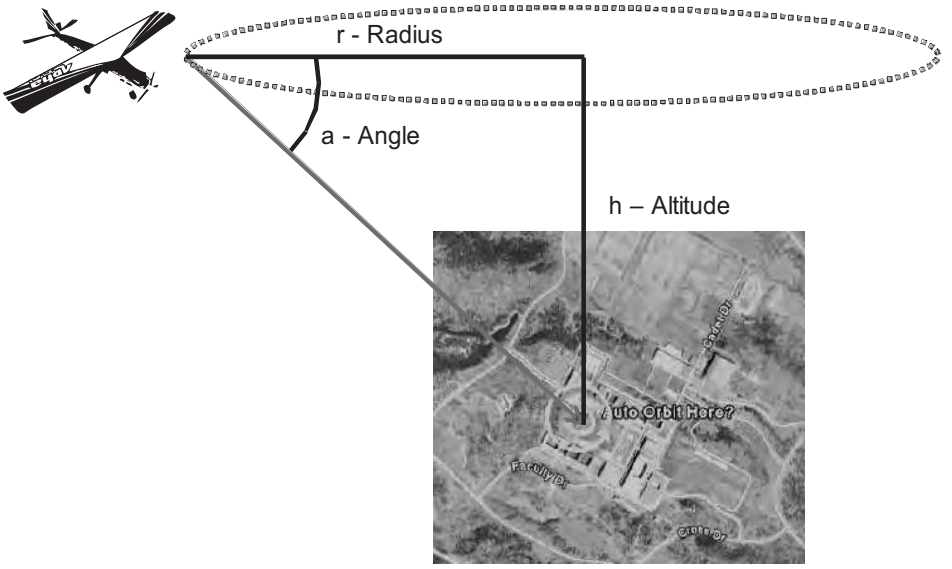
In the following section, we describe a small example that shows the SPARK development process in action. This example also highlights how the SPARK tools detect errors. At the U.S. Air Force Academy, we use SPARK to develop code in the senior-level software engineering course as we develop code to build flight plans for UAVs.

Figure 2: *Auto-Orbit Tool in UAVSAT*

UAV Situational Awareness Tool

In order to improve the situational awareness of a commander during a crisis situation, we have developed the UAV Situational Awareness Tool (UAVSAT) [7]. This year we have ported UAVSAT to Google Earth[2], which shows the location of the UAV in three dimensions along with a near real-time video feed from the UAV. The UAVSAT tool also provides the commander the ability to select a location on the map for the UAV to orbit. Figure 2 shows how the commander selects a location on Google Earth and indicates where the UAV should orbit. To position the UAV and gimbaled camera, the software we developed automatically calculates the optimal orbit altitude and radius to position it in the correct orbit to *stare* at the location selected by the commander. The software builds a new flight plan for the UAV in order to transition it from its current latitude and longitude to an optimal orbit radius and altitude.

Figure 3: *Calculating the Auto Orbit*



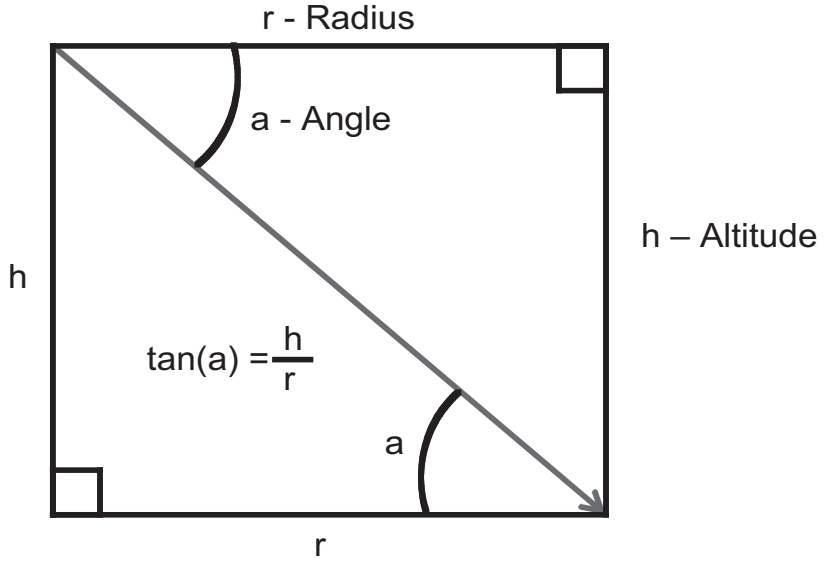


Figure 4: *Mathematical Depiction*

Using SPARK for Flight Control

Since the orbit flight plan for the UAV is built automatically by UAVSAT, it is imperative to calculate the correct altitude and radius for the UAV's orbit. If these values are not calculated correctly, the UAV will not be able to position the camera properly to observe the area of interest. If these calculations include flight critical phases, such as terrain avoidance or aircraft spacing, errors in the calculations could cause loss of the aircraft or destruction of objects on the ground. In order to ensure that the auto-orbit calculations are coded properly, we are using SPARK to verify the implementation.

Figure 3 (see page 11) shows which calculations need to be done in order to determine the proper flight plan for the UAV. In the figure, *h* is the altitude of the orbit, *r* is the radius of the orbit, and *a* is the angle of the gimbaled camera. The latitude and longitude of the orbit location are provided to UAVSAT as selected through the auto-orbit tool and positioned by the commander. To simplify the orbit problem, we have initially fixed the altitude of the orbit at 500 feet above ground level and have fixed the gimbal's angle at 30 degrees. The code for calculating the auto-orbit flight plan must simply determine the radius *(r)*, given the altitude and the gimbal angle.

Figure 4 depicts the problem as two similar right triangles. The tangent of the angle *(a)* is equal to the opposite side of the triangle over the adjacent side of the triangle. In the figure, this is represented by $tan\ (a) = h/r$. Solving for *r*, the result is $r = h/tan\ (a)$. That is, to calculate the radius of the orbit, we simply divide the altitude by the tangent of *a*.

In order for UAVSAT to receive the current latitude and longitude of the UAV and also to upload new flight plans to the UAV, our software must interface with C++ code provided by the autopilot vendor. We could have implemented UAVSAT in C/C++, but we wanted to preserve our ability to use the automated verification provided by SPARK. We therefore used Ada to build a Dynamically Linked Library (DLL) called from the C++ code. This DLL interface utility is well documented in the Ada Language Reference Manual[3].

Figure 5 shows the Ada_Radius function built in the DLL interface to calculate the radius *r* for the auto-orbit utility. Now the developer annotates preconditions and postconditions for the Ada_Radius function.

Figure 6 shows the specification of the Ada_Radius function. The specification includes the annotations for the preconditions and the postconditions. For functions in SPARK, the postconditions are annotated by using the *return* annotation [1]. The precondition for Ada_Radius allows heights greater than or equal to zero. The input angle is restricted from zero to avoid a potential division by zero error. The postcondition for Ada_Radius expresses the mathematical formula for the radius calculation. The angle is multiplied by Pi over 180 to convert the angle from degrees into radians as expected by the tangent function.

Figure 7 shows the body of the Ada_Radius function. Since the preconditions and postconditions are defined in the specification, no further annotations are needed in the body. The code is now ready to be analyzed by the SPARK Examiner. The developer executes the Examiner tool passing in the Ada_Radius function to be analyzed. The Examiner returns the verification conditions, and the developer executes the Simplifier, which returns the simplified verification conditions shown in Figure 8.

In the figure, the lines beginning with *H* represent *hypotheses* and the line beginning with *C* represents a *condition*. The symbol ->indicates *implication*. In this verification condition, H1 and H2 are derived directly from the precondition of Ada_Radius. H3 is determined during the process of hoisting the postcondition through the code and states that the result of the tangent function is restricted from zero. Since the radius is given as $h/tan(a)$, H3 is avoiding a potential division by zero. C1 indicates the final part of the verification condition built from the hoisting process. The code takes the first line of C1, and the postcondition takes the second line of C1. In order to prove the code is correct, we must show that this implication is true (i.e. that H1-H3 imply C1). So far, the developer has built the code and the annotations. The Examiner and Simplifier have built this simplified verification condition automatically for the developer.

The next step is to attempt to build a proof that this verification condition is true. The developer executes the Proof

Figure 5: *Ada_Radius*

```
-- function to calculate the radius in Ada
function Ada_Radius (
   Height : in   Float;
   Angle : in   Float)
 return Float is
begin
 return (Height / Ada_Tangent((3.14159/180.0)) * Angle);
end Ada_Radius;
```

Figure 6: *Specification for Ada_Radius*

```
-- function to calculate the radius in Ada
function Ada_Radius (
      Height : in  Float;
      Angle : in  Float)
   return Float;
--# pre (Height >= 0.0) and (Angle /= 0.0);
--# return (Height/Ada_Tangent((3.14159/180.0) * Angle));
```

Checker passing in the simplified verification condition. In this example, the Proof Checker is not able to build a proof and the developer must consider the possibility that the code is incorrect. Careful inspection of C1 shows that the *angle* variable has been moved to the front of the expression during the simplification process. This inadvertently highlights a discrepancy between the code and the postcondition.

Figure 9 shows the original Ada_Radius specification and body. Note the parenthesis on line 14 of Figure 9 for the call to Ada_Tangent. The postcondition formula on line 7 includes parentheses around the Angle variable, but the code implementation does not. This is a simple error in parentheses. This error is discovered during development because the verification condition for Ada_Radius cannot be proven to be correct. SPARK has found an error during the development phase where it can easily be fixed. Had the error not been found until the testing or implementation phase, it would have proven more costly. This simple example illustrates how SPARK reduces the cost of software development by finding errors during the development phase.

Figure 10 shows the corrected Ada_Radius code. The parentheses have been correctly placed around the Angle variable. Now when the program development team executes the SPARK tools on the code, a proof can be built that shows the verification condition is true. SPARK is able to automatically prove this code is a correct implementation for the preconditions and postconditions.

This simple example shows the power of the SPARK correctness by construction methodology. Even on a simple example such as this, an error in the code was discovered. Students in a senior-level software engineering course developed this example code. They eventually noticed the error in their code during testing and corrected it in a later version. Had they been using the SPARK approach from the start, they would have found the error while constructing the code and delivered correct code the first time.

Conclusion

As we have seen, the SPARK approach to constructing code is a powerful way to prove that the code being developed is a correct implementation given the preconditions and postconditions. This approach is now being used to develop safety critical flight control software for UAVs at the U.S. Air Force Academy as part of a UAVSAT that is designed to enhance a commander's ability to respond to dynamic situations effectively. The added benefit of using the SPARK approach to software development is that it helps assure the system is correct by construction.◆

```
-- function to calculate the radius in Ada
function Ada_Radius (
      Height : in Float;
      Angle : in Float)
   return Float is
begin
 return (Height / Ada_Tangent((3.14159/180.0)) * Angle);
end Ada_Radius;
```

Figure 7: *Body of Ada_Radius*

```
function_ada_radius_3.
H1:  height >= 0 .
H2:  angle <> 0 .
H3:  ada_tangent(314159 / 100000 / 180) <> 0 .
    ->
C1:  angle * (height / ada_tangent(314159 / 18000000)) =
    height / ada_tangent(angle * (314159 / 18000000)) .
```

Figure 8: *Results of Simplification for Ada_Radius*

References

1. Barnes, John. High Integrity Software: The SPARK Approach to Safety and Security. London, UK: Addison-Wesley, 2003.
2. Croxford, Martin, and James Sutton. "Breaking Through the V and V Bottleneck." Lecture Notes in Computer Science. 1031 (1996).
3. Croxford, Martin, and Dr. Roderick Chapman. "Correctness by Construction: A Manifesto for High-Integrity Software." CROSSTALK, Dec. 2005 <www.stsc.hill.af.mil/crosstalk/2005/12/index.html>.
4. Amey, Peter. "Correctness by Construction: Better Can Also Be Cheaper." CROSSTALK, May 2002 <www.stsc.hill.af.mil/crosstalk/2002/05/index.html>.
5. Shaw, M. "Abstraction Techniques in Modern Programming Languages." IEEE Software Oct. (1984): 10-26.
6. Guttag, John V., and James J. Horning. Larch: Languages and Tools for Formal Specification. New York, NY: Springer-Verlag, 1993.
7. Sward, Ricky, Tim Beerman, and Clint Sparkman. "Unmanned Eyes in the Sky." Military Geospatial Technologies 3.3 (2005).

Notes

1. Retrieved from Praxis High-Integrity Systems <www.praxishis.com> on Apr. 24, 2006.
2. Retrieved from Google Earth, a 3-D Interface to the Planet <http://earth.google.com> Apr. 24, 2006.
3. Retrieved from Ada Language Reference Manual <www.adahome.com/rm95/>.

Figure 9: *Original Ada_Radius Specification and Body*

```
1  -- function to calculate the radius in Ada
2  function Ada_Radius (
3     Height : in   Float;
4     Angle : in   Float)
5   return Float;
6  --# pre (Height >= 0.0) and (Angle /= 0.0);
7  --# return (Height/Ada_Tangent((3.14159/180.0) * Angle));

8  -- function to calculate the radius in Ada
9  function Ada_Radius (
10    Height : in   Float;
11    Angle : in   Float)
12  return Float is
13 begin
14   return (Height / Ada_Tangent((3.14159/180.0)) * Angle);
15 end Ada_Radius;
```

Figure 10: *Corrected Ada_Radius Code*

```
-- function to calculate the radius in Ada
function Ada_Radius (
   Height : in   Float;
   Angle : in   Float)
 return Float is
begin
 return (Height / Ada_Tangent((3.14159/180.0) * Angle));
end Ada_Radius;
```

About the Authors

Lt. Col. Ricky E. Sward, Ph.D., USAF, retired from the Air Force as an Associate Professor of Computer Science at the U.S. Air Force Academy. He retired as the Deputy Head for the Department of Computer Science and the Course Director for the senior-level two-semester Software Engineering capstone course. Sward received his doctorate in Computer Engineering at the Air Force Institute of Technology in 1997 where he studied program slicing and reengineering of legacy code.

Department of
Computer Science
2354 Fairchild DR STE 6G101
USAF Academy, CO 80840
E-mail: ricky.sward@wavmax.com

Lt. Col. Mark J. Gerken, Ph.D., USAF, is an Assistant Professor and the Deputy Department Head for Technology in the Department of Mathematical Sciences at the U.S. Air Force Academy. He currently teaches probability and statistics and has taught calculus, differential equations, and engineering mathematics. Gerken received his doctorate in Computer Engineering at the Air Force Institute of Technology in 1995 where he studied software architecture and formal program development.

Department of
Mathematical Sciences
2354 Fairchild DR STE 6D112
USAF Academy, CO 80840
E-mail: mark.gerken@usafa.af.mil

2nd Lt. Dan Casey, USAF, graduated from the U.S. Air Force Academy in May 2006 where he majored in computer science with an emphasis in information assurance. Casey is currently stationed at Ramstein AFB, Germany where he works as a communications and information officer.

435th Communications Squadron
Ramstein AB, Germany
E-mail: daniel.casey@
ramstein.af.mil

Web Sites

BuildSecurityIn

http://BuildSecurityIn.us-cert.gov

As part of the Software Assurance program, BuildSecurityIn (BSI) is a project of the Strategic Initiatives Branch of the National Cyber Security Division (NCSD) of the Department of Homeland Security. The Software Engineering Institute was engaged by the NCSD to provide support in the Process and Technology focus areas of this initiative. The Software Engineering Institute team and other contributors develop and collect software assurance and software security information that helps software developers, architects, and security practitioners to create secure systems. BSI content is based on the principle that software security is fundamentally a software engineering problem and must be addressed in a systematic way throughout the software development life cycle. BSI contains and links to a broad range of information about best practices, tools, guidelines, rules, principles, and other knowledge to help organizations build secure and reliable software.

Software Assurance Technology Center

http://satc.gsfc.nasa.gov/

The Software Assurance Technology Center (SATC) was established in 1992 as part of the Systems Reliability and Safety Office at NASA's Goddard Space Flight Center (GSFC). The SATC was founded with the intent to become a center of excellence in software assurance, dedicated to making measurable improvement in both the quality and reliability of software developed for NASA at GSFC. SATC is self-supported with internal funding coming from research and application of current software engineering techniques and tools. Research funding primarily originates at NASA headquarters and is administered by its Software Independent Verification and Validation Facility in Fairmont, WV. Other support comes directly from development projects for direct collaboration and technical support.

National Institute of Standards and Technology (NIST) Computer Security Division (CSD)

http://csrc.nist.gov/

The CSD-(893) is one of eight divisions within Information Technology Laboratory. The mission of NIST's Computer Security Division is to improve information systems security by raising awareness of information technology (IT) risks, vulnerabilities, and protection requirements, particularly for new and emerging technologies. NIST researches, studies, and advises agencies of IT vulnerabilities and devising techniques for the cost-effective security and privacy of sensitive federal systems; developing standards, metrics, tests and validation programs to promote, measure, and validate security in systems and services, to educate consumers, and to establish minimum security requirements for federal systems; and developing guidance to increase secure IT planning, implementation, management and operation.

CERIAS

www.cerias.purdue.edu

CERIAS is currently viewed as one of the world's leading centers for research and education in areas of information security that are crucial to the protection of critical computing and communication infrastructure. CERIAS provides multidisciplinary approaches to problems, ranging from technical issues (e.g., intrusion detection, network security, etc) to ethical, legal, educational, communicational, linguistical, and economical issues, and the subtle interactions and dependencies among them.