Accelerating Error Handling Verification of Complex Systems: A Formal Approach

Bhushan Parikh, Peter Graniello, Neha Rajendra
Intel Corporation
5000 W. Chandler Blvd,
Chandler, AZ 85226

***Abstract*-Error handling verification is one of the key phases in determining reliability of any embedded system. It involves verifying that the system correctly detects and gracefully reports various errors. This is especially critical for Smart Network Interface Cards (NICs) as they are usually located in an isolated environment and need to be continuously online with a very little to no human interaction. Failure to report an error may expose security vulnerabilities such as denial of service. Due to the technological advancement in recent years, the complexity of Smart NICs has increased, resulting in a greater number of error scenarios. This has made the task of error handling verification even more challenging using constraint based random verification (CBRV). In this paper we will demonstrate how leveraging Formal Property Verification (FPV) can address these challenges using our work on error handling verification of a hardware (HW) Decompression IP.**

I. IMPORTANCE OF THE ERROR HANDLING VERIFICATION

Computer systems have achieved significant progress in the areas of technology, performance, and capability during the last quarter century. These systems are now distributed systems instead of a standalone system which has created several challenges to meet high Reliability requirement of 99.999% [2]. A reliable system does not silently continue and deliver results that include corrupted data. Instead, it detects, reports and, if possible, corrects the corruption.

Error handling is one of the key components in evaluating Reliability of the system. It is the process comprised of anticipation, detection, and resolution of errors. Error handling ensures that the system has correctly identified and reported the error. In addition, it guarantees that the system gracefully completes erroneous execution and stays available for serving the next task, thus maintaining the normal flow of execution.

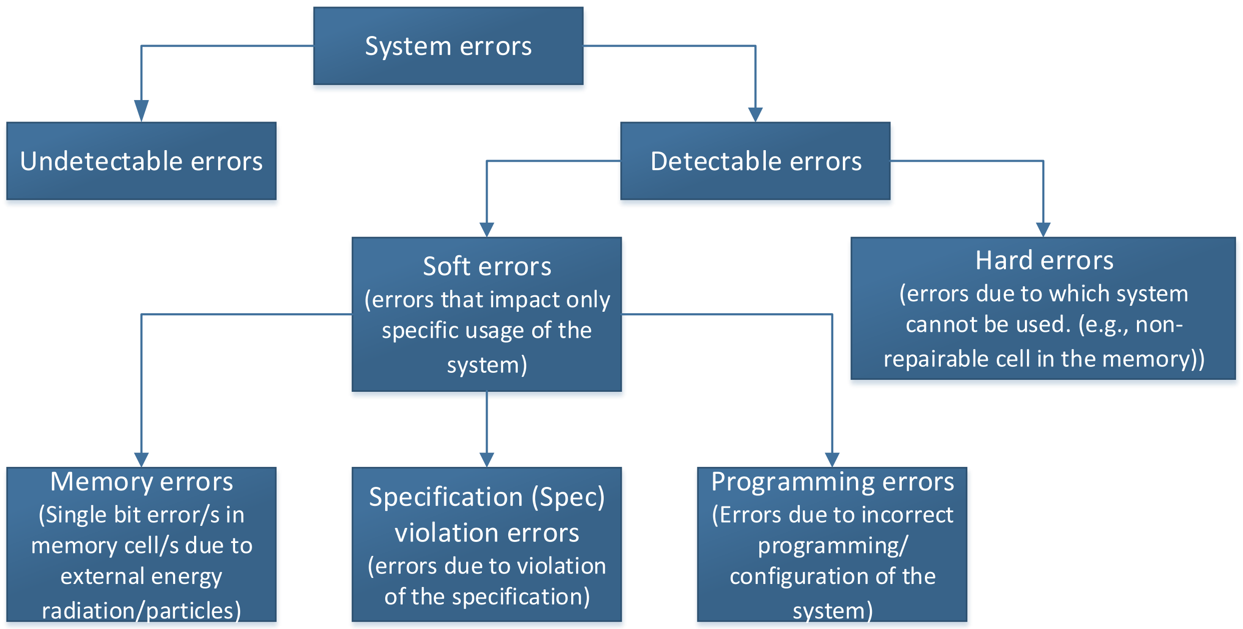
System errors
Undetectable errors
Detectable errors
Soft errors (errors that impact only specific usage of the system)
Hard errors (errors due to which system cannot be used. (e.g., non-repairable cell in the memory))
Memory errors (Single bit error/s in memory cell/s due to external energy radiation/particles)
Specification (Spec) violation errors (errors due to violation of the specification)
Programming errors (Errors due to incorrect programming/ configuration of the system)

Figure 1. Classification of System Errors

As shown in Figure 1, system errors can be classified in two main categories, Undetectable and Detectable. Since only detectable errors can be reported, handled, and corrected, handling of these errors falls in scope of the error handling verification.

Categorization of detectable errors gives an idea of the vast possibilities that can result in an error which the system needs to report, handle and, if possible, correct. This makes error handling verification extremely difficult to plan, execute and deliver. In addition, any gaps/holes in verification will result in late breaking pre silicon bugs or last-minute post silicon issues. We will use a lossless HW Decompression IP as an example to illustrate this.

II. Challenges associated with error handling verification

A HW Decompression IP consumes compressed data stream as input and produces uncompressed data. Figure 2 shows a block diagram of a conceptual compressed data stream.

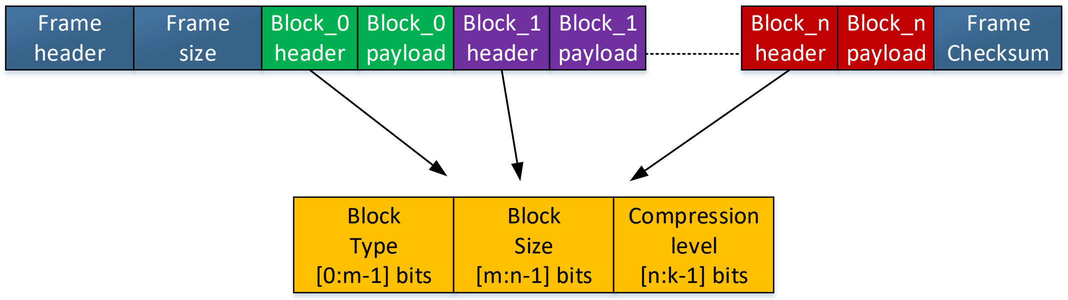

Figure 2: Block diagram of a conceptual compressed data stream

The stream consists of various fields – Frame Header and Frame Size followed by 1 or more Block Header and Block Payload pair and terminated with a Frame Checksum. Usually most of these fields are multibit fields. The field widths and possible field values are determined by the targeted algorithm specification. In addition, some of the values and combinations of these fields are marked as reserved for future use. For example, in RFC 1951 specification, block type is a two-bit field and has defined 2'b00, 2'b01, 2'b10 as valid values while 2'b11 as reserved value [1]. The presence of any such reserved/undefined values or combinations in the compressed data stream should be considered as a violation of the specification. It is the responsibility of HW Decompression IP to detect and report such combinations as errors and gracefully assert completion of the task.

In the CBRV methodology, various compressed data streams are used as test stimulus to verify a lossless HW Decompression IP. Hence, a Pre-Si Verification (PSV) engineer must identify the right uncompressed data that can generate desired compressed data streams. Typically, various SW compressors are used to process the desired uncompressed data and generate compressed data streams. The SW compressor comes with limitations, as listed below, which becomes a major roadblock in verification of error detection capability of a lossless HW Decompression IP.

1) *It will only generate a subset of all possible legal encodings*
2) *It has subtle optimizations to improve performance which are not easy to reverse-engineer from the implementation.*
3) *It is spec compliant and will not generate a compressed data stream with illegal encoding or field values.*

One approach to address this is to use data generator – a utility that can produce desired compressed data stream based on input data and some other knobs/options, but this requires additional time and resources for the development and verification of such data generator. Also, it does not address the problem of defining and executing various test-cases targeting numerous combinations of compressed data stream required to thoroughly verify error detection task of error handling verification activity. To summarize, error handling verification of a lossless Decompression IP is very challenging and may require months of effort/resources to complete it using traditional CBRV methodology.

III. Proposed methodology

Formal Property Verification (FPV) is a technique that is widely acknowledged and accepted for improving validation effectiveness [3]. This methodology is commonly used to accomplish functional verification goals and identify corner case bugs in the normal (i.e., non-error) functionality of the design. In contrast, we extended this methodology for the error handling verification task of our lossless HW Decompression IP and completed the task in half the number of weeks compared to CBRV. To present this methodology we will start with the high-level plan for the error handling verification task, a brief discussion of the challenges associated with it followed by how various FPV techniques can address the challenges.

A high-level plan for error handling verification is comprised of,

1) *Error detection verification:* verify that the system is detecting error correctly
2) *Error reporting verification:* verify that the system is reporting any detected error correctly
3) *Graceful completion verification:* verify that the system completed erroneous execution gracefully and is ready for next task/request

A. Error detection verification

As previously seen, the biggest challenge with error detection verification of the decompression IP is to create corrupt compressed data streams for the desired error condition. In FPV methodology one needs to specify only the expected behavior for the given error condition instead of identifying the required stimulus to exercise the behavior and the formal tool will ensure to exercise every possible stimulus. This makes FPV very fast and effective for error detection verification.

Let's understand this in a little more detail using an example. Consider the Block header field in Figure 2, it contains the size of the block payload (BLOCK_SIZE). The maximum value of block payload (BLOCK_MAX_SIZE) is defined in the targeted lossless compression standard specification (e.g., [4]). A spec compliant and reliable HW Decompression IP is required to assert an error signal (e.g., *block_max_size_error*) when it detects BLOCK_SIZE > BLOCK_MAX_SIZE.

In the CBRV methodology, the PSV team will need to test every single compressed stream which can result in (BLOCK_SIZE > BLOCK_MAX_SIZE) condition. Unfortunately, this results in many test cases. With FPV, the problem can be solved with two assertion properties as shown in Figure 3 - an assertion to detect the (BLOCK_SIZE > BLOCK_MAX_SIZE) error condition and an assertion to detect the block_max_size_error signal. Here the formal tool identifies all possible conditions for (BLOCK_SIZE > BLOCK_MAX_SIZE) and verifies the respective error signal trigger.

```
parameter BLOCK_MAX_SIZE              =     BLOCK_MAX_SIZE_NUMBER;
parameter BLOCK_MAX_SIZE_WIDTH        =     $clog2(BLOCK_MAX_SIZE+1);

logic [BLOCK_MAX_SIZE_WIDTH-1:0]            block_size;
logic                                       block_max_size_error;

assert_block_max_size_error_when_detected :
   assert property ((block_size > BLOCK_MAX_SIZE) |-> ##[0:M]  block_max_size_error);
                                            //M >= 0, dependent on the implementation
assert_block_max_size_error_only_when_detected :
   assert property (block_max_size_error |-> (block_size > BLOCK_MAX_SIZE));
```

Figure 3. Assertions Required to Verify a Spec Error

The TABLE I shows some more examples of complex spec errors for which we have leveraged FPV methodology to verify that the lossless HW Decompression IP detects these errors correctly.

TABLE I
SPEC ERRORS AND THEIR BRIEF EXPLANATION

Spec error	Explanation
Unbalanced Huffman tree	The Huffman tree used for compression is not balanced
Distance out of range	The copy location in token is too far behind
Received incorrect code	The code does not match with encoding of any symbol from the compressed stream
Invalid block header	The block header field/s description does not match with the specification
Invalid encoding	Decoded symbol value does not match any value specified in the specification
Incomplete stream	The compressed stream is not complete
Padding byte error	Padding bytes in the compressed stream are corrupted

B. Error Reporting Verification

To reduce the complexity of the error reporting structure for the system, we implemented a scheme that utilizes a single error valid signal accompanied by an 8-bit error code bus reflecting an error code for the targeted error as shown in Figure 4. Thus, the task of error reporting verification required verification of 256 different test-cases.

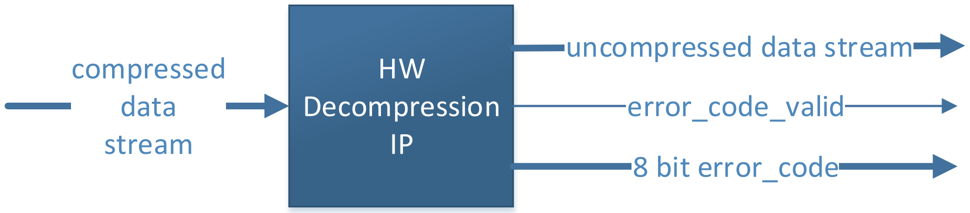

Figure 4. High Level Block Diagram of the HW Decompression IP with Error Code Valid and Error Code Bus

Like the verification of error detection, here also PSV will need to generate many tests to cover the breadth of conditions to verify error reporting. With FPV, the two assertions shown in Figure 5 would sufficiently complete the error reporting verification for any given error condition (ERROR_CONDITION_N, where N >= 0).

It is important to note that when using FPV methodology the verification is more exhaustive. For example, assertions defined in Figure 3 do not only check that error is asserted when there is an error condition but also verify that error is asserted if and only if error condition is seen. Similarly, assertions defined in Figure 5 check that the error code is generated if and only if its respective error condition is observed in addition to the generated error code is correct.

```
parameter ERROR_CODE_BUS_WIDTH                  =    8;
parameter ERROR_CODE_INCOMPLETE_BLOCK           =    255;
parameter ERROR_CONDITION_INCOMPLETE_BLOCK      =    INCOMPLETE_BLOCK_DETECTED;

logic [ERROR_CODE_BUS_WIDTH -1:0]        error_code;
logic                                    error_incomplete_block;
logic                                    error_condition;

assert_error_code_incomplete_block_when_detected_error_condition_incomplete_block_detected :
   assert property ((error_condition == ERROR_CONDITION_INCOMPLETE_BLOCK) |->
                                     ##[0:M] ((error_code == ERROR_CODE_INCOMPLETE_BLOCK) && (error_incomplete_block)));
                                                   //M >= 0, dependent on the implementation
assert_error_code_incomplete_block_only_when_error_condition_incomplete_block_detected :
   assert property (((error_code == ERROR_CODE_INCOMPLETE_BLOCK) && (error_incomplete_block)) |->
                                     (error_condition == ERROR_CONDITION_INCOMPLETE_BLOCK));
```

Figure 5. Assertions Required for Verifying an Error Code

C. Graceful Completion Verification

Now that the error detection and reporting for all errors are verified, the next step in error handling verification is to ensure that the system will gracefully exit to a known (or an IDLE) state and be serviceable. This is a very important reliability aspect for network accelerators such as a lossless HW Decompression IP. For example, if an attacker gets access to a compressed stream that can put the system in an unrecoverable (also known as hang or an error state where it always produces incorrect output) state then he/she can bring down the entire network by constantly sending the bad compressed stream and keeping the system in hang state thus making it unavailable for rest of the users. This is known as Denial of Service (DoS) attack. Therefore, it is essential for the lossless HW Decompression IP to complete its task and return to a known state in the event of an erroneous stream.

As we have seen in the previous section our design of lossless HW Decompression IP has a total of 255 error codes. Now for any of the 255 possible errors, an error can occur at any cycle and the timing of the error assertion is an asynchronous event with respect to the rest of the design functionality. Also, there may be cases where more than a single error is present. Hence, it may require weeks/months of random regressions and coverage analysis to verify only a subset of the possibilities and impossible to verify all possible combinations without missing the IP development schedule using the CBRV methodology.

Formal tools have the capability to create cut-points in the design i.e., disconnect any signal/net from its driver and drive that signal to every possible value during the formal run. This can be very powerful in verifying graceful completion. To understand this better let’s consider Figure 6. It shows a simplified block diagram of the lossless HW Decompression IP error handling logic along with the control logic as a Finite State Machine (FSM) diagram on the right.

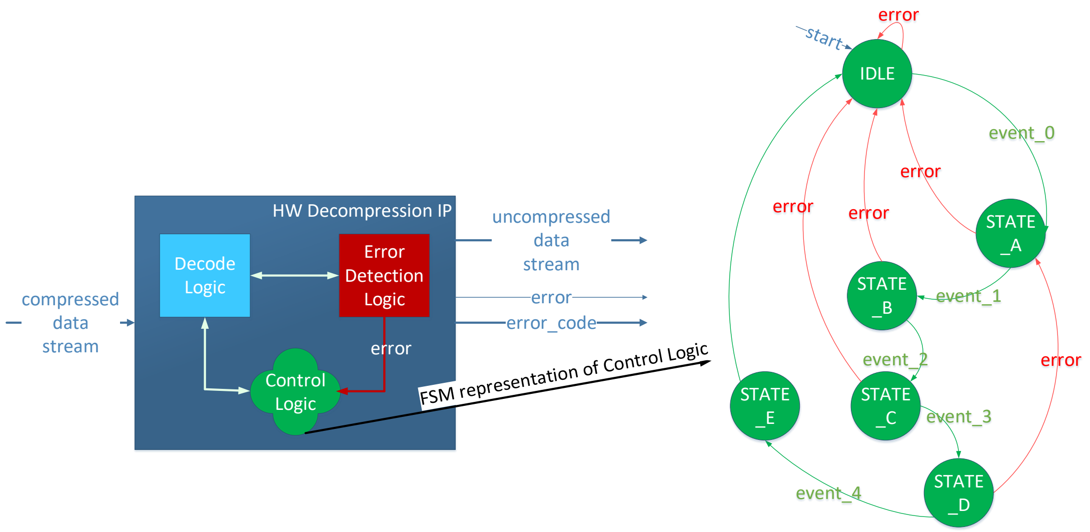

Figure 6. Simplified Block Diagram of the Lossless HW Decompression IP Error Detection and Control Logic

To verify that the FSM returns to the IDLE state after an error condition, the CBRV methodology requires the following:

1) *Develop a test that can cause the error signal to assert.*
2) *Add hooks to the test such that the error signal can only assert when the FSM is in the specific state.*
3) *Repeat this for all states of the FSM.*
4) *Develop and analyze functional coverage to verify that all possible cases are covered.*

In the FPV methodology, a cut-point can be created for the error signal as shown in Figure 7. The formal tool then has the freedom to assert the error signal whenever it is required. If there are no assumptions added to the error signal, the formal tool will exercise all possible combinations. Using this strategy, only two assertions, shown in Figure 8, are needed to verify graceful completion logic of the design.

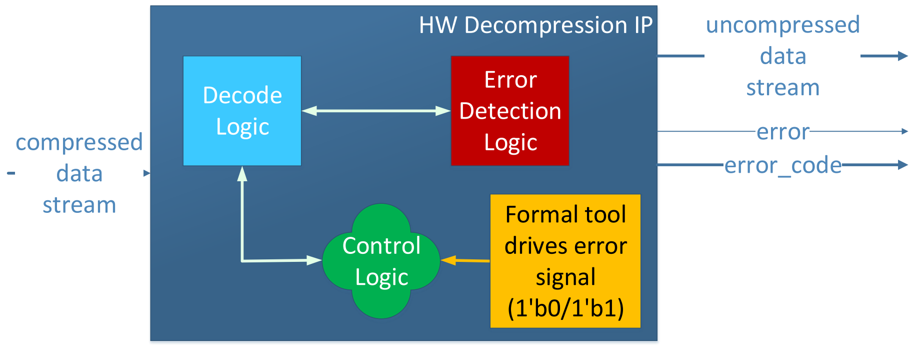

Figure 7. Simplified Block Diagram of the Lossless HW Decompression IP Error Detection with Cut-Point

```
//Formal environment setup to create cut point for signal error

assert_fsm_1_transitions_to_idle_state_after_error_assertion :
   assert property ($rose(error) |-> ##[0:M] (fsm_1_present_state == IDLE));
                                                      //M >= 0, dependent on the implementation
assert_fsm_1_stays_in_IDLE_state_if_error_is_asserted :
   assert property ((error && (fsm_1_present_state == IDLE)) |-> ##1 (fsm_1_present_state == IDLE));
```

Figure 8. Assertions Required to Verify FSM Transition to IDLE State When Error Occurred

IV. RESULTS

Figure 9 shows the total number of bugs found using FPV and CBRV methodology for the error handling verification task of the lossless HW Decompression IP. It is important to note that we deployed FPV methodology for error handling verification almost a quarter after the CBRV started. Despite the late deployment of FPV, we had a very high Return on Investment (RoI) using this methodology. The exhaustive nature of FPV found 75% of the overall error handling bugs. We also did complexity analysis of the bugs identified using FPV and found that most of these bugs were complex and required multiple events to align correctly which would have been difficult to achieve using the CBRV methodology. Out of the bugs found by CBRV methodology, we missed only 1% of the bugs because of missing formal checkers due to misunderstanding of the specification. For completeness we developed these checkers and discovered the same bugs using FPV methodology. The remaining 24% of the bugs were identified using FPV methodology as well. However, due to its early deployment, these 24% of the bugs were first identified using CBRV methodology hence, we counted it as part of the bugs found by CBRV methodology.

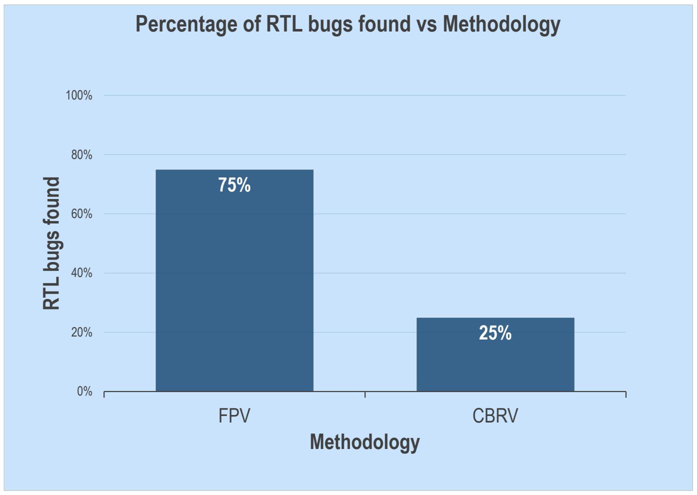

Figure 9. Comparison of Bugs Found vs Methodology

We used various formal coverage metrices (functional and code) and did iterative reviews of FPV test-plan to conclude our FPV activities. We deployed various industry standard techniques and achieved 100% convergence for all our assertions. Figure 10 shows the comparison of verification progress between FPV and CBRV. This clearly reflects the challenges associated with the CBRV, some of these are complex test-bench development and directed/focused tests to exercise specific scenarios.

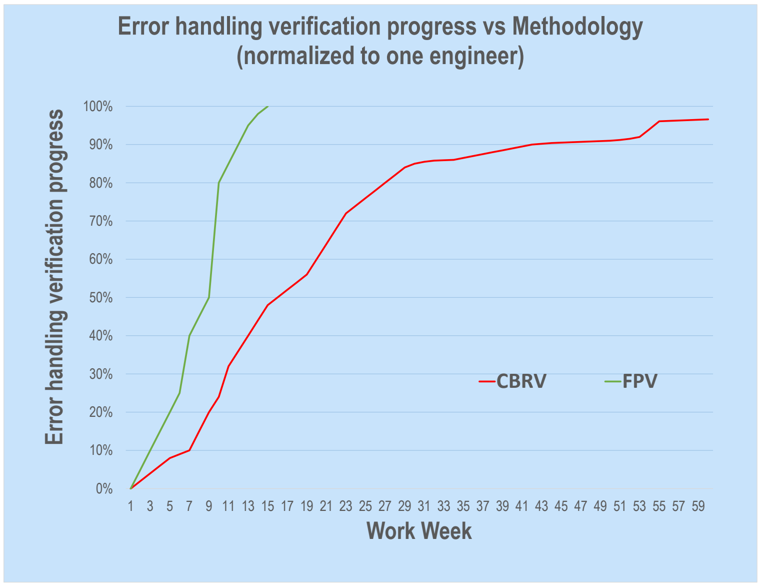

Figure 10. Error Handling Verification Progress vs Methodoloby

V. Conclusion and future work

In this paper, we discussed the importance of error handling verification and its challenges associated with the constraint based random verification methodology. Based on the lossless HW Decompression IP, we demonstrated the effectiveness of applying FPV to error handling verification. We identified 75% of the error handling bugs in short time of twelve weeks compared to twenty-eight weeks of time in the constraint based random verification methodology. Leveraging this methodology in the error handling verification of the decompression IP was one of the key factors to reach the verification quality metrics six weeks ahead of the planned tape-out date. We strongly believe that this methodology will also provide similar schedule savings without increasing project budget for a wide variety of designs, especially those which involve industry standard protocols/interfaces (e.g., PCIe, AMBA, etc.).

Future work includes deploying this methodology at System on Chip (SoC) to verify system level error handling requirements. In addition, due to its exhaustive nature we believe FPV methodology can be very effective in security verification and fuzz testing.

References

[1] P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3”, https://tools.ietf.org/html/rfc1951, 1996.

[2] Hoang Pham, " Handbook of Reliability Engineering", 2003, p. 201.

[3] Erik Seligman, Carl Dreyer, Ken Hare, Raman Nayyar, "Zero Escape Plans: Tying Together Design, Simulation and Formal Methods for Bulletproof Stepping Validation", DVCON USA, February 2008.

[4] Y. Collet, \LZ4: Extremely Fast Compression Algorithm", https://lz4.github.io/lz4/, 2011