Assessing Critical SW as “Proven in Use”: Pitfalls and Possibilities

Peter Bernard Ladkin
RVS White Paper 4
20130614

The Issue

Suppose you are the supplier of a component C of a system S which could exhibit unsafe behavior (behavior which could result in injury or death to a person or persons, or adversely affect the environment in some significant way). C contains some digital electronics, and those digital electronics are programmed.

Suppose, for example, C is a robust temperature sensor which is to operate in extreme environments, and which is specified to continue operating in even more extreme environments when something goes wrong with S, as part of a defined safety function. (“Safety function” is a technical term defined in the international standard addressing functional safety of systems involving electric, electronic and/or programmable-electronic components, IEC 61508.)

Functions that C provides will be assigned a Safety Integrity Level (SIL), from 1 to 4, according to the requirements of IEC 61508. Since C is a temperature sensor, let us suppose that function is to deliver an output value, representing the actual measured temperature to some specified degree of accuracy, at defined times or time periods, to a specified communications port on C.

That SIL will specify a maximum “rate” of dangerous failures allowed for C, varying from less than 1 in 100,000 operating hours for SIL 1, to less than 1 in 1,000,000 operating hours for SIL 2, up to less than1 in 100 million operating hours for SIL 4. The specified SIL comes from analysing the required behavior of the safety function within the operations of S. In what exactly a “dangerous failure” consists will be derived from the specification and analysis of S. The notion of “failure” is well defined, and general (C provides a specified service, namely that described above; a failure is when that service in some respect or other is not fulfilled), but the attribution of “dangerous” comes from analysis of S. It could be a sufficiently-wrong temperature datum; it could be omission of an output value or output values over some time period; it could be .... well, whatever is determined by analysis to be a “dangerous failure”.

You, the supplier of C, have been building and installing C's for many years. The parameters under which C delivers temperature readings to specification are well understood; the specification is mature; the failure rates of C are well understood. You have had it running in environments for which there is a SIL 1 or SIL 2 requirement for years. It's robust. It doesn't quit. Everybody wants one. You are one of the world's leading suppliers.

C contains a central processor CP, but also some peripherals chips. The peripherals chips are 8-bit, but the chip manufacturer has discontinued the model and is producing a “backwards-compatible” 16 bit chip. You have no choice but to use a different chip than you have been using, and you decide to use the same manufacturer's new chips. The chip manufacturer has documentation involving reams of data on formal architecture verification, static analysis, and tests, of its opcodes to support the claim that the “new” chip behaves on the “old” opcodes exactly the way the “old” chip behaved: no deviation. You take all of that, assess it along with your (and their) drivers, and come to the conclusion that the drivers will behave the way they did, without exception. IEC 61508 doesn't allow you to say “without exception”, but it will allow you to say “to the level of one failure in 100 million operating hours” if you incorporate this in a SIL claim. (There is in itself no SIL to be attached to the peripherals-chip driver because this is not designated as a “safety function” of S.) You are fairly certain that any assessor of S will accept your claim and the accompanying documentation on good – let's say impeccable - engineering grounds.

Now, what about the SW running on CP? Call it CP-SW. You don't want to touch it. It works. It has worked, you have loads of documentation about C in all sorts of environments, including safety functions with high SILs, and C retains the crucial data about its operation and performance, which you have downloaded from all the installed C's over the years you have been in business. You have masses of it.

CP-SW has been incrementally modified over the years. The version you want to offer for S is V1.4. The field-performance data on V1.4 only goes back a year. You have been installing C's with V1.4 regularly over the past year, and you have field-performance data on all 100 of them, so you have data on somewhat over half a million hours of operational use. No failures of the function deemed critical in S (that is, whose failure would be a “dangerous failure” for S).

All those C's had the “old” peripherals chips, though.

What you want to say is that you have half a million hours of operational data on, inter alia, the behavior of CP-SWV1.4. And it has never shown an error. You are convinced, and you have convinced the assessor, that your new drivers on the new peripherals behave exactly as the old drivers with the old peripherals, to the highest level (at most one failure in 100 million operational hours). So it stands to reason that if you run CP-SWV1.4 on the new kit (which is otherwise unaltered except for the peripherals chips and drivers) it is going to behave the same way it's behaved for the half-million hours you have on record, doesn't it?

Doesn't it?

Well, actually, the current version of IEC 61508 Part 3 says sure, go ahead. It doesn't even require all that stuff about the peripherals behaving “the same” way on the opcodes that CP-SWV1.4 throws at them; certainly not to the level of 1 failure in 100 million operational hours. It just says “sufficiently similar” specified use. Way to go.

The Problem

The thing is, all your kit has been running in air environments which are normally around 100°C and 95% humidity, with no corrosive materials in the air. They occasionally spike up to 500°C and 100% humidity with some caustics, when things go wrong. But that's only for a fraction of the time; some hours. Then the plant is shut down, and your kit is visually inspected, and the plant is started up again.

But S does that regularly, for a few hours a day. It's not an exceptional environment for S. It's not an exceptional environment for C either – you designed for it and it's within specification. No prob.

But there is a problem. It turns out that the new peripherals are somewhat more susceptible to the environment than the old peripherals. They throw out exception conditions more often. Of course, CP-SWV1.4 can deal with exception conditions. But it has never had to deal with four at once before. It turns out there is a configuration issue which means that CP-SWV1.4 croaks quite often on four simultaneous exceptions from the peripherals. You have a stack to handle the exceptions, and the exception conditions are logged (CP-SWV1.4 is well-instrumented!). But the space for logging the exception conditions, as well as the parameters passed to the exception handler turns out to be just too small to handle four exception conditions with parameters more than twice. And on the third time the buffer to which the logs are written overflows and .... CP-SWV1.4 croaks. It never happened before, because whenever the exception conditions were triggered, it was in a circumstance in which the plants were then shut down and CP-SWV1.4 reinitialises on start-up.

Lo and behold C is engaging in dangerous failure about once every two weeks in S. But its SIL 1 classification in S only allows it to do that once every ten years! This behavior is neither wanted nor anticipated by the assessors. Something has gone badly wrong.

That is why the conditions currently specified in IEC 61508-3:2010 for assessing and accepting CP-SWV1.4 as "proven in use" need to be tightened up.

The Solution?

So what would the statisticians say?

If you want to use the history of CP-SWV1.4 as evidence that it is fit for SIL 1 purpose in S, they would say that

1. The distribution of inputs to CP-SWV1.4 for the proposed use in S must be identical to the distribution of inputs to CP-SWV1.4 in the history. Well, they aren't. The distribution in S has a lot higher frequency of extreme inputs than what is in the history. Some assessor thought they were "sufficiently similar", because they were still within the spec of C, but they manifestly weren't "sufficiently" similar to ensure similar behavior of CP-SWV1.4.

2. You must have guaranteed-perfect failure detection for failures of CP-SWV1.4. It was the logger that failed, and it failed with a buffer overflow. Does the failed logger correctly log its own failure and the consequences? Gee, you never thought of that.

3. For SIL 1, with perfect failure detection, and identical distribution of inputs to CP-SWV1.4, you need C to have a failure rate of at most one dangerous failure in 100,000 hours, that is, a little over ten years, of operation. Now, any failure of C is "dangerous" (we have agreed) and when the CP-SWV1.4 stops working because of an exception condition it couldn't handle, then C stops delivering the goods also. So we can't have CP-SWV1.4 croaking more than once every 100,000 hours on average. The statisticians say that, given 1 and 2 above, you need 300,000 hours of history to be 90% confident that CP-SWV1.4 will exhibit that failure rate or better in the future, and 460,000 hours of history to be 95% confident. Well, you do have that. You scrape by even at the 95% level with your just over half a million logged operational hours. But if S had been a SIL 2 application you would have needed 3 million hours history for 90% confidence and 4.6 million hours history for 95% confidence, and you certainly don't have that.

So much for C in S. You've fixed the configuration problem. And you have performed an impact analysis of your fix to ensure that you haven't introduced any new bugs through fixing the one. Can you go ahead and continue to use C with its software CP-SWV1.5 in S?

Well, how many other glitches are awaiting? You don't know, and neither does anyone. The same stuff might happen. It's a new SW version; it's an environment with a new distribution of input parameters. And you didn't have perfect failure detection – do you now? It follows that you can't qualify C with CP-SWV1.5 as "proven in use", because C with CP-SWV1.4 was "disproven in use".

With any luck, CP-SWV1.4 had some level of modular structure, and you have been able to separate out the logging functions where there was a configuration problem from the sensor-data-processing that is CP-SW's main function. Say that sensor-data-processing was performed in CP-SWV1.4Kernel and the logging function in CP-SWV1.4Log. Indeed, CP-SWV1.5 consists of CP-SWV1.4Kernel with CP-SWV1.5Log. Your impact analysis has shown a lack of malignant interaction between those two SW components.

And you went into the SW code – luckily, you had from the development the full specifications along with some argumentation, some of it in the form of formal verification, that showed that CP-SWV1.4Kernel did what it was supposed to do. You trotted that out and satisfied the assessors. And CP-SWV1.5Log you did ab initio, as might be expected since CP-SWV1.4Log was buggy. Now, you are pretty much sure, and can persuade assessors, that not only is your failure detection of CP-SWV1.4Kernel perfect but that of CP-SWV1.5Log, and the interactions between the two, are also perfect.

So far so good. Actually, people were very impressed by what you did by way of assuring your SW in the wake of the debacle with S. First, S's suppliers, and then all the clients of S. Now you've sold 100 C's with CP-SWV1.5 installed, all in similar installations to S. They have been running for up to a year, so you have your half-million hours of history in the S-type environment.

The New Client

Your new client has an application with an S-type environment. You have half a million hours of history, a similar environment (so identical distribution of inputs to CP-SWV1.5), perfect failure detection. C with CP-SWV1.5 has never failed. Your new client really wants your kit, not only because of the history but also because of the care which you have taken in qualifying your kit with its SW.

You have done some “proven in use” qualifications with the last few installations of C with CP-SWV1.5, once you had accumulated the 300,000 failure-free hours needed for 90% confidence in SIL-1-levels of reliability. Your new client has the similar environment, similar S-type configuration – but a SIL 2 safety requirement.

What do you do? You don't have the 3 million historical operational hours to satisfy the statisticians on Point 3 that you can be 90% confident. In the past, upon introduction of C with CP-SWV1.5 you had justified the use with reference to your documentation of the static analysis and formal verification you performed. It was regarded as satisfactory for SIL 1. And on top of that you now have a half-million hours of failure free operational use in exactly these conditions, statistically justifying that SIL 1 assessment most rigorously.

What more could you do? You can't do any more formal analysis of the SW than you had done – you did all you could on CP-SWV1.5, after the failure of CP-SWV1.4. That failure was very expensive and you needed to build client confidence in your kit back up. And your analytical diligence had enabled you to succeed at that.

Now, IEC 61508 Part 3 says that's all you need to do for SW to develop it to – well, SIL 4! But that is just formal stuff. You could have had another subtle bug in there, and all that math wouldn't have helped. There is a hard constraint on C – SIL 2 says dangerous-failure rate of at most 1 in one million operational hours. And since CP-SWV1.5 drives the behavior of C now, you're short two and a half million operational hours of experience.

Sorry, you say to your customer, come back in two or three years and we'll have C qualified to SIL

2 for you to 90% confidence.

Customer goes to the competition. They don't do half the logging you do, so they have a meagre selection of input data in the history. And they don't change their SW. They ticked all the right boxes in SW development, and they just put the SW on their box, of which the HW they had qualified to SIL 2. Here you are, they said to the customer. This is SIL-2 appropriate. And the assessor said so too – and she wasn't wrong, according to the IEC 61508 standard. Your customer buys the competition's kit; you've lost them.

What did you do wrong? Did you do anything wrong, technically? What did the competition do right? Did they do it right?

The New Client Who Worships You

Now, you have a new, discerning customer. The customer says she came to you because you have the reputation of doing everything thoroughly and not compromising. You are way more thorough than the competition. Your histories are more detailed. She has way more confidence that you will have detected any failures were there to have been any. The bug-fixing you engaged in after the experience with S was exemplary; nobody else does it so thoroughly; she is amazed you can do that and still make money (actually, so are you).

She's got a SIL 2 application. She has far more confidence that your kit C with CP-SWV1.5 is more thoroughly analysed and tested than anything else out there. Further, you have the operational experience that shows rigorously that C with CP-SWV1.5 is SIL 1 qualified.

Please will you sell her your kit, she asks. She has, justifiably, so much more confidence in it than in anything else on the market, and if other people are willing to sell her SIL 2-qualified kit, why not you, because yours is – so obviously to her - just so much better analysed and assessed than theirs.

What do you do?

An Intermediate Approach?

SW in the usual imperative programming languages such as C and Ada is deterministic, isn't it? The program executes, one source-code command after another, does things with the data, jumps here and there; here you are in one state (data, stack, command to be executed) and you execute the command and there you are in the next state determined by the semantics of the command you just executed. There's no *choice* of commands; the CPU doesn't toss a die to find out what to do next.

Well, actually not. Compilers can introduce quirks into the object code that are not at all in the semantics of the source code. *If* the source code has a unique semantics (the SPARK executable subset of Ada does; C as a whole doesn't, but MISRA C is better, as is Hatton-style Safer C). And then there are linkers, and then maybe the intended semantics of the opcodes does not exactly mirror the way the chip HW actually behaves in all aspects. So, the object code executing on the HW mostly does *this* on this specific data, but it may occasionally do *that* instead.

Basically, one can think of it as hidden inputs. Part of the HW state that is invisible to the SW; part of the low-level SW state that is invisible at the source-code level, and so on.

But, you can argue, it still works off the concept of state, part of which state is ostensible and part of which is hidden. The HW+SW is in some state; it does something mechanically (actually, electrically) determined on that state, with or without some external input, and it goes into some new state, with or without some external output. Something which operates like that is called a *state machine*. Almost all formal verification technologies treat executions as something happening on a state machine – even hybrid/analogue processes can be so modelled, albeit not without some complexity. This one is not deterministic – sometimes it goes *this* way on this input in this state, and sometimes *that* way.

In your new client's application, C is only ever going to see combinations of input data which it has seen many, many times before. Half a million hours' worth. What about the hidden inputs? They are in there too, in the half-million hours of operational data, in some distribution which is equally hidden.

What about the state? You can't see the state – further, it would be too complex to calculate explicitly for this HW+SW architecture, even if you could do it accurately. But you got into some state because the system started in some well-defined initial state (you have ensured that, as a good computer scientist) and executed a sequence of commands until it got to where it is now. So the sequence of commands it has executed to get to this state can stand proxy for the state it is in. If you take a given sequence of commands, because the operation is non-deterministic it may be that you can attain many states, but execute that sequence sufficiently many times and you will visit them all. Similarly, there may be more than one sequence of commands which will put the machine in a given state. The correspondence between sequences of commands and states reachable from the initial state is not exact. But each set covers the other.

What commands, though? Opcodes? Source code? Something higher-level?

The best approach may be to go to the specification of the system S.1 in which C will be running. The specification of S.1 will contain certain functions which C is to fulfil (in part or in whole, alone or in conjunction with other components). Those functions will be described in more detail and ultimately refined into subfunctions which C is to execute. Those subfunctions are likely the most efficient construal of "function" in the phrase "sequence of function calls" above. They are precisely the system-architectural operations for which C is needed: it is those operations which the client cares about.

So what you are looking at in the operation of C is a non-deterministic state machine, which goes sometimes *this* way and sometimes *that* way given specific input with a specific sequence of function invocations. You have enough historical operational data, though, that these non-determined behaviors exhibit some sort of statistical regularity: it goes *this* way X% of the time, and *that* way (100-X)% of the time. (Or three ways, or five ways, or fifty ways.)

You have yourself something called a Markov process. A Markov process is exactly a non-deterministic state machine as described above, with specific probabilities attached to each of the possible transitions out of a state.

Note that this is a piece of theory. We have concluded that real SW (object code) running on real HW will cause the HW to behave functionally (that is, just with respect to producing the values the SW commands it to produce) like a Markov process. The argumentation given above has been quite general. There are no specific properties of CP-SWV1.5 which have been adduced. So this is a

Folk Theorem: SW in object code form running on specific HW behaves as a Markov process, provided that the HW remains failure-free.

Do you have a unique Markov process – can you say pretty well what the X/(100-X) values are for all those input data/sequence of function invocation combinations? (Or X, Y, 100-(X+Y), or more .....) How much confidence may you have that your estimates of X, etc, are passably close to the real probabilities?

One cannot say in general. It depends on the particulars of the SW. Obviously, the simpler the functions provided by the SW (as defined in the specification of the SW functionality), the easier it is going to be to define a Markov process which emulates the behavior of your SW.

There are a variety of techniques for assessing whether you have enough data on a Markov process to be confident to a given degree that you have sufficiently good estimates of the transition probabilities. There are a whole variety of methods based on things called Bayesian networks which might help as well. Even if I knew what they all were, here is not the place to say, for I haven't finished my tale yet.

Suppose you can say that your historical data determines to some fair degree of approximation the Markov process that is your SW execution on your HW, to some degree of confidence D which you specify.

Then you can look at all the times the execution of C resulted in a dangerous failure. It didn't. Not once. And suppose it won't, as far as you can tell, within the bounds of the approximation. Or maybe it might occasionally, but you can bound that within the SIL 2 numerical requirement.

So now you can say you are confident to degree D that C won't result in a dangerous failure. Can't you? And you and your client don't have to wait for another number of years of data on C's behavior. You can both sign a contract right after this Markov analysis and you and the other stakeholders can live happily ever after.

Can't you? Can't they?