Embedded Systems Conference
April 3-7, San Jose
[ESC-447]

Safety-Critical Design Techniques for Secure and Reliable Systems

Robert B. K. Dewar
AdaCore
dewar@adacore.com

What is a Safety Critical Program?

The traditional definition of a safety-critical program is one in which human life depends on the correct operation of the program. If there is a bug in such a program, then death or serious injury can result. Typical examples are signaling systems on trains, avionics control systems, medical instrumentation, and space applications. Since the focus is on human safety, we apply requirements to such programs that essentially require that they be error free.

That's a strong requirement, especially given the common wisdom that all large programs contain serious errors. But in our modern technological age we place our safety at the mercy of computer software programs every time we board a train or plane, or enter a hospital, or even drive a car. We simply have to ensure the reliability of such programs, 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.

Although we are definitely focusing on safety-critical software in this paper, it is worth noting that in our modern complex world, more and more critical functions depend on computers. For example, banks rely on programs for controlling the international financial system. People may not die directly as a consequence of major failures in this area, but there are still awful consequences if such systems fail. Similarly, much of our general infrastructure, including phones, the internet, water and electricity supplies, and many other critical needs of modern life are dependent on software. If we can indeed devise methods of writing completely reliable code, we can certainly find more applications for such techniques than traditional safety-critical programs encompass.

One of our important theses in presenting this material is that all programmers should be aware of these techniques so that some appropriate subset of them can be applied much more widely. Almost no one will say that reliability is uninteresting for a planned software project. We may not be able to afford to make every application completely reliable, but for sure we can do a better job in the future in reaching this goal, and we can definitely extend the notion of high integrity systems beyond the domain of safety-critical programming alone.

General Approaches and Observations

We now have over fifty years of experience in writing large programs. During that period we have developed many techniques which can be refined to play a part in the design and implementation of safety-critical software. Perhaps the most important and fundamental requirement is that everyone involved in such a design effort must orient themselves to 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 kind of culture.

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 critical aspects 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 safety-critical software. First, we should start by noting that it is possible to do anything in any language. One can even prove that statement in some theoretical sense. However, we know from a lot of experience that programming language design is definitely significant and can affect the ease of writing programs.

When it comes to writing error-free code, it is most certainly the case that we want all the help we can get, and to the extent that programming language design can help avoid errors, we definitely want that help. So let’s look at which languages need to provide in this area.

As a starting point, we will note that C, and Java and C++ are not suitable languages for writing safety-critical software. And before you start thinking that these are peculiar statements from an Ada enthusiast, we rush to add Ada to the list of languages not suitable for safety-critical software. What do we mean by this rather outrageous statement? The point we are making is that all of these languages, in their entirety, are too complex to be used in this arena. We can’t let programmers use the full power of any of these languages; even C has too much functionality. The extended functionality of

modern programming languages does make it easier to write code in the first place, but we have to worry about demonstrating 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 subset the language, so that we write our programs in a very well understood subset of the chosen language, which avoids unnecessarily complex semantics. For instance, in Ada, we most likely avoid using the full power of Ada tasking. In C, we exclude some of the C library routines which are unlikely to be available 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-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. In the case of Ada for example, the built in notions of pragma Restrictions and pragma Profile make it much easier to specify and control the resulting profile. Two interesting examples of such language subset designs are MISRA C:

www.misra-c2.com

and the SPARK Ada subset:

www.praxis-his.com/sparkada

about which we will have more to say later. These are examples which show how a coherent subset can be designed that makes the use of the language more effective for safety critical purposes. The MISRA group is currently busy designing a similar subset of C++ to be called MISRA C++.

So what features should we look for in a language to be used for safety critical programming? Most obviously we want to favor compile time checking that can find as many problems as possible at compile time. Ada is an example of a language that is designed with this criterion in mind. Programmers learning Ada for the first time often comment that it is hard work to get the compiler to accept a program, but when it does, the program is far more likely to run correctly the first time. That characteristic may be a bit annoying for rushing out programs rapidly where reliability is not paramount, but for safety-critical programming it is just what we want. Ada achieves this partly by implementing a much more comprehensive type system, in which for example there are multiple integer types, and the compiler can check at compile time that you are not doing something that makes no sense like adding a length to a time.

When we subset the language, we try to avoid aspects of the language which violate this important criterion, and as MISRA C shows, even a language which starts out with almost the opposite criterion can be improved considerably by careful subsetting.

Another important issue is run time checking. Again, using Ada as an example, the Ada language defines many run time checks that are required to raise exceptions if they fail. 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 enabled in the final product is an interesting one. On the one hand, we would prefer to demonstrate that a program is free of any possibility of run time errors. On the other hand, it provides an extra safety belt in case of a problem sneaking through our careful procedures. But for sure such run time checking is invaluable during the testing process.

Though we have emphasized simplicity in language subset selection, we nevertheless have to recognize that safety-critical applications are getting more complex, and we have to be able to accommodate these requirements. We mentioned that the full tasking capabilities of Ada are probably not appropriate for safety-critical applications. However, support for multi-tasking is becoming more and more important. 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 safety-critical use:

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

This provides an excellent introduction to this feature, with some useful insights into the design criteria and usage. Another interesting effort is the Java real-time development work. The original Java thread specification is inadequate, and this work aims to correct that. 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 any language, and that is certainly true. Safety-critical applications have been written in many different languages. Nevertheless choice of language is important 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 that we want to demonstrate that a program is completely reliable, a natural approach is to decide that we should prove the program correct in a mathematic sense. That way we won't have to rely on testing or any other subjective measures. Some twenty years ago, 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? Well most significantly, we have to figure out what correct means. The standard model is that we need a formal specification of the problem, and then we will prove that the program properly implements this formal specification. Unfortunately there is a huge hole in this approach. How do we come up with the formal specification? For small academic problems, like sorting an array of numbers, we can indeed write down a formal specification in an appropriate language and construct a mathematical proof that a given program meets this specification. In order to actually have confidence in the proof, we need to verify the proof using a mechanical process, but that is also quite feasible for small cases.

But what of 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 do we know the specification is right? The answer is we don't, and the problem of writing a reliable correct program has simply been transformed to the problem of writing a reliable correct specification.

For these reasons, the notion of proving entire large applications correct has largely disappeared from view. That's particularly true in the US, 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! It is just that we have to recognize that proof techniques and the use of formal methods tools are not the only answer, but that does not mean they cannot play a very important part. In England, there is much more emphasis on formal methods. For example, the MoD standard for safety-critical programs requires the use of formal methods (although it is not very specific on what this means). So it is not surprising that to learn more about this approach, we should find that a British company, Praxis High Integrity Systems, is one of the leading practitioners in the area. For general details on this company, see:

www.praxis-his.com

What Praxis is able to show is that although total proof of correctness may not be feasible, it is still very useful to be able to prove specific properties hold for a program. As an example, in Ada, there is a very clear definition of a set of run time conditions that cause exceptions to be raised. An exception in Ada generally corresponds to an error situation, and we certainly don't want a critical program to contain such errors. The task of proving that a program is free of any possibility of exceptions is a well defined task, and has actually been achieved 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 very well defined language. For this purpose, Praxis has designed SPARK, a subset of Ada, which is enhanced by the addition of static annotations, which for example say what 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. Many other companies are also working in this area, but Praxis is definitely a leader in this area, and what we can learn from this is that the use of formal methods and proof tools is not just an academic exercise, but is useable in practice as an important tool in the arsenal of the safety-critical programmer.

Testing, Testing, Testing

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

Can we in practice devise 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 certification standard includes many aspects, but the most significant is a thorough testing approach that tries to answer this question affirmatively. It does this in a two-pronged approach.

First, it specifies an approach for generating systematic functional tests. These tests must test 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.

Then we insist that full coverage testing be done. This means that in our test suite every statement is executed at least once. That doesn't guarantee anything, but we really don't have much confidence in statements that have never been executed. This 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 for such testing. 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 a number of different levels, corresponding to different requirements for safety. For level A certification, the highest level for the DO-178B standard and the one we associate with life-critical systems, there is an additional requirement regarding flow of control. Consider:

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

Now here simple coverage testing will only ensure that the statements have been executed and it could be that 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;
```

Now here we really want to test various combinations of conditions to make sure that all possibilities are covered. But we don't need to test all possible conditions. In particular, if condition1 is false then we don't care about condition2, but we would like to test the following three cases:

condition 1 false
condition 1 true, condition 2 true
condition 1 true, condition 2 false

The testing regime that ensures this is 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 on this approach, see:

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

which contains a very thorough bibliography on this technique.

One interesting issue is whether to apply the coverage testing to the source 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. We can either do 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 on this approach. The other approach is to do coverage at the source program level, but in this case it is necessary to establish full traceability between the source program and object program. Both approaches have been used successfully, and both have their advocates (we have seem some fierce arguments between these two schools of thought in some of the projects we have worked on).

One important aspect of DO-178B is that it is not simply a mindless set of objective rules. At the heart of the process is a human exercising judgment. These DERs

(Designated Engineering Representatives) are independent authorities whose job it is to make sure 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 well does the testing regime that DO-178B imposes work? The pragmatic answer is that it is pretty successful. Remember that we don't really require software to be 100% guaranteed to be totally error free. Rather we want to make sure that we can write software that is 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 we can determine) as a result of software bugs in this arena. That's a pretty impressive record. Of course there can always be a first time, but so far we have done pretty well.

That does not mean we are content with the current state of affairs. We are working hard on improving our understanding of formal techniques, so we can use proof techniques more effectively. We are working on improving our programming languages so that they make it easier to write reliable programs, and we are improving our testing approaches. For example, the current work on DO-178C is addressing the issue of object oriented techniques, which we examine in more detail in the next section. Still, we are doing pretty well. Often you will hear people say that our software technology is terrible and all big programs contains serious bugs. Well that may be true of some areas, and for example we worry a lot about automobiles, where safety standards have not caught up with the increasing use of computers in cars. But in areas where we apply strict certification standards, we have a string of successes.

Of course we can't cover the DO-178B standard in detail here, but we have outlined some of its important features. For further details on this standard, a good source is:

www.software.org/quagmire/descriptions/rtcado-178b.asp

There are many other resources on the net, that describe approaches that various vendors have taken in meeting the requirements of this standard.

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 a lot by careful reading of programs with software experts doing the reading, but often we do even better by using automated tools to help with this process. There are many general categories of such tools.

Static analysis tools analyze the structure of a program to detect 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 tools of this kind can guarantee that your program is correct, but everything we do helps to increase our confidence, and 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 our 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 we are less concerned with the buffer overflow problem, since the built-in exception mechanism will detect any such problems.

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

There are many different kinds of tools for analyzing such properties as schedulability, worst-case timing, run-time use of storage, freedom from race conditions, freedom from unwanted side effects, automated testing, metrics etc. An important part of the preparation for a project aiming at a high-integrity product is to investigate and acquire a coherent set of tools. The use of integrated development environments is often a useful way of organizing such a set of tools. For example, the GNAT Programming Studio (GPS) product from AdaCore provides a convenient way of organizing a wide variety of tools in a coherent manner, and there are many other such products.

Object Oriented Programming and Safety Critical Systems

Object oriented programming methods have become an important part of the arsenal of tools in the hands of a modern programmer, and a wide variety of languages support these notions (C++, Java, Ada and many others). In this section we will look at the special considerations of using these methods in a safety critical environment.

The notion of OO programming is a little ill-defined. On the one hand it refers to a design method in which objects communicate via message passing. Such a design method in and of itself poses no special problems or safety concerns. On the other hand, it refers to the use of a set of features in programming languages, originally conceived to facilitate the use of the object oriented design approach, but more widely useable for many purposes. This set of features typically comprises 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, allowing automatic choice of the right object

The first two features offer no special obstacles in a safety critical environment, and it is worth noticing that the use of these two features is helpful even without dynamic dispatching. For example, a type and associated functions (methods) may be defined in a library. The program can then import this type, extend it to specialize it for a particular application, and then use the inherited operations on this type. In Ada, this useful set of capabilities is recognized by the provision of a pragma No_Dispatch, whose purpose is precisely to check that a program does not use dynamic dispatching. An Ada compiler can recognize this pragma, and enforce the restriction, as well as improving 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 are not part of the standard.

Now let's look at dynamic dispatching. This offers a challenge. The problem is two-fold. First a typical implementation is to have a table of pointers and then index into this table. That's a bit worrisome. What if the table gets clobbered some how? The dispatching operation can cause a wild jump. Now of course we don't expect any such clobbering in a certified program (although the demonstration of correctness of the dispatch table raises some nontrivial issues). Still indirect calls make us nervous, since now to ensure the integrity of the control flow we have to prove properties relating to data access.

The second problem 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 a program. What exactly are we supposed to check when we see a dispatching call.

One possibility would be to treat each call as though it were a case statement with branches going to every possible function that corresponds to the dispatching call. This seems like a completely fair translation, but the trouble is that in a real program, we might easily find that most calls can only go to a small subset of the possible targets. Then we end up with a lot of deactivated code (code that can never be executed), and we have trouble testing such cases, or proving that they can never occur.

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 function will share a single case statement. On the positive side, we argue that we could have written the program this way in the first place, and traditional testing would have been fine, so it should be fine here. On the negative side, we worry that our coverage testing is only showing that each method is used somewhere, and we are not really verifying the possible flows of control.

The fact that we could have written the program that way is not decisive. The testing schemes we have described are not perfect, no testing scheme is perfect, but they work pretty well in practice. However, they can be subverted by a programmer concentrating on the letter of the standard, and ignoring its intent. Here is a way of essentially removing all **if** statements from a program. Given:

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

we replace this with:

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

Where the second and third arguments are now pointers to functions that if called will execute the appropriate statements. Now Eval_If itself looks like:

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

Here the .**all** notation calls the relevant procedure when the condition is met. Now we have only one **if** statement in the entire program, the one inside this routine. Our conventional coverage approach, where the MC/DC protocol tries hard to ensure that all conditionals are tested thoroughly is now subverted, and our coverage now only proves that some **if** somewhere is true and some **if** somewhere is false.

Well this is pretty clearly cheating. Even though it meets the letter of the certification law, it does not meet the spirit, and we suspect no DER will let this or any similar subversion sneak by.

So here is the question, is the conversion of dispatching calls to a single shared case statement cheating? We don't really know the answer to this yet. We need more experience. At least one program has been certified using this approach as far as we understand, but other programs are worrying a lot about this issue, and the lesson to be learned here is that dynamic dispatching is best avoided if possible in high integrity applications that are to be certified, and if they can't be avoided, you need to worry about these issues, and follow the evolving state of the art in this area.

For information on the development of the follow on DO-178C standard, see

www.rtca.org/CMS_DOC/Original%20TOR%20mod.PDF

There is a lot of work being done 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” that is a must-read reference 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 reliable systems. This technology will continue to improve in the future. Now 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 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 straight-forward program, it would make sense to adopt a much more strenuous view of reliability in this case. Lives were not at stake, but a few billion dollars is real money! I did not receive a reply, but I think in future that we will come to demand a level of reliability and security in a wide variety of critical programs.