SILs and Software

PG Bishop
Adelard and Centre for Software Reliability, City University

Introduction

The SIL (safety integrity level) concept was introduced in the HSE (Health and Safety Executive) PES (programmable electronic system) guidelines and subsequently extended in the development of IEC 61508. This article sets out our diagnosis of some of the issues associated with the SIL concept and provides some ideas for the improvement of IEC 61508.

The SIL concept

The HSE originally introduced the concept of a "level of safety integrity" in its programmable electronic systems (PES) guidelines to deal with the fact that programmable systems are likely to contain systematic design faults (primarily software defects). It was thought that systematic failures (especially those in software) could not be treated probabilistically, and safety integrity level was used as an alternative. In IEC6 1508 the SIL concept was extended to a set of levels, ranging from SIL0 to SIL4. Even at that stage, the SIL had two distinct roles:

- to denote the required behaviour of the overall safety function (i.e. avoidance of failures due the systematic faults)
- as a label to denote an appropriate set of software development methods

The SIL subsequently found its way into IEC 61508, where the meaning was extended to encompass *dangerous failure rate* targets for the safety function (e.g. SIL1 has a dangerous failure rate band of 10^{-1} to 10^{-2} per year). While this is a top-level target for the whole system, there is an implication that the failure rate of the software should not exceed this target either. IEC 61508 also specifies sets of "highly recommended" and "recommended" techniques that can be applied for a given SIL. By implication these techniques should be able to deliver software failure rates that satisfy the dangerous failure rate targets for the SIL.

Problems with SIL

Two problems with the current approach to SIL allocation and software are:

1) The SIL is actually associated with a *safety function*, i.e. some plant-related risk reduction measure like an emergency trip. The mapping from the SIL assignment to requirements for the subsystems and associated software does not explicitly recognise the contribution of the system *architecture.* For example, there is no credit for implementing diverse trip subsystems—the software in both subsystems still has to meet the requirements associated with the top-level SIL "label". Similarly, no account is taken of the fact that some software components are less critical to safety than others.

2) There is no technical basis for the linkage between recommended software techniques and target software failure rate implied by the SIL. Any linkage can at best be shown on average, and it is not certain that a specific development process will achieve a given dangerous failure rate. This leads to an abuse of the standard where compliance with SIL-mandated techniques is deemed sufficient to claim the dangerous failure rate is in the stated SIL band.

Unpicking the SIL concept

I think problems with SIL are due to a confusion between different levels of abstraction, namely:

- Safety function
- Safety equipment
- Safety software

At the safety function level it is meaningful to talk a safety-related *service*. Depending on the service type, the safety requirement is expressed in terms of failures per demand (e.g. due to a plant excursion) or in terms of failures per hour (e.g. for continuous plant control).

A safety function is implemented by *equipment subsystems* and a *support infrastructure* to operate, maintain and proof-test the equipment. Changes to the infrastructure (e.g. the proof-test interval) can alter the safety function *service*, i.e. reducing the proof test interval will reduce the probability of failure per demand (due to random hardware failures and software crashes).

At the software level, we have *software faults* that can cause a dangerous equipment failure, e.g.:

- unsafe handling of equipment hardware failure (which could occur at any time)
- software lockups and crashes (which could also occur at any time)
- unsafe response to a given input sequence (e.g. a demand, or a required control action)

Depending on the context, activation of a fault will have an instant effect on the service or a delayed effect (e.g. failure to operate at the next demand). The software failures depend on the specific *operational profile* seen by the software (this can include hardware failure events handled by software, and input data from the plant).

Linking faults to failures

Is there a justifiable way of linking dangerous software faults to dangerous equipment failures? To some extent there is. There is a theory [1, 2] that shows that, under the following conditions:

- a test interval T using an unchanging *operational profile*
- N faults in the software
- a fault is fixed once *d* failures have occurred

the failure rate λ after time T is bounded by:

$$\lambda \leq Nd / eT$$

where *e* is the exponential constant (2.718…). Normally d=1 during system development, but usually $d >> 1$ during subsequent operation. Note that this theory gives a *worst case* bound (for a given operational profile), but there are empirical arguments that the *best case* failure rate could be no more than an order of magnitude better than the bound.

Clearly the dangerous failure rate of software must be less than its SIL target. So we need to show:

$$Nd / eT \leq \lambda_{SIL}$$

where λ_{SIL} is the maximum dangerous failure rate permitted in the SIL band. Given that testing of the final operational system is limited e.g. T is no more than 10^4 hours and faults are always fixed (d=1), we have the following requirements for *dangerous faults* in the software:

$$N_{danger} \leq e\ \lambda_{SIL}\ T$$

In some cases, this calculation will yield a target value of N_{danger} that is less than unity, but the theory still applies [2]. Such "fractional faults" can be related to the probability of software "perfection", i.e.:

$$P_{no\_danger} = (1 - N_{danger})$$

The implications of this theory for requirements on residual faults are shown in the table below.

SIL	Max	N^*	$P^*_{no\_danger}$
1	10^{-5} /hr	0.2718	0.7282
2	10^{-6} /hr	0.0272	0.9733
3	10^{-7} /hr	0.0027	0.9973
4	10^{-8} /hr	0.0003	0.9997

Table 1. Residual dangerous faults vs. SIL failure targets (* assuming $T=10^4$ hours)

So it can be seen that for SIL1, in this simplified example, there should be a better than 50:50 chance that there are no dangerous faults in the software. For higher SILs the requirements for perfection are even more stringent.

If we accept this fault-to-failure model, we could (in the most extreme case) replace the SIL "software label" for recommended development methods by targets for the *number of dangerous software faults*. The developer would then have to provide evidence that the chosen software processes can achieve this target. This is, admittedly, a novel interpretation of the SIL "software label" but it might be closer to the original intent (i.e. the avoidance of dangerous systematic faults). Note that a low dangerous software fault target does not preclude the existence of a larger number of non-critical faults that do not affect safety.

Implications

If we accept the theory described above, we could view all the IEC 61508 recommended software techniques for a given SIL (like module testing, programming standards, static analysis, formal proof) as a means for achieve the required number of dangerous faults, N_{danger} (or equivalently, achieving the required probability of perfection, P_{no_danger}). In addition, in system architectures that use diverse subsystems to mask failure, we could use the same targets for *common* dangerous faults, which enable some relaxation of the dangerous fault targets for the individual subsystems.

A further implication of our fault-to-failure model is that, to increase the SIL compliance of the software by one level, additional techniques have to be used that can decrease the residual dangerous software fault density by a factor of 10. So to move from SIL1 to SIL4, the residual fault density has to decrease by three orders of magnitude. This is clearly a tough target to achieve and, to some degree, this justifies the increases in rigour demanded in the standard (although it is not easy to determine how much reduction is likely to be achieved in practice).

The new definition also highlights another problem with the current SIL-related technique recommendations—the number of faults tends to scale linearly with size. For example, experience with good commercial practice shows that it is possible to achieve 3 faults per kilo-line of code. So using the *same* development process and techniques might yield 3 faults in a 1000 line program or 3000 faults in a million line program. So the idea that a given set of techniques must, by definition, achieve a given dangerous failure rate is extremely suspect.

Do these ideas work in practice?

So far we have only discussed the theory, but will it be useful in practice? To demonstrate the approach in practice, we have taken have some actual (anonymised) data relating to a SIL1 compliant process for a safety-related system. Experience with other SIL1 developments indicates that the data are fairly typical. The application development data is summarised below.

Program size	200 Kloc (new application code)
Fault density	4 faults/kloc (prior to V&V)
Faults found	800
Dangerous faults	25
Fraction dangerous	3%

Using fault discovery trends, we estimated that the following number of faults remained at release to operational testing:

Total residual faults:	20 … 40

Dangerous faults: 1 … 2

The ranges reflect the uncertainty in the fault estimates. The dangerous faults were estimated from the percentage observed in earlier development. Based on final testing period of 1060 hours and the residual fault estimates, we used the worst case bound formula to derive the operational failure rate bounds:

MTTF (all failures): 72 … 144 hours
MTTF (dangerous): 1440 … 2880 hours

It can be seen that the predicted MTTF is actually a lot worse than the SIL 1 band of 10^5 to 10^6 hours between dangerous failures, but this was sufficient to meet the actual system safety requirements as there is a lot of external mitigation of dangerous failures.

Subsequently, the equipment operated without dangerous failure for 3000 hours with the following results:

Residual faults found: 7
Dangerous faults found: 0
MTTF (all faults): ~ 430 hours
MTTF (dangerous) ≥ 3000 hours (at 63% confidence)

So the worst case bound predictions do appear to be conservative provided the residual faults (and especially dangerous faults) can be correctly estimated. If we (quite unjustifiably) assume that dangerous failures are 3% of the total (like dangerous faults), the dangerous MTTF could be as high as 14000 hours (but this is still in the SIL0 band).

From the foregoing example it is clear that even a SIL1 dangerous failure rate target is quite hard to achieve. But the failure rate target is easier to achieve if the application is smaller. For example, with the same process but only 10 kilolines of code, we might expect around 40 development faults (with around one fault being dangerous), and a proportionately smaller number of post release faults (e.g. N_{danger} ~ 0.05). This would be sufficient to meet the SIL1 target (see Table 1).

In addition, it is apparent that the current SIL bands fail to cover quite legitimate safety-related systems with dangerous failure rates targets in the 10^{-3}…10^{-5} per hour band. Perhaps the bands need to be adjusted or extended to accommodate "low end" safety applications.

We can also see that, for this application at least, not all faults lead to dangerous failure. This suggests that designing to limit the dangerous consequences of software faults could be a valuable alternative to reducing the total number of faults.

Concluding remarks

I hope the rather provocative suggestions made here will stimulate a debate on the IEC 61508 approach to software. The main points that I would like addressed are:

- A clear distinction between SIL as used at the safety function level and SIL as a "method index" at the software level. Ideally there should be a separate term for the integrity of components that implement the service, e.g. SSIL—Software Safety Integrity Level.
- A better definition of what a SIL "software index" (or SSIL) means, we would advocate a definition based on the number or probability of dangerous systematic *faults* remaining in the software. This definition is certainly more meaningful in a development context.
- IEC 61508 should take account of architectural diversity, i.e. it should allow claims on *common* dangerous faults as well as claims for dangerous faults in a given piece of software. This could permit less stringent development methods to be used in diverse subsystems.
- A better rationale should be given for the choice of techniques for a given SIL, in particular it should be related to claims on the expected *fault density* achieved by the development and verification techniques in the process.
- It is also important to recognise that "size matters", i.e. with the same techniques it is easier to achieve a particular dangerous fault target for a smaller piece of software.

- The failure rate of the *software* should be distinguished from the failure rate of the *safety function*. These are not identical if there is external mitigation or diverse software implementations).
- Perhaps *software* failure rate bands could be set more realistically, with the lowest level starting at 10^{-3} failures/hour rather than the current value of 10^{-5}.
- There should be more focus on design practices that prevent or limit dangerous failures in the software.

References

[1] P.G. Bishop and R.E. Bloomfield, "A Conservative Theory for Long-Term Reliability Growth" Prediction. IEEE Trans. Reliability, vol. 45, no. 4, Dec. 96, pp. 550-560, ISSN 0018-9529

[2] P.G. Bishop and R.E. Bloomfield, "Worst Case Reliability Prediction Based on a Prior Estimate of Residual Defects", Thirteenth International Symposium on Software Reliability Engineering (ISSRE '02), pp. 295-303, November 12-15, Annapolis, Maryland, USA, 2002

[3] P.G. Bishop and R.E. Bloomfield, T.P. Clement, A.S.L.G. Guerra, 'Software Criticality Analysis of COTS/SOUP", Reliability Engineering and System Safety, 81 (2003) pp. 291-301