Embedded Systems Conference – Silicon Valley
April 2-5, 2007
San Jose, CA

[ESC-310]

Design Techniques for Safety- and Security-Critical Systems

Robert B. K. Dewar

AdaCore

dewar@adacore.com

A year ago, the ESC proceedings contained my paper Safety-Critical Design Techniques for Secure and Reliable Systems. This paper extends the considerations to security-critical programs. Although traditionally a strong distinction has been drawn between safety and security, in the post-9/11 era many systems with safety requirements must also take security into account. Safety-critical software will need to satisfy security-based standards, and high-security software reused in safety-critical systems will need to satisfy safety-based standards. This paper explores the impact of these dual trends on software development.

What is a Safety Critical Program?

A program is safety-critical if human life depends on its correct operation. If there is a bug that results in a hazard, then death or serious injury can result. Typical examples are train signaling systems, avionics control, medical instrumentation, and space applications. Since the focus is on human safety, such programs must essentially be error free.

That's a strong requirement, especially given the common assumption that all large programs contain errors. But in our modern technological age we place our safety at the mercy of computer programs every time we board a train or plane, or enter a hospital, or even drive a car. We simply have to ensure that such programs are reliable, and as we will see in this paper, it is in fact possible and practical to achieve the seemingly very difficult goal of writing essentially error-free completely reliable software.

What is a Security Critical Program?

A program is security-critical if it must not only operate correctly, but do so in a potentially hostile environment. Certainly such a program must be free of internal defects, but it must also be designed to survive under all possible usage scenarios, including those with hostile elements such as humans intent on corrupting the results. In a sense the requirement for security is more demanding than that for safety: a bug in a safety-critical program may be tolerable if it does not lead to a hazard, but in a security-sensitive environment it must be assumed that virtually any bug can be exploited by antagonists.

Security-critical applications arise in many different settings. Obvious examples are banking, management of classified or trade-secret data, and management of personal data such as social security and credit card numbers. Another major area, which has been the focus of a great deal of attention recently, is voting machines, where it is essential that the system not allow tampering from the outside. Furthermore, in today's interconnected world, networking software and also any code that runs on a networked system, should be considered security critical.

The Relation between Safety and Security

Not all security-critical applications are safety-critical. If a voting machine is compromised, then there is no immediate injury or loss of line. On the other hand, the indirect results can be serious. No one could doubt in the turbulent political climate of today's world that the outcome of elections is in the long run a life and death issue. Similarly, compromising credit card records is not an immediate safety-critical issue, but for someone whose identify is stolen, the consequences can be extremely serious. We can't really imagine a situation in which one of these security-critical applications is somehow taken less seriously on the grounds that there is no immediate loss of life on a failure. Such programs should be just as reliable and error free as the most critical avionics control system: there is little point in worrying about outside interference if the program itself is unreliable. So in this regard, we have to be at least as vigilant in ensuring that the program is free of defects.

Looking at things the other way round, traditionally, safety-critical systems have focused on freedom from defects and fidelity to the specifications for the system, and security considerations have been secondary. For example, the Boeing 777 has a safety-critical avionics system (written entirely in Ada) that was subject to stringent certification and testing to meet the highest levels of safety-critical performance. But the issue of security as such was not a primary focus.

Since the days of the 777, the world has changed. One of the effects of the devastating 9/11 attacks was to make clear that everyone everywhere is at risk, and that all systems (especially those whose failure would cause great loss of life or property) are potential candidates for attack. An aircraft in flight can certainly be destroyed by a physical device such as a bomb, but it could also be brought down by compromising its avionics control system. Just how secure are such systems? On one episode of the TV series *Alias*, Marshall, the genius hacker, is flying for the first time and is nervous. At takeoff, he is furiously typing into his notebook computer, despite admonitions from the cabin attendant. He explains to Sydney that he has hacked into the plane's avionics system to ensure that the captain has not forgotten any items on the take off checklist. An amusing scene – but quite unrealistic – or is it? These days we should be reluctant to take anything for granted, and it would seem that any system on which human life depends is a potential target for deliberate corruption (nuclear power station control, air traffic control, and other society infrastructure systems all come to mind).

The bottom line is that safety and security concerns are merging. Traditionally these have been rather separate considerations. During the development of Ada 95, a language that specifically focuses on safety and security (there is a section of the Ada standard devoted to this topic), the language committee talked to both constituencies and was struck on the

one hand by how similar the technical concerns were, and on the other by how separate the two communities were. These days we can hardly afford such separation, and we need to have both safety and security in mind for a wide range of applications.

This paper will consider specifically the impact of safety and security requirements on software development. Note that this is only one of many concerns in total system development. Clearly hardware reliability and security are equally important, but this paper will focus on software aspects. Software can't solve all the problems, but unreliable software will undermine an otherwise perfect hardware solution.

General Approaches and Observations

The computing industry now has over fifty years of experience in writing large programs. During that period it has developed many techniques that can be refined to play a part in the design and implementation of security- and safety-critical software. Perhaps the most important and fundamental requirement is that everyone involved in such a design effort must adopt a disciplined view that is entirely quality-oriented. I once had a programmer working for me who said "It's a waste of time worrying about whether a loop is one-off, since you will find out during testing anyway." Such an attitude is the very antithesis of what we need if we are to succeed in writing reliable software.

As I am sure many of you know, I am an enthusiastic Ada supporter, and I should disclose that right away, though what I have to say here is certainly not Ada specific. But I will say that one of the advantages of Ada in this area, apart from some important objective features, is that Ada was designed with this kind of quality orientation in mind, and the culture that surrounds Ada tends to have this emphasis. Even if you are not using Ada in a critical application, you will do well to borrow this mindset. As mentioned earlier, the Ada 95 standard has a section entitled "Safety and Security", and even if you are not an Ada programmer you will find this section interesting reading.

This may seem like a trivial observation, but in my experience the issue of culture and attitude is a critical one. If a team is totally dedicated to quality, it is far more likely to achieve its goal. Nevertheless, even the most dedicated team needs the tools and procedures that will help ensure success, and we will now examine some of the main factors that help ensure success in writing totally reliable programs.

Programming Language Design

In this section, we examine the influence of programming language design on the production of security- and safety-critical software. Of course it is possible to do basically anything in any programming language, and one can even prove that statement in some theoretical sense. However, we know from experience that the programming language design is definitely significant and can affect the ease of writing programs and demonstrating their correctness. Let's look at what languages need to provide, from the vantage points of both safety and security.

We first note that C, Java, and C++ are not suitable languages for such critical software. Before you dismiss these as biased claims from an Ada enthusiast, we rush to add Ada to the list of unsuitable languages. What do we mean by this rather outrageous statement? The point is that each of these languages, in its entirety, is too complex. We can't let

programmers use the full power of any of these languages; even C has too much functionality. The considerable capabilities of modern programming languages make it easier to write code in the first place, but make it more complicated to demonstrate that the resulting code is error-free.

So what do we need to do? The answer is that for any of these languages we need to choose an appropriate subset (or subsets), so that we write our programs avoiding features with overly complex semantics. For instance, in Ada, we most likely avoid using the full power of the tasking model. In C, we exclude some of the library routines which are unlikely to be certifiable in a safety-critical environment. For C++ we avoid the complex use of templates. For Java, we avoid the use of dynamic features that allow the program to modify itself while it is running. (Of course, there are issues besides size that can interfere with a language's ability to support safety- or security-critical development. For example, C has a number of error-prone features that can hinder a program's readability. If we see the construct "`if (X=Y) …`" are we sure that the programmer really meant to assign `Y` to `X`, and not compare the two for equality?)

The exact choice of the set of features to be used is a challenging language design task, and the base language may be more or less helpful in this process. Ada provides considerable flexibility; the built in notion of pragma `Restrictions` allows the programmer to choose the features to exclude on an *a la carte* basis, and pragma `Profile` facilitates the standardization of common sets of restrictions. Another example of a language subset intended for critical applications is MISRA C:

www.misra-c2.com

And a further example is the SPARK Ada subset from Praxis High-Integrity Systems:

www.praxis-his.com/sparkada

which we will talk about in more detail later. These examples show that coherent subsets can be designed that makes the use of the language more effective for safety- or security-critical purposes. The MISRA group is currently designing an analogous subset of C++, to be called MISRA C++.

What features should we look for in a language to be used for safety- or security-critical programming? Most obviously we want to favor compile time checking so that we can find as many problems as possible at compile time rather than at run-time. Ada is an example of a language that is designed with this criterion in mind. Programmers learning Ada often comment that it is hard work to get the compiler to accept a program, but once the program compiles successfully, it is far more likely to run correctly the first time than an analogous program in C or C++. Ada achieves this effect in part by providing a much more comprehensive type system. For example, users can define their own floating-point types distinct from the built-in ones, allowing the compiler to check (at compile time) that you are not doing something that makes no sense like adding a length to a velocity. With less type-safe languages (including C, C++ and Java) such errors would only be detected during testing.

Another important issue is run-time checking. Again, using Ada as an example, there are many run time checks that are required to raise exceptions if they fail. This is in contrast with C and C++, where (for example) out-of-range array indices are not detected, leading

to the well-known buffer overflow problem that is the source of so many insecurities. As any Ada programmer knows, these checks and resulting exceptions are enormously valuable in finding errors in the early stage of testing, rather than later on in the development process. The issue of whether such checks should be suppressed in the final product is an interesting one. From a pure reliability point of view, it is preferable to demonstrate statically that a program is free of any possibility of run time errors, which argues against including the checks. On the other hand, run time checks can provide an important extra measure of security-oriented checking.

Let's look at an example from the world of voting machines. Typical voting machines record the results on some kind of magnetic medium, such as a flash memory card. Clearly one important criterion is that the vote total for each candidate is zero before the election starts, and we expect the software to check this. In one well-publicized case a machine performed such a check, but did so by summing the votes for all the candidates and then checking that the total was zero. Unfortunately, there was nothing in the program to prevent negative numbers of votes (a good guess is that this software was written in C, and the relevant variables were signed `int` variables). Quite easily someone discovered that the card could be preloaded, for example with -1000 votes for candidate A, and +1000 votes for candidate B, and the test for a zero total worked.

Now obviously this is a bug, but the key to security is multi-layered redundancy so that errors like this get caught somewhere along the line. If this software had been written in Ada, the programmer would have been forced to consider an appropriate range for the relevant variables and would have probably written something like

```
type Vote_Count is range 0 .. 100_000;
--  Vote count cannot be negative, and anything more than 100,000
--  votes on a single machine is not realistically possible.
```

A variable of type `Vote_Count` has an integer value that must be at least 0 and no greater then 100,000. Under the scenario described above, when the initial vote counts were read in, the attempt to pre-load with -1000 would raise a run time exception. At worst this run time exception would not have been anticipated, causing the system to shut down, but it is much preferable to have the machine fail (and, e.g., cause a reversion to paper ballots) than to silently record wrong votes.

Though we have emphasized simplicity in language subset selection, we nevertheless have to recognize that security- and safety-critical applications are getting more complex, and these requirements must be accommodated. We mentioned that Ada's full tasking capabilities are not appropriate for safety- or security-critical applications. However, support for multi-tasking is becoming more and more important in these areas. One of the important additions to Ada 2005 (the latest version of the Ada language) is the Ravenscar tasking profile which is specifically intended for high-integrity systems including those with safety- or security-related requirements:

www.stsc.hill.af.mil/crosstalk/2003/11/0311dobbing.html

This article provides an excellent introduction to the Ravenscar profile, with some useful insights into the design criteria and usage. Another interesting effort is the Real-Time Java activities currently underway. This work is an attempt to address a number of

problems that interfere with Java's use in real-time applications (e.g., inadequacies in the Java thread model, and unpredictability from Garbage Collection). For details, see:

www.embedded.com/showArticle.jhtml?articleID=16100316

The choice of language is always a hotly debated issue. We started out by noting that any problem can be solved in pretty much any language, and that is certainly true. Indeed, safety-and security-critical applications have been written in many different languages. Nevertheless the language choice does make a difference, and it is no accident that Ada finds its widest use and support in the context of large safety-critical applications such as air traffic control.

The Use of Formal Methods

Given the desire to demonstrate that a program is completely reliable, a natural approach is to prove correctness in a mathematic sense. That would avoid relying on testing or any other subjective measures. During the 1970s and 1980s, the notion of proof of correctness was all the rage in academic circles, and still today there are academic computer scientists who assume that this is *the* solution to the problem of writing reliable code.

What's wrong with this viewpoint? Most significant is the issue of what "correct" means. The standard model is to first define a precise formal specification of the problem, and then prove that the program correctly implements this formal specification. Unfortunately there is a huge hole in this approach. How does one come up with the formal specification? For small academic problems, like sorting an array of numbers, it is tractable to write down a formal specification in an appropriate language and then construct a mathematical proof that a given program meets this specification. In order to actually have confidence that the proof is correct, it is necessary to verify the proof using a mechanical process, but that is also quite feasible for small cases.

But how about large applications? First of all there are aspects of large programs that are just not easy or even possible to formalize. For example, a pilot's cockpit must present a user-friendly interface. The notion of "user-friendly" is hardly a formal one. Another problem is that for a large program, the specification is itself a huge document. Furthermore it is written in a formal specification language that may be harder for many people to read than a normal program in a conventional programming language. How does one know that the specification itself is complete and consistent? In fact we don't, and the problem of writing a correct program has simply been transformed to the (arguably more difficult) problem of writing a correct specification.

For these reasons, the notion of proving entire large applications correct has largely disappeared. That's particularly true in the U.S., where typical academic programs are far more likely to offer courses in Unix Tools, and Web Programming than in formal logic and proof of correctness.

So is this approach a dead end? Not at all! Although proof techniques and formal methods/tools are not the only answer, they can still play a very important role. This seems to be more appreciated in Europe than in the U.S.; for example, the U.K. Ministry of Defence standard for safety-critical programs requires the use of formal methods (although it is not very specific on what this means). So it is perhaps not surprising that a British company, Praxis High Integrity Systems – www.praxis-his.com – is one of the

leading practitioners in the area. Praxis has shown that, although total proof of correctness might not be feasible, proving that specific properties hold for a given program is both useful and practicable. An example is the issue of dealing with exceptions. Ada defines a number of run time conditions that cause exceptions to be raised (such as an array index out of range). An exception occurrence generally corresponds to an error, and we certainly don't want a safety- or security-critical program to contain such errors. Proving that a program is free of any possibility of exceptions is a well-defined, tractable problem that has actually been solved for non-trivial software. For details, see:

www.praxis-his.com/pdfs/Industrial_strength.pdf

In order to construct such proofs, it is essential that the program be written in a relatively simple, precisely-defined language. For this purpose, Praxis has designed SPARK, a subset of Ada that is enhanced with static annotations that, for example, specify which variables can be accessed where. The SPARK Examiner tool verifies these conditions, and other Praxis tools allow the proof of specific properties of a program. The lesson learned is that formal methods and proof tools are not just an academic exercise, but are usable in practice as an important tool in the arsenal of the safety-critical programmer.

Let's apply these ideas to our previous example of voting machine software reacting to potential tampering (negative vote counts). As we saw, the run time checks would cause a failure here, but really the programmer should anticipate this potential insecurity and deal with it cleanly.

Our model Ada implementation of this software would have a statement similar to:

```
Candidate_Count (Candidate_Number) := Read_Initial_Value_From_Card;
```

Where `Candidate_Count` is an array, `Candidate_Number` is an index into this array, and `Read_Initial_Value_From_Card` is a function that retrieves the pre-stored vote count for the given candidate.

Static analysis of this statement would show that a check was needed to ensure that the value read from the card met the range constraint for a candidate count, and hence excluded negative values. But an attempt to prove that the program could not raise any run-time exceptions would fail, because the function call might indeed return a negative value. As a consequence it is necessary to insert an explicit test that the returned value is in the expected range. If this check fails, the program could then display an appropriate diagnostic noting that the inserted card was corrupted, requiring the election official to obtain a new card, or reinitialize the existing one.

Testing, Testing, Testing

If we cannot in practice prove all the properties that we need to demonstrate, how can we ensure the program's safety? One answer is given in the title of this section. Admittedly, generations of programmers have been taught the simple observation that testing can never demonstrate the absence of bugs; it can only show their presence. This is certainly true from a theoretical point of view, but still, we definitely put more trust in a program that has been tested than in one that has not, and the more thorough the testing, the more we trust it.

In practice, are there testing approaches that are sufficiently thorough that we are willing to literally risk our lives on the resulting demonstration that there are no known problems? That's an enormously significant question.

The DO-178B standard, used by the FAA to certify commercial aircraft avionics systems, provides an affirmative answer through a comprehensive testing-based approach comprising two major elements. It first specifies requirements on generating systematic functional tests. These tests must exercise all functional aspects of the program, at all levels of abstraction. The tests are derived in general terms from the problem statement and specification, and at a more detailed level from the actual code of the program, to make sure that every detail of the logic works correctly.

DO-178B then requires full coverage testing: the test suite must cause every statement in the software to be executed at least once. That of course doesn't demonstrate correctness, but statements that have never been executed do not inspire much confidence. That may seem like an obvious and simple observation and requirement, but in practice, most large non-safety-critical programs are not tested in this way, even though tools are available to help automate the process. For example, the failure of the AT&T long lines system was due to the execution of error recovery software that had never been tested.

The DO-178B standard has several different levels, labeled A through E, corresponding to different requirements for safety. Level A certification, the highest (most stringent) level and the one associated with life-critical systems, imposes an additional requirement regarding the tests for flow of control. Consider:

```
if condition then
      statements
end if;
```

Simple coverage testing will only ensure that the statements have been executed, but perhaps the test suite always has `condition` set to true. That's not really enough. We also want to know that if `condition` is false, it is safe to skip the statements.

A more complicated example is

```
if condition1 and condition2 then
      statements;
end if;
```

Various combinations of conditions need to be tested, to make sure that all possibilities are covered. But not all possible conditions need to be tested. In particular, if `condition1` is false then we don't care about `condition2`, but we would like to test the following three cases:

`condition1` false
`condition1` true, `condition2` true
`condition1` true, `condition2` false

The testing regime that ensures this is coverage called *MC/DC* (modified condition/decision coverage), and there are tools to enforce the requirement that the set of tests include all cases. For additional information, see:

www.dsl.uow.edu.au/~sergiy/MCDC.html

which contains a very thorough bibliography on this technique.

An interesting issue is which version of the program to analyze through coverage testing: the source code or the object code. We can't fully trust compilers because they are far too complex to be themselves fully certified (or "qualified as development tools", in DO-178B parlance). So what should we do? There are two approaches. One is to conduct all testing at the object code level. (This is for example, the approach used by the Verocel tools, see:

www.verocel.com/do178b.htm

for details.) The other approach is to perform coverage testing at the source code level, but then it is necessary to demonstrate full traceability between the source program and object program. Both approaches have been used successfully, and both have their advocates (at AdaCore we have seem some fierce arguments between these two schools of thought in some of the projects we have worked on).

It must be emphasized that DO-178B is not simply a mindless set of objective rules to be applied or checked. At the heart of the process are humans exercising judgment. These DERs (Designated Engineering Representatives) are independent authorities responsible for ensuring that the rules have been followed to the letter and in spirit. They are the "building inspectors" of the critical software engineering industry, and their extensive experience helps to make sure that the standard works in practice.

How effective is the testing regime that DO-178B imposes? The pragmatic answer is that it is rather successful. Remember that software need not be 100% guaranteed as totally error free. Rather it must be hazard-free (i.e., errors cannot compromise safety), and thus reliable enough so that it is not the weak link in the chain. If we take commercial avionics as an example, many lives have been lost due to various hardware failures on scheduled commercial flights, but no lives have been lost (as far as can be determined) as a result of software bugs. That's an impressive record. Of course there can always be a first time, but so far the industry (using DO-178B) has done a good job.

However, a good job now might not be good enough in the future: as hardware speed and memory capacity continues to increase, applications will grow more complex to exploit these advances. Software technology is trying to keep pace. As one example, work in the formal methods area is attracting increased interest, which should lead to the more effective and widespread use of proof techniques. As another example, programming languages continue to evolve, as illustrated most recently by the new Ada 2005 standard, introducing specific facilities that make it easier to write reliable programs. And as a third example, the whole area of software testing is progressing to account for methodologies that have spread into common practice. Work is underway on DO-178C, the eventual successor to DO-178B, and one of the major discussion items is how to address object-oriented techniques, which historically have not been used in avionics software but which are now attracting considerable attention. We will address this issue in more detail in a later section.

Testing and Security

The previous section on testing addressed traditional safety-critical testing, but it did not mention security. Indeed that section is largely unchanged from last year's paper, which

focused exclusively on safety. So, let's ask the question: can testing verify security? The answer: not really. There is a big difference between a program that is operating in a stand alone manner with a well defined environment, and one that is operating in a hostile environment with hackers attempting to derail it. Let's look at the Microsoft Windows system as an example. This system is furiously tested by thousands of testers. Out of the box, and operating in an isolated mode, it is remarkably reliable. But connect it to the Internet and the world of thousands of mischievous and criminal hackers descends on your machine. Even with layers of security oriented software, these hackers can get through and cause severe damage. In the context of security considerations, Windows is a disappointment, and it is no surprise that Microsoft has reoriented itself to a new dedication to security considerations.

Testing is simply not sufficient when it comes to security. We can realize some benefits from testing in which a system is deliberately subjected to attack by an army of hackers hired for this purpose. Indeed experts in the voting machine area have suggested that this approach be included in the mandated testing procedures for federal voting machines, but we can never be sure that the hackers on our side will find all the security flaws.

We said earlier that no lives had ever been lost because of errors in commercial avionics software. That's true, but it's just a little misleading. There has in fact been at least one instance, involving a Malaysia AirlinesB777 flight from Perth to Kuala Lumpur in August 2005, where a software error led to a hazardous midair situation. Details on this incident may be found in a report from the Australian Transport Safety Bureau:

www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx

Luckily the only effect in this case was a very bad scare for the passengers and crew, but the existence of this error was worrisome. The question arises as to whether evil-intentioned individuals could have exploited such an error to cause real damage.

While testing will always be a useful component, not only in finding errors, but in building confidence (the general public will always be more confident in systems that have been tested, and they have good reason for this point of view), it must be acknowledged that testing by itself is not going to be sufficient.

The current situation with voting machines is that commercial companies develop these systems (including the software) internally, and keep everything secret. The software's source code is inaccessible to outsiders, and there is no information on how it was developed. Then the machine is shipped to a testing lab. If it passes the lab's tests, then it can be certified for use.

In other words, for voting machines, society is indeed relying only on testing (actually the weak "black-box" testing in which the testers do not have access to the source code), and hoping for the best on all other fronts. This cannot be a successful approach and can never demonstrate that the machines are reliable or secure. This would be true even if no malfunctions had been detected. In fact there are numerous examples demonstrating severe vulnerabilities in these machines, further showing that testing alone is not going to be sufficient. Formal methods, and other end-to-end development techniques, are necessary.

The Use of Tools

When we are aiming at perfection, we need to take full advantage of all the tools at our disposal. We can achieve much by through careful reading of programs by software experts, but often we do even better by using automated tools. There are many general categories of such tools.

Static analysis tools analyze the structure of a program to detect potential errors and to provide information that will help find problems before they cause trouble. An example of such a tool is CodeSonar from Grammatech:

www.grammatech.com/products/codesonar/overview.html

This tool automatically finds errors such as buffer overruns in C++ programs. There are many such tools from many suppliers. Of course no tool of this kind can guarantee that a program is bug free, but each analysis identifying the absence of a certain type of error increases one's confidence in the overall correctness of the software. It is the sum total of this information and effort that leads us to be willing to get on the plane that will be deploying the software at the end of the process. Choosing an appropriate set of tools and developing experience in their use can be as important as language and compiler selection. Note that the tool set is also likely to be language dependent. For example, in Ada there is less concern with buffer overflow, since the built-in language semantics and exception mechanism will detect such problems at run time.

Compiler vendors often provide useful suites of such tools, and evaluation of the full tool suite should be an important part of the selection process for languages and compilers. For example, my company, AdaCore, provides a complete suite of tools, including a static stack usage analyzer that addresses the specific and important requirement that a program does not cause any stacks to overflow.

There are many different kinds of tools available for analyzing program properties such as thread/task schedulability, worst-case timing, run-time use of storage, freedom from race conditions, and freedom from unwanted side effects. There are also other tools that help in automating the testing process, generating program metrics, etc. An important part of the preparation for a project with safety- or security-critical requirements is to investigate and acquire a coherent set of tools. An Integrated Development Environment (IDE) can be used to organize such a set of tools. For example, the GNAT Programming Studio (GPS) product from AdaCore provides a menu-driven IDE giving access to the full set of tools that a project will use, and there are many other such products.

Again, tools that just check for internal consistency are not sufficient to verify safety or security properties of a program. That requires a more systematic approach in which specifications and formal verification play an important role. The SPARK language and its associated tools give an important indication of what can be achieved.

Object-Oriented Programming and Safety-/Security-Critical Systems

Object-Oriented Programming ("OOP") has become an important methodology for modern programmers and is supported in a wide variety of languages including C++, Java, and Ada. In this section we will look at the special considerations of using these methods in security- and safety-critical environments.

The notion of OOP is somewhat ill-defined. On the one hand it refers to a design method in which objects communicate via message passing. Such an approach in and of itself poses no special problems or safety/security concerns. On the other hand, it also refers to the use of a set of features in programming languages, typically comprising three important components:

- The ability to extend existing types by adding new data elements
- Automatic inheritance of existing methods when types are extended, along with the ability to override such methods and/or add new ones
- Dynamic dispatching, entailing automatic choice of the right method at run time, based on the data type of the object referenced by the method's argument

The first two features offer no special obstacles in a security- or safety-critical environment, and it is worth noting that they are useful even without dynamic dispatching. For example, a type and its associated methods may be defined in a library. An application can then import this type, extend it to specialize it for a particular purpose, and then use the inherited operations on this type. In Ada, one can obtain the type extension and method inheritance/overriding capabilities, and exclude dynamic dispatching, by specifying pragma `No_Dispatch`, which, as its name implies, checks that a program does not use dynamic dispatching. An Ada compiler can recognize this pragma and enforce the restriction, as well as improve the code knowing that this restriction is in place (for example, by eliminating dispatch tables). Similar switches or pragmas could be implemented in other languages, though they would not be part of the standard.

Now let's look at dynamic dispatching. As background, note that the typical implementation is a table of pointers to methods, where an index into this table identifies the method that is to be invoked. However, that approach raises two safety/security issues. First, what if the table gets corrupted somehow? The indexing / dispatching operation could then cause a wild jump. Now of course such corruption would not be expected in a certified program (although the demonstration of correctness of the dispatch table raises some nontrivial issues). Still, as will be described below, indirect calls make safety certification more difficult: to ensure the integrity of the control flow we need to prove properties relating to data access. In the security context, we know how a table might get corrupted, namely by an evil corrupter deliberately attacking the system from outside. By changing the address in one dispatch table, a bad guy can compromise the entire behavior of a program.

The second problem with dynamic dispatching is more significant. In a sense dynamic dispatching is all about not having to know what routine you are calling. But certification and coverage testing is all about knowing and checking the control flow of your program. What exactly needs to be done for a dispatching call?

One possibility would be to treat each call as though it were a `case` statement, with a branch for each routine that might be the target of the dispatching call. This translation seems completely fair, but the trouble is that only a small subset of the possible targets might actually be invoked. The uncalled routines would then be deactivated code (code that can never be executed), raising DO-178B certification issues.

A simpler approach is to treat the dispatching call as a call to a single routine that contains such a `case` statement. In this approach all calls to a given dispatching routine will share a single `case` statement. On the positive side, one can argue that the program could have been written in this way in the first place and thus that traditional testing is sufficient. On the negative side, coverage testing is really only showing that each method is used somewhere, and thus the possible flows of control are not being completely verified.

The fact that the program could have been expressed in an alternative style is not a convincing argument. The testing schemes implied by DO-178B are not perfect (no testing scheme could be), but they work well in practice. However, they can in principle be subverted by a programmer concentrating on the letter of the standard, and ignoring its intent. Here is a way of essentially removing **`if`** statements from a program. In Ada syntax, replace each **`if`** statement:

```
if condition then
    then-statements
else
    else-statements
end if;
```

with the semantically equivalent procedure call:

```
Eval_If (condition,
         then-statements-proc'Access,
         else-statements-proc'Access);
```

where the second and third arguments are now pointers to functions that will execute the appropriate statements. The `Eval_If` procedure itself has the form:

```
procedure Eval_If (Cond : Boolean; T, E : access procedure) is
begin
      if Cond then
            T.all;
      else
            E.all;
      end if;
end Eval_If;
```

(The "*ptr*`.all`" notation means an invocation of the procedure designated by *ptr*.) Now we have only one **`if`** statement in the entire program, the one inside this routine. MC/DC analysis, which is intended to ensure that all conditionals are tested thoroughly, is now subverted. Coverage testing now only proves that some **`if`** somewhere is true and some **`if`** somewhere is false.

This is pretty clearly cheating. Even though it meets the letter of the certification requirements, it does not meet the spirit, and we suspect no DER will allow this or any similar subversion.

So here is the question: is it cheating if dispatching calls are converted to a single shared `case` statement? The answer is not yet known. At least one system has been certified using this approach as far as we understand, but other projects are analyzing this issue and have not yet reached a decision. The lesson to be learned here is that dynamic dispatching is best avoided if possible in applications that need to be certified against

requirements such as in DO-178B. If dynamic dispatching must be used (for example if you are planning to reuse some OOP code in a safety-critical application), you need to deal with the resulting issues and follow the evolving state of the art in this area.

An excellent summary that deals with the whole issue of certification of object oriented software can be found in:

www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/

This references a four volume set *Handbook for Object Oriented Technology in Aviation*, which is a "must-read" for anyone considering the use of object oriented techniques in safety-critical programs.

Conclusion

As we have noted earlier, the certification of safety-critical software is at this stage a well understood activity, and has allowed us to repeatedly produce large scale systems that are safe and reliable. This technology will continue to improve in the future. However, the introduction of security concerns means that we have to go beyond these techniques.

Of course, not every one is working on safety critical systems. However, as we noted at the start of this paper, nearly everyone is in favor of reliable software, and it seems to us that many of the techniques that have been developed in the security and safety critical area deserve wider use. When eBay went down for nearly a week at one point due to software problems, causing the valuation of the company to lose several billion dollars, I wrote a note to the founders of eBay suggesting that since they had a huge company depending on one relatively straightforward program, it would make sense to adopt a much more strenuous view of reliability. Lives were not at stake, but a few billion dollars is real money! I did not receive a reply, but I think in the future that we will come to demand a level of reliability and security in a wide variety of critical programs. Indeed the eBay program can be considered a good example of software which should be treated as security critical given its accumulation of personal data, and its responsibility for billions of dollars in transactions.

Reliability, safety, and security are hard goals to achieve, and require up-front attention in terms of software architecture, design and implementation. The choice of appropriate programming languages (in fact, programming language subsets), development tools, and verification methods that go beyond testing and include formal methods, can help make the challenge manageable.