CAPITALIZE ON COMPLEXITY

Nicholas Mc Guire, Markus Kreidl*, Dr. Sheng Cheng* [†]

**OpenTech EDV Research GmbH,
Bullendorf, Austria
hofrat,mkreidl@opentech.at*

*[†]Beijing Shenzhou Aerospace Software Technology Co., Ltd.
Beijing, China
chengsheng@bjsasc.com*

released under FDL V1.2

Keywords: complexity, KISS, masking, safety logic, SOUP

Abstract

One of the, maybe main, problems with understanding complexity is that we, for historic reasons, have been entirely focused on fighting it rather than utilizing it. Our way of looking at complexity is focused on evading the consequences it has on our way of thinking on our route of rational on our ability of deterministic deconstruction - With other words we have been focused on sharpening our prime weapon - Abstraction.

Safety related systems are focused on giving an acceptable risk of failure and the prime means has been “maximize determinism” - functional, procedural, temporal, organizational. With the ever increasing levels of abstractions being piled on top of each other, KISS (Keep it simple, stupid) is starting to be more wishful thinking than an implemented reality.

The prime thesis of this paper is that if one changes the perspective and looks at complexity as a resource then it well may be feasible to find technologies that can enhance safety through utilizing complexity. This seemingly paradox approach is motivated by starting at the root cause of faults in digital systems, then looking at classical mitigation and extending this classical mitigation to profit from system complexity through loose coupling. Finally we generalize this approach and use it to describe a (maybe) novel concepts we call dynamic data types which shows that one can protect against faults effectively by capitalizing on complexity.

In this paper we introduce an implementation of the principle of ”capitalize on complexity” and outline some practical example ”proof-of-concept” level implementation rather than trying to continue the long lost battle against complexity.

1. Introduction

Our problem with complex systems is that they are able to exhibit behaviour that we can’t predict, some- times don’t even know of. If we now try to define a ”good” property and make it reliable we run into the problem of how to deal with the unknown unknowns. The principle alternative is to revert the problem and define the ”bad” and then use the inherent non-determinism of complex systems to make this bad arbitrarily unlikely. At it’s core that is what architectural protection is doing.

Architectural protection though is still based on the individual node being functionally deterministic. Temporal relaxations, known as loose coupling, are well established (i.e. Thales TAS Platform) and have shown the ability to evade some potential

CCFs (Common Cause Fault). Taking this one step further we would give up the deterministic function in the sense that the implementation is non-deterministic - the functionality well may be deterministic at the logical level. This transition is one that security has done more than a decade ago (e.g. address space randomization) [4],[8]. The principle behind this is that we know there are ”bugs” in the code, but to exploit these bugs we need knowledge of locality or sequences, control flow information, and the like. The intent of securities use of randomness is to make this information sufficiently improbable to attain (or guess) and so protect against the attack.

Applying this to safety is equally possible - just that we must replace the intelligent threat agent by an unintelligent hazardous event. The unintelligent event being unfortunately more powerful as it is allowed to arbitrarily often retry and does not need any information to cause alterations and it does not limit alterations to being meaningful aka exploits. Even though the probability of a single event causing a meaningful system state (meaningful in the context of the intended logical function of the system) is small, the unbounded retry makes it a significant potential hazard that must be addressed. To stick to the security model above, we replace knowledge by correlation and the system stays in a sufficiently safe state as long as the ”bad” states don’t occur in a correlated way.

1.1 Problem Statement

The first ”root cause” of the problem is that there is a dense mapping from logical values to data type primitives in software; a boolean is represented as a 0 or 1 - both states are within the domain of the logic so alterations can’t be detected if they happen by unintended (and uncontrolled) interfaces like SEE or systematic faults (design/implementation errors) with other words there hamming distance is 0. The therapy thus would be to introduce a loose or spare coupling between logical values and data-representation.

The second ”root cause” of software implemented functionality misbehaving is the temporal domain impacting the validity or correctness of the systems state. One possible cause of this can be identified in the absence of temporal information in the data representation used. In safety related software systems we ”fix” this by using sequence numbers or time-stamps (i.e. EN 50159 methods [6]) but this is restricted to a generally small subset of operations like explicit communication. A more general approach would be to integrate the temporal behaviour into the primitive data object itself and build the safety related logic with a so extended data object which we call dynamic data types and introduce in section 3.

2. Developing the principle

Random faults are surprisingly simple to handle provided we can put a limit on the number of concurrent occurrences. If we can then basically we can employ a variety of architectural protection schemes to either cover them or at least detect them.

- 2oo2: single fault detection (at the price of halving availability – thas is halving MTTF (Mean Time To Failure))
- 2oo3: single fault coverage (trippling physical resources to get 5/6 availability)[7]

These well established methods are already utilizing the complexity of the system by making assumptions about the probability of correlated faults in all replicas. To reduce the probability further one has introduced loose coupling [5].

loose coupling

Basically make the bad case of synchronous ”lock-step” operation - which is a pre-requisite to certain CCFs in a TMR (Tripple Modular Redundancy) type protection system, sufficiently unlikely so as to make the system level ”bad” event, that this CCF would constitute, a tolerable risk. Essentially loose-coupling is an example of capitalizing on complexity - the underlying assumption is that the systems are temporally independent so that the bad event would not occur within a time window too small to detect it - with other words we are sampling the three ”urns” of the TMR for the bad event and asking what is the probability of getting two undistinguishable wrong entities at the same time ? Effectively we are building on the non-deterministic properties of the loosely coupled components of the system.

The fundamental difference between the lock- step TMR and the loosely coupled TMR now is that it is in fact, covering some systematic faults as well as random faults! What is the probability that you have three desktop PCs side by side, with a race condition in the OS (Operating System) and all three systems hit this same race condition at the same time (or within the detection time window) so as to produce majority of false-positives? It’s not 0 but it can be almost arbitrarily reduced by increasing the complexity of the 3 nodes. Increasing complexity here means increasing the state-space the nodes can operate in as well as the dynamics of the movement in this state space - essentially they simply should never be in the same state; more precisely - never be in the same substate space that is associated with the occurrence of the ”bad” event.

This transition to loosely coupled TMR vs lock- step TMR is a generalizable model of capitalizing on complexity. It not only retained the coverage of random faults (like SEUs – Single Event Upset) but it also extended the coverage to some types of systematic faults. In effect it translated the systematic fault to manifest itself as a random-fault - which we know how to handle - Architectural protection like a TMR.

2.1 Generalizing the principle

To generalize the principle the first step might be to look at what level of abstraction are we using for positioning the mitigation? A TMR is a node-level mitigation, a two-channel software solution would be a functional mitigation, can we go lower?

3. Encoding

3. Encoding

Value and primitive types:

In computer science, a value is an expression which cannot be evaluated any further (a normal form). The members of a type are the values of that type. For example, the expression ”1 + 2” is not a value as it can be reduced to the expression ”3”. This expression cannot be reduced any further (and is a member of the type Nat) and therefore is a value. [from Wikipedia.org/wiki/lvalue]

Coded processors are now fundamentally based on representing computation by expressions (in the above sense) rather than by values. And by doing this we increase the minimal modification of the representation necessary to still be a valid expression - with other words we increased the complexity of the necessary fault pattern needed to lead to a false positive result. Encoding is then by putting syntactic and semantic requirements on the expressions and defining operations that allow to manipulate the expressions, without transition to values, while retaining the syntactic and semantic property constraints. Checking these property constraints then is what allows to detect unwanted modifications during computations or thereafter.

While this is a possibility, it does not capitalize on the system complexity. Detection of alteration is what ensures the safety properties of the transformations encoded, but the probability of such alterations occurring increases with the complexity of the system. Thus this form of encoding will reduce the availability of the so encoded functionality.

A similar situation, just significantly more dramatic, is encountered by traditional (deterministic) replication approaches - like a TMR. The availability of a 2oo2 system is only 1/2 of the 1oo1 system, increasing complexity to a 2oo3 system yields an availability of 5/6 compared to the (unsafe) 1oo1 system. But as safety can’t be entirely decoupled from availability - notably in systems that then transit to a ”fall-back mode” like manual operations [6], the overall safety of a replicated systems with a reduced availability is also impacted.

The goal now is to have systems that do not have this reduction of availability with increased system complexity. To achieve this we must have systems that fail in complex/correlated situations only - with other words we must design systems where the failure case is the complex case and thus unlikely to happen. If this can be done, then the likelihood of this unsafe case happening would decrease with increasing complexity. We can again employ the analogy with security - the longer the password you chose the less likely it is that it can be guessed, provided it does not have a system that is easily discerned - so a long random password is more secure than a ”human-readable” (or rather meaningful sequence of characters based) one. If we now identify the guessing of the password with the hazardous ”false- positive” data item then the recipe is simply to increase complexity of the representation while retaining as simple as possible logical structure.

To complete the coverage of the two above cited ”root-causes” we must add time though. Current strategies for encoding of time are explicit time encoding, i.e. by time-stamps, which immediately raises the issues of time-stamp correctness and imposes requirements like maximum drift etc. Fundamentally time is though not encoded in the data but is only represented as a further data object in a sparse fashion - notably not all data translations can typically be timed/time- monitored. With respect to temporal faults encoding mitigates noise - noise describes random temporal faults does not include systematic temporal faults (offset, drift, oscillation, etc.). To mitigate this at the data type level we extend the concept of encoding to describe temporal properties in the dynamics of the encoding. This concept, which we call dynamic data types, is introduced in the next section.

4. Dynamic data types

Addressing the two above mentioned ”root causes” (with all the ambiguity of this term I do hope it is an acceptable use). The core is simply to map logical values to dynamic properties of a primitive data types rather than to individual values of primitive data types - in some way simply an extension of well known encoding methods. A simple example of this mapping would be to represent a signal by an array of integers and encode the logical value TRUE by a discrete sine-wave of n Hz while logic FALSE is encoded in n/2 Hz. Thus now any alteration introduced in the data representation would either lead to an unknown value (something other than n Hz or n/2 Hz) or to a ”spike” without actually altering the underlying valid encoding of n or n/2 Hz.

This model we call dynamic data types (DDT). The question to answer here now is, how to encode logic on top of these alternative representations in a digital system and how to demonstrate that it is resilient against a fault class of interest.

In this section we introduce a novel (?) concept of encapsulating the general signal in a way that en- forces inherent mitigation of faults. At this point we are not (yet) claiming that the fault coverage is actually complete - but we do think that the majority of relevant faults are covered notably without impacting availability.

The system model behind dynamic data types is to view the input signals as streaming through the system and accumulating changes based on the operations done on them encoded in the dynamics only. Thus each sequence of operations is entirely encoded in the data type itself and can be validated entirely by placing acceptance criteria on the emitted data. The basic operations that are currently being studied are:

- Signal expansion: i.e. adding of signals
- Signal reduction: i.e. filtering of signals • Delaying the signal: i.e. phase shifting

Note that all of these operations are vector or matrix operations - thus there are not single values (ideally) that could switch between two valid outputs.

4.1. Signal Requirements

The requirements listed here are only from a safety perspective and not from a functional perspective. Further more they are most likely not yet complete as this project is in an early stage.

- all valid states must be active states
- all elements may only react to valid states
- SEUs shall not impact the valid frequency domain
- Operations on signals may not include a single point-of-failure

To satisfy these - admittedly very crude - requirements, we propose mapping logical values to discreet frequencies, with a tolerated range. In this example we mapped:

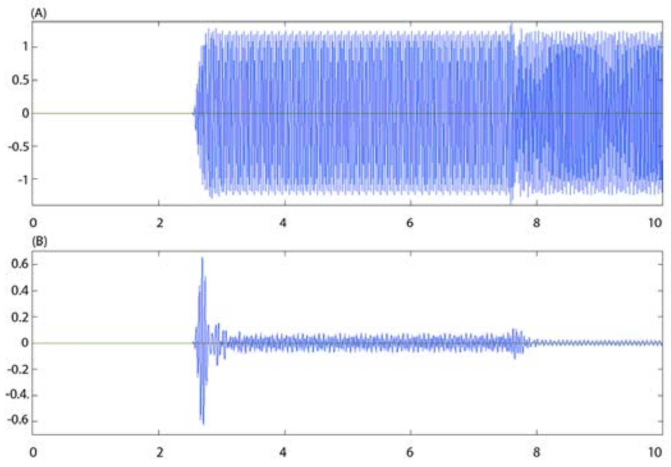

Figure 1: Response to input omission failure

TRUE	*6Hz*
FALSE	*12Hz*

Further signal interaction is by multiplication, thus any signal omission would lead to multiplication with a 0 (or static) value and thus no valid output.

Stuck at failures are a bit more critical, selecting proper frequencies in the generated intermediate spectrum in combination with bandpass filtering allows covering both omission and stuck-at faults.

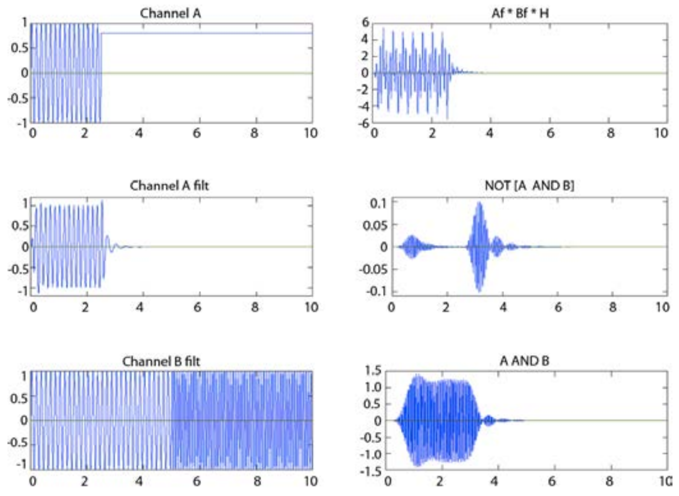

FIGURE 2: Input response to stuck-at failure of

4.2. Core concepts of dynamic data types

From a safety perspective dynamic data types concept can be summarized as

- Values are bound to a frequency spectrum and so conserve temporal information of the signal and signal history.
- Complex logic can be described by signal composition
- Compromising dynamic data types would require arbitrarily complex correlated alterations of data/code to yield a valid signal
- Increased system complexity reduces the probability of a false-positive signal generation
- Essentially the goal of dynamic data types is to ”scale” with system complexity growth which we consider inevitable.

4.3. Composable logic

A further issue for any control system is composability. While we are restricting the model here to boolean logic at present extensions do seem possible. Never the less the first step is to demonstrate composability of basic logic elements. The structure of the example is depicted below:

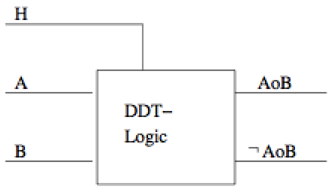

FIGURE 3: Logic structure of DDT example

Even though the presented example is not a wildly impressive level of complexity - it does demonstrate the principle capabilities of the concept nicely. In this example we have two signal inputs - A and B - and a logic input H, which allows switching the logic from AND operation to XOR operation. The input signals again use:

TRUE	*6Hz*
FALSE	*12Hz*

and logic input H used to select the logic operations to be performed on the inputs uses:

AND Logic	*9Hz*
XOR Logic	*18Hz*

Note that the frequencies selected here during simulation, technically are not sensible, but the principles don't change when transformed to other ranges of the spectrum - any real-life system would be operating in the kHz range.

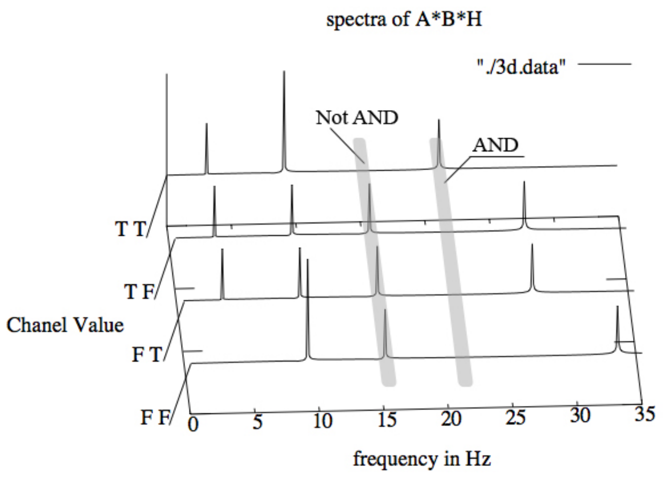

FIGURE 4: Spectra of A*B*H for different input values of Channel A and B

Thus for the given simple logic the spectrum of A * B * H contains the four different spectra for the given input settings of channel A and B. If these product signal are now passed through two parallel output lowpass filters of 15Hz (logical NOT AND) and 21Hz (logical AND) then the output of the DDT logic represents an AND gate. Similar for XOR (by setting the H frequency to 18Hz) spectra are generated and filtered to extract the XOR logic.

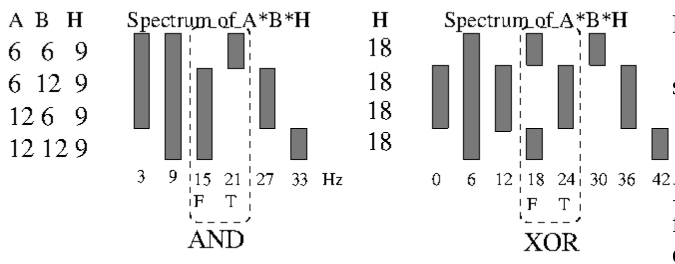

FIGURE 5: Frequency spectra of ABH for different H settings

Inspecting the generated frequency spectrum one can extract further logic operations on the inputs from the output spectrum through adjusted filters as well as additional logic inputs. At this point we neither know the scalability of this composition nor do we know its limits - but the current state of experiments is sufficient to already produce some basic boolean safety logic that could be practicable for safety related applications running on complex (non-safety) hardware software platforms.

4.4. Masking through complexity

As noted this is in the context of efforts to utilize complexity of underlying hardware/software systems to enhance safety rather than fighting them. With the mapping presented in the last section we obtain a process to construct outputs that is not susceptible to faults in the underlying software/hardware systems because any false-positive requires an arbitrarily long correlated sequence of faults to cause manifestation. Any fault in the underlying system can impact the processing of dynamic data types at arbitrary points, but as there is no single point of failure in the transformation of the input stream to the output stream, no such single fault could impact the logical correctness - it can of course diminish availability. Assuming that each period of the sinosoidal input signal were represented by 32 samples, and 28 samples were needed to still reliably extract a correct signal, then the MTTF of this system of n diverse channels would be:

$$MTTF=\frac{1}{\lambda}*\sum_{x=k}^{n}\left(\frac{1}{x}\right)$$

For n=32 and k=28, we get ~2.2 times the MTTF of a single signal with the same "bit-flip-rate", thus while this solution is computationally expensive, it may well compensate this disadvantage by increasing availablility and most notably eliminating the reliability demands on the underlying resources of the OS to ensure safety properties (e.g. reliable voting in a 2-out-of-3 system). If the reduced system level resource demands can compensate the increased complexity of the safety-logic processing is hard to answer in general. From our current work we think that for simple safety-logic the resource expense pays off as one no longer needs to rely on guaranteed safety properties of the underlying OS and hardware. Regarding multiple faults, there is of course the possibility of accumulated dormant faults emerging and these then striking at some mode change - i.e. in the above example when switching the mode- input from AND logic to XOR logic - but any such fault would have to again generate a complex sequence of values to lead to a false positive output which is not impossible but unlikely - how unlikely it is is still under active investigation.

5. Conclusion

It is a bit early to draw much of a conclusion with regards to the specific concept of DDT, but the conclusion that we do see suitable at this point is that complexity can be utilized to improve safety and reliability properties of critical logic. Effectively what DDT shows is that by increasing the complexity of the operations and the data objects used to describe the desired operations - simple as they may be - results in false positives depending on arbitrarily complex fault patterns and thus can be made arbitrarily unlikely - increasing safety and reliability.

We believe that this concept that seems to diametrically oppose KISS is actually well in line with safety paradigms - keep the requirements and the design of the safety related components simple and clear. While keeping the implementation simple has resulted in many models of ”safe coding” (MISRA C) or all sorts of wrapper protections (EN 50159 methods) that ultimately yield high code-complexity as well.

Essentially we need methods that will scale with growing complexity of the underlying system components or safety related systems will loose the benefit of utilizing widely deployed and thus well-tested technologies. DDT is a demonstration that capitalizing on complexity is technically feasible - more work is still to be done.

Authors Biographies

ChengSheng received his Ph.D. from Zhejiang University in China and is currently Director of Embedded R&D Division of Beijing Shenzhou Aerospace Software Co.Ltd. His research interests are in real-time operating system, software V&V, system-on-chip, on-board electronics system. He has published two books and over 50 papers.

Markus Kreidl (M.Sc.) is currently working at OpenTech EDV Research GmbH. During his time with Liebherr Aerospace and Transportation he was responsible for Linux OS and architectural Software Design. He also worked for Thales Transportation Group where he was responsible for safety related soft and hardware development. His main research interests are located in safety related low-level software and FPGA development.

Nicholas Mc Guire received no degree what so ever. Has worked as self tought consultant for Thales, Siemens, Liebherr and others. Founder of OpenTech EDV Research GmbH in 2003 has been working on RTOS and safety issues for the past decade. Established the Distributed and Embedded Systems Lab of Lanzhou University in 2005 and has focused on the problem of using complex systems for safety in the past years.

References

References

[5] Jens Braband, “Risikoanalysen in der Eisenbahn- Automatisierung”, Eurailpress, 2005

[2] S. Forrest, A. Somayaji, “Building Diverse Computer Systems” D. H. AkleyComputer Systems, 1997

[3] A. Gerstinger, H. Kantz and C. Scherrer, “TAS Control Platform: A Platform for Safety- Critical Railway Applications”, ERCIM, 2008

[4] Nancy Leveson, “Safeware: System Safety and Computers”, Addison-Wesley, 1995

[5] Ozello Patrick, “The Coded Microprocessor Certification”, SafeComp, 2008

[6] EN 50159-1 Railway applications, “Communication, signalling and processing systems Part 1: Safety-related communication in closed transmission systems”, CENELEC, 2003

[7] Marvin Rausand,Arnljot Hoyland, “System Reliability Theory, Second Edition”, Wiley Inter-Science, 2004

[8] Hovav Shacham, Matthew Page, Ben Pfaff, “On the Effectiveness of Address-Space Randomization”, USENIX, 2003