Lessons Learned Using Formal for Functional Safety

Doug Smith
doug.smith@doulos.com
Doulos

***Abstract*- Functional safety requires the functional verification of safety mechanism to eliminate any systematic failures and the evaluation of safety violations due to random hardware failures. Formal technology is a great tool for both. It can be used to verify safety functionality or ensure designs are equivalent when inserting new safety mechanisms. Moreover, it is ideal for generating fault lists for random fault campaigns and it can be used either in tandem with other tools like simulation or emulation or as a stand-alone tool for random fault campaigns. In this paper, the process of using formal for functional safety, the various issues that arise, and the several lessons learned from using formal for automotive functional safety are shared for others to benefit in their formal functional safety analysis.**

I. How Formal Is used with Functional Safety

There are two areas within functional safety hardware development where formal technologies provide great value—functional verification of the safety elements and safety mechanisms ([1], Part 5, 5.7) and fault injection testing ([1], Part 5, 5.9).

A. Functional verification

The use of formal technology for hardware functional verification is a broad topic and beyond the scope of this paper. However, using formal to verify safety components is relevant. Often designs are retrofitted with safety mechanisms to meet ASIL requirements ([1]) so parts can be sold into the automotive industry. Some safety mechanisms lend themselves nicely to *formal model checking* like parity, ECC, and checksum calculations. Since these mechanisms are primarily combinational, formal is ideal at quickly proving the correctness of these implementations. Other mechanisms like lockstep and redundancy lend themselves nicely to formal *sequential equivalency checking* (SEC). In fact, when retrofitting safety mechanisms into an existing design, I recommend using SEC to verify that the original design remains unaffected by the new safety mechanism.

> *Lesson #1: CRC is hard for formal so consider using C-to-RTL sequential equivalency checking.*

One safety mechanism, CRC (cyclic redundancy check), poses a significant challenge for formal. By nature, CRC is cyclical, typically requiring multiple clock cycles to multiply (via shifts) or divide—two operations that formal model checking struggles with due to the large state space. In my experience, both formal model checking and RTL SEC have difficulty analyzing CRC blocks.

CRC may be hard for formal, but not impossible. Of course, small implementations are more doable, but generally larger polynomials used in safety applications or communication protocols typically call for polynomial sizes of 9, 17, 33, and so on, resulting in large design complexity for formal. In such cases, I recommend considering what is known as C-to-RTL sequential equivalency checking, also known as *data path verification* (DPV). By abstracting to the C algorithm level, formal verification of CRC may be achievable.

B. Random fault testing

A second use of formal in functional safety is random fault testing. The goal of random fault testing is to determine a safety metric known as *diagnostic coverage* (DC). Diagnostic coverage indicates the percentage of the design that is safety related and covered by a safety mechanism that detects any random failures. To determine diagnostic coverage, a list of faults is first constructed. Formal can easily generate a fault list since it synthesizes the design before performing analysis. In fact, safety flows often use formal as their front-end to generate the fault list and then run fault simulations (or emulations) based on those lists.

Since formal has a built-in mechanism to determine the cone-of-influence (COI) of its targets, this COI can be used to determine if a fault is safe or not. Any fault outside of a safety path's cone-of-influence (COI) is by definition "safe" since it cannot possibly affect the safety output (see Figure 1). This built-in behavior facilitates

fault pruning, and formal may further eliminate faults possibilities through *fault collapsing*, where it eliminates faults by finding equivalent or dominant faults.

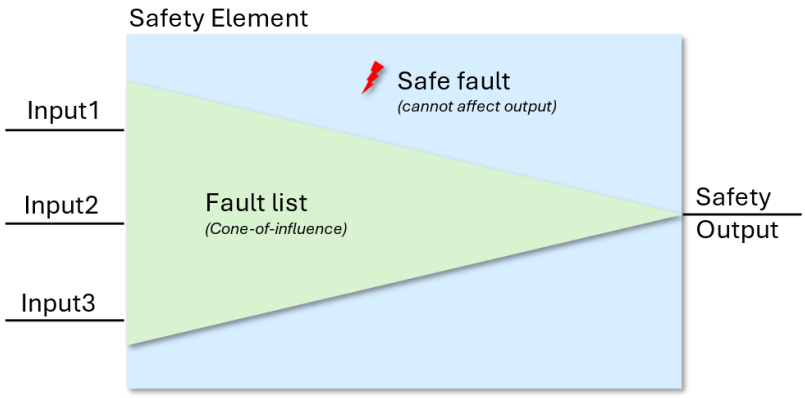

Figure 1: Illustration of the cone-of-influence to determine a fault list.

Once a fault list is created, fault injection is performed to determine if the faults can propagate through the design and affect the safety output. The easiest way to accomplish this in formal is using sequential equivalency checking (Figure 2). The safety element is compared with itself as both the specification and implementation, and a fault is injected into the implementation. If the safety output is unaffected, then the fault is by definition "safe." If the fault affects the safety output, then the safety mechanism or alarm should detect it. If the fault falls outside of the cone-of-influence of the safety alarm, then the fault is referred to as a *single-point fault* (SPF). If the fault goes undetected, it is considered a *residual fault* (RF). If the fault is detected, then it falls into the category of *multi-point fault* (MPF) or more specifically, a *dual-point fault* (DPF) since the standard requires testing of no more than two concurrent faults ([1], Part 1, 1.123; [1], Part 7, 7.4.3.2).

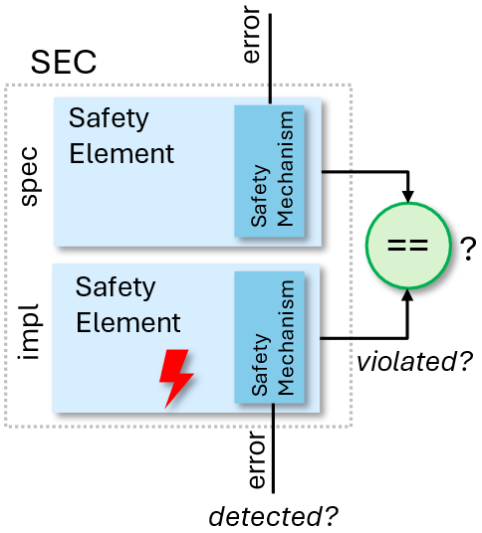

Figure 2: Random fault testing using SEC to inject a fault and compare.

Formal tool vendors have created specialized safety apps to automate the creation of the fault lists, run formal random fault testing, integrate with other tool flows, and merge the results across the different tool flows. Typically, these formal apps do not calculate a diagnostic coverage (DC), but their results can be used within an FMEDA[1] for performing the various metric calculations.

[1] Failure modes, effects, and diagnostic analysis.

II. EXPERIENCES WITH RANDOM FAULT TESTING

C. Generating a fault list

Lesson #2: Identify safe faults using the formal cone-of-influence.

Using formal to generate a fault list is one of the primary purposes of using formal in a safety flow. Again, the reason is that formal performs synthesis of the design. Obviously, a fault list generated from the project's design synthesis tool would also suffice. However, formal has a built-in feature that makes it particularly useful for generating fault lists—the *cone-of-influence tracing* of its targets. The safety path output can be described in a simple cover property, which formal traces back to all primary inputs:

```
c1: cover property ( @(posedge clk) safety_output );
```

All formal tools can generate a cone-of-influence report listing the various design elements affecting the safety output (in this case, cover property c1). In fact, most tools have a Tcl API that allows scripting and automating the generation of the fault list apart from a separate license for a formal safety app. A formal tool is ideal for generating the fault list because it eliminates all safe faults outside of the cone-of-influence ([1], Part 1, 1.123). Hundreds or even thousands of faults can be safely removed from a fault list, which is a distinct advantage over using a standard synthesis tool for the fault list generation.

Lesson #3 – Not all faults are necessary for a random fault campaign—typically around 4,000 faults.

A question that arises when performing random fault testing is, "How many faults need to be tested?" Engineers by nature want to test everything. Designs typically have thousands to millions of faults that need to be tested, which is not realistic with a standard event-driven simulator. Therefore, many companies resort to emulation for faster testing or specialized simulators that focus on fault propagation with massive parallelization to churn through thousands of faults.

However, the end-goal of random fault testing is to determine if a part meets or exceeds a specific ASIL level requirement (Table 1). Therefore, the number of faults to test really depends on the margin of error you are comfortable with in your diagnostic coverage calculation.

Table 1: ASIL level metric requirements.

	ASIL B	ASIL C	ASIL D
Single Point Fault Metric (SPFM)	>= 90%	>= 97%	>= 99%
Latent Fault Metric (LFM)	>= 60%	>= 80%	>= 90%

R. Leveugle and others [3] showed that a statistical population sampling can be used to randomly select faults and produce a low margin of error with a high-level of confidence. Using an unbiased and uniform random selection of faults, the standard statistical confidence interval formula can be used as shown in Figure 3.

$$e = t \cdot \sqrt{\rho \cdot \frac{1-\rho}{n}} \cdot \sqrt{\frac{N-n}{N-1}}$$

where

- e = margin of error
- t = 2.58 (critical value for 99% confidence interval)
- ρ = 0.5 (50% chance of selecting a fault - the most pessimistic value)
- n = sample size
- N = total number of faults

Figure 3: Margin of error statistical formula for of the random sampling of faults.

When the margin of error is computed for various sample sizes, an interesting observation can be made:

Table 2: Margin of error for various sample sizes with a 99% level of confidence.

	Total Design Elements							
Sample size	**10,000**	**20,000**	**30,000**	**40,000**	**50,000**	**100,000**	**500,000**	**1,000,000**
500	5.62%	5.70%	5.72%	5.73%	5.74%	5.75%	5.77%	5.77%
1,000	3.87%	3.98%	4.01%	4.03%	4.04%	4.06%	4.08%	4.08%
2,000	2.58%	2.74%	2.79%	2.81%	2.83%	2.86%	2.88%	2.88%
3,000	1.97%	2.17%	2.23%	2.27%	2.28%	2.32%	2.35%	2.35%
4,000	1.58%	1.82%	1.90%	1.94%	1.96%	2.00%	2.03%	2.04%
5,000	1.29%	1.58%	1.67%	1.71%	1.73%	1.78%	1.82%	1.82%
6,000	1.05%	1.39%	1.49%	1.54%	1.56%	1.61%	1.66%	1.66%
7,000	0.84%	1.24%	1.35%	1.40%	1.43%	1.49%	1.53%	1.54%
8,000	0.65%	1.12%	1.24%	1.29%	1.32%	1.38%	1.43%	1.44%
9,000	0.43%	1.01%	1.14%	1.20%	1.23%	1.30%	1.35%	1.35%
10,000	0.00%	0.91%	1.05%	1.12%	1.15%	1.22%	1.28%	1.28%

Notice that around 4,000 sampled faults, the margin of error drops to 2% or lower. Figure 4 further illustrates how a sampling size of 4,000 faults reduces the margin of error to around 2% or less for practically any number of faults.

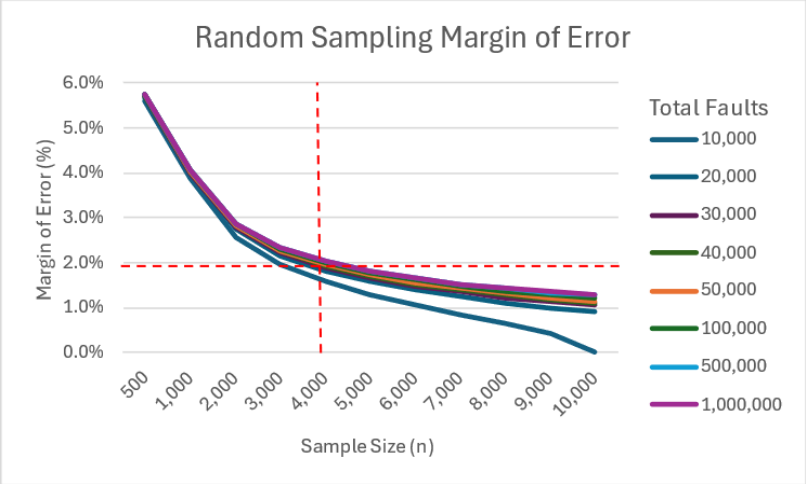

Figure 4: Margin of error as a function of sampling size.

For most fault campaigns, a 2% margin of error is acceptable. For example, ASIL B only requires a DC of 90%. If most faults are detectable, then a +/- 2% is still well within the ASIL requirements. It may be possible that a higher margin of error is acceptable for ASIL B using even less faults. Likewise, ASIL C requires 97%. If all sampled faults prove detectable, then the DC will equal 100% +/- 2% or 98%, which is still within the acceptable range. For ASIL D, a larger sample size may be required or all faults tested or other techniques employed. In fact, in other areas of safety such as nuclear reactor controllers or aircrafts, random fault testing is never used because it just requires too much effort. Instead, more robust safety mechanisms like triple or quadruple redundancy and processor lockstep are implemented, but at a much greater cost. Unfortunately, in the commercial automotive world, costs need to be kept at a minimum so redundancy and lockstep are not often economical options. Therefore, designs are often broken into safety islands and ASIL D portions kept small so that full fault testing is possible (*ASIL decomposition* [1], Part 9, 9.5). The lesson learned here is that while you can invest a lot in expensive tools to run

many faults in parallel and spend lots of time in fault simulations, the effort may be unnecessary since results will not vary much from just using a smaller sampling size of 4,000.

Lesson #4 – Fault collapsing is unnecessary and may actually violate the intent of the random testing.

Another technique commonly used by ATPG tools and formal safety apps is fault collapsing. Fault collapsing is the process of reducing the number of faults by identifying faults that are known to be equivalent or dominant (Figure 5). Gate-level fault lists may be cut in half by using fault collapsing [3],[4].

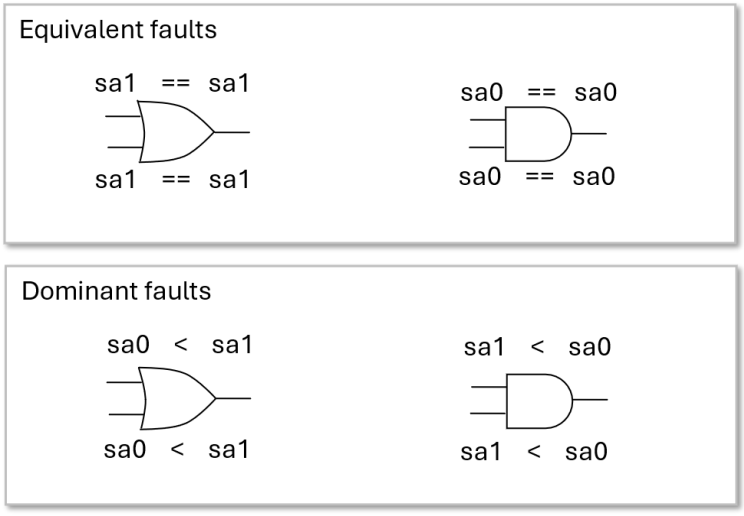

Figure 5: Equivalent and dominant stuck-at faults used in fault collapsing.

In general, fault collapsing achieves the goal of reducing the fault and making a more exhaustive fault campaign possible. However, there are some important considerations when using fault collapsing. First, RTL does not lend itself to fault collapsing as readily as gate-level netlists. Internal nodes within RTL constructs like flip-flops, muxes, and so on are not typically accessible by a simulator, emulator, or formal tool, impacting what can be collapsed. Furthermore, the ISO26262 standard requires that a random fault campaign have faults uniformly selected and without bias (i.e., truly random). When faults are collapsed, intelligence is added into the fault selection process that is not normally there. Likewise, the distribution of faults changes as faults are removed from the fault list, which does not give the uniform distribution of faults that is required. Without unbiased selection and uniform distribution, the random sampling margin of error explained above can no longer be assumed with 99% confidence.

Even if a strong case can be made for equivalent fault collapsing, dominant fault collapsing should be avoided in a random fault campaign. With dominant faults, testing one fault dominates the result of the other fault. For example, a stuck-at-1 (sa1) on the output of an OR-gate would mask any stuck-at-0 (sa0) on its input; therefore, the stuck-at-0s on the inputs can be removed from testing if the stuck-at-1 on the output is tested. By removing the stuck-at-0 faults, the dominant output fault (stuck-at-1) *must* be tested or there is a hole in the stuck-at-0 fault list. Here in lies the issue—once a fault is required be tested, the selection is no longer truly random nor uniform so it violates the intention of random fault testing. Obviously, if all faults were tested, then fault collapsing may be a good choice since the fault campaign is no longer random; however, when using random sampling to run a fault campaign, fault collapsing violates the intended outcome and is unnecessary since only around 4,000 faults are needed for a small margin of error. The extra effort required to collapse faults is unnecessary and fault collapsing on RTL results in only a small reduction of faults in my experience.

Lesson #5 – Use formal's COI to prove independence within a design when partitioning a safety element.

A few other lessons can be learned from using formal for fault list generation. For example, by placing cover properties on safety outputs and generating a COI report, areas of design or data independence can be structurally proven. Showing design independence is usually done using a Fault Tree Analysis (FTA), which is a bit tedious to construct. It may be difficult to show what blocks depend on each other at the SoC level, but using a formal tool's

COI report, dependent blocks can easily be seen. This is extremely helpful when structuring an FMEDA and deciding what blocks are needed for the various safety metrics.

> *Lesson #6 – RTL is more pessimistic than gates; therefore, just run fault testing on RTL.*

Another question that arises is whether to run a fault campaign on gates or RTL? The answer is rather straightforward. The diagnostic coverage formula is as follows:

$$DC_{SPF/RF} = 1 - (N_{SPF/RF} / N_{all})$$

The denominator of the fraction is N_{all}, which represents the total sum of all of the faults. Gate-level netlists can easily have 10x more faults than RTL. As the number of faults increases, the percentage of residual faults will decrease, thereby increasing the value of DC. In other words, RTL is much more pessimistic than gates. Therefore, if ASIL requirements can be met with RTL, then they certainly should be met with gates. The ISO standard explains, *"Depending on the purpose, fault injection can be implemented at different abstraction levels (e.g., semiconductor component top-level, part or sub-part level, RTL, etc.) A rationale for the abstraction level is provided"*. It further states as an example, *"Selection of the abstraction level can also depend on the nature of the fault that is intended to be modelled by fault injection: the stuck-at fault can be injected at gate level netlist, whereas for bit-flips an RTL level abstraction is sufficient."* ([1], Part 11, p. 64). In other words, the level of abstraction is based on project requirements. As stated, RTL is more pessimistic and harder to achieve its metric goals than gates so it should be adequate for most applications.

D. Running a fault campaign with formal

Running a fault campaign with formal is easy to do with a formal safety app. The user provides the tool the design and specifies the safety paths (outputs) and safety mechanisms (alarms). Within the formal app, a fault list is automatically created, the design is instantiated twice with one instanced faulted, and then sequential equivalency checking is performed as illustrated in Figure 2.

In the absence of a formal safety app, FPV (model checking) can be used to (1) generate a fault list with the COI report, (2) generate a fault injector to inject faults and detect safety violations and alarms, and (3) create tool-specific cut point directives to conditionally inject stuck-at or transient faults based on a fault ID formal choses using a non-deterministic variable. With these 3 files created, SEC can be run against the faulted design. For reference, a simple fault injector is shown in Figure 6.

```
module fault_injector(...);
    default clocking cb @($global_clock); endclocking
    bit [$clog2(N)-1:0] fault = '0;
    ...
    always @($global_clock)
       if ( inject ) injected = 1;

    asm_stable_sf_fault : assume property ( injected     |-> $stable(fault)  );
    asm_delay_injection : assume property ( $fell(reset) |-> !inject[*START] );

    always @($global_clock) begin
       violated = injected && (  spec.output != impl.output );
       detected = injected && ( !spec.error  && impl.error  );
    end

    // Find residual fault(s)
    cov_res1: cover property (( fault == 1 ) && violated && !detected );
    cov_res2: cover property (( fault == 2 ) && violated && !detected );
    ...
```

```
// Pseudo-code - Tool directive for conditional cut point (sa0 fault injection)
cut { impl.signal1 } -cond { injected && fault == 1 } -value 0
cut { impl.signal2 } -cond { injected && fault == 2 } -value 0
...
```

Figure 6: Example fault injector used with formal SEC.

Lesson #7 – Constrain formal to the design's "normal operating modes" and "conditions".

Formal users inherently want to run formal unconstrained to test all possible scenarios. However, running a formal fault campaign may be difficult depending on the complexity of the design. In addition to reducing the fault list, reducing the state space may help the fault campaign succeed. The ISO standard only requires fault testing during "normal operating modes" and "conditions". In other words, all the modes may not need tested. Indeed, FMEDAs break out different modes of operation as well as different fault modes (i.e., stuck-at, transient, etc.) It is perfectly acceptable to constrain a design as much as possible when running a fault campaign if the normal operating modes and conditions are known. This could be the difference between converging on properties or not. Remember, the purpose of the fault campaign is not to test all the design's functionality, but to verify that faults can be detected under normal operating conditions.

Lesson #8 – Never "cherry pick" formal friendly faults nor taint a random sampling set by modifying it.

A common temptation when properties fail to converge is to look for other faults to test that formal can easily determine. For example, you might run 10,000 faults through formal and pick the 4,000 faults that converge and yield conclusive results. Unfortunately, this is called "cherry picking" the faults. By selecting faults that are easy for formal, the fault sampling is now extremely biased and no longer uniform. Again, this violates the intention of random fault testing and destroys the confidence level of the statistical margin of error.

The obvious question is then, "What do you do if formal never converges on the faults?" Of course, you could try abstraction techniques (for example, abstract a CRC block and assume it detects everything), but abstractions must be done carefully to not invalidate the results. The easiest solution is to punt on formal and dump the remaining inconclusive faults over to simulation or emulation. The important point is—once a random sampling is made, *it should not be modified*. Certainly, a different random sampling could be made if a particular sampling is struggling with convergence. Statistically speaking, any number of random samplings should be able to find residual faults within a margin of error with a 99% confidence level. However, if the random sampling is uniform, it is very likely that any additional random sampling will pick faults from the same pool of faults that had difficulty propagating through the design in the first sampling. Regardless, do not bias a random sampling set by modifying it.

Lesson #9 – Formal has good capability for multi-point latent fault analysis.

Another area of consideration is latent fault analysis. With latent fault analysis, a fault is injected into the safety mechanism to see if it can still detect the fault(s) in the design. Often, this is accomplished by injecting 1 fault in the safety mechanism and no faults in the design. The idea is that injecting every design fault for every safety mechanism fault is just too large and too hard. In fact, individuals often skip latent fault analysis since its influence in the PMHF[2] calculation is so small that it is usually negligible. Moreover, injecting multiple faults may be unnecessary. If it is assumed that faults can propagate to the safety mechanism's inputs, then injecting a single fault into the safety mechanism is probably sufficient to reveal any input combinations that could be masked by a fault in the safety mechanism. In theory, this should produce a more pessimistic latent fault diagnostic coverage than when combined with the safety element because the safety element may not be able to reproduce the same latent fault scenario.

For a more optimistic latent fault diagnostic coverage, formal may be the better option. Multi-fault analysis is actually easier for formal. Built into formal verification is the concept of nondeterminism. By leaving a variable undriven, formal can set it to any value it chooses. For example, in Figure 6 above, formal assigns any value to the variable 'fault' to inject any fault in the design. When performing latent fault analysis, the design fault can be left under the control of formal while injecting a second fault in the safety mechanism. The more control given to formal, the easier the formal analysis. Unlike simulation, formal tests all design faults simultaneously while injecting a fault in the safety mechanism. Indeed, formal accomplishes what may not be practically possible using simulation or emulation due to the sheer number of dual-point faults to test. Still, there is the issue of design complexity and inconclusive properties. Since ASIL requirements are much less for LFM than SPFM, all properties may not need to converge. For example, if more than 60% of the formal properties finish successfully without undetectable faults, then the requirements of ASIL B are met even if 40% of the properties never converge.

[2] PMHF = Probabilistic Metric for random/Hardware Failure. See ISO26262-10 for the formula and calculation.

Nonetheless for completeness, a more pessimistic approach may be employed using either abstraction techniques in the safety element or single-point fault injection as mentioned above to achieve convergence on all properties.

E. Formal fault campaign results

> *Lesson #10 – Formal for functional safety is best combined with other verification tools.*

Obtaining random fault campaign results is not always straightforward. Some design pose challenges for specific tools, and some tools have special built-in capabilities such as formal's ability to synthesize and report a cone-of-influence. In addition, not all fault testing yields results within practical time and resource limits. Therefore, it is best to combine tools together within a safety flow. Formal works well for front-end synthesis and fault list generation and probably is the best option in a safety flow. Its ability to prune faults using the COI make it worth the price. Formal can also perform analysis and merge results. However, relying solely on formal for all fault testing may result in disappointment when designs contain constructs that pose complexity challenges for formal. Nonetheless, if only a small random sampling is required, then formal may be more than adequate to perform the analysis.

> *Lesson #11 – Consider ISO guidelines for DC instead of exerting unnecessary effort on a fault campaign.*

Another lesson learned using formal for random fault testing is to avoid formal random fault testing of certain safety measures like CRC. As already discussed, CRC is not formal friendly, and it poses significant challenges to formal propagating faults through it. An easy misconception is to believe that all safety elements require a fault campaign. The ISO standard [1], Part 5, Appendix D lists several tables that can be used as a starting point for evaluating diagnostic coverage when accompanied by the proper safety rationale. The tables use a diagnostic coverage of 60% for "low", 90% for "medium", and 99% for "high". In table D.6, a CRC safety measure is considered a high DC or 99% (of course, care must be taken when choosing the appropriate polynomial length). In other words, not all safety mechanisms require a fault campaign, and often their DC is selected from Appendix D tables and used directly in the FMEDA. The lesson here is to ask first, *"Is a fault campaign even necessary? Is it worth all the effort?"* In the case of a CRC mechanism, if it is implemented properly then it is very difficult to find any faults that it cannot catch. Using the guidelines from Appendix D is probably the better choice.

> *Lesson #12 - Learn your tool vendor's terms because none use the ISO terminology.*

Finally, learn your formal tool's terminology for faults and how they relate back to the ISO standard. None of the commercial formal safety apps use the exact terminology from the standard. For example, the ISO standard [1], Part 5, Figure B.2 illustrates fault classification and their terms:

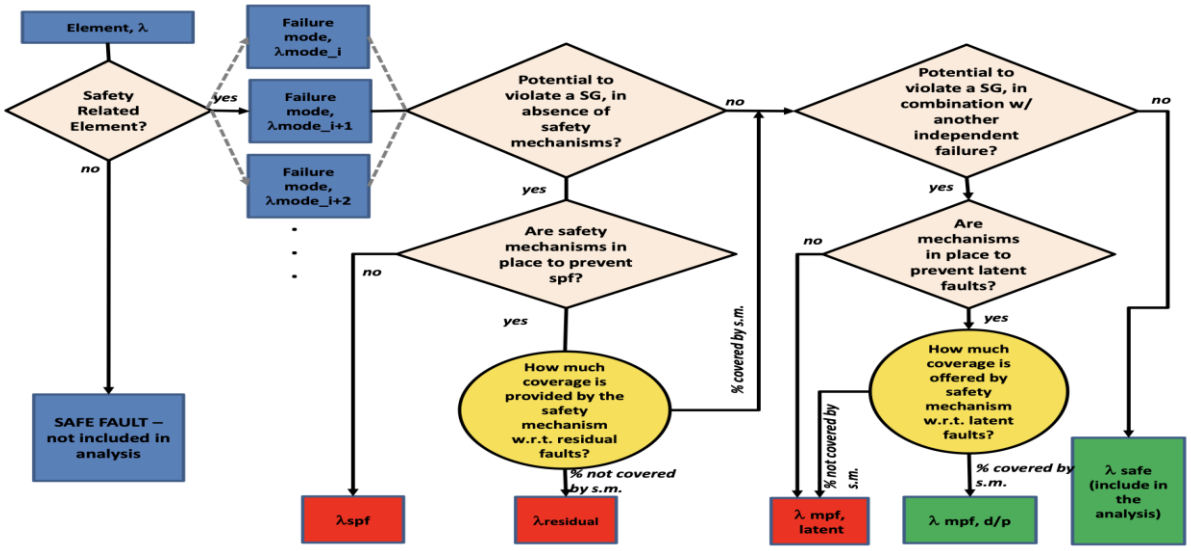

Figure 7: Fault classification diagram from ISO26262-5:2018(en), Figure B.2.

The diagram shows the fault classifications of SPF (single-point fault), RF (residual fault), MPF,D/P (multi-point fault, detected/perceived), MPF,L (multi-point fault, latent), and SF (safe fault). Formal safety apps report results as “detected”, “undetected”, “activated”, “non-propagated”, or “non-detected”. Certainly, these descriptions are more natural and intuitive, but formal safety apps leave calculating the diagnostic coverage up to the end-user, which makes it hard to know what values to plug into the DC formula. Other tools like fault simulators may use different terminology as well; in particular, care should be taken with the word “safe”. In the case of formal apps, using the term “safe” is accurate if a fault falls outside of the COI, or it has been proven to not affect the safety output. A formal tool can absolutely prove this. In simulation, however, it is very hard to prove that a fault will *never* affect a safety output. Therefore, be sure to check with your vendor regarding what is really meant by the word “safe”. When an EDA tool classifies a fault as “safe”, it may be that the tool was implemented by individuals who do not understand the safety process. “Safe” or “unsafe” is determined by a project’s HARA (Hazard And Risk Assessment) and ultimately depends on the safety goals.

III. Conclusion

Formal verification should be included in every safety project. Formal’s ability to prove properties is ideal for functional verification of safety elements and their safety mechanisms. Formal’s built-in synthesis and cone-of-influence tracing makes it the perfect front-end tool for generating fault lists for random fault testing. It also works great for merging results with simulation and emulation. Sequential equivalency checking technology enables fault propagation testing through a safety element and safety alarm, and nondeterminism enables multi-point fault injection for latent fault analysis. While a formal safety app simplifies the process, FPV and SEC are enough to effectively use formal for random fault campaigns. Whichever method you use, simplify your efforts by picking a low number of faults with a high level of statistical confidence. Likewise, consider using ISO guidelines for DC determinations instead of time-consuming random fault testing. The simple lessons presented in this paper will hopefully help you be more successful in your next safety project.

References

[1] ISO26262:2018(en), *Road vehicles - Functional safety.* International Standards Organization, 2018.

[2] Prasad, A.V.S.S. & Agrawal, Vishwani & Atre, Madhusudan. “A new algorithm for global fault collapsing into equivalence and dominance sets.” *IEEE International Test Conference*, 2002. 391-97.

[3] R. Leveugle, A. Calvez, P. Maistri and P. Vanhauwaert. "Statistical fault injection: Quantified error and confidence." *Design, Automation & Test in Europe Conference & Exhibition.* Nice, France, 2009 . 502-506.

[4] Robert Chang, Sep Seyedi, A. Veneris, M. Abadir. *Exact Functional Fault Collapsing in Combinational Logic Circuits.* Journal of Electronic Testing, 2003.