Software Safety
A journey across domains and safety standards

Jean-Paul Blanquart(1), Emmanuel Ledinot(2), Jean Gassino(3),
Philippe Baufreton(4), Jean-Louis Boulanger(5), Stéphane Brouste(6),
Jean Louis Camus(7), Cyrille Comar(8), Philippe Quéré(9), Bertrand Ricque(4)

(1): Contact author, Airbus Defence and Space, jean-paul.blanquart@airbus.com;
(2): Dassault Aviation; (3): Institut de Radioprotection et de Sûreté Nucléaire;
(4): Safran; (5) CERTIFER; (6) Groupe PSA; (7): ANSYS-Esterel Technologies; (8): AdaCore; (9): Renault

Abstract: The position of software regarding the global system safety is subject to significant variations among the various application domains and their safety standards. As a consequence, the position regarding whether, how and to which extent software safety analyses could or should contribute to the global safety assessment also varies.

In Civil Aviation [ARP 4754A; ARP 4761; DO 178C], Nuclear [IEC 61513, IEC 60880] and to some extent Space [ECSS Q40; ECSS Q80], safety analyses are performed at system level and on functions, sub-systems and equipment, but not under the form of dedicated safety analyses applied to software. In these domains, the rationale is that software contributes to system safety through adherence to software development and validation rules i.e. through an argument on confidence in software correctness to an extent adapted to the consequences of failures. However it is worth noting that the assessment of the consequences of failures, and hence the determination of the Development Assurance Level, or Software Criticality Category, etc., result from safety omnibet analyses performed at system and not at software level.

Conversely in domains such as railway [EN 50129] or automotive [ISO 26262], whereas the overall safety rationale is very similar, it is still required to perform in addition dedicated safety analyses applied to software.

In the Base Safety Standard [IEC 61508], the software safety analysis encompasses a set of normative means supporting the functional safety assessment. In the process industries domain [IEC 61511], the concept is only emerging.

In this paper, from our experience in safety practice and standard in several domains and discussions within a working group dedicated to cross-domains comparison of safety standards, we propose a description of classical software safety analysis techniques and discuss the nature of arguments they can provide, according to their various objectives and situation in the global process and safety argumentation. We also discuss why software complexity increase has progressively made completeness of system functional safety requirements an important issue. Inspired by STPA and contract-based design in software engineering, we make constructive propositions to mitigate the incompleteness risk. We sketch out a general specification setting, in which another kind of "software safety" analysis would come into play. We conclude on whether and how this could fit in the global system safety assessment.

Keywords: software safety, software safety analysis, Software FMEA, SEEA, STPA, rationale of safety standards, DAL, SIL, ASIL, SSIL, cross-domain comparison.

1. Introduction – Position of the paper

Embedded France is an initiative launched in France by major industrial companies involved in the development of critical embedded systems in a very wide spectrum of application domains. Its objectives are to improve its members' capabilities to meet the major challenges of the development of embedded systems, in particular software intensive safety critical embedded systems. It elaborates propositions, recommendations, roadmaps etc. based on collaborative work and discussions in dedicated thematic Working Groups.

One of these Working Groups is dedicated to safety standards[1] and gathers industrial safety experts in many application domains such as automotive, aviation, industrial processes, nuclear, railway and space, and in software tools and technology provider companies. The objectives are to identify their main similarities and dissimilarities with an aim to a potential cross-domains harmonization, when possible and relevant, in terms of processes, approaches, methods and tools.

This paper continues a series of publications through which the Working Group members disseminate and encourage feedback about their findings. After general comparison of safety standards [WG-2010], and comparison of the safety assurance at system and at software levels [WG-2012a, WG-2012b] completed by a dedicated focus on the notion of criticality category (or DAL etc.) [WG-2012c], the Working Group focused on some aspects of system safety related to software through links between formal methods and testing [WG-2014] and a critical discussion of the so-called probabilistic safety assessment of software [WG- 2016a; WG-2016b].

Going back to a more general perspective but still dedicated to software, this paper proposes to investigate and discuss the notion of safety at software level.

In some domains, the safety standards do not explicitly refer to safety at software level. Software is expected to be compliant to the system requirements allocated to it, and safety is regarded only at the level of the system.

[1] The Working Group "Safety Standards" was created in 2008 and initially attached to the "Club des Grandes Entreprises de l'Embarqué (CG2E)".

In other domains, the safety standards explicitly identify safety requirements and safety activities at software level such as software safety analysis.

This paper describes some points of similarities and differences in different industrial domains regarding the way to consider safety at software level. We first address the global consideration about safety at software level through a cross domain comparison of software safety at different steps of the software design (requirements, analysis and implementation) and validation, and its positioning with respect to the global (system level) safety rationale. Then we focus on a category of software safety analyses recommended by many safety standards under names such as Software Failure Modes and Effects Analysis (Sw-FMEA) or Software Error Effects Analysis (SEEA) and discuss their possibilities and limitations. Finally we discuss and propose some perspectives in terms of software safety, inspired by the System Theoretic Process Analysis (STPA) [Leveson 2017] and contract-based approaches.

Notes on terminology:

- Safety is used in this paper following its definition, consistent with most safety standards, as “freedom from unacceptable risks”. Even though the safety assessment may be and is often supported by probabilistic measures (when appropriate), it must be understood that this can only be a part of the safety assessment, complemented by deterministic qualitative arguments. This is particularly true for what concerns the software. This paper does not aim at showing how to consider software in a probabilistic safety measure, but what is the position of software in the global safety rationale (and therefore, due to the current position of the safety standards which the paper intends to reflect, it is mostly the deterministic and qualitative arguments which are addressed).
- We do not discuss in this paper some issues sometimes considered as controversial about whether software can fail, or be unsafe. We will use wordings such as software safety which, if so desired, may be understood as a shortcut for the contribution of software to system safety (and similarly for software failure, error, fault).
- We use the expression “software correctness” as a short-cut encompassing the correctness and completeness of the software requirements (with regards to the system (safety) requirements and needs), and the correctness of the software implementation with respect to its requirements.

2. Safety rationale and software in standards across domains

2.1. Civil aviation, nuclear

In civil aviation and nuclear domains and in the associated applicable safety standards ([ARP 4754A, ARP 4761, DO 178C] for civil aviation, [IEC 61513, IEC 60880] for nuclear), the role of software regarding safety and its possible contribution to safety related events are of course fully recognised and addressed. However one can say that they are addressed mostly, if not only, through the notion of software correctness (as defined in the introduction). Notably, these standards never use expressions such as “software safety” (nor “software safety analysis”) and propose a framework where system safety, for what concerns software, is ensured and assessed through processes and activities dedicated to software correctness i.e., the avoidance of software failures (thanks to the avoidance or elimination of their causes, the errors or faults in software).

As a logical consequence, these standards do not recommend nor even mention at least explicitly any particular analysis of the occurrence, causes, propagation and consequences of incorrect software behaviour, other than, of course for the determination of software criticality (which we use here as a generic term for what is called Item Development Assurance Levels or Software Levels in civil aviation standards, and Software Classes in nuclear standards). Indeed the notion of software criticality can be seen as a counter-example to the claim that these domains and standards focus on software correctness. Moreover, the approach to determine software criticality category is based on an analysis of the consequences of potential failures of software i.e., an example of what we call here a “software safety analysis”. However it is worth noting that in these standards:

- The allocation is performed without explicit consideration, description and analysis of any specific failure mode of the concerned software. In fact it is performed by inheritance of categories allocated to upper level elements (functions in civil aviation, or systems and their role in the protection in nuclear), on the basis of system level safety analysis not considering software failures, and not even software itself.
- The rules applicable to the software development and validation according to the criticality category have as their only objective the correctness of the software. Even though the extent and rigour vary according to the category, it is still correctness that is sought and not an obligation to identify the possible software failures and mitigate their consequences.

2.2. Automotive, railway, industry

In the Base Safety Standard [IEC 61508] and in the derived domain specific standards for e.g., automotive [ISO 26262], process industry [IEC 61511] and railway [EN 50126, EN 50128, EN 50129], the focus is also put on software correctness (with specific focus on correctness regarding a sub-set of requirements identified as “safety requirements” in automotive [ISO 26262]). Also, these domains and standards require the allocation of criticality categories to software through the Safety Integrity Levels (SIL [IEC 61508, IEC 61511]), Software Safety Integrity Levels (SSIL [EN 50128]) and Automotive Safety Integrity Levels (ASIL [ISO 26262] and in the same way as for civil aviation and nuclear, they are allocated through a top-down analysis at system level without explicit consideration of software failure modes.

However, these standards also require or recommend the performance of dedicated “software safety analyses”. This should not be seen as challenging the validity of the general “correctness-based approach”, but rather as a complement towards better global efficiency. Therefore, the identification of software failure modes at low level and the analysis of their propagation and potential consequences, are expected to support the provision of dedicated protection or mitigation means during design and their further assessment.

2.3. Space

In the (European) space domain and related standards [ECSS Q40, ECSS Q80], the approach is also very similar, with a large focus put on correctness and an adaptation of the development and validation rigor according to software criticality categories (called “software categories” in these standards). It is worth noting that in their recent revision (revision 1, February 2017), these standards clarified the criticality allocation approach which is based, in the same way as for the other domains and standards described above, on a system level safety analysis leading to allocation of criticality categories to high-level functions, followed by an allocation to software by inheritance of the category of the function(s) that the software implements. Therefore, as for the other domains, the allocation is performed without explicit consideration of software failure modes and without any analysis performed at the level of the software and explicitly considering and analysing the software, its contents and its potential faults and failures.

However these standards also require the performance of software safety analyses, for software identified as safety-critical, with the objectives to:

- consolidate the system level safety analyses, in particular through the potential identification of new failure modes due to software, which could have not been identified from top-down analysis at system level,
- continue (if so desired) the allocation of criticality categories down to low-level software components.

It is worth noting that the ECSS Handbook on Software Safety and Dependability [ECSS Q80-03] mentions that according to the results of these software safety analyses, the allocation of software criticality category may have to be revisited.

2.4. Synthesis

The comparison of the safety standards (for systems and for software) in automotive [ISO 26262], aviation [ARP 4754A, ARP 4761, DO 178C], process industry [IEC 61511], nuclear [IEC 61513, IEC 60880], railway [EN 50126, EN 50128, EN 50129], space [ECSS Q40, ECSS Q80, ECSS Q80-03] and the base safety standard [IEC 61508] regarding the position of software in the global safety assessment, results in the main following points:

- A largely common core rationale where the potential for so-called “systematic failures” due to errors in the software are addressed through arguments on the correctness and completeness of the software requirements, and on the correctness of the software implementation with respect to these requirements. In theory, following this view, there would be no need to address safety at software level and address explicitly software safety and notably software failures.
- Of course this view suffers from some well-known limitations, covered in various standards through some adaptations to the theoretical rationale, with some differences between standards and domains:
  - The “criticality categories” (DAL etc.), to focus the effort in particular to ensure the completeness and correctness of software requirements and implementation, according to the consequences of (software) failures. We note however that in most standards, the categories are determined by an analysis at system level rather than at software level.
  - The explicit identification of safety requirements in the set of software requirements in some domains (e.g., automotive, as opposed to e.g., aviation), to focus the effort on some requirements identified as important for (system) safety. Note that here also, the identification is performed as a top-down approach from top-level system safety needs and requirements, without explicit consideration of software failures, errors and faults.
  - A focus on some particular detailed characteristics of software behaviour which are recognised as not always completely identifiable and controllable through a purely top-down approach. We can put in this category the notion of robustness, the so-called “derived requirements” in aviation, the analysis of “dependent failures” in automotive, or the definition of some specific development rules intended to prevent failure propagation.
  - A focus on some programming attributes understood as contributors to software safety such as robustness, maintainability, traceability and reliability can be identified in process industries [IEC 61511].

At this stage it can be noted that whereas the safety rationale is mostly considered as a system issue, and software (and more generally design, and all sources of systematic failures, development errors etc.) is addressed through correctness and completeness, many complements are added in order to cover the limitations of this view. This is where the notion of software safety, and analyses of software safety, come into the picture.

3. Common core: software correctness

As a common baseline approach across domains, the development process allocates criticality to system functions (or requirements, in automotive) according to their importance to safety. Each hardware and software component which contributes to the implementation of a function inherits the corresponding functional requirements (and additional requirements to ensure proper cooperation with the other components) as well as its criticality.

The software must comply with its requirements: the criticality level does not define some tolerance to deviation but the qualitative confidence level that the requirements are fulfilled.

The software life-cycle is therefore built to provide due evidences of correctness based on design and verification activities (as “verification” and “validation” have different meanings in different standards, we use here “verification” as a generic term).

Design activities are constrained by process rules (e.g. split the design in a number of formalized phases commensurate with the criticality, in order to proceed by small steps, less prone to error) and design rules intended to avoid common mistakes and make verification more efficient.

Verification activities have to confirm the design correctness; they do not consist in “bug hunting”. Thus, their effectiveness cannot be measured by the number of bugs found but must be justified by their ability to find hypothetical errors.

The methods and tools used for verification all aim at finding hypothetical errors:

- document review;
- simulation, e.g. to assess the completeness and the consistency of the software requirements;
- functional testing with structural coverage measurement, to provide evidence that the implemented behaviour matches the required one at least on the covered paths;
- specific analyses depending on the design choices, e.g. worst case resource use (processor time, memory, stack, etc.), state-transition analyses, etc.;
- formal methods are increasingly used and recognized by the standards, thanks to the scientific and technical progress; they are now frequently used to prove the absence of run-time errors and the fulfilment of low-level functional properties; it is expected that they scale up to higher-level properties in a near future.

Overall, all software analyses are performed to demonstrate the software correctness. Of course, software and system designers are aware that residual errors may exist in the requirements or in the implementation of software or hardware, or in their combination; this is mitigated, as far as appropriate, at some higher level of the system architecture, typically by providing some other means to reach the functional objective.

Regarding how software errors may lead to system failures, it is noted that methods (such as FMEA) used to analyse the consequences of hardware component failures are not applicable to software because:

- such methods require the knowledge of all plausible failure modes of the elementary components, which is feasible for hardware (e.g. relay stuck open or stuck close) but not for software: the instructions themselves do not fail and it is not possible to identify all plausible design errors (i.e. all plausible wrong sequences of instructions);
- they also require the knowledge of all propagation paths from the root failures to the outputs; this is feasible for hardware (by following the physical connections), but not for software where an error in a component may e.g. unduly modify some memory used by another component (e.g. through stack overflow or incorrect pointer value), even if there is no apparent connection between them.

Thus, the system analyses of the consequences of software errors do not try to identify specific errors or to trace their propagation through the software; they rather postulate that some function depending on software may fail for an unknown reason, assess the consequences and, if needed, requires complementary means to reach the functional objectives.

4. Software FMEA, SEEA

Complementing the correctness-oriented approach for software in safety critical systems, the base safety standard [IEC61508] and some domain-specific standards (notably automotive, railway and space) recommend the performance of dedicated software safety analyses.

Many of the safety analysis techniques proposed for software are indeed the application to software of safety analysis techniques used for other elements than software (generally hardware), but often missing a clear assessment that such an extrapolation is sound with respect to the particular characteristics of software and of software faults, errors and failures, as compared to hardware. For instance the Software Failure Modes and Effects Analysis (Sw-FMEA), recommended by safety standards in automotive, industry, railway and space, is the application to software of the well-known Failure Modes and Effects Analysis (FMEA) (see e.g., [ECSS Q30-02]), widely used mostly for random faults of hardware. This technique consists in analysing a system as an assembly of components where the assembly itself is considered correct, and all components but one are considered correct. The potential failure modes of this component are postulated, one at a time, and their propagation over the assembly and their consequences are analysed. The analysis is repeated for all failure modes of the component, and for all components.

In addition to other well-known strong limitations such as the lack of sound and recognised fault models for software, an important issue can be identified in the extrapolation to software of the FMEA, and in particular on a detailed description of low-level software components: it comes from the fact that most of its fundamental principles are no longer valid. Notably, when some component is considered as potentially faulty, it is unclear why we could postulate that the assembly (i.e. here, the software architecture) is correct and that all the other components (i.e., other elements inside the analysed software) are correct. Therefore one may have some doubts about the effectiveness of an analysis focusing on only one element, all other ones and their assembly being assumed correct, without even a clear definition of what “being incorrect” may mean for the considered element. By the way, it is very possible that as described in the representation of the software used for the analysis, this element is indeed incorrect. In this case, postulating a potential failure mode on it and analysing its propagation and consequences as usually done in FMEA (i.e., in an abstract way, as a deviation with respect to what would be observed without the postulated failure mode, which is assumed to be correct), may appear as quite questionable. Paradoxically, one could even note that it may be one of the postulated failure modes which indeed corresponds to the “theoretically correct software”.

Moreover when performing the analysis on a very detailed representation of the software, as recommended by some standards e.g. in space, one may question the real meaning of a postulated failure mode on for instance, an instruction. Indeed, when an instruction “does not behave as written”, what is analysed is not a fault in the software but rather a fault in its execution support. Conversely, postulating failure modes on a very detailed software representation is likely to miss fault cases such as a given group of instructions is not the right one, or should not exist, or should be somewhere else, or another group of instructions is missing etc. In fact it misses the analysis of the potential faults relevant to all the design steps out of which the analysed software representation is obtained. In other words, the more detailed the software representation, the more typical software faults escape the analysis.

Besides that, there is a clear interest in analysing fault propagation “on the real thing” and even, despite the fact that the analysis becomes rapidly tedious and costly, on a detailed representation of it. This suggests that there might be an interest for Sw-FMEA in distinguishing more clearly than in FMEA the representation on which faults, or failure modes, are postulated, and the representation on which their propagation and consequences are analysed. This could take the form of two complementary analyses:

- A FMEA-like analysis performed only on a representation where all or at least some components correspond to software i.e.,
    - at a level of abstraction where the considered failure modes may be considered as reasonably representative,
    - and where the assembly of the components may be postulated correct;
- A more detailed analysis of the error propagation that may complement the former analysis, if necessary.

It is worth noting that the latter analysis differs from a classical FMEA in the sense that it addresses the propagation of erroneous data through a given software component from its inputs, and not the propagation of errors due to faults inside this component, which is exactly the opposite for a classical FMEA. It seems then closer to the technique known as Software Error Effect Analysis (SEEA) at least as practiced in some domains, even though to our knowledge, the safety or software standards which mention or recommend it do not provide details about how it should be performed.

5. Focus on completeness of system functional safety requirements

Though widely used in railway and automotive standards, the term “functional safety” is not used in that of aviation and nuclear for instance. Some preliminary definition of “functional safety”, that could also be called “behavioural safety”, is needed.

We distinguish functional safety from architectural safety.

Architectural safety, on the deterministic side, is in charge of preventing single cause failures by architectural mitigation. Fault Tree Analysis is one of the main analysis methods associated to it. Functional safety is in charge of defining safety regions[2] and controls that make them invariants of system dynamics. Behavioural modelling and simulation (M&S) is one of the main analysis methods associated to functional safety.

As control is now almost always software-based, functional safety might be the joint concern of system assurance and software assurance.

As explained in section 3, system assurance is in charge of ensuring correctness and completeness of the functional safety requirements. Software assurance, in aviation and nuclear at least, *has to* take them for granted.

We question the following aspects:

- Is current M&S-based analysis of system functional safety on par with systems' complexity?
- How does DAL-modulated rigor bear on functional safety *specification* verification?
- Do the software FMEA and SEEA contribute to functional safety specification verification?

Following [Leveson 2011] we advocate that safety standards lag behind complexity of *software-intensive* systems, and even more so for systems of systems (IoT, fleet of autonomous vehicles, new air traffic control etc.). The system/software dichotomy on functional safety specification seems outdated, and even paradoxical.

We mention some prospective work intended to overcome this situation, especially in the context of civil aviation where a tentative reformation of system, software, and hardware assurance standards is in progress [Lingberg 2017], [Delseny 2017].

System Modelling and Simulation for Functional Safety Specification Verification

System M&S has been continuously progressing over the past 30 years. Mono-system/mono-physic control modelling appeared in the mid 80's (e.g., MALAB/Simulink, SciLab, etc.), as well as specialized tools for specific and hard to simulate physics (e.g. SABER for electricity, Amesim for thermo-fluidic systems). Multi-system/multi-physic modelling, required for Cyber-Physical Systems (CPS), has been emerging only in the 21st century [Tiller 2001]. Simulink and Modelica-based tools have true hybrid system modelling capabilities: discrete time operators for software-based control, and continuous time operators for the differential equations of physics. However, these tools are fundamentally continuous time modellers. Representation of the cyber part is necessarily partial and limited to interaction with physics. The behaviour of hundreds of thousands of lines of code yet to be specified is not (and can't be) represented in such verifications.

So M&S functional safety specification verification is necessarily *partial* at system level.

Moreover, the cost of developing such models, and the required close collaboration between prime and suppliers to assemble extensive system functional simulations, may make system modelling unaffordable or intractable. Incompleteness likelihood, of functional safety specifications is even greater in these model-free situations. Common practice is paper specification, verified by review.

In both cases, model-free and model-based system engineering, we do question the validity of the assumption made by the assurance standards about completeness of functional safety specification, when done at system level only. On ground of this assumption some standards *require* safety analysis *not* to be addressed at software level, even as an awareness duty (e.g., safety agnosticism of DO 178).

Safety Regions

Safety Regions are defined as logical constraints over controlled or uncontrolled, physical or digital, state variables.

Safety is an "always" state property: the dynamics of the system (or SoS) must always meet these constraints i.e., be kept within the safety region. Hence the use of *invariants* in the rest of this section.

Use of invariants can be extended from software to systems. Like for software, most of them can be programmed as executable assertions for M&S verification activities [Otter 2015]. They are amenable to a *uniform* approach to functional safety specification, across system, software and hardware. It is worth mentioning that formal verification of invariants has been standard industrial practice for more than a decade in hardware engineering [Chang 1999].

Extension of *contract-based* specification to system may be seen as an attempt at introducing *compositionality* and *refinement* of invariants [Benveniste 2015]. A contract is a set of invariants anchored to an interface (system, software, or hardware). Most importantly it is also a way of compelling specifiers to separate behaviours from their expected properties, to separate the 'how' from the 'what'. Safety regions are on the 'what' side.

This separation and compositional approach introduced in software engineering [Meyer 1997] starts being considered in software assurance standards.

[2] Conditional predicates over continuous states (phase space safety regions) and over discrete states (configuration space safety regions).

This is only emerging in system engineering and system assurance. Such a separation is an incentive for defining *explicit* safety regions, and for managing more effectively functional safety throughout very complex systems. Otherwise, safety regions are likely to remain unstated properties ensured by control behaviours whose formulation is the only explicit part of functional safety specification.

System-Software uniform approach to functional safety specification

System on chip [Chang 1999] has definitely blurred the formerly sharp frontier between system / software / hardware.

For functional safety specification, behavioural scope or "grain" or "size", in other words a notion of *scale* of 'dynamic complexity' is more relevant than implementation technology.

Key concepts to specify functional safety are:

- *Interface*: service-based, signal-based, message-based etc.
- *States*: physical or logical, discrete or continuous, inner or outer, their mapping on architecture and their associated observability conditions. They may include interface inputs and outputs.
- *Safety invariants*: conditioned predicates over observable states. They define safety regions on continuous states, and configuration subspaces on discrete states. So-called "*non-functional*" properties like timing properties or absence of Run-Time Errors (RTE) are included within the *functional* safety invariants,
- *Safety contracts*: structured sets of safety invariants. A contract is associated to an interface that encloses a set of inner states. It is partitioned in two groups of invariants: assumptions and guarantees. For a comprehensive theory of hybrid system contracts that fit system contracts, software contracts and hardware contracts, see [Benveniste 2015, Benveniste 2014]. For a case study related to civil aviation see [Wilkinson 2016].
- *Functional safety controls*: the software-based, hardware-based or physics-based actions that evolve the states subject to safety invariants.

A system / software / hardware uniform approach to functional safety specification, including contract decomposition and refinement guarantees, is possible on the basis of these concepts. It is under experimentation on the use case of a collaborative project dedicated to system certification reformation in civil aviation [Lingberg 2017], [Delseny 2017], [Ledinot 2017].

Enhancement with STPA

STPA [Leveson 2011] goes a step further and proposes a uniform top-down "correct by construction" approach to functional safety specification, at any scale (i.e. from SoS down to FPGA, ASIC, RTOS or software library). It has been inspired by forensics of aerospace accidents and by control theory. Its key concept is *control loop*.

All current safety assurance standards are reliability oriented and "cascading effects" oriented.

By "reliability oriented" we mean that causes of safety escapes, i.e. violation of some safety invariant, are primarily envisioned as consequences of breakdowns.

By "cascading effects oriented" we suggest that *causal loops* are likely to be addressed by chance. "Cascading" suggests waterfall, and waterfall suggests DAG[3] topology of causal chains. Cascading suggests first wave propagation, possibly with fork and join of causal flows, but not iteration of effects over long time horizons.

Notice that causality cycle is an issue in system dysfunctional modelling (e.g., AltaRica [Rauzy 2017], and even more so in Fault Tree Analysis).

Figure 1 presents the generic pattern used to analyse and model accident causation (forensic), or to design functional safety (fault prevention and elimination). This pattern may be used in a hierarchical manner (nested control loops) or in a "flat" way (several instances at the same level, possibly with mutual dependences).

The states mentioned in the previous section are either located in the *Controlled process*, or in the *Controller*. Both can be recursively decomposed into lower level control loops. The loops are the structure of influence (informational, physical, human) that evolve the states subjected to safety invariants. Detection of possible safety escapes (i.e., invariant violations) is addressed through detection of how wrong control action could occur within these loops.

STPA proposes check-lists and structured analysis best-practices [Leveson 2011] that we anticipate could help reaching correctness and completeness of functional safety specification.

Notice that the pattern includes *human factor* and process *modelling errors*.

[3] Direct Acyclic Graph

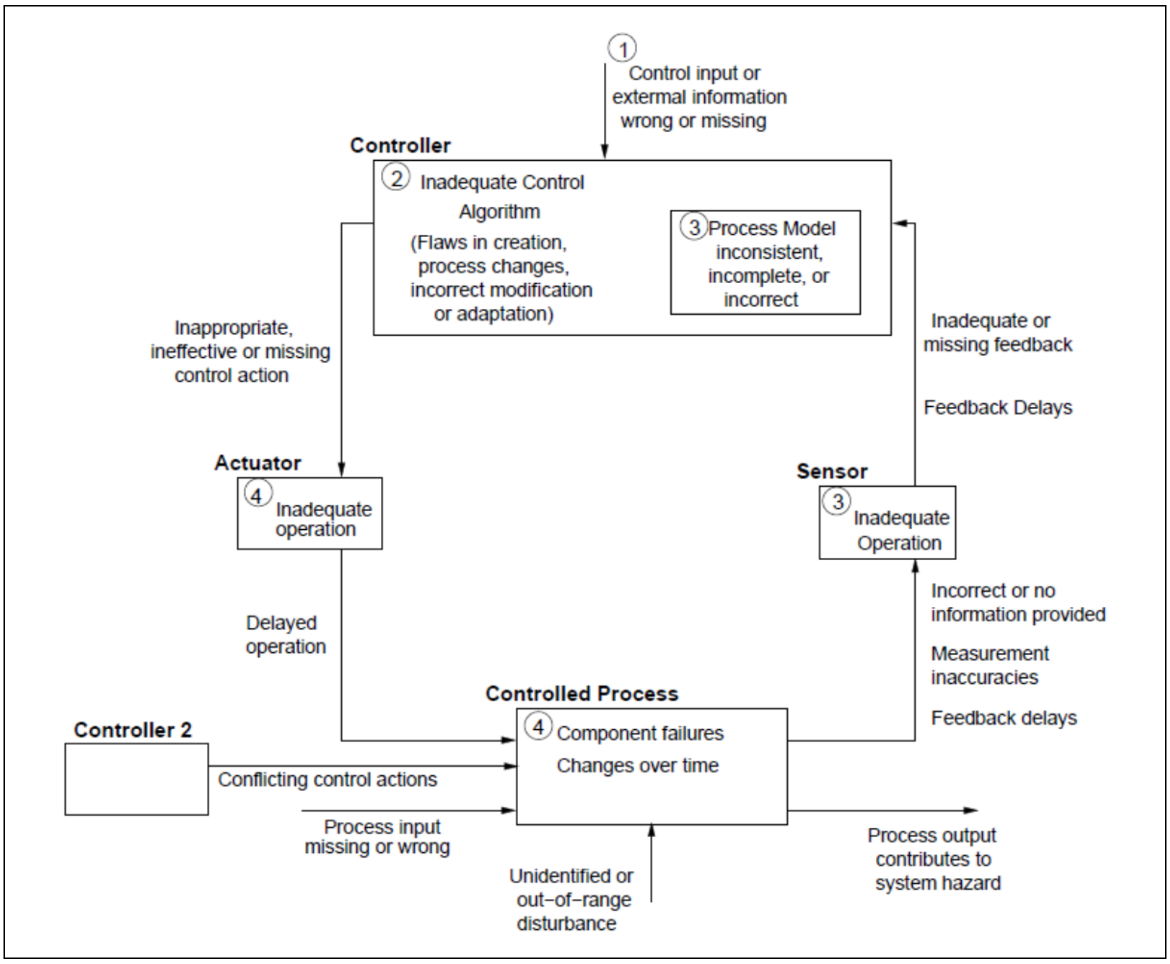
1
Control input or external information wrong or missing
Controller
2 Inadequate Control Algorithm
(Flaws in creation, process changes, incorrect modification or adaptation)
3 Process Model inconsistent, incomplete, or incorrect
Inappropriate, ineffective or missing control action
Inadequate or missing feedback
Feedback Delays
Actuator
4 Inadequate operation
Sensor
3 Inadequate Operation
Delayed operation
Incorrect or no information provided
Measurement inaccuracies
Feedback delays
Controlled Process
4 Component failures
Changes over time
Controller 2
Conflicting control actions
Process input missing or wrong
Unidentified or out-of-range disturbance
Process output contributes to system hazard

Figure 1: STPA's fundamental analysis and design pattern

Software Safety Analysis

A third approach to this notion, in addition to software FMEA and SEEA discussed previously, would be rejection of the *completeness hypothesis* associated to system functional safety specification.

The vision would be the following: functional safety is first and primarily specified at system level. Software correctness i.e., *conformance* to specified functional safety is the primary assurance goal. On this part, let us assume 80% of functional safety specification, *safe software is correct software*. But, considering modern software complexity, detailed software specification necessarily defines behavioural enrichments and safety needs that are unknown when this first system level specification stage is carried out, so that completion of functional safety specification (the remaining 20%) would need an additional system-software *co-engineering* step.

Aviation standards address these possible enrichments through the "derived requirements" mechanism in a fully asymmetric manner: safety agnosticism at software level and subordination to system level. In other standards e.g., [IEC 61508] safe software is an objective and correctness is a mean.

Potential convergence of system and software engineering through contract & model-based engineering, in addition to redundancy considerations[4], lead us to advocate a more symmetrical, top-down and bottom up vision of functional safety specification at the system-software frontier.

Software-intensive distributed real-time control, intertwined with distributed fault tolerance mechanisms, is the prototypical situation where we feel such symmetrical safety-aware co-engineering is needed.

Pragmatically speaking, and ideally speaking, this third software safety analysis would consist in:

- Extending the explicit definition of the safety invariants, in a contract-based setting, at software architecture level, with or without STPA-based enhancements,
- Introducing the safety-contract compositionality verification goals into the software verification goals.

[4] i.e. Swiss cheese considerations in development assurance

Note that in this third software safety analysis, *refinement correctness* of safety contracts is the analysis' driver. It is not a "what if" analysis driven by assumptions of perturbation initiators. Detection of interaction failures in nominal cases [Leveson 2011] is the primary goal. Since invariants mean "always", abnormal cases are also within the scope.

6. Conclusion

Software safety as such seems to be understood in all standards as a short-cut for "the contribution of software to system safety". This contribution is primarily obtained through software compliance to its requirements. This cannot be enough as full compliance and completeness may be difficult to ensure, and different domains have proposed additional measures to try to fill the gap through notions such as e.g., robustness, dependent failures or derived requirements.

Some standards recommend the performance of software safety analyses. A simple extrapolation to software of system or hardware analyses techniques such as FMEA is unlikely to provide meaningful results, considering that the underlying assumptions such as the fault model do not apply to software. However techniques such as SEEA (Software Error Effect Analysis) may provide some support to robustness analysis.

The functional behaviour of large software-intensive systems tends to become more and more complex, because software allows incorporating more and more functions, with higher flexibility and (supposedly) lower cost than other technologies.

In this context, system safety increasingly depends on complex interaction loops between software and other system parts, which complicates the achievement and the demonstration of correctness and completeness of the software requirements and of the software itself.

Thus, some system-software co-engineering (i.e., not only a strict top-down approach) becomes mandatory to ensure the adequacy of software requirements, and the proper development of such pieces of software needs the generalization of techniques such as contract-based design with compositional verification, consistent safety invariants at all design levels, and a more control-oriented approach to safety.

7. References

[ARP 4754A] "Guidelines for Development of Civil Aircraft and Systems", EUROCAE ED-79A and SAE ARP-4754A, 21/12/2010.

[ARP 4761] "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment", EUROCAE ED-135 and SAE Aerospace Recommended Practice ARP-4761, 12/1996.

[Benveniste 2014] A. Benveniste, D. Nickovic, T. Henzinger "Compositional Contract Abstraction for System Design". INRIA Research Report RR 8460, January 2014.

[Benveniste 2015] A. Benveniste, B. Caillaud et al. "Contracts for System Design: Theory". INRIA Research Report RR 8759, July 2015.

[Chang 1999] Chang H. Cooke L et al. "Surviving the SOC Revolution – A guide to platform-based development", Kluwer Academic Publishers 1999.

[Delseny 2017] H. Delseny "Defining the Overarching Properties: The RESSAC Project". Certification Together Conference (CTIC), Toulouse, March 21-23 2017.

[DO 178C] "Software considerations in airborne systems and equipment certification", ED 12 (EUROCAE WG-12) and DO 178 (RTCA SC-105), issue C, 5/1/2012.

[ECSS Q30-02] "Space product assurance – Failure modes, effects (and criticality) analysis (FMEA/FMECA)", European Cooperation for Space Standardisation, ECSS-Q-ST-30-02C, 6/3/2009.

[ECSS Q40] "Space product assurance – Safety", European Cooperation for Space Standardisation, ECSS-Q-ST-40C rev.1, 15/2/2017.

[ECSS Q80] "Space product assurance – Software", European Cooperation for Space Standardisation, ECSS-Q-ST-80C rev.1, 15/2/2017.

[ECSS Q80-03] "Space product assurance – Software dependability and safety", European Cooperation for Space Standardisation, ECSS-Q-ST-80-03A rev.1, 20/11/2017.

[EN 50126] “Railway applications – The specification and demonstration of reliability, availability, maintainability and safety (RAMS)”, CENELEC, EN 50126, 1999 AMD 16956, 28/2/2007.

[EN 50128] “Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems”, CENELEC, EN 50128:2001, 15/5/2001.

[EN 50129] “Railway applications – Communications, signalling and processing systems – Safety related electronic systems for signalling”, CENELEC, EN 50129:2003, 7/5/2003.

[EN 50159] “Railway applications – Communications, signalling and processing systems, Parts 1 and 2, CENELEC, EN 50159-1:2001 (11/2001) and EN 50159-2:2001 (12/2001).

[IEC 60880] “Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions”, IEC 60880, edition 2.0, 2006-05.

[IEC 61508] “Functional safety of electrical/electronic/ programmable electronic safety-related systems, IEC 61508 Parts 1-7, Edition 2.0, 4/2010.

[IEC 61511] “Functional safety – Safety instrumented systems for the process industry sector. IEC 61511 Parts 1-3, edition 1.0, 3/2003.

[IEC 61513] “Nuclear power plants – Instrumentation and control for systems important to safety – General requirements for systems”, edition 1.0, 22/3/2001.

[ISO 26262] “Road vehicles – Functional safety” ISO 26262 Parts 1-9, first edition, 2011-11-15, ISO 26262 Part 10, 2012-08-01.

[Ledinot 2017] E. Ledinot “Experimenting with the Overarching Properties: a use case by the RESSAC Project”, Certification Together Conference (CTIC), Toulouse, March 21-23 2017.

[Leveson 2011] N. G. Leveson, “Engineering a Safer World – Systems Thinking Applied to Safety”. MIT Press 2011.

[Lingerg 2017] B. Lingberg, “FAA Software, AEH, Systems Activites and Roadmap”. Certification Together Conference (CTIC), Toulouse, March 21-23 2017[Meyer 1997] B. Meyer “Object-Oriented Software Construction” 2nd Edition Prentice Hall 1997.

[Otter 2015] M. Otter, N. Thuy “Formal Requirements Modeling for Simulation-Based Verification” 11th International Modelica Conference, Versailles, September 21-23, 2015.

[Rauzy 2017] A. Rauzy, “Model-Based Safety Assessment with Altarica 3.0” ESREL Conference, Portoroz Slovenia, June 18-22, 2017.

[Tiller 2001] M. Tiller, “Introduction to Physical Modeling with Modelica”, Kluwer Academic Press 2001.

[WG- 2010] P. Baufreton et al., “Multi-domain comparison of safety standards”, ERTS-2010.

[WG- 2012a] J. Machrouh et al., “Cross domain comparison of System Assurance”, ERTS-2012.

[WG- 2012b] E. Ledinot et al., “A cross-domain comparison of software development assurance standards”, ERTS-2012.

[WG- 2012c] JP. Blanquart et al., “Criticality categories across safety standards in different domains”, ERTS-2012.

[WG- 2014] E. Ledinot et al., “Joint use of static and dynamic software verification techniques: a cross-domain view in safety critical system industries”, ERTS-2014.

[WG- 2016a] E. Ledinot et al., “Perspectives on Probabilistic Assessment of Systems and Software”, ERTS-2016.

[WG- 2016b] JP. Blanquart et al., “Software Safety Assessment and Probabilities”, DSN-2016.

[Wilkinson 2016] C. Wilkinson “Integration of complex digitally intensive systems” – FAA Streamlining Assurance Processes Workshop – Dallas, September 13-15, 2016.